IX BRAZILIAN
CONFERENCE ON SOFTWARE

CBSOFT

SÃO CARLOS 2018

PROCEEDINGS

VI Workshop de Visualização, Evolução e Manutenção de Software

VI Workshop on Software Visualization, Evolution and Maintenance

ORGANIZATION:

ICMC USP SÃO CARLOS

ufscar

INSTITUTO FEDERAL
São Paulo
Câmpus São Carlos

PROMOTION:

SBC Sociedade Brasileira de Computação

SUPPORT:

CAPES

FAPESP

Prefeitura de SÃO CARLOS

Anais – *Proceedings*

VI Workshop de Visualização, Evolução e Manutenção de Software

VI Workshop on Software Visualization, Evolution and Maintenance

IX Congresso Brasileiro de Software: Teoria E Prática (CBSoft 2018)
IX Brazilian Conference on Software: Theory and Practice (CBSoft 2018)

Dados Internacionais de Catalogação na Publicação
Sociedade Brasileira de Computação
Universidade de São Paulo, São Carlos

Workshop de visualização, evolução e manutenção de software (6. : 2018 : São Carlos).

Anais do 6º Workshop de visualização, evolução e manutenção de software [recurso eletrônico] / 6º Workshop de visualização, evolução e manutenção de software, 19 de setembro, 2018; editado por Thiago Gottardi. – São Carlos: Universidade de São Paulo, 2018. 197 p.

Tema central: Software: Teoria e Prática.
Disponível em: `https://vem2018.github.io`;.
ISBN: 978-85-7669-447-2

VI Workshop de Visualização, Evolução e Manutenção de Software

19 de setembro, 2018
São Carlos, SP, Brazil

ANAIS | *PROCEEDINGS* Sociedade Brasileira de Computação (SBC)

COORDENADORES DO PROGRAMA | *PROGRAM COMMITTEE CHAIRS*

Marco Tulio Valente (UFMG, Brazil)
Elder Cirilo (UFSJ, Brazil)

COORDENADORES GERAIS | *GENERAL CHAIRS*

Elisa Yumi Nakagawa (General Chair ICMC/USP)
Rosana Vaccare Braga (General Co-Chair ICMC/USP)
Valter Vieira de Camargo (General Co-Chair DC/UFSCar)
Auri Marcelo Rizzo Vincenzi (General Co-Chair DC/UFSCar)
Daniel Lucrédio (General Co-Chair DC/UFSCar)
Lucas Bueno Ruas de Oliveira (General Co-Chair IFSP/São Carlos)

COMITÊ GESTOR DO CBSOFT 2018 | *CBSOFT 2018 STEERING COMMITTEE*

Fernando Trinta (UFC) (President and General Chair CBSoft 2017)
Marum Simão Filho (UNI7) (General Co-Chair CBSoft 2017)
Rossana Andrade (UFC) (General Co-Chair CBSoft 2017)
Elisa Yumi Nakagawa (ICMC/USP) (General Chair CBSoft 2018)
Rosana Braga (ICMC/USP) (General Co-Chair CBSoft 2018)
Auri Vincenzi (DC/UFSCar) (General Co-Chair CBSoft 2018)
Valter Vieira de Camargo (DC/UFSCar) (General Co-Chair CBSoft 2018)
Daniel Lucrédio (DC/UFSCar) (General Co-Chair CBSoft 2018)
Lucas Oliveira (IFSP) (General Co-Chair CBSoft 2018)
José Carlos Maldonado (ICMC/USP) (Coordinator CEES and Program Chair SBES 2017)
Fabiano Cutigi Ferrari (DC/UFSCar) (Coordinator CEES Coordinator SBES 2017)
Fabio Mascarenhas (UFRJ) (Coordinator CELP Coordinator SBLP 2017)
Uirá Kulesza (UFRN) (Program Chair SBES 2018)
Carlos Camarão (UFMG) (Program Chair SBLP 2018)
Márcio Ribeiro (UFAL) (Program Chair SBCARS 2017)
Ingrid Nunes (UFRGS) (Program Chair SBCARS 2018)
Arilo Dias Neto (UFAM) (Program Chair SAST 2017)
Patricia Machado (UFCG) (Program Chair SAST 2017)
Guilherme Horta Travassos (COPPE/UFRJ) (Program Chair SAST 2018)
Avelino Francisco Zorzo (PUC-RS) (Program Chair SAST 2018)

COORDENADOR DOS ANAIS | *PROCEEDINGS CHAIR*

Rosana Teresinha Vaccare Braga (ICMC/USP)

EDITORAÇÃO DO LIVRO DE ANAIS | *PROCEEDINGS BOOK EDITOR*

Thiago Gottardi (ICMC/USP)

ARTE DA CAPA | *COVER ART*

Jonathan Silva (IFSP/São Carlos)

DESENVOLVIMENTO WEB | *WEB DEVELOPMENT*
Thiago Luiz Varella (IFSP/São Carlos)

GERÊNCIA DE COMUNIDADES E WEB | *WEB AND COMMUNITY MANAGERS*
Lina Garcés (ICMC/USP)
Tiago Volpato (ICMC/USP)
Ana Allian (ICMC/USP)
Warteruzannan Soyer Cunha (DC/UFSCar)
Jonathan da Silva (IFSP/São Carlos)
Thiago Luiz Varella (IFSP/São Carlos)

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

ORGANIZAÇÃO | *ORGANIZATION*
Instituto de Ciências Matemáticas e de Computação – Universidade de São Paulo (ICMC/USP)
Universidade Federal de São Carlos (UFSCar)
Instituto Federal de São Paulo (IFSP-São Carlos)

APOIO | *SUPPORT*
Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP)
Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES)
Faber-Castell
Prefeitura de São Carlos

PATROCINADORES | *SPONSORS*
UOL/PagSeguro, B2W Digital
Serasa Experian, Monitora, IBM, Google
S2.it, CeMEAI

Prefácio | *Foreword*

The VI Workshop on Software Visualization, Evolution and Maintenance (VEM 2018) is part of the IX Brazilian Congress on Software: Theory and Practice (CBSoft 2018), held in São Carlos – SP, from September 17 to 21, 2018. 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. The VEM 2018 Program Committee (PC) is composed of 47 active researchers in the areas of software visualization, evolution and maintenance, who come from several regions of Brazil. The PC members selected 24 interesting and promising short papers to be presented at VEM 2018, from a total of 39 submissions. Each submission was evaluated by at least three PC members, based on their originality, technical quality and adequacy to the event's scope.

In addition to the 24 papers selected by the PC, the VEM 2018 technical program also includes two invited keynote talks where the event participants could discuss the main problems and solutions related to software visualization, evolution and maintenance.

Finally, we would like to express our deepest gratitude to all authors who submitted their work to VEM 2018, for their interest, to the PC members, for their effort and invaluable collaboration during the paper selection process, and to the CBSoft 2018 organizers and sponsors, for their support and contribution.

Marco Tulio Valente (UFMG, Brazil)
Elder Cirilo (UFSJ, Brazil)
Organizadores do Programa | *Program Co-Chairs* - VEM 2018

Coordenadores do Comitê de Programa | *Program Chairs*

Marco Tulio Valente (UFMG, Brazil)
Elder Cirilo (UFSJ, Brazil)

Organizadores do Comitê de Programa | *Organizing Committee*

Marcelo Maia (UFU, Brazil)
Marcelo Schots (UERJ, Brazil)
Claudio Sant'Anna (UFBA, Brazil)
Ricardo Terra (UFLA, Brazil)
Eduardo Figueiredo (UFMG, Brazil)
Nabor Mendonça (UNIFOR, Brazil)

Comitê de Programa | *Program Committee*

Andre Hora (Federal University of Mato Grosso do Sul)
Auri Vincenzi (Federal University of São Carlos)
Baldoino Fonseca (Federal University of Alagoas)
Bruno Cafeo (Federal University of Mato Grosso do Sul)
Bruno da Silva (California Polytechnic State University)
Christina Chavez (Federal University of Bahia)
Claudio Sant'Anna (Federal University of Bahia)
Eduardo Figueiredo (Federal University of Minas Gerais)
Eduardo Guerra (National Institute of Space Research)
Eiji Adachi Barbosa (Federal University of Rio Grande do Norte)
Elder Cirilo (Federal University of São João del-Rei)
Elisa Huzita (State University of Maringa)
Fabio Petrillo (Federal University of Rio Grande do Sul)
Fernando Castor (Federal University of Pernambuco)
Guilherme Avelino (Federal University of Piauí)
Gustavo Pinto (Federal University of Pará)
Heitor Costa (Federal University of Lavras)
Henrique Rocha (INRIA)
Humberto Marques-Neto (Pontifical Catholic University of Minas Gerais)
Igor Steinmacher (Federal University of Technology (Paraná)
Igor Wiese (Federal University of Technology (Paraná)
Ingrid Nunes (Federal University of Rio Grande do Sul)
Ivan Machado (Federal University of Bahia)
Kecia Ferreira (CEFET-MG)
Leopoldo Teixeira (Federal University of Pernambuco)
Lincoln Rocha (Federal University of Ceará)
Luciana Silva (Federal Institute of Minas Gerais)
Marcelo Maia (Federal University of Uberlandia)
Marcelo Schots (Rio de Janeiro State University)
Marco Tulio Valente (Federal University of Minas Gerais)
Maria Istela Cagnin (Federal University of Mato Grosso do Sul)
Marx Viana (Pontifical Catholic University of Rio de Janeiro)
Nabor Mendonca (University of Fortaleza)
Patrick Brito (Federal University of Alagoas)
Rafael Durelli (Federal University of Lavras)
Regina Braga (Federal University of Juiz de Fora)
Renato Novais (Federal Institute of Bahia)
Ricardo Terra (Federal University of Lavras)
Roberta Coelho (Federal University of Rio Grande do Norte)

Rodrigo Bonifacio (University of Brasilia)
Rodrigo Souza (Federal University of Bahia)
Rogerio de Carvalho (Fluminense Federal Institute)
Rogerio Garcia (São Paulo State University)
Rosana Braga (São Paulo University)
Uirá Kulesza (Federal University of Rio Grande do Norte)
Valter Camargo (Federal University of São Carlos)
Vinicius Durelli (Federal University of São João del-rei)

Revisores Adicionais | *Additional Reviewers*

Allan Mori (Federal University of Minas Gerais)
Fernanda Madeiral (Federal University of Uberlândia)
Fischer Ferreira (Federal University of Minas Gerais)
Jairo Souza (Federal University of Alagoas)
Jailton Coelho (Federal University of Minas Gerais)
João Montandon (Federal University of Minas Gerais)
João Paulo Diniz (Federal University of Minas Gerais)
Rodrigo Lima (Federal University of Alagoas)

Palestrantes Convidados | *Invited Speakers*

Marcelo Maia

UFU, Brazil

Prof. Marcelo Maia is full professor at the Faculty of Computing at the Federal University of Uberlândia, Brazil. He started his academic career in 1993 at Federal University of Ouro Preto. He received his PhD in Computer Science from the Federal University of Minas Gerais (UFMG) in Brazil in 1999. He has been awarded with a Google Research Award Latin America in 2016. He has advised more than 20 graduate students, including two PhDs. Currently, he heads LASCAM – Laboratory of Software, Comprehension, Analytics and Mining[1], collaborating with researchers in USA, Canada, Italy, France and Sweden, advising 9 PhD students.

[1] `http://lascam.facom.ufu.br`

Massimiliano Di Penta

University of Sannio, Italy

Massimiliano Di Penta is an associate professor at the University of Sannio, Italy. His research interests include software maintenance and evolution, mining software repositories, empirical software engineering, search-based software engineering, and testing. He is author of over 250 papers appeared in international journals, conferences and workshops, and received various awards for his research and reviewing activity, including two most influential paper awards (SANER 2017 and GECCO 2015) and three ACM SIGSOFT Distinguished Paper Awards (ICSE, FSE and ASE). He serves (and has served) the organizing and program committees of over 100 conferences such as ICSE, FSE, ASE, ICSME, ICST, MSR, SANER, ICPC, GECCO, WCRE, and others. He is currently member of the steering committee of ASE, MSR, and PROMISE. Previously, he has been steering committee member of other conferences, including ICSME, ICPC, SSBSE, CSMR, SCAM, and WCRE. He is in the editorial board of ACM Transactions on Software Engineering and Methodology, the Empirical Software Engineering Journal edited by Springer, and of the Journal of Software: Evolution and Processes edited by Wiley. He has served the editorial board of IEEE Transactions on Software Engineering.

Further information on his research can be found at the following links: Google Scholar[2]; DBLP[3].

[2] `https://scholar.google.com/citations?user=j6ucyOAAAAAJ`
[3] `http://dblp2.uni-trier.de/pers/hd/p/Penta:Massimiliano_Di`

Sumário | *Table of Contents*

Palestras | *Keynotes*

Trabalhos Completos | *Full Papers*

The promises and perils of mining

Marcelo Maia
UFU, Brazil

Keynote

Abstract

Mining software repositories (MSR) has matured as an important research field. The body of knowledge provided by empirical work on MSR is becoming increasingly important to support a wide range of software engineering tasks. In this talk, we provide a personal reflection on the attempt to organize a research group around this theme. As expected, we will show some results, but also raise possible perils that may hinder long-term impact of the group.

Are we Software Engineers or Lawyers? How Licenses Influence our Daily Development Tasks

Massimiliano Di Penta
University of Sannio, Italy

Keynote

Abstract

Software licenses govern the way software can be used and above all re-distributed, and range across different levels of restriction, i.e., from very restrictive licenses such as Affero-GPL towards permissive ones like Apache or BSD. The recent history of software projects is permeated of cases in which a suitable (or unsuitable) decision about software licenses played a major role in the success of a project. Starting from recent literature and from problems actually encountered by developers, this talk conjectures how legal and technical expertise need to be properly combined when developing and evolving software projects. This has noticeable implications for practitioners, which require to cope with factors they often neglect when choosing components, but also for educators, responsible of introducing software licenses in computer science curricula, and, last but not least, for researchers, which might make developers' life easier through the development of licensing-aware recommender systems.

Caracterizando o Consumo de Energia de APIs de E/S da Linguagem Java

Gilson Rocha[1], **Gustavo Pinto**[1], **Fernando Castor**[2]

[1]Universidade Federal do Pará (UFPA)
Belém – PA – Brasil

[2]Unversidade Federal de Pernambuco (UFPE)
Recife – PE - Brasil

`gilsonrocha@gmail.com,gpinto@ufpa.br,castor@cin.ufpe.br`

Resumo. *APIs que implementam operações de entrada e saída (E/S) são a base para a construção de grande parte dos sistemas de software desenvolvidos que precisam, por exemplo, acessar um banco de dados, a internet ou periféricos. No entanto, há poucos trabalhos conduzidos com o objetivo de melhor entender e caracterizar o consumo de energia de APIs de que implementam operações de E/S. Neste trabalho, foram instrumentadas 23 classes Java com essas características (micro-benchmarks), além de dois benchmarks que fazem uso de algumas das classes instrumentadas, com o fim de entender o comportamento de consumo de energia das mesmas.*

Introdução

O consumo de energia em software tem atraído recente interesse não só de grande empresas de tecnologia, mas também de pesquisadores interessados em entender as causas e, eventualmente, mitigar suas raízes [Pinto et al. 2014, Sahin et al. 2016, McIntosh et al. 2018]. Apesar de importantes avanços terem sido introduzidos no passado recente [Bartenstein and Liu 2013, Liu et al. 2015, Oliveira et al. 2017], pouco ainda se sabe sobre o consumo de energia de APIs que implementam operações de entrada e saída (E/S). Essas APIs são amplamente utilizadas por desenvolvedores de software, em particular, pela necessidade que sistemas de software não-triviais têm em realizar tais operações, seja para acessar um banco de dados, periféricos, ou serviços externos.

O presente trabalho tem como objetivo minimizar essa lacuna através de uma caracterização do comportamento do consumo de energia de APIs que realizam operações de E/S na linguagem Java. Através da instrumentação de 23 classes do pacote `java.io`, foi possível responder questões de pesquisa sobre qual classe Java apresenta um melhor consumo de energia (QP1), se as classes com maior consumo de energia são também aquelas mais frequentemente empregadas (QP2) e se classes intercambiáveis podem ser alternadas de forma a melhorar o consumo de energia de benchmarks não-triviais (QP3).

Os resultados preliminares desse trabalho apontam para uma significante variação do consumo de energia entre as classes instrumentadas. Por exemplo, enquanto a classe `Files` apresenta o menor consumo de energia, a classe `FileInputStream` apresenta consumo de energia 3 vezes pior. Foi também observado que as classes frequentemente empregadas em projetos de código fonte aberto não são necessariamente as que tem o melhor consumo de energia (`Scanner`, apesar de ser frequentemente empregada, apresenta o quarto pior consumo de energia). Por fim, quando classes intercambiáveis são empregadas em benchmarks não-triviais, o consumo de energia pode aumentar mais de 100%. Isso significa que há uma oportunidade para reduzir o consumo de energia de aplicações existentes que demandaria pouquíssimo esforço dos desenvolvedores.

Método

Questões de Pesquisa

Este trabalho é guiado pelas seguintes questões de pesquisa (QP):

QP1: Qual o consumo de energia das APIs Java que implementam operações de E/S?
QP2: As APIs que implementam operações de E/S mais frequentemente utilizadas são as que tem menor consumo de energia?
QP3: APIs intercambiáveis podem ser alternadas de forma a melhorar o consumo de energia de benchmarks não-triviais?

Para responder a primeira questão de pesquisa (**QP1**), foram estudados 23 APIs que implementações operações de entrada e saída na linguagem Java. Para responder a segunda questão de pesquisa (**QP2**), os dados da primeira questão de pesquisa foram correlacionados com a frequência da utilização das classes em projetos de código aberto. Esta ocorrência foi medida através da infraestrutura de código `BOA` [Dyer et al. 2013]. Por fim, por intercambiáveis, na **QP3**, entende-se implementações que possam ser alteradas como mínimo ou nenhum esforço (por exemplo, troca entre classes que implementam a mesma interface). Essas modificações foram então empregadas em benchmarks consolidados e utilizados por estudos anteriores [Lima et al. 2016, Oliveira et al. 2017].

Micro-Benchmarks e Benchmarks

Nesta Seção descrevemos os Micro-Benchmarks e Benchmarks utilizados neste estudo.

Micro-benchmarks. A linguagem de programação Java é particularmente rica quando se trata de APIs que manipulam operações de entrada e saída. Em particular, a implementação das operações de E/S está concentrada nas subclasses de quatro classes abstratas: `OutputStream`, `InputStream`, `Reader` e `Writer`. As classes `InputStream` e `OutputStream` implementam E/S de um vetor de bytes, enquanto que as classes `Reader` e `Writer` para E/S de caracteres. A Tabela 1 sumariza as classes e os métodos estudados.

Cada classe estudada é responsável por implementar um método de leitura ou um método de escrita. Foram estudadas a maior parte das classes que implementam E/S que residem no pacote `java.io`. Por questões de simplicidade, nossos benchmarks incluem todas as subclasses das quatro classes citadas anteriormente, com exceção de:

- `DataOutputStream`: por ter o método `readLines()` depreciado;
- `LineNumberInputStream` e `StringBufferInputStream`: por terem sido depreciadas;
- `ObjectOutputStream`: por focar em E/S de objetos java;
- `PipedOutputStream` e `PipedInputStream`: por funcionarem apenas em conjunto. As duas precisam ser implementadas juntas uma vez que uma gera dados para a outra consumir;

Tabela 1. Classes e a assinatura das APIs que realizam operações de E/S. A coluna versão indica a versão do Java em que a classe foi introduzida.

Nome da classe	Método Instrumentado	Herda de	Versão
BufferedWriter	`void write(String str)`	java.io.Writer	1.1
FileWriter	`void write(String str)`	java.io.Writer	1.1
StringWriter	`void write(String str)`	java.io.Writer	1.1
PrintWriter	`void write(String str)`	java.io.Writer	1.1
CharArrayWriter	`void write(String str)`	java.io.Writer	1.1
BufferedReader	`int read()`	java.io.Reader	1.1
LineNumberReader	`int read()`	java.io.Reader	1.1
CharArrayReader	`int read()`	java.io.Reader	1.1
PushbackReader	`int read()`	java.io.Reader	1.1
FileReader	`int read()`	java.io.Reader	1.1
StringReader	`int read()`	java.io.Reader	1.1
FileOutputStream	`void write(byte[] b)`	java.io.OutputStream	1.0
ByteArrayOutputStream	`void write(byte[] b)`	java.io.OutputStream	1.0
BufferedOutputStream	`void write(byte[] b)`	java.io.OutputStream	1.0
PrintStream	`void print(String str)`	java.io.OutputStream	1.0
FileInputStream	`int read()`	java.io.InputStream	1.0
BufferedInputStream	`int read()`	java.io.InputStream	1.0
PushbackInputStream	`int read()`	java.io.InputStream	1.0
ByteArrayInputStream	`int read()`	java.io.InputStream	1.0

- `PipedReader` e `PipedWriter`: pelo mesmo motivo anterior;
- `SequenceInputStream`: por ele ler em sequências dois inputs, como aponta o construtor `SequenceInputStream(InputStream s1, InputStream s2)`. Ele inicia e termina o s1 e depois pula automaticamente para o s2;

As demais classes do pacote `java.io` são de exceção ou classes de configuração. Adicionalmente às classes apresentadas na Tabela 1, foram também incluídas a classes `java.util.Scanner` (introduzida no Java 1.5) e `java.nio.file.Files` (introduzida no Java 1.7). Estas classes não herdam de nenhuma das quatro classes abstratas supra-citadas, logo implementam operações de E/S de forma ligeiramente diferente das classes apresentadas anteriormente. Por exemplo, a classe `Files` implementa três métodos diferentes para leitura de dados (`List<String> readAllLines(Path path)`, `Stream<String> lines(Path path)` e `BufferedReader newBufferedReader(Path path)`). Ainda, a classe `Scanner` conta com dois métodos (`String nextLine()` & `boolean hasNext()`) que devem ser utilizados em conjunto para leitura de dados.

Para cada uma das classes, foi criado um teste que lê um arquivo de formato `HTML` de 20 MBs (ou escrevendo, nas operações de escrita). As 23 classes apresentadas nessa seção compreendem o nosso conjunto de **micro-benchmarks**.

Benchmarks. Além dos micro-benchmarks, foram utilizados também dois benchmarks do Computer Language Benchmark Game[1], que tem por objetivo comparar os desempenhos de diversas linguagens de programação. São eles:

FASTA: Este benchmark organiza as estruturas de DNA, RNA ou proteínas. FASTA escreve os resultados (que são textos com sequências do tipo "GGGATACCG-TACA", etc) através do método `write(byte[] b)` de um `OutputStream`. Este benchmark tem 329 linhas de código.

[1] https://benchmarksgame-team.pages.debian.net

K-NUCLEOTIDE: Este benchmark trabalha em conjunto com o FASTA. Através do método `String readLine()` de um `BufferedReader`, este benchmark lê todas as linhas do arquivo gerado pelo FASTA contabilizando as diversas sequências de DNA contidas neste arquivo. Este benchmark tem 205 linhas de código.

Estes benchmarks foram escolhidos pois fazem uso de pelo menos um dos micro-benchmarks descritos nessa seção. Ao contrário dos micro-benchmarks, os benchmarks são programas que, embora pequenos (em torno de 200 linhas de código), foram projetados para ter bom desempenho. Estes benchmarks, inclusive, são frequentemente utilizados em estudos que visam entender ou otimizar o desempenho de linguagens de programação [Lima et al. 2016] e máquinas virtuais [Barrett et al. 2017]. Para cada um dos benchmarks escolhidos, seu código fonte foi modificado de forma a utilizar diferentes implementações de micro-benchmarks intercambiáveis.

Execução de Experimentos

Para executar os experimentos, foi utilizado um notebook com processador Intel® Core i7-2670QM CPU @ 2.20GHz com 4 núcleos físicos e memória de 16GB DDR3 1600MHz. O sistema operacional foi o Ubuntu 16.04 LTS (kernel 4.4.0-112-generic) e a linguagem de programação Java(TM) SE Runtime Environment, na versão 1.8.0-151.

Para medições de consumo de energia, foi utilizada a biblioteca `jRAPL` [Liu et al. 2015]. Esta biblioteca é uma interface para o módulo MSR (Machine-Specific Register), o qual está disponível para arquiteturas Intel que dão suporte a RAPL (Running Average Power Limit). Este módulo é responsável por armazenar informações relacionadas ao consumo de energia, as quais posteriormente podem ser acessadas pelo sistema operacional. Desta maneira, com o `jRAPL` é possível coletar de um trecho de código-fonte Java a dissipação de potência (P) ao longo do tempo (t). Através da relação entre P (medido em watts) e t (medido em segundos) temos o consumo de energia E (medido em joules), ou seja, $E = P \times t$ [Pinto and Castor 2017].

Os experimentos foram realizados sem nenhuma outra carga de trabalho em execução simultânea no computador que realizou os experimentos (exceto processos do próprio sistema operacional). Uma vez que se necessita de tempo para que o Just-In-Time (JIT) compilador da Máquina Virtual Java (JVM) identifique as partes de código que podem ser otimizadas, a JVM, em particular, e VMs que utilizam JIT compiladores, em geral, tratam as primeiras execuções de um programa como *warmup*. Recentes estudos apontaram que a execução de um programa é mais lenta durante o período de *warmup* e com melhor desempenho nas execuções subsequentes ao *warmup* [Georges et al. 2007, Pinto et al. 2014]. Dessa forma, experimentos que utilizam linguagens de programação que se beneficiam de compiladores JIT devem descartar as execuções realizadas durante o período de *warmup* e analisar somente as execuções subsequentes. Neste trabalho, cada (micro-) benchmark foi executado 10 vezes ao longo de um mesmo processo de uma JVM; as três primeiras execuções são descartadas, enquanto a média das sete subsequentes é reportada.

Resultados

Os resultados estão organizados em termos das perguntas de pesquisa.

QP1: Qual o consumo de energia das APIs que implementam operações de E/S?

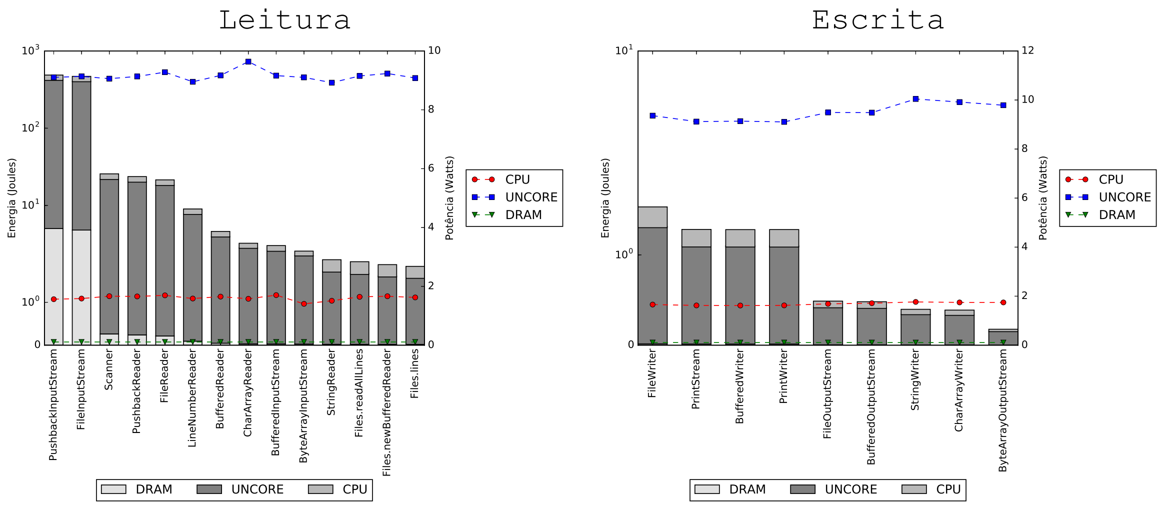


Figura 1. Consumo de energia das APIs de E/S Java. Consumo de energia é apresentado em escala logarítmica.

Na Figura 1, as barras representam consumo de energia, enquanto as linhas representam potência. O consumo de energia é a soma dos consumos de CPU, UNCORE (parte da CPU que não inclui os núcleos) e DRAM. À esquerda estão os micro-benchmarks que implementam operações de leitura, enquanto que os que implementam operações de escrita estão à direita. O micro-benchmark `PushbackInputStream` é o que apresenta maior consumo de energia dentre aqueles que realizam operações de leitura (492 joules consumidos), seguido de `FileInputStream` (474 joules). Por outro lado, `Files` é o micro-benchmark com melhor consumo energético usando os métodos: `lines` (1,86 joules), `newBufferedReader` (1,90 joules) e `readAllLines` (1,97 joules). O micro-benchmark `FileWriter` é o mais ineficiente do ponto de vista energético entre os que realizam operações de escrita (1,55 joules consumidos), seguido de `PrintStream` (1,30 joules) e `PrintWriter` (1,29 joules). `ByteArrayOutputStream`, por outro lado, é o que apresentou o menor consumo de energia (0,18 joules). Estes experimentos foram replicados com outras cargas de trabalho (1mb e 10mb) e os resultados se mostraram estáveis.

QP2: As APIs que implementam operações de E/S mais frequentemente utilizadas são as que tem menor consumo de energia?

A infraestrutura de código `BOA` [Dyer et al. 2013] foi utilizada como base para avaliar as ocorrências dos métodos que implementam E/S em projetos de código aberto. `BOA` conta com o código fonte de 7.830.023 projetos de código fonte aberto. Através de uma DSL, é possível navegar pela AST dos projetos, de forma a quantificar dados e meta-dados dos projetos armazenados. A Figura 2 contabiliza a utilização das classes estudadas.

Na figura abaixo, as linhas representam a quantidade de projetos que fazem ao menos um uso da classe, enquanto as barras contabilizam o total das ocorrências das classes Java. Ocorrências são medidas como o número de declarações de variáveis (tanto em escopo de classe quanto de método), além das assinaturas dos métodos; `imports` não

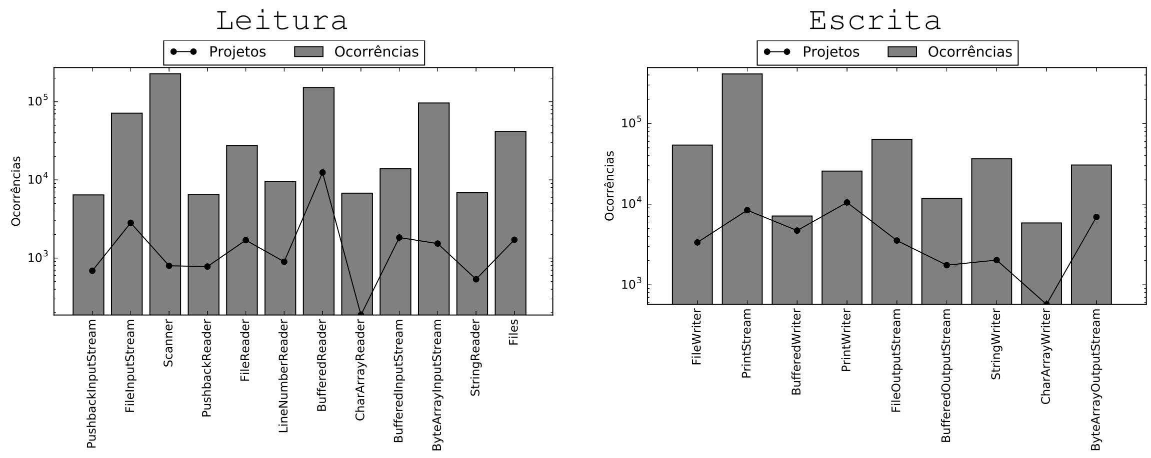


Figura 2. Utilização dos Micro-benchmarks Java em projetos de código aberto.

são levados em consideração. A classe `PrintStream` é a classe com mais ocorrências nos projetos estudados (412.163 ocorrências encontradas em 8.424 projetos diferentes), enquanto a classe `BufferedReader` é a mais utilizada em número absoluto de projetos de código aberto (12,441 projetos fazem 151.911 usos dessa classe).

A classe `Scanner` é a segunda mais utilizada (227.949 instancias) mas tem o quarto pior consumo de energia (26 joules). Dentre as classes mais energeticamente ineficientes que implementam operações de leitura, a `FileInputStream` é a mais utilizada, empregada em 71.339 instancias de 2.823 projetos de código aberto. Em contrapartida, a classe `FileWriter` é a terceira mais utilizada e apresentou o maior consumo de energia das que realizam operações de escrita.

QP3: APIs intercambiáveis podem ser alternadas de forma a melhorar o consumo de energia de benchmarks não-triviais?

Com esta QP, queremos saber se é possível reduzir o consumo de energia de aplicações existentes simplesmente refatorando-as para que utilizem classes de E/S alternativas. A Figura 3 apresenta o consumo de energia dos benchmarks após a aplicação dos micro-benchmarks intercambiáveis. Por intercambiável, entende-se APIs que possam ser alternadas com esforço mínimo, por exemplo, subclasses de uma mesma classe abstrata.

O benchmark FASTA faz uso de um `OutputStream`, logo as opções de APIs intercambiáveis se restrigem a: `FileOutputStream` (F1), `ByteArrayOutputStream` (F2), `BufferedOutputStream` (F3) e `PrintStream` (F4). Na implementação original, o FASTA utiliza `System.out` para escrever os resultados na saída do terminal. Como pode-se observar na Figura 3, o melhor consumo de energia se manteve na versão padrão (F) do benchmark, enquanto a implementação utilizando `FileOutputStream` foi 26% menos eficiente em termos de consumo de energia. Já o K-NUCLEOTIDE faz o uso de um `BufferedReader` (K), que é uma das implementações da classe abstrata `Reader`. Para este benchmark, as classes `Scanner` (K1) e `LineNumberReader` (K2) ficam sendo as opções de APIs intercambiáveis, uma vez que estas são as únicas que apresentam o método `String readLines()` (as demais classes apresentam o `int read()`, que tem

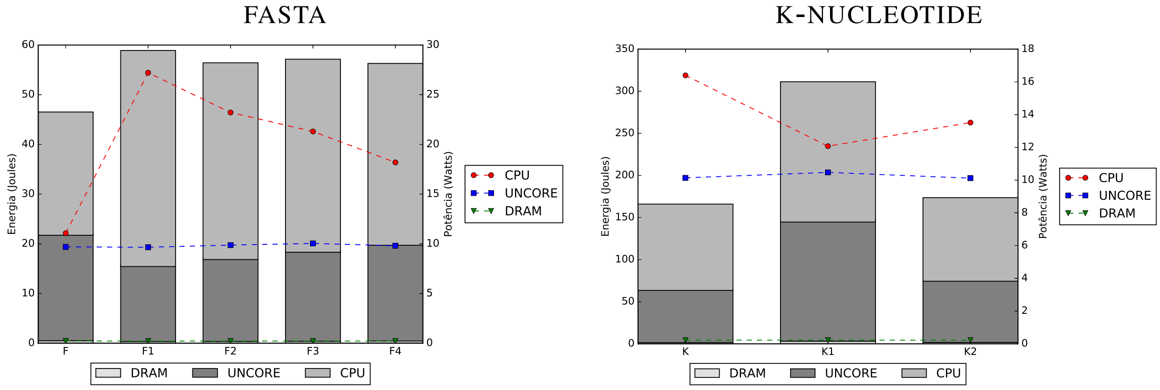


Figura 3. Consumo de energia dos benchmarks após refatoração para utilizar outras classes de E/S.

comportamento ligeiramente diferente). Como se pode observar na Figura 3, a versão padrão (K) apresentou o melhor consumo de energia, enquanto a versão (K1) apresentou um consumo de energia 87% maior, quando comparado ao baseline.

Esse resultado, onde as implementações padrão foram as mais eficientes, já era esperado, tendo em vista que os benchmarks do Computer Language Benchmarks Game são otimizados agressivamente para ter bom desempenho. Por outro lado, mostra que há oportunidades para se economizar energia com pouco esforço, já que aplicações que usem as classes menos eficientes podem ser refatoradas para usar alternativas menos custosas.

Limitações

Este trabalho se limita a estudar as classes Java que residem no pacote `java.io`. Esse pacote contém classes que implementam métodos de E/S por padrão na linguagem Java. No entanto, não foram investigadas outras classes de comportamento similar que residem em outros pacotes, nem classes desenvolvidas por terceiros disponibilizadas em bibliotecas de software. Além disto, apesar de algumas variações de configuração terem sido objeto de estudo (como a variação do tamanho de entrada do arquivo), algumas das classes apresentadas contam com diversas opções de ajustes de configuração (como na presença de vários construtores). Outras variações dessas configurações podem ser exploradas em trabalho futuros. Uma outra ameaça está relacionado aos projetos armazenados na na infraestrutura BOA, que tiveram sua ultima atualização em 2015. Ademais, como o objetivo do trabalho é trazer uma caracterização inicial dos métodos que implementam operações de E/S, os benchmarks utilizados nesse trabalho são simples repetições dos métodos estudados, o que não necessariamente caracteriza o perfil de uma aplicação real em utilização. O estudo de como os achados se comportam em aplicações reais não foi contemplado neste trabalho, bem como o estudo aprofundado dos fatores que impactam um maior/menor consumo de energia.

Trabalhos Relacionados

Existem estudos que concentram-se em ferramentas para estimar o consumo de energia em software [Liu et al. 2015], os quais apresentam ferramentas que permitem aos desenvolvedores de software estimar o consumo de energia de seu software sem a necessidade de um profundo conhecimento dos detalhes de mais baixo nível. Outros trabalhos analisam o comportamento energético em projetos de software através de estudos empíricos [Pinto et al. 2014, Lima et al. 2016, McIntosh et al. 2018]. Estes trabalhos apontam que pequenas mudanças podem introduzir grandes diferença em termos de consumo de energia e que refatorações simples, que fazem mudanças em tipos de dados podem ter bons resultados na eficiência energética de um software. Embora alguns estudos abordem eficiência energética em softwares que fazem uso intenso de dados [Bartenstein and Liu 2013, Liu et al. 2015], este trabalho se diferencia dos demais no seu foco nas APIs da liguagem Java que implementam operações de E/S.

Conclusões

Este trabalho apresentou uma caracaterização de 23 classes que implementam operações de E/S na linguagem Java. Dentre os achados, destacam-se: (1) a significativa variação do consumo de energia entre as classes instrumentadas (enquanto a classe `Files` apresenta o menor consumo de energia, a classe `FileInputStream` apresenta consumo 3 vezes pior), (2) classes que são frequentemente utilizadas, como a classe `Scanner`, não necessariamente têm um bom comportamento em termos de consumo de energia e (3) quando classes intercambiáveis são empregadas em benchmarks não-triviais, o consumo de energia pode aumentar mais de 80%.

Referências

[Barrett et al. 2017] Barrett, E., Bolz-Tereick, C. F., Killick, R., Mount, S., and Tratt, L. (2017). Virtual machine warmup blows hot and cold. *Proc. ACM Program. Lang.*, 1(OOPSLA):52:1–52:27.

[Bartenstein and Liu 2013] Bartenstein, T. W. and Liu, Y. D. (2013). Green streams for data-intensive software. In *ICSE*, pages 532–541.

[Dyer et al. 2013] Dyer, R., Nguyen, H. A., Rajan, H., and Nguyen, T. N. (2013). Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *ICSE*, pages 422–431.

[Georges et al. 2007] Georges, A., Buytaert, D., and Eeckhout, L. (2007). Statistically rigorous java performance evaluation. In *OOPSLA*, pages 57–76.

[Lima et al. 2016] Lima, L. G., Soares-Neto, F., Lieuthier, P., Castor, F., Melfe, G., and Fernandes, J. P. (2016). Haskell in green land: Analyzing the energy behavior of a purely functional language. In *SANER*, pages 517–528.

[Liu et al. 2015] Liu, K., Pinto, G., and Liu, Y. D. (2015). Data-oriented characterization of application-level energy optimization. In *FASE*.

[McIntosh et al. 2018] McIntosh, S., Hassan, A., and Hindle, A. (2018). What can android mobile app developers do about the energy consumption of machine learning? *Empirical Software Engineering*.

[Oliveira et al. 2017] Oliveira, W., Oliveira, R., and Castor, F. (2017). A study on the energy consumption of android app development approaches. In *MSR*, pages 42–52.

[Pinto and Castor 2017] Pinto, G. and Castor, F. (2017). Energy efficiency: a new concern for application software developers. *Commun. ACM*, 60(12):68–75.

[Pinto et al. 2014] Pinto, G., Castor, F., and Liu, Y. D. (2014). Understanding energy behaviors of thread management constructs. In *OOPSLA*, pages 345–360.

[Sahin et al. 2016] Sahin, C., Wan, M., Tornquist, P., McKenna, R., Pearson, Z., Halfond, W. G. J., and Clause, J. (2016). How does code obfuscation impact energy usage? *Journal of Software: Evolution and Process*, 28(7):565–588.

Um Estudo em Larga-Escala sobre Características de APIs Populares

Caroline Lima[1], **Pedro Henrique de Moraes**[1], **Andre Hora**[1]

[1] Faculdade de Computação (FACOM)
Universidade Federal de Mato Grosso do Sul (UFMS)

carollimaxp@gmail.com, pedhmoraes@gmail.com, hora@facom.ufms.br

Abstract. *Software libraries are commonly used via APIs to improve productivity and decrease development costs. While some APIs are very popular, others face much lower usage rates. In this context, one important question emerges: are there particular characteristics that differentiate popular APIs from other APIs? In this paper, we assess several characteristics of popular APIs, at code and evolution level, aiming to reveal development aspects of globally used APIs. This analysis is performed in the context of 897 APIs provided by libraries such as JDK, Android, and JUnit. As a result, we find that popular APIs are better encapsulated, have more documentation, and have more updates. Finally, based on our findinds, we propose several implications to library designers.*

Resumo. *Bibliotecas de software são comumente utilizadas, através de suas APIs, para aumentar a produtividade e diminuir os custos de desenvolvimento. Enquanto algumas são muito populares, outras equivalentes apresentam taxas de uso relativamente menores. Nesse contexto, uma questão importante surge: existem características particulares que diferenciam as APIs populares das demais APIs? Neste artigo, analisa-se diversas características de APIs populares, em nível de código e de evolução, visando revelar aspectos de desenvolvimento de APIs utilizadas globalmente. Essa análise é realizada no contexto de 897 APIs, fornecidas por bibliotecas como JDK, Android e JUnit. Como resultado, detecta-se que as APIs populares possuem melhor encapsulamento, mais documentação e mais alterações que as demais APIs. Por fim, com bases nesses resultados, diversas implicações são sugeridas para projetistas de bibliotecas.*

1. Introdução

Hoje em dia, bibliotecas de software são frequentemente utilizadas através de suas APIs (*Application Programming Interfaces*), para apoiar a criação de sistemas, melhorar a produtividade e diminuir os custos de desenvolvimento [Moser and Nierstrasz 1996]. Enquanto algumas APIs são muito populares e utilizadas por milhares de sistemas clientes, outras equivalentes apresentam taxas de uso relativamente menores. Por exemplo, em uma análise considerando 4.532 sistemas, detectou-se que 97% desses sistemas utilizavam a biblioteca de teste JUnit, enquanto apenas 12% utilizavam o seu concorrente TestNG [Zerouali and Mens 2017]. Considerando 260 mil sistemas, a diferença entre as taxas de uso continuam altas: 20% usam o JUnit, enquanto 1% usam o TestNG [Dyer et al. 2013].

Nesse contexto, uma questão importante surge: *existem características particulares que diferenciam as APIs populares das demais APIs?* Por exemplo, elas se diferenciam com relação a documentação, complexidade, estabilidade? Responder essas perguntas pode revelar aspectos de desenvolvimento presentes em APIs utilizadas globalmente, de forma a ajudar projetistas de APIs em suas atividades de manutenção de bibliotecas de software. Neste artigo, analisa-se diversas características presentes em APIs populares. Especificamente, investiga-se aspectos de código (*ie*, tamanho, complexidade e documentação) e de evolução (*ie*, estabilidade) dessas APIs. Logo, duas questões de pesquisa centrais são propostas: *QP1: Quais as características de código das APIs populares?* e *QP2: Quais as características evolucionárias das APIs populares?*

Para responder essas questões de pesquisa, estuda-se 897 APIs (fornecidas por bibliotecas como JDK, JUnit, Android e Facebook Fresco), que são utilizadas por 1.000 sistemas. Essas APIs são categorizadas em três grupos com relação aos seus níveis de popularidade e comparadas entre si. Logo, as principais contribuições desse trabalho são: (1) um estudo em larga-escala sobre as características de código e de evolução de APIs populares, (2) uma análise contrastando APIs populares das demais APIs e (3) um conjunto de lições aprendidas e implicações para projetistas de bibliotecas.

2. Popularidade de APIs

Bibliotecas fornecem interfaces para componentes de software criados para serem reutilizados por sistemas clientes [Reddy 2011]. Na linguagem Java, essas interfaces (também conhecidas como APIs) podem ser importadas por sistemas clientes. Exemplos de APIs comumente utilizadas em Java são `java.util.Map` e `java.util.HashMap`.

A Tabela 1 apresenta dentre todos os tipos APIs as mais frequente em Java, extraídas de 260K sistemas [Dyer et al. 2013]. Nota-se que uma alta taxa de uso está concentrada em poucas APIs e que essa taxa diminui rapidamente. Por exemplo, a API mais utilizada (`ArrayList`) é importada por 143.454 (54%) dos 260K sistemas, enquanto a API na 500ª posição (`AbstractTableModel`) é usada por apenas 3.977 (∼2%).

Posição	API	Uso Absoluto	Uso Relativo (%)
1ª	`java.util.ArrayList`	143.454	54%
100ª	`android.graphics.Color`	16.382	6%
200ª	`org.junit.Assert`	10.857	4%
300ª	`android.location.Location`	6.786	3%
400ª	`android.view.SurfaceHolder`	5.149	∼2%
500ª	`javax.swing.table.AbstractTableModel`	3.977	∼2%

Tabela 1. APIs Java mais utilizadas nos 260K sistemas [Dyer et al. 2013].

Diversas pesquisas levam em consideração a popularidade de APIs para aprender com elas. Por exemplo, estudos mineram tendências de uso de APIs populares, com a finalidade de sugerirem se determinada API deve ser adotada ou evitada pelos clientes [Mileva et al. 2009]. Outros trabalhos avaliam a evolução de APIs populares para apoiar a migração de bibliotecas [Hora and Valente 2015]. No melhor do nosso conhecimento, entretanto, aspectos relacionados as APIs populares (*eg*, complexidade, tamanho, documentação e evolução) foram pouco explorado pela literatura.

3. Metodologia

3.1. Detectando APIs Populares

Esta pesquisa foca na análise de sistemas escritos na linguagem de programação Java (uma das mais populares na atualidade) e hospedados na plataforma GitHub (a mais utilizada hoje em dia para armazenar sistemas *open-source*). Para responder as questões de pesquisa, primeiramente, precisa-se detectar um conjunto de APIs populares. Essa detecção é realizada em três fases: (1) coleção de um conjunto de sistemas relevantes; (2) detecção das APIs usadas por esses sistemas; e (3) categorização das APIs de acordo com sua popularidade, conforme descrito a seguir.

Etapa 1: Coleta de Sistemas Clientes. Primeiramente, realizou-se a coleta de um conjunto de sistemas Java; essa coleta foi necessária para descobrir quais APIs esses sistemas estavam utilizando. Especificamente, coletou-se os primeiros 1.000 sistemas com mais *stars*[1] para focar nos sistemas mais populares. Alguns desses sistemas são criados por grandes empresas, como Google, Facebook, Twitter e Microsoft. Os sistemas coletados pertencem a diversos domínios, tais como linguagens de programação (*eg*, Kotlin e Clojure), bibliotecas (*eg*, Facebook Fresco e Google Guava), frameworks (*eg*, Android e Spring). Esses sistemas apresentam na mediana 1.797,5 *stars* e 54 arquivos Java.

Etapa 2: Detecção das APIs Usadas pelos Sistemas Clientes. O próximo passo na metodologia é detectar as APIs utilizadas pelos 1.000 sistemas (em sua última *release*). Para isso, extraiu-se as APIs importadas, e, em seguida, para cada API, calculou-se o número de sistemas distintos que a utilizavam. No total, detectou-se 230.365 APIs distintas importadas por pelo menos um sistema. Para filtrar APIs locais, descartou-se aquelas utilizadas por menos de 10 sistemas (APIs com pelo menos 1% do total de sistemas). Essa filtragem resultou em 2.994 APIs com pelo menos 10 clientes. A Figura 1(a) apresenta a popularidade das APIs filtradas em relação ao números de sistemas clientes. O primeiro quartil é 13 clientes, a mediana é 20 e o terceiro quartil é 39. Também percebe-se uma grande quantidade de *outliers*, que representam APIs altamente populares.

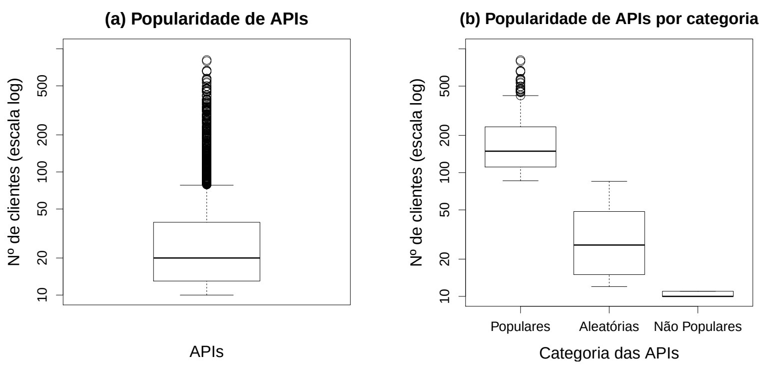


Figura 1. (a) Popularidade das APIs em relação ao número de clientes. (b) Popularidade das APIs em relação ao número de clientes por categoria.

Etapa 3: Categorização das APIs em Populares, Não Populares e Aleatórias. O último passo consiste em categorizar as APIs detectadas em *populares* e *não populares*. Nesse contexto, ordenou-se de forma decrescente as 2.994 APIs pelo número de clientes. Em seguida, seguindo o mesmo projeto experimental adotado para avaliar *apps* populares e não populares [Tian et al. 2015], classificou-se como *populares* os primeiros 10% (*ie*, as APIs com mais clientes) e como os *não populares* os últimos 10% (*ie*, as APIs menos clientes); cada categoria, portanto, inclui 299 APIs. Adicionalmente, selecionou-se um conjunto *aleatório* de 299 APIs para contrastar com as *populares* e as *não populares*. Dessa forma, foram consideradas três categorias (*populares*, *aleatórias* e *não populares*),

[1]Métrica que indica a popularidade de um projeto na plataforma GitHub.

totalizando **897** APIs distintas (299 x 3). A Figura 1(b) apresenta a popularidade das APIs considerando as três categorias. Na mediana, as APIs *populares* possuem 149 clientes, as *aleatórias* 26 clientes e as *não populares* 10 clientes. Observou-se também que as três categorias são estatisticamente diferentes entre si (*p-value* $< 0{,}01$ para o teste de Mann-Whitney e *effect-size* 0,99 para o Cliff's Delta).

3.2. Mensurando Características das APIs

QP1 (Código das APIs). Para responder a Questão de Pesquisa 1, sobre as características de tamanho, complexidade e documentação das APIs estudadas, extraiu-se 5 métricas de código, com suporte da ferramenta Understand[2], conforme descrito na Tabela 2. Para o cálculo da métrica de complexidade considerou-se todos os métodos e para a métrica documentação considerou-se todas linhas com comentários.

QP2 (Evolução da API). Para responder a Questão de Pesquisa 2, sobre as características de evolução das APIs, utilizou-se os dados fornecidos pelo controle de versão git. Nesse contexto, foram extraídas métricas relacionadas ao nível de estabilidadedas APIs a partir do histórico de versões. Utilizou-se o comando `git log` as API e implementou-se uma ferramenta para calcular as métricas número de *commits* e *lifetime* (ver Tabela 2). *Lifetime* é o número de dias entre o commit em que a API foi inserida até atualmente. Na métrica *commits* contabilizou-se o número de commits que alteraram o arquivo da API.

RQ	Categoria	Métrica	Descrição
Código das APIs (QP1)	Tamanho	NPM	Número de métodos públicos.
		RPM	Número relativo de métodos públicos (razão entre NPM e o número total de métodos).
	Complexidade	CC	Média da complexidade ciclomática dos métodos (expressões e condicionais são consideradas).
	Documentação	NCL	Número de linhas com comentários.
		RCL	Número relativo de linhas com comentários (razão entre NCL e número total de linhas).
Evolução das APIs (QP2)	Estabilidade	Lifetime	Número de dias entre o primeiro e último *commit*.
		Commits	Número de *commits* que modificar a API.
		RLCM	Razão entre *lifetime* e *commits*.

Tabela 2. Métricas estudadas ao nível de API.

4. Resultados

QP1: Quais as características de código das APIs populares?

Tamanho. A Figura 2 compara o tamanho das APIs através do número de métodos públicos (NPM) e do número relativo de métodos públicos (RPM). Observa-se que as APIs *populares* fornecem mais métodos públicos (mediana 19) para os clientes do que as *aleatórias* (mediana 10) e as *não populares* (mediana 9) (*p-value*$<0{,}01$ e *effect-size* pequeno). Considerando o valor relativo, nota-se o oposto; a razão entre métodos públicos e o número total de métodos é menor para as APIs *populares* (mediana 81%) do que para as *aleatórias* (mediana 89%) e *não populares* (mediana 86%). Desse modo, as APIs *populares* incluem outros tipos de modificadores de visibilidade (*ie*, `private`, `package` e `protected`), indicando maior preocupação com encapsulamento.

[2]https://scitools.com

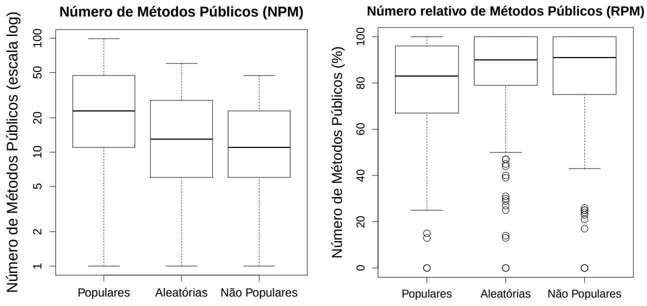


Figura 2. Tamanho das APIs.

Complexidade. A Figura 3 apresenta os resultados da métrica média da complexidade ciclomática (CC). Essa métrica conta cada estruturada (*eg*, `if`, `else` e `while`) e cada operador lógico (`&&` e ||). O resultado é a razão entre a soma da complexidade ciclomática e o número de métodos da API. As *populares* são um pouco mais complexas (mediana 2) do que as demais, ambas com mediana 1 (*p-value*<0,01 e *effect-size* pequeno).

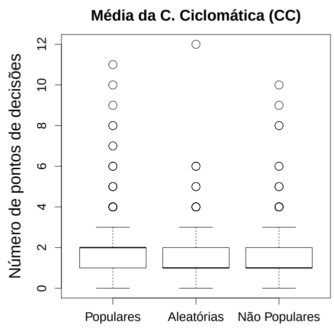


Figura 3. Complexidade das APIs.

Documentação. A Figura 4 avalia a documentação das APIs considerando duas métricas: número de linhas com comentários (NCL) e número relativo de linhas com comentários (RCL). As APIs *populares* possuem mais linhas com comentários (mediana 347) do que as *aleatórias* (mediana 140) e *não populares* (mediana 97) (*p-value*< 0,01, *effect-size* médio e grande). Relativamente, as APIs *populares* possuem mais comentários (mediana 60%) do que as *não populares* (mediana 52%) com *p-value*<0,01 e *effect-size* pequeno. Esse resultado mostra que os desenvolvedores de APIs *populares* possuem maior preocupação com documentação, o que pode facilitar a utilização dessas APIs.

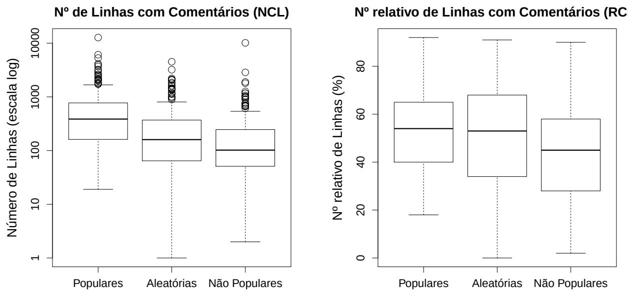


Figura 4. Documentação das APIs.

QP2: Quais as características evolucionárias das APIs populares?

Estabilidade. A Figura 5 apresenta a estabilidade das APIs em relação a três métricas: *lifetime* (em dias), número de *commits* e razão entre *lifetime* e *commits*. As APIs *populares* são mais antigas (mediana 3.593 dias) do que as demais (medianas 3.114 e 2.538 dias), sendo *p-value*<0,01 e *effect-size* médio. Considerando o número de *commits*, as APIs *populares* possuem mais alterações (mediana 14) do que as demais (medianas 8 e 7), (*p-value*<0,01 e *effect-size* pequeno). Para entender melhor esses resultados, a métrica razão entre *lifetime* e *commits* permite inferir que as APIs *populares* mudam mais frequentemente (mediana de 251 dias por *commit*) do que as demais APIs (mediana 277). Em suma, as APIs *populares* são mais antigas, possuem mais *commits* e mudam com mais frequência ao longo do tempo.

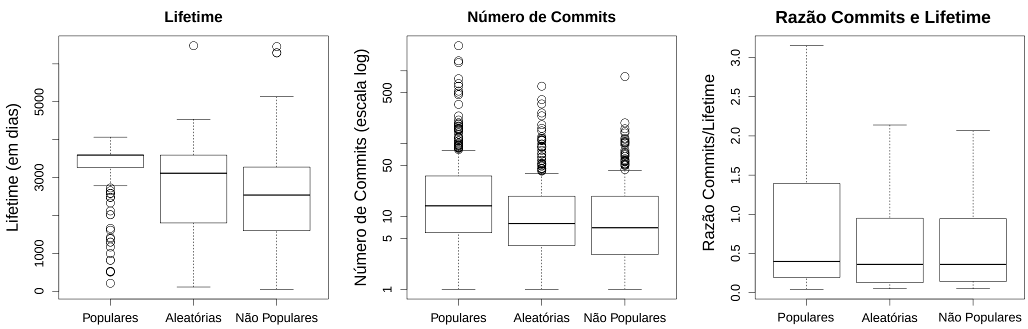


Figura 5. Estabilidade das APIs.

5. Discussão

APIs populares são maiores e mais encapsuladas. Em números absolutos, detectou-se que as APIs *populares* possuem mais métodos públicos. Em termos relativos, entretanto, observou-se que as APIs *populares* possuem proporcionalmente menos métodos públicos (ou seja, elas possuem mais métodos com outras visibilidades), indicando um melhor encapsulamento. De fato, projetistas de bibliotecas devem refletir cuidadosamente sobre a visibilidade das APIs para reduzir da exposição de elementos internos. Elementos internos que são desnecessariamente expostos aos clientes podem causar retrabalho e perda de qualidade na biblioteca [Hora et al. 2016].

APIs populares possuem mais documentação. Tanto em valores absolutos quanto em relativos, detectou-se que as APIs *populares* possuem mais comentários em seus códigos. Desse modo, as APIs *populares* podem facilitar o uso das suas funcionalidades. De fato, desenvolvedores de bibliotecas devem dar atenção especial a documentação; por exemplo, idealmente, todo elemento público deve possuir documentação.

APIs populares são mais antigas e mudam mais frequentemente. As APIs *populares* possuem mais tempo de vida e *commits*. Além disso, elas mudam com maior frequência ao longo do tempo, possivelmente para acomodar demandas dos seus clientes [Xavier et al. 2017, Brito et al. 2018]. Na prática, APIs não são elementos imutáveis, mas evoluem ao longo do tempo; mesmo as APIs populares, com milhares de clientes, podem ser alteradas. Desse modo, desenvolvedores e projetistas de APIs devem avaliar os impactos antes de alteração no código, visando reduzir possíveis quebras de contratos.

6. Ameaças à Validade

Categorização das APIs. Durante o processo de categorização, removeu-se algumas APIs, por exemplo, sem código fonte disponíveis, ou que não eram classes ou interfaces.

Generalização dos Resultados. Analisou-se 1K sistemas Java *open source* hospedados no GitHub. Logo, os resultados não podem ser diretamente generalizados para outros sistemas, especificamente para os implementados em outras linguagens e comerciais.

7. Trabalhos Relacionados

Diversos trabalhos investigam bibliotecas e APIs em relação a aspectos de uso e popularidade. Por exemplo, estudos focam verificar tendências de uso de APIs [Mileva et al. 2009], identificar qual versão atende as necessidades de um cliente [Kim and Notkin 2009] e melhorar documentação de forma automática [Treude and Robillard 2016]. Nota-se também que algumas bibliotecas são mais populares que outras com funcionalidades semelhantes [Zerouali and Mens 2017]. Estudos também avaliam a popularidade de APIs, a fim de sugerir migração de interfaces (*eg*, [Hora and Valente 2015]). No contexto do Android, investigou-se a correlação entre a popularidade dos aplicativos na *PlayStore* com a utilização de APIs menos propensas a erros [Bavota et al. 2015]. No GitHub, investigou-se as caracteristicas de projetos populares [Borges et al. 2016a, Borges et al. 2016b]. Além da aplicação na Engenharia de Software, popularidade é um tema recorrente em outras áreas de pesquisa, principalmente relacionadas à redes sociais. Por exemplo, estudos analisaram a popularidade de vídeos do YouTube [Ahmed et al. 2013, Figueiredo et al. 2014] e hashtags do Twitter [Lehmann et al. 2012].

8. Conclusão

Neste estudo foi apresentado uma análise inicial sobre algumas características de código e evolução encontradas em APIs globalmente utilizadas. Como resultado, detectou-se que APIs populares são maiores, mais encapsuladas, melhor documentadas e alteradas com maior frequência. Como trabalhos futuros, pretende-se: (i) analisar mais sistemas clientes, (ii) expandir a análise para outras categorias de APIs (*eg*, Android e JDK) e (iii) analisar outras métricas de código (*eg*, legibilidade), evolução e uso das APIs.

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

Referências

Ahmed, M., Spagna, S., Huici, F., and Niccolini, S. (2013). A peek into the future: Predicting the evolution of popularity in user generated content. In *International Conference on Web Search and Data Mining (WSDM)*.

Bavota, G., Linares-Vasquez, M., Bernal-Cardenas, C. E., Di Penta, M., Oliveto, R., and Poshyvanyk, D. (2015). The impact of API change and fault-proneness on the user ratings of android apps. *Transactions on Software Engineering*, 41(4).

Borges, H., Hora, A., and Valente, M. T. (2016a). Predicting the Popularity of GitHub Repositories. In *International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*.

Borges, H., Hora, A., and Valente, M. T. (2016b). Understanding the factors that impact the popularity of GitHub repositories. In *International Conference on Software Maintenance and Evolution (ICSME).*

Brito, A., Xavier, L., Hora, A., and Valente, M. T. (2018). Why and how Java developers break APIs. In *International Conference on Software Analysis, Evolution and Reengineering (SANER).*

Dyer, R., Nguyen, H. A., Rajan, H., and Nguyen, T. N. (2013). Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *International Conference on Software Engineering (ICSE).*

Figueiredo, F., Almeida, J. M., Gonçalves, M. A., and Benevenuto, F. (2014). On the dynamics of social media popularity: A youtube case study. *Transactions on Internet Technology (TOIT)*, 14(4):24.

Hora, A. and Valente, M. T. (2015). apiwave: Keeping track of api popularity and migration. In *Inter. Conference on Software Maintenance and Evolution (ICSME).*

Hora, A., Valente, M. T., Robbes, R., and Anquetil, N. (2016). When should internal interfaces be promoted to public? In *International Symposium on Foundations of Software Engineering (FSE).*

Kim, M. and Notkin, D. (2009). Discovering and representing systematic code changes. In *International Conference on Software Engineering (ICSE).*

Lehmann, J., Gonçalves, B., Ramasco, J. J., and Cattuto, C. (2012). Dynamical classes of collective attention in twitter. In *International Conference on World Wide Web (WWW).*

Mileva, Y. M., Dallmeier, V., Burger, M., and Zeller, A. (2009). Mining trends of library usage. In *International Workshop on Principles on Software Evolution (IWPSE) and ERCIM Workshop on Software Evolution (EVOL).*

Moser, S. and Nierstrasz, O. (1996). The effect of object-oriented frameworks on developer productivity. *Journal of Computer*, 29(9).

Reddy, M. (2011). *API Design for C++*. Morgan Kaufmann Publishers.

Tian, Y., Nagappan, M., Lo, D., and Hassan, A. E. (2015). What are the characteristics of high-rated apps? a case study on free Android applications. In *International Conference on Software Maintenance and Evolution (ICSME).*

Treude, C. and Robillard, M. P. (2016). Augmenting API documentation with insights from Stack Overflow. In *International Conference on Software Engineering (ICSE).*

Xavier, L., Brito, A., Hora, A., and Valente, M. T. (2017). Historical and impact analysis of API breaking changes: A large-scale study. In *International Conference on Software Analysis, Evolution and Reengineering (SANER).*

Zerouali, A. and Mens, T. (2017). Analyzing the evolution of testing library usage in open source Java projects. In *International Conference on Software Analysis, Evolution and Reengineering (SANER).*

Minerando Mensagens de Depreciação Faltantes em APIs: Um Estudo de Caso no Ecossistema Android

Pedro Henrique de Moraes[1], **Caroline Lima**[1], **Andre Hora**[1]

[1] Faculdade de Computação (FACOM)
Universidade Federal de Mato Grosso do Sul (UFMS)

pedhmoraes@gmail.com, carollimaxp@gmail.com, hora@facom.ufms.br

Abstract. *To facilitate library migration, developers should deprecate APIs before any change that may impact on client systems. However, the literature reports that APIs are commonly deprecated without any message to support their clients. Therefore, in this paper, we propose an approach to recommend missing replacement messages in deprecated APIs. We assess the history version of the Android framework and 100 client systems. We found that some replacement messages can be detected by mining the clients, but alternatives solutions are common. Finally, based on our learning, we suggest improvements to the design of an API deprecation migration tool.*

Resumo. *Para facilitar a migração entre versões de bibliotecas de software, desenvolvedores devem depreciar APIs antes de qualquer alteração que possa impactar os clientes. No entanto, a literatura reporta que APIs são geralmente depreciadas sem mensagens de substituição para auxiliar os clientes. Diante desse problema, neste artigo, propõe-se uma abordagem para a recomendação de mensagens faltantes em APIs depreciadas. Analisa-se o histórico de versões do framework Android e de 100 sistemas clientes. Como resultado, detecta-se que algumas mensagens podem ser encontradas nos clientes, mas soluções alternativas são comuns. Por fim, com base no nosso aprendizado, sugere-se melhorias para o projeto de uma ferramenta de migração de APIs depreciadas.*

1. Introdução

Bibliotecas de software são utilizadas por aplicações clientes para reuso de funcionalidades e aumento de produtividade, resultando na diminuição dos custos de desenvolvimento [Moser and Nierstrasz 1996, Tourwé and Mens 2003]. Como qualquer outro sistema, bibliotecas de software também evoluem com o passar do tempo [Hora et al. 2015, Xavier et al. 2017]; naturalmente, elas são alteradas para fornecer novas funcionalidades [Brito et al. 2018]. Consequentemente, as aplicações clientes podem ser impactadas e devem migrar para se beneficiarem das novas APIs [Bogart et al. 2016]. Nesse contexto, visando facilitar a migração entre versões, *uma boa prática é depreciar APIs antes de qualquer alteração que possa impactar os clientes* [Robbes et al. 2012].

No entanto, a literatura reporta que APIs são geralmente depreciadas *sem* mensagens de substituição para auxiliar os clientes [Brito et al. , Sawant et al. 2017]. Diante desse problema, neste artigo, propõe-se uma abordagem para a recomendação de mensagens faltantes em APIs depreciadas. Especificamente, mesmo quando não existe mensagem de substituição explícita, *os clientes tomam suas próprias decisões para substituir* *uma API depreciada por outra* [Brito et al.]. Desse modo, uma técnica de recomendação pode ser projetada para aprender automaticamente a partir dessas decisões, particularmente quando uma decisão é compartilhada por vários clientes. Logo, três questões de pesquisa são propostas:

- *QP1: Qual a taxa de APIs sem mensagens de depreciação atualmente?*
- *QP2: Pode-se inferir regras de evolução de código minerando os clientes?*
- *QP3: Pode-se inferir mensagens de depreciação faltantes minerando os clientes?*

Neste estudo, foca-se na análise do ecossistema Android devido a sua grande relevância na atualidade. Particularmente, para responder as questões de pesquisa, analisa-se o histórico de versões do framework Android e de 100 sistemas clientes. Desse modo, as principais contribuições deste trabalho são: (1) a proposição de uma abordagem para inferir mensagens de depreciação faltantes em APIs; (2) uma primeira avaliação dessa abordagem através da mineração de dezenas de sistemas clientes; (3) uma análise sobre o estado atual de depreciação nas APIs do Android; e (4) um conjunto de lições aprendidas e implicações para projetistas e clientes de bibliotecas.

2. Depreciação de APIs

Idealmente, *Application Programming Interfaces* (APIs), tais como classes, métodos e atributos públicos, devem ser estáveis ao evoluir. Em outras palavras: APIs não devem ser removidas, alteradas ou renomeadas *sem uma prévia comunicação aos seus clientes* [Brito et al. 2018, Xavier et al. 2017]. Essa comunicação deve ser realizada através da depreciação da API, que pode conter mensagens para ajudar os desenvolvedores (conforme apresentado na Figura 1). De fato, antes de ser removida, uma API deve primeiramente ser depreciada com uma mensagem clara para que os seus usuários (i) sejam notificados sobre a sua futura remoção e (ii) saibam como reagir a tal alteração.

```java
/**
 * @deprecated Use {@link #getParams()} instead.
 */
@Deprecated
protected Map<String, String> getPostParams() throws AuthFailureError {
    //...
}
```

Figure 1. API depreciada com mensagem de substituição [Brito et al.].

Infelizmente, a literatura reporta que esse processo nem sempre é seguido [Brito et al. , Sawant et al. 2017]. Ou seja, APIs são frequentemente depreciadas sem mensagens para auxiliar os seus sistemas clientes (conforme apresentado na Figura 2). Por exemplo, em um estudo em larga escala realizado anteriormente [Brito et al.], detectou-se que apenas 66,7% das APIs são depreciadas com mensagens em sistemas Java e 77,8% em sistemas C#. Isso significa que em torno de 34% das APIs analisadas em Java e 23% em C# são depreciadas sem mensagens.

```java
@Deprecated
public Database(Context context) {
//...
}
```

Figure 2. API depreciada sem mensagem de substituição [Brito et al.].

3. Abordagem Proposta

Uma solução para esse problema é aprender *como* os sistemas clientes reagem quando se deparam com APIs depreciadas sem mensagens. Nesse contexto, pode-se analisar as alterações de código dos clientes em dois níveis de granularidade: (i) importações de classes e (ii) corpos de métodos. A mineração das alterações em nível de importação é uma técnica mais leve pois necessita apenas descobrir os `imports` adicionados e removidos na modificação de código. Por exemplo, a Figura 3 (esquerda) apresenta um trecho de código onde a mineração de imports pode ser utilizada para extração de informação relevante. Nesse caso, pode-se inferir que a classe `junit.framework.Assert` foi substituída pela classe `org.junit.Assert`.

```
 package org.jboss.as.modcluster;

-import junit.framework.Assert;
 import org.jboss.as.controller.PathAddress;
 import org.jboss.dmr.ModelNode;
 import org.jboss.dmr.ModelType;
+import org.junit.Assert;
 import org.junit.Test;
```

```
   public Collection<FormulaData> getChildren() {
-     List<FormulaData> result = new ArrayList<FormulaData>();
+     List<FormulaData> result = Lists.newArrayList();
      for (DecoratorContext childContext : decoratorContext.getChildren()) {
        result.add(new DefaultFormulaData(childContext));
      }
```

Figure 3. Esq: alteração na importação. Dir: alteração no corpo do método.

Já para a segunda abordagem (alterações nos corpos de métodos), é preciso uma análise mais fina do código alterado, uma vez que deve-se comparar duas versões da *Abstract Syntax Tree* (AST) para extração de informação relevante. Por exemplo, a Figura 3 (direita) apresenta um trecho de código onde a mineração do corpo do método `getChildren()` pode ser utilizada; nesse caso, pode-se inferir que o uso do construtor da classe `ArrayList` foi substituído pelo método `Lists.newArrayList()`.

A abordagem proposta neste trabalho (Figura 4) utiliza Mineração de Regras de Associação [Agarwal and Srikant 1994] para detectar ambos os tipos de alterações (*ie*, nas importações e nos corpos de métodos). A seguir são descritos as quatro etapas principais.

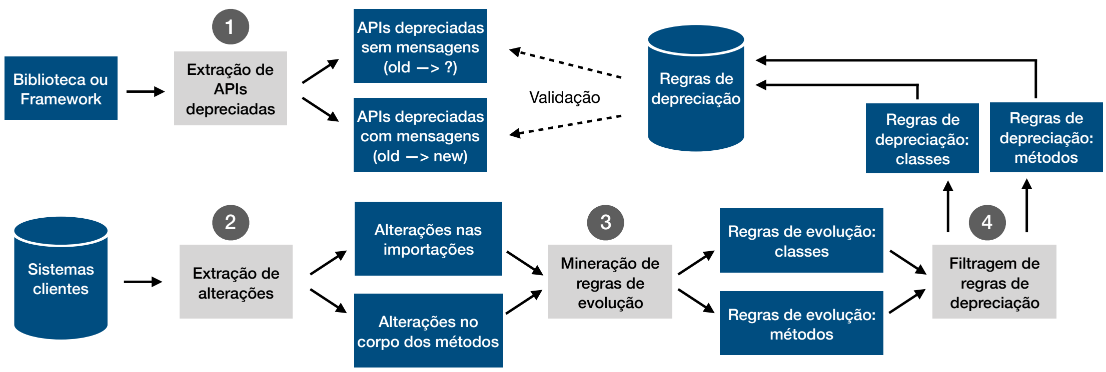


Figure 4. Abordagem para mineração de mensagens de depreciação faltantes.

1. Extração de APIs depreciadas. A primeira etapa da solução proposta consiste em extrair as APIs depreciadas de uma determinada biblioteca ou framework. Para tal, utiliza-se a ferramenta de mineração de APIs proposta por Brito et al. [Brito et al. 2016, Brito et al.] para detecção depreciação. A ferramenta adotada determina se uma API depreciada possui mensagem ou não, com base em um conjunto de regras preestabelecidas (*ex*: o Javadoc possui a tag `@deprecated` e a palavra *use*).

2. Extração de Alterações. Visando "aprender" com os sistemas clientes, para cada *commit* de um dado sistema, extrai-se: (i) as classes removidas/adicionadas nas importações e (ii) as invocações removidas/adicionadas no corpo dos métodos. Para descobrir as alterações nas *importações*, realiza-se uma análise textual entre as duas versões de uma classe de modo a detectar as importações removidas e adicionadas. Para descobrir as alterações no *corpo dos métodos*, utiliza-se a ferramenta FAMIX [Ducasse et al. 2011] para criação de modelos de código na versão anterior e posterior. Nesse caso, ao comparar duas versões de um método, extrai-se as invocações removidas e adicionadas.

3. Mineração de Regras de Evolução. Após a extração dos dados, aplica-se o algoritmo Apriori para detectar regras de associação [Agarwal and Srikant 1994] no formato `elemento-removido` → `elemento-adicionado`, indicando que o elemento na esquerda deve ser substituído pelo elemento na direita. Desse modo, com base nos dados gerados na etapa anterior, pode-se inferir regras que representam a evolução de classes e de métodos. Por exemplo, a mineração do código apresentado na Figura 3 (esquerda) pode gerar a regra: `junit.framework.Assert` → `org.junit.Assert`. Nota-se, no entanto, que nem todas as regras geradas são necessariamente no contexto de depreciação.

4. Filtragem de Regras de Depreciação. A última etapa consiste em filtrar as regras de evolução para selecionar apenas regras no contexto de depreciação de APIs. Especificamente, seleciona-se dois conjuntos de regras. No primeiro, nomeado *match parcial*, seleciona-se as regras de depreciação em que o lado esquerdo corresponde a uma API depreciada com mensagem. No segundo, nomeado *match total*, seleciona-se as regras de depreciação em que o lado esquerdo corresponde a uma API depreciada com mensagem e o lado direito corresponde a mensagem de substituição dessa API. Logo, pode-se validar a capacidade da abordagem acertar/errar APIs depreciadas com mensagens de depreciação.

4. Metodologia

4.1. Extração de APIs Depreciadas: Framework Android

Neste estudo, focou-se na análise do Android por representar um ecossistema altamente relevante. Minerou-se o Android Base e o Android Support, cujo os códigos estão espelhados no GitHub.[1] Realizou-se a extração das APIs depreciadas (Etapa 1, Seção 3) para os 345 mil *commits* do Android Base e 35 mil *commits* do Android Support. Em seguida, categorizou-se, manualmente, as APIs depreciadas em três grupos: com mensagem, com mensagem parcial e sem mensagem. Na QP1, essas categorias são explicadas e os resultados são discutidos.

4.2. Mineração de Regras de Evolução e Depreciação: Sistemas Clientes Android

Visando descobrir como os clientes lidam com APIs depreciadas, realizou-se uma primeira avaliação com sistemas clientes do Android (Etapas 2, 3 e 4, Seção 3). Foram selecionados no GitHub os 100 projetos Java mais populares baseados no Android; como critério de inclusão, selecionou-se os projetos com a palavra "Android" em suas descrição. Rodou-se o algoritmo de mineração de regras de associação com suporte 3 e confiança 50%. Esses valores foram configurados após uma avaliação manual realizada pelos autores, de forma a minimizar a quantidade de regras irrelevantes.[2] Esses resultados são detalhados na QP2 e na QP3.

[1] Projetos no GitHub: platform_frameworks_base e platform_frameworks_support

[2] Discutir os detalhes dessa avaliação está fora do escopo deste artigo.

5. Resultados

QP1: Qual a taxa de APIs sem mensagens de depreciação?

A Tabela 1 apresenta a taxa de APIs depreciadas com mensagens, parcialmente e sem mensagens. Nota-se a predominância de APIs depreciadas com mensagens completas para ajudar os clientes (57.88%). Observa-se que 30.80% das APIs depreciadas não possuem mensagens relevantes para auxiliar os clientes. 11.32% das APIs são depreciadas com mensagens parciais. Nota-se também que a grande maioria das entidades depreciadas ocorrem em nível de métodos (62%, 1,961 de 3,153 casos).

API	Total	Com Msg.	%	Parcial	%	Sem Msg.	%
Classe	63	4	6.35	7	11.11	52	82.54
Método	1,961	1,229	62.67	102	5.20	630	32.13
Atributo	1,129	592	52.43	248	21.97	289	25.60
Todas	**3,153**	**1,825**	**57.88**	**357**	**11.32**	**971**	**30.80**

Table 1. Taxa de APIs depreciadas com, parcial e sem mensagens.

Quando uma mensagem está completa, poupa tempo do cliente em busca de soluções alternativas. A Figura 5 (esquerda) apresenta um exemplo em que o método depreciado `getValidNotBefore()` pode ser substituído por `getValidNotBeforeDate()`. Entretanto, mensagens relevantes nem sempre estão presentes nas APIs depreciadas. Nessa condição, um cliente deve buscar por soluções alternativas. A Figura 5 (direita) apresenta uma exemplo em que método `onNewPicture()` está depreciado com mensagem irrelevante: "*Deprecated due to internal changes*". Além desses dois casos, mensagens de depreciação podem vir parcialmente, sugerindo locais onde procurar soluções alternativas para a depreciação. A Figura 5 (abaixo) mostra o método `getSelectedNavigationIndex()` depreciado sem mensagem direta, mas com uma *url* para orientar seus clientes.

```
/**
* @deprecated Use {@link #getValidNotBeforeDate()}
*/
@Deprecated
public String getValidNotBefore() {

    return formatDate(mValidNotBefore);
}
```

```
/**
 * @deprecated Deprecated due to internal changes.
 */
@Deprecated
public void onNewPicture(WebView view, Picture picture);
```

```
/**
* @deprecated Action bar navigation modes are deprecated and not supported by
inline
* toolbar action bars. Consider using other
* <a href="http://developer.android.com/design/patterns/navigation.html">common
* navigation patterns</a> instead.
*/
@Deprecated
public abstract int getSelectedNavigationIndex();
```

Figure 5. Esquerda: depreciação com mensagem. Direita: depreciação sem mensagem relevante. Abaixo: depreciação com mensagem parcial

Categoria	Regra de Evolução	Regra de Depreciação	Match Parcial	Match Total
Classe	121	12	11	1
Método	14	1	1	0
Todas	**135**	**13**	**12**	**1**

Table 2. Regras de evolução e depreciação de APIs.

QP2: Pode-se inferir regras de evolução de código?

A segunda coluna da Tabela 2 apresenta a quantidade de regras de evolução detectadas nos 100 sistemas Android analisados. Verifica-se um total de 135 regras geradas, onde 121 são no nível de classe (extraídas das alterações nas importações) e 14 são de métodos (extraídas das alterações nos corpos dos métodos). Logo, de fato, regras de evolução são frequentemente encontradas nos sistemas clientes do Android. Em seguida, analisa-se quais dessas regras são relacionadas a depreciação de APIs.

QP3: Pode-se inferir mensagens de depreciação faltantes minerando os clientes?

A terceira coluna da Tabela 2 apresenta a quantidade de regras no contexto de depreciação. Nota-se que 13 (10%) das 135 regras de evolução são relacionas a depreciação. Dentre essas 13 regras, 12 são no nível de classes enquanto 1 é de método. Verifica-se também que dentre as 13 regras de depreciação, 12 possuem *match parcial*, *ie*, regras em que o lado esquerdo corresponde a uma API depreciada com mensagem. Nesses casos, o lado direito das regras não batem com a sugestão da API. Por exemplo, a classe `MenuItemCompat` do Android foi depreciada sugerindo a utilização da classe `MenuItem`. No entanto, a regra de depreciação detectada foi `MenuItemCompat` → `org.telegram.ui.-Views.ActionBar.ActionBarMenuItem`. Logo, nota-se que os clientes tendem a utilizar soluções alternativas como substituição das APIs depreciadas. Por fim, detectou-se apenas 1 regra com *match total*. Por exemplo, a classe `Contacts` foi depreciada sugerindo a utilização da classe `ContactsContract`. Nesse caso isolado, de fato, a regra de depreciação detectada foi `Contacts` → `ContactsContract`.

6. Discussão

APIs são comumente depreciadas sem mensagens. Foi verificado que 30.8% das APIs depreciadas no framework Android não contém mensagens para ajudar os clientes. Estudos anteriores (*eg*, [Brito et al.]), cujo o foco não é o Android, detectaram uma taxa equivalente, de 34%. Esses dados evidenciam a necessidade de uma ferramenta para ajudar os clientes.

APIs são ocasionalmente depreciadas com mensagens parciais. 11.32% das APIs depreciadas apenas contém mensagens parciais (*eg*, urls de ajuda). Desse modo, somado com a taxa de APIs sem mensagens, nota-se que em torno de 42% das APIs depreciadas não ajudam diretamente seus clientes.

Regras de evolução podem ser detectadas. Confirmando estudos anteriores (*eg*, [Wu et al. 2010, Hora et al. 2015, Hora et al. 2018a]), foi verificado que regras de evolução de APIs (principalmente em nível de classe) podem ser detectadas através da técnica de mineração de regras de associação.

Regras em nível de método são mais difíceis de serem detectadas. O maior desafio da abordagem proposta foi produzir regras em nível de método. Isso ocorre pois é necessário lidar com a resolução de tipos nos clientes. Para tal, é necessário analisar suas dependências externas. Logo, sugere-se que as dependências dos clientes (*ie*, `jars` externos) sejam incluídas para aumento na quantidade de regras geradas. Além disso, para evitar os ruídos causados alterações durante a evolução, sugere-se que refatorações sejam resolvidas para uma detecção de regras mais completa [Hora et al. 2018b].

Regras de depreciação podem ser detectadas. Foram detectadas 12 regras de depreciação com *match parcial* e apenas 1 *match total*. Apesar de baixo, esses números sugerem que regras de depreciação podem ser detectadas. De fato, nesse estudo, foram analisados apenas 100 sistemas clientes. Uma hipótese, é que para para aumentar a quantidade de regras, deve-se minerar mais clientes. Por exemplo, aumentando a quantidade de clientes em um fator de 100, teoricamente, pode-se gerar milhares de regras de depreciação com *match parcial* e uma centena com *match total*.

Soluções alternativas são adotadas pelos clientes. Os clientes utilizam soluções alternativas para lidar com a evolução de APIs depreciadas. Isso aumenta o leque de possibilidades para a substituição da API depreciada, indicando que soluções não oficiais são adotadas.

7. Ameaças à Validade

Generalização dos resultados. Este estudo analisou o framework Android e 100 sistemas clientes implementados em Java. Assim, não é possível generalizar os resultados para outras linguagens de programação nem para outros ecossistemas de software.

Resolução de tipos. A ferramenta FAMIX (para a criação de modelos de código, adotada na Etapa 3) possui algumas limitações para a geração da AST. Quando o código analisado possui dependências externas, naturalmente, o nome completo de algumas APIs não são resolvidos, o que poderia impactar na qualidade das regras geradas. Portanto, para evitar regras com nomes incompletos, esses casos foram removidos da nossa base de alterações.

8. Trabalhos Relacionados

Diversas abordagens são propostas pela literatura para analisar evolução de APIs [Henkel and Diwan 2005, Hora et al. 2015, Hora et al. 2018a, Wu et al. 2010] e estudar seus efeitos colaterais [Bogart et al. 2016, Brito et al. 2018, Xavier et al. 2017]. O contexto de depreciação de APIs também é explorado [Brito et al. , Robbes et al. 2012, Sawant et al. 2017], mostrando que os clientes sofrem com a falta de mensagens e alto tempo de reação. Esses estudos, entretanto, não propõem soluções para lidar com APIs depreciadas sem mensagens.

9. Conclusão

No melhor do nosso conhecimento, a abordagem proposta é a primeira para detectar mensagens de depreciação faltantes. Verificou-se que 42% das APIs depreciadas do Android não possuem mensagens. Através da mineração de 100 clientes, algumas regras de depreciação foram geradas. Como trabalhos futuros, pretende-se: (i) realizar uma análise em larga escala (10K clientes), (ii) minimizar o problema da resolução de tipos, (iii) analisar outros ecossistemas e (iv) avaliar outras técnicas para geração das regras.

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

References

Agarwal, R. and Srikant, R. (1994). Fast algorithms for mining association rules. In *International Conference on Very Large Data Bases (VLDB)*.

Bogart, C., Kästner, C., Herbsleb, J., and Thung, F. (2016). How to break an API: cost negotiation and community values in three software ecosystems. In *International Symposium on Foundations of Software Engineering (FSE)*.

Brito, A., Xavier, L., Hora, A., and Valente, M. T. (2018). Why and How Java Developers Break APIs. *International Conference on Software Analysis, Evolution and Reengineering (SANER)*.

Brito, G., Hora, A., Valente, M. T., and Robbes, R. On the use of replacement messages in API deprecation: An empirical study. *Journal of Systems and Software*.

Brito, G., Hora, A., Valente, M. T., and Robbes, R. (2016). Do developers deprecate APIs with replacement messages? A large-scale analysis on Java systems. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*.

Ducasse, S., Anquetil, N., Bhatti, M. U., Hora, A., Laval, J., and Girba, T. (2011). MSE and FAMIX 3.0: an interexchange format and source code model family.

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

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 (ICSME)*.

Hora, A., Robbes, R., Valente, M. T., Anquetil, N., Etien, A., and Ducasse, S. (2018a). How do developers react to API evolution? a large-scale empirical study. *Software Quality Journal*, 26.

Hora, A., Silva, D., Robbes, R., and Valente, M. T. (2018b). Assessing the threat of untracked changes in software evolution. In *International Conference on Software Engineering (ICSE)*.

Moser, S. and Nierstrasz, O. (1996). The effect of object-oriented frameworks on developer productivity. *Journal of Computer*, 29(9).

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 (FSE)*.

Sawant, A. A., Robbes, R., and Bacchelli, A. (2017). On the reaction to deprecation of clients of 4+ 1 popular Java APIs and the JDK. *Empirical Software Engineering*.

Tourwé, T. and Mens, T. (2003). Automated support for framework-based software. In *International Conference on Software Maintenance (ICSM)*.

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

Xavier, L., Brito, A., Hora, A., and Valente, M. T. (2017). Historical and impact analysis of API breaking changes: A large-scale study. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*.

STF: uma abordagem Social para estimar Truck Factor no GitHub

Hercules Sandim[1], **Michele A. Brandão**[1], **Mirella M. Moro**[1]

[1]Universidade Federal de Minas Gerais, Belo Horizonte, Brasil

herculessandim@ufmg.br, {micheleabrandao,mirella}@dcc.ufmg.br

Abstract. *Truck Factor (TF) is the number of developers who would disrupt a project if they abandoned it. Calculating it is a complex task, and there are few approaches to estimate it. Here, we propose an approach to estimate TF in a GitHub collaboration network based on the ties strength between developers. We also evaluate it against the state of art with promising results.*

Resumo. *Truck Factor (TF) é o número de desenvolvedores que perturbariam um projeto se o abandonassem. Calculá-lo é uma tarefa de alta complexidade, e são poucas iniciativas para estimá-lo. Aqui, propomos um método para estimar o TF em uma rede de colaboração do GitHub com base na força dos relacionamentos entre os desenvolvedores. Sua avaliação produz resultados promissores.*

1. Introdução

Na Engenharia de Software, *Truck Factor – TF* (também conhecido como *Bus Factor*, *Lottery Factor*, *Bus/Truck Number* ou *Lorry Factor*) é uma medida de risco que analisa o grau de conhecimento compartilhado em um projeto de software. Apesar da alta complexidade [Hannebauer and Gruhn 2014], o cálculo do TF é importante para identificar a formação de silos de conhecimento (quando apenas uma pessoa ou grupo de pessoas detêm o conhecimento) entre equipes de desenvolvimento, no intuito de antever riscos de descontinuidade em projetos de software [Avelino et al. 2016]. Porém, há poucas iniciativas para estimar o TF de um sistema, e não escalam para grandes equipes. As exceções são os estudos em [Avelino et al. 2016, Ferreira et al. 2017], os quais estimam o TF com boa acurácia de acordo com evidências empíricas, utilizando dados de atividade de manutenção extraídos do sistema de controle de versões de repositórios do GitHub[1].

Os dados coletados do GitHub podem ser utilizados para os mais variados propósitos. Por exemplo, é possível modelar uma rede social de desenvolvedores que interagem através de criação, contribuição e compartilhamento de repositórios e projetos de software [Alves et al. 2016, Batista et al. 2017, Li et al. 2017]. Dessa forma, modelando a rede como um grafo, várias análises podem ser feitas através de métricas topológicas e semânticas sobre os nós e/ou as arestas. A força de colaboração entre os desenvolvedores do GitHub possui ampla aplicabilidade em tarefas de ranqueamento, recomendação e detecção de comunidade [Batista et al. 2017, Easley and Kleinberg 2010, Li et al. 2017].

Aqui, propomos o *STF - Social Truck Factor*, uma abordagem social para estimar o TF a partir de uma rede de colaboração sobre o GitHub. STF se baseia no ranqueamento de desenvolvedores a partir da agregação de métricas topológicas e semânticas da

[1]Plataforma para hospedagem e controle de versões de codigos-fonte: `https://github.com`

rede de colaboração. Considerar tal aspecto social permite melhor capturar o nível de interação entre os desenvolvedores [Batista et al. 2017]. Ademais, o STF não necessita que sejam realizadas pesquisas com desenvolvedores, como em algoritmos do estado-da-arte [Avelino et al. 2016, Ferreira et al. 2017]. Após apresentá-la, também descrevemos os resultados de uma análise preliminar, comparando seu desempenho ao estado-da-arte.

2. Soluções Atuais para *Truck Factor*

De maneira geral, TF é o número de desenvolvedores que interromperiam o projeto se abandonassem o mesmo. Sistemas com baixo valor de TF apresentam forte dependência de poucas pessoas do time de desenvolvimento, formando silos de conhecimento entre as equipes de desenvolvedores [Avelino et al. 2016]. Apesar de sua importância, soluções ótimas para seu cálculo necessitam de algoritmos de alta complexidade [Hannebauer and Gruhn 2014]. Mesmo assim, considerando o melhor de nosso conhecimento, não existe uma definição precisa e há poucas maneiras para estimá-lo.

Por exemplo, Ricca et al. [2011] e Zazworka et al. [2010] abordam estratégias para estimar o TF de um sistema, porém sem apresentar evidências empíricas e, portanto, carecendo da confiabilidade dos sistemas reais. Recentemente, Avelino et al. [2016] e Ferreira et al. [2017] propõem e comparam algoritmos para estimar TF usando dados de atividade de manutenção extraídos de sistemas de controle de versão. Por outro lado, apresentamos uma abordagem diferente para estimar o TF de um sistema por meio de métricas topológicas e semânticas extraídas de uma *rede de colaboração* do GitHub. Dessa forma, nossos resultados (ainda em fase inicial) são comparados aos resultados e ao oráculo proposto por Avelino et al. [2016] e Ferreira et al. [2017] na Seção 5.

3. Ranqueamento dos Desenvolvedores

Essa seção apresenta uma modelagem da rede de colaboração do GitHub e a sumarização de métricas topológicas e semânticas apresentadas em [Adamic and Adar 2003, Batista et al. 2017]. Após, introduz uma estratégia para ranqueamento de desenvolvedores com base na agregação das classificações obtidas através das métricas para força de colaboração entre desenvolvedores de um mesmo repositório. Tal ranqueamento é parte crucial para o Algoritmo STF (Seção 4).

Modelagem e Métricas para Força de Colaboração. O GitHub pode ser considerado como uma rede complexa, a qual é modelada como um grafo $\mathcal{G} = (\mathcal{V}, \mathcal{E})$: $\mathcal{V}$ é o conjunto de nós que representam os desenvolvedores, e $\mathcal{E}$ é o conjunto de arestas não-direcionadas que conectam pares que colaboram em um mesmo repositório. A partir do grafo, é possível identificar propriedades topológicas (referem-se às características da *estrutura* da rede) e semânticas (características particulares da rede que capturam o significado de seus elementos). Considerando o GitHub, a Tabela 1 resume as propriedades topológicas definidas em [Adamic and Adar 2003, Brandão and Moro 2017] e que auxiliam a analisar as colaborações em desenvolvimento de software, bem como as propriedades semânticas de colaboração com as métricas propostas em [Alves et al. 2016, Batista et al. 2017].

Ranqueamento *CombSUM* – CSR. Métodos de classificação podem ser aplicados ao GitHub para identificar especialistas, ou seja, classificar os desenvolvedores de acordo

Tabela 1. Propriedades Topológicas e Semânticas aplicadas ao GitHub.

Para um nó X da rede: $\mathcal{N}(X)$ é o conjunto de vizinhos de X, $w(X)$ é a soma dos pesos das arestas conectadas a X, $w(X,Y)$ é o peso da aresta entre X e Y, e $\mathcal{R}$ é o conjunto de todos os repositórios onde X e Y colaboram

Propriedades topológicas	
Métrica	**Definição e interpretação**
Adamic-Adar Coefficient (**AA**) [Adamic and Adar 2003]	Estipula maior peso aos vizinhos que não se relacionam com muitos outros, e é definida pela equação: $AA_{(X,Y)} = \sum_{\forall Z \in \mathcal{N}(X) \cap \mathcal{N}(Y)} \frac{1}{log\|\mathcal{N}(Z)\|}$.
Tieness (**T**) [Brandão and Moro 2017]	Mede a força das relações de coautoria (i.e., em rede formada por autores de trabalhos científicos). No contexto da rede de colaboração do GitHub: $T_{(X,Y)} = \frac{\frac{\|\mathcal{N}(X) \cap \mathcal{N}(Y)\|+1}{1+\|\mathcal{N}(X) \cup \mathcal{N}(Y) - \{X,Y\}\|} \|\|w(X,Y)\|\|}{2}$. A métrica *Tieness* é um caso particular de propriedade topológica, pois é ponderada por um peso $w(X,Y)$. Entretanto, no contexto do GitHub, tal peso pode ser representado pelas métricas semânticas descritas a seguir*.
Propriedades semânticas [Alves et al. 2016, Batista et al. 2017]	
Métrica	**Definição e interpretação**
Previous Collaboration (**PC**)	Seja $ND_{(r_i,t)}$ o número de desenvolvedores no repositório r_i no tempo t, $PC_{(X,Y,t)}$ é a colaboração oferecida por X e Y no tempo t: $PC_{(X,Y,t)} = \frac{\sum_{\forall r_i \in \mathcal{R}} \frac{1}{ND_{(r_i,t)}}}{\|\mathcal{R}\|}$.
Jointly Developers Contribution to Shared Repositories (**JCSR**)	Seja $JCSR_{(X,Y,r_i)}$ a razão entre a quantidade de desenvolvedores envolvidos no relacionamento e o total de desenvolvedores em r_i: $JCSR_{(X,Y)} = \frac{\sum_{\forall r_i \in \mathcal{R}} JCSR_{(X,Y,r_i)}}{\|\mathcal{R}\|}$.
Jointly Developers Commits to Shared Repositories (**JCOSR**)	Sejam $NC_{(X,r_i)}$ o número de *commits* por um usuário X em um repositório r_i e $NC_{(r_i)}$ o total de *commits* no repositório r_i, temos que: $JCOSR_{(X,Y)} = \sum_{\forall r_i \in \mathcal{R}} \frac{NC_{(X,r_i)}+NC_{(Y,r_i)}}{NC_{(r_i)}}$.

*As métricas semânticas, se combinadas com a métrica *Tieness*, denotam novas métricas: T_PC, T_JCOSR e T_JCSR.

com algum recurso relacionado à experiência. Aqui, para cada métrica, os desenvolvedores são classificados a partir do somatório da força de seus relacionamentos para identificar os principais desenvolvedores sob o aspecto da métrica utilizada. Por exemplo, o ranqueamento dos desenvolvedores pela métrica JSCR (denotado por R_JSCR) é definido através da ordenação decrescente do conjunto: $\{\sum_{\forall j \in \mathcal{N}_{(X)}} JSCR_{(X,j)} \mid \forall X \in \mathcal{V}\}$.

De forma suplementar, denotamos R_JSCR(r_i) como o ranqueamento de desenvolvedores que atuam num mesmo repositório r_i. Assim, os ranqueamentos para as demais métricas são os conjuntos R_AA(r_i), R_PC(r_i), R_JCOSR(r_i), R_T_JSCR(r_i) e R_T_JCOSR(r_i). Após ranquear, combinamos as diferentes classificações para gerar um ranqueamento único através do método *CombSUM* que agrega as classificações pela soma dos valores [Ganjisaffar et al. 2011]. A classificação final (denotada por CSR) é definida por ordem decrescente da soma dos valores obtidos por cada desenvolvedor nas métricas. Há outros métodos para agregação de ranqueamentos, como *Borda Count* [Emerson 2013] e *Condorcet* [Young 1988], mas o método *CombSUM* é o único que mantém o valor agregado, ao invés de simplesmente retornar o novo ranqueamento. Além disso, ressaltamos que todas as métricas apresentam-se normalizadas dentro do intervalo [0,1] para garantir que não haja viés causado pela distribuição diferente de valores.

4. Algoritmo *Social Truck Factor* – STF

O Algoritmo 1 é a nossa solução para estimar o TF de um sistema com base nas propriedades topológicas e semânticas de uma rede social de colaboração do GitHub. Denominado *Social Truck Factor* – STF, tal algoritmo recebe uma rede de colaboração do GitHub, um repositório alvo e um parâmetro ϵ (critério de parada). O STF computa os ranqueamentos de desenvolvedores com base nas métricas (linha 2) e produz um ranqueamento único

Algoritmo 1: SOCIAL_TRUCK_FACTOR

```
   Entrada: RedeDeColaboracaoGitHub, repos, ϵ
   Saída: Lista estimada de Truck Factors do repositório repos.
1  início
2      R[] = ComputarTodosRanks(RedeDeColaboracaoGitHub, repos)
3      CSR = CombSum (R)
4      STF_LISTA=[ CSR[1] ]
5      i = 2
6      enquanto (i ≤ |CSR|) e ((CSR[i]-CSR[i-1]) ≤ ϵ) faça
7          STF_LISTA[i] = CSR[i]
8          i = i + 1
9      fim
10 fim
11 retorna STF_LISTA
```

(CSR) com base na estratégia apresentada na Seção 3 (linha 3). O primeiro desenvolvedor do ranqueamento CSR inicializa a lista estimada de *Truck Factors* (STF_LISTA, na linha 4). Então, o método iterativo (linhas 6 a 9) seleciona os próximos desenvolvedores do ranqueamento CSR para compor a STF_LISTA, até satisfazer um dos dois critérios de parada (linha 6): percorrer todos os desenvolvedores do ranqueamento CSR; ou atingir um limiar de convergência ϵ, que representa o fator de tolerância máxima para a diferença entre as métricas agregadas dos desenvolvedores das *i*-ésima e (*i+1*)-ésima posições de CSR. O STF retorna a lista de desenvolvedores integrantes do TF, em ordem decrescente pelo valor do CSR (linha 11). Consequentemente, o TF é estimado pelo tamanho da lista.

Em suma, o STF seleciona, de maneira gulosa, os desenvolvedores de maior peso agregado no ranqueamento CSR. Contudo, a calibração do parâmetro ϵ é um desafio para alcançar resultados satisfatórios, pois é importante compreender a natureza dos repositórios. Repositórios grandes e relevantes para a comunidade de software livre (por exemplo, torvalds/linux[2]) tendem a ter um TF maior. Essa afirmação se deve à importância do projeto para a comunidade, onde não pode haver forte dependência de poucos desenvolvedores para mitigar os impactos de possíveis abandonos durante o ciclo de vida do projeto. Por outro lado, a maioria dos demais repositórios tendem a ter TF menor, pois são geralmente instanciados e mantidos por equipes pequenas ou indivíduos. Essas evidências empíricas são relatadas por [Avelino et al. 2016, Ferreira et al. 2017], e a calibração do parâmetro ϵ é discutida a seguir.

5. Análise Experimental

Para apresentar a análise experimental, primeiro descrevemos a metodologia com escolha do conjunto de dados, métricas utilizadas, configuração do parâmetro ϵ e considerações iniciais para a comparação com o estado-da-arte. Em seguida, apresentamos uma análise preliminar do STF em comparação com os resultados obtidos no estado-da-arte. Então, apresentamos considerações que reafirmam a validade do nosso estudo.

Metodologia. Nossa metodologia é composta por uma sequência de passos necessários para a análise preliminar comparativa com o estado-da-arte, como segue. Primeiro, o conjunto de dados utilizado é o GitSED (*GitHub Socially Enhanced Dataset*) [Batista et al. 2017], um conjunto de dados do GitHub curado, expandido e enrique-

[2] Repositório GitHub com os códigos-fonte do *Kernel* do Linux.

Tabela 2. Número de contribuidores por repositório e linguagem.

Linguagem Javascript		Linguagem Ruby		Linguagem Java	
Repositório	**# Contribuidores**	**Repositório**	**# Contribuidores**	**Repositório**	**# Contribuidores**
r.js	10	Google-Maps-for-Rails	10	code_swarm	10
code-standards	10	practicing-ruby-web	10	join-plugin	10
html5shiv	10	orm_adapter	10	jogl-demos	10
grunt	30	parallel_tests	30	kernel	29
node_redis	30	sprockets	30	playn	30
Font-Awesome	30	janky	30	thredds	30
node	703	gitlabhq	521	elasticsearch	435
angular.js	1440	rails	2296	frameworks_base	506

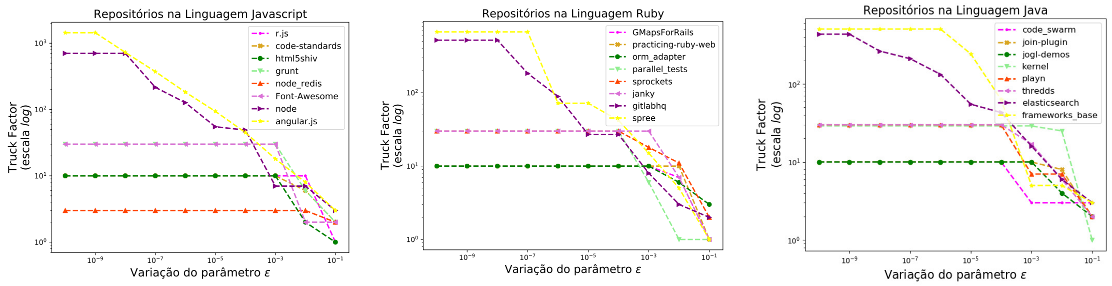


Figura 1. Resultados do STF variando o ϵ no intervalo [10^{-10},10^{-1}]. Os eixos das ordenadas (ϵ) e das abscissas (*Truck Factor*) estão na escala log.

cido a partir do GHTorrent [Gousios 2013][3]. Este conjunto contém dados até setembro de 2015 e apresenta informações sobre desenvolvedores e repositórios, de acordo com a modelagem e o cálculo das métricas discutidas na Seção 3, o que poupa relevante esforço computacional. Tal versão do GitSED considera repositórios desenvolvidos em apenas três linguagens de programação (Javascript, Ruby e Java). Apesar disso, o conjunto de dados é adequado para uma comparação preliminar com os resultados em [Avelino et al. 2016, Ferreira et al. 2017], que contém dados até fevereiro de 2015.

Segundo, sobre métricas para força de colaboração no GitHub, Batista et al. [2017] propõem um modelo (com base no estudo de correlações entre as métricas) para selecionar as métricas mais adequadas para mensurar a força de colaboração no mesmo contexto. Assim, nós selecionamos as métricas AA, PC, JCSR, JCOSR, T_JCOSR e T_JCSR, para a construção do ranqueamento de desenvolvedores.

Terceiro, para a calibração do parâmetro ϵ, realizamos experimentos iniciais em 24 repositórios do GitHub (oito repositórios para cada linguagem de programação do GitSED). A seleção desses repositórios segue a variação na linguagem de programação e no número de contribuidores ativos, conforme a Tabela 2. A Figura 1 apresenta os resultados dessa avaliação preliminar sobre o comportamento do STF, variando o ϵ no intervalo [10^{-10}, 10^{-1}]. Corroborando os resultados obtidos por [Avelino et al. 2016, Ferreira et al. 2017], o valor de ϵ deve ser calibrado para o intervalo [10^{-3}, 10^{-1}], pois assim os repositórios com maior número de contribuidores apresentam TF menores, mas ainda maiores em comparação aos repositórios com menor número de contribuidores (onde o TF é próximo de 1). Além disso, não identificamos diferença significativa entre as três linguagens analisadas.

[3] Vários estudos sobre produção de software consideram dados extraídos do GHTorrent em vez de extrair diretamente do GitHub, como por exemplo em [Jarczyk et al. 2018].

Quarto, para comparar com o estado-da-arte, os estudos em [Avelino et al. 2016, Ferreira et al. 2017] se destacam por se apoiar em evidências empíricas obtidas pela construção de um oráculo para aferir uma boa acurácia (medida pelo erro absoluto: $TF_{algoritmo} - TF_{oráculo}$) de seus resultados. Inicialmente, Avelino et al. [2016] propõem uma nova abordagem para estimar o TF de um sistema, executando-a em 133 repositórios do GitHub e avaliando-a por meio de uma pesquisa realizada com desenvolvedores de 67 destes repositórios. Neste caso, a acurácia em descobrir os principais desenvolvedores alcançou 84%, enquanto que a acurácia em estimar corretamente o TF atingiu 53%. Similarmente, Ferreira et al. [2017] realizam uma comparação entre três algoritmos para estimar o TF, verificando a acurácia por meio de um oráculo construído através de pesquisas realizadas com desenvolvedores em 35 repositórios de software-livre no GitHub. Neste caso, a acurácia chega a 100% em 20 repositórios que obtiveram TF=1, porém a acurácia cai para cerca de 50% nos demais repositórios. A disponibilidade pública dos resultados obtidos pelo estado-da-arte[4] e do conjunto de dados do GitSED[5] possibilitam que nossos resultados sejam reproduzíveis em estudos correlatos. Como aqui apresentamos uma investigação inicial para estimar o TF de um sistema, as análises preliminares foram realizadas apenas com nove repositórios que estão descritos na Tabela 3.

Comparação com o Estado-da-Arte. A Tabela 3 sumariza nossos resultados quantitativos. O Algoritmo STF estima corretamente o TF em 55% dos casos analisados (celluloid/celluloid, gruntjs/grunt, Leaflet/Leaflet, excilys/androidannotations, netty/netty), e o resultado foi muito próximo nos demais casos (erro médio de 1,25). Apesar disso, percebemos que tais resultados são obtidos com variação do parâmetro ϵ, afirmando a necessidade de maior discussão a respeito da adequação de tal parâmetro. Enquanto o algoritmo do estado-da-arte apresenta maior acurácia para repositórios com TF=1, o STF apresenta maior acurácia para repositórios com TF no intervalo [1,4].

Além da análise quantitativa, o oráculo possibilita uma análise qualitativa no sentido de verificar quais são os desenvolvedores melhor ranqueados para compor o TF. Por exemplo, no repositório emberjs/ember.js, o ranqueamento dos desenvolvedores pelas métricas topológicas e semânticas está descrito na Tabela 4, destacando os ranqueamentos R_JCOSR, R_JCSR e R_AA por identificar o maior número de desenvolvedores (até sete) dentre os elencados pelo oráculo. Tais métricas ainda se destacam por identificar desenvolvedores listados pelo oráculo e não identificados pelo estado-da-arte (MatthewBeale, ErikBryn, AlexMatchneer e TrekGlowacki). Além disso, o STF não retorna o desenvolvedor CharlesJoolley, o qual foi identificado pelo estado-da-arte mas não confirmado pelo oráculo. Note que, por considerar os *commits*, R_JCOSR possui maior proximidade ao conceito de *Truck Factor*. Por outro lado, as demais métricas (R_PC, R_T_JCSR e R_T_JCOSR) não apresentam resultados satisfatórios e poderiam ser descartadas para a composição do ranqueamento agregado.

Ameaças à Validade. A nossa avaliação com dados reais foi possível (e realizada) devido à disponibilidade pública dos mesmos. O STF é original, com implementação inspecionada por nosso grupo de pesquisa. A diferença temporal entre o conjunto de dados utilizado e o conjunto de dados do estado-da-arte é pequena diante do esforço computa-

[4]Disponível em `http://aserg.labsoft.dcc.ufmg.br/truckfactor`.
[5]Disponível em `http://bit.ly/proj-apoena`.

Tabela 3. Repositórios utilizados na análise preliminar do Algoritmo STF.

Repositório	Linguagem	NC*	TF_O*	TF_EA*	STF	
					STF*	ϵ
alexreisner/geocoder	Ruby	233	1	1	2	10^{-1}
celluloid/celluloid	Ruby	90	1	1	1	10^{-1}
rails/rails	Ruby	2296	12	9	13	10^{-3}
gruntjs/grunt	Javascript	64	1	1	1	10^{-2}
Leaflet/Leaflet	Javascript	446	1	1	1	10^{-1}
emberjs/ember.js	Javascript	364	11	6	8	10^{-3}
nicolasgramlich/AndEngine	Java	21	1	1	2	10^{-2}
excilys/androidannotations	Java	58	4	2	4	10^{-1}
netty/netty	Java	238	2	2	2	10^{-1}

*NC: Número de contibuidores, TF_O: TF do oráculo do Estado-da-Arte,
TF_EA: TF do Estado-da-Arte, STF: TF estimado pelo Algoritmo STF.

Tabela 4. Os 10 principais desenvolvedores no repositório emberjs/ember.js **através das métricas apresentadas na Seção ??.**

Pos.	R_AA	R_PC	R_JCSR	R_T_JCSR	R_JCOSR	R_T_JCOSR
1º	Ryunosuke SATO	Rob Monie	Yehuda Katz	Roy Daniels	**Stefan Penner**	**Kristofor Selden**
2º	**Peter Wagenet**	Roy Daniels	**Peter Wagenet**	**Kristofor Selden**	**Peter Wagenet**	Eric Schank
3º	Yehuda Katz	Chris Conley	**Stefan Penner**	Falk Pauser	**Robert Jackson**	Mitch Lloyd
4º	**Trek Glowacki**	**Kristofor Selden**	**Erik Bryn**	Christoph	Yehuda Katz	Roy Daniels
5º	Tom Dale	Gustavo Beathyate	Clemens Müller	darkbaby123	**Matthew Beale**	Yoran Brondsema
6º	**Robert Jackson**	Brandon Turner	Tom Dale	HipsterBrown	**Erik Bryn**	Godfrey Chan
7º	**Stefan Penner**	Doug Mayer	**Tomhuda Katzdale**	Godfrey Chan	**Alex Matchneer**	Selva Ganesh
8º	**Matthew Beale**	Falk Pauser	Francesco Rodríguez	Nook Orphan	Stanley Stuart	Travis Hoover
9º	**Tomhuda Katzdale**	**Alex Matchneer**	**Matthew Beale**	Richard Lopes	**Martin Muñoz**	Adam Luikart
10º	Stanley Stuart	Michael Latta	James Rosen	Craig Teegarden	Tom Dale	Rob Monie

Estado-da-arte: Robert Jackson, Peter Wagenet, Charles Jolley, Tomhuda Katzdale, Stefan Penner e Martin Munoz.
Oráculo: Robert Jackson, Peter Wagenet, Tomhuda Katzdale, Stefan Penner, Martin Munoz, Kristofor Selden, Erik Bryn, Alex Matchneer Matthew Beale, Trek Glowacki e Edward Faulkner.
Observação: Nomes em cor azul são nossos acertos em relação ao oráculo.
O nome em cor vermelha representa um erro do estado-da-arte comparado ao oráculo e às nossas métricas.

cional para composição de um novo conjunto de dados, com a modelagem e o cálculo das métricas. A análise comparativa foi realizada com um pequeno sub-conjunto dos repositórios analisados pelo estado-da-arte, porém houve o cuidado em selecionar repositórios com variação em linguagem de programação, número de contribuidores e TF_O (respeitadas as limitações do conjunto de dados de entrada). Todas essas considerações são importantes e apontam para a continuidade deste estudo.

6. Conclusão

Computar o *Truck Factor - TF* de um sistema é uma tarefa pouco escalável e de alta complexidade. Por exemplo, a maioria das soluções existentes apresentam resultados estimados para pequenas equipes de desenvolvimento (com menos de 40 desenvolvedores) e sem apoio de evidências empíricas. Assim, propostas para estimar o TF são pertinentes para identificar riscos de descontinuidade em projetos de software. Neste artigo, foi descrita uma solução através do ranqueamento de desenvolvedores a partir da agregação de métricas topológicas e semânticas de uma rede de colaboração do GitHub. Realizamos ainda uma análise comparativa preliminar com o estado-da-arte para a validação do nosso modelo. Em resumo, o STF apresentou resultados muito próximos (ou iguais) aos obtidos no estado-da-arte, principalmente para repositórios com TF menor (entre 1 e 4). Além disso, em um repositório específico, os ranqueamentos de desenvolvedores (pelas métricas AA, R_JCSR e R_JCOSR) foram capazes de elencar melhor os desenvolvedores em comparação com o estado-da-arte. Assim, nossa proposta configura-se como uma alternativa viável para estimar o TF, principalmente pela simplicidade de cálculo ao considerar o uso das métricas pré-processadas pelo GitSED. Como trabalhos futuros, planeja-se considerar repositórios de mais linguagens de programação, calibrar melhor o ϵ, identificar métricas que considerem outras granularidades de relacionamento (número de linhas adicionadas ou removidas nos *commits*, por exemplo), bem como utilizar outros métodos de agregação.

Agradecimentos. Trabalho parcialmente financiado por CAPES, CNPq e FAPEMIG.

Referências

[Adamic and Adar 2003] Adamic, L. A. and Adar, E. (2003). Friends and neighbors on the web. *Social Networks*, 25(3):211–230.

[Alves et al. 2016] Alves, G. B. et al. (2016). The strength of social coding collaboration on github. In *SSBD*, pages 247–252.

[Avelino et al. 2016] Avelino, G., Passos, L. T., Hora, A. C., and Valente, M. T. (2016). A novel approach for estimating truck factors. In *ICPC*, pages 1–10.

[Batista et al. 2017] Batista, N. A. et al. (2017). Collaboration Strength Metrics and Analyses on GitHub. In *WI*, pages 170–178.

[Brandão and Moro 2017] Brandão, M. A. and Moro, M. M. (2017). The strength of co-authorship ties through different topological properties. *JBCS*, 23(1):5.

[Easley and Kleinberg 2010] Easley, D. A. and Kleinberg, J. M. (2010). *Networks, Crowds, and Markets - Reasoning About a Highly Connected World*. Cambridge Un. Press.

[Emerson 2013] Emerson, P. (2013). The original borda count and partial voting. *Social Choice and Welfare*, 40(2):353–358.

[Ferreira et al. 2017] Ferreira, M. M. et al. (2017). A comparison of three algorithms for computing truck factors. In *ICPC*, pages 207–217.

[Ganjisaffar et al. 2011] Ganjisaffar, Y. et al. (2011). Bagging gradient-boosted trees for high precision, low variance ranking models. In *SIGIR*, pages 85–94, Beijing, China.

[Gousios 2013] Gousios, G. (2013). The GHTorrent Dataset and Tool Suite. In *MSR*, pages 233–236.

[Hannebauer and Gruhn 2014] Hannebauer, C. and Gruhn, V. (2014). Algorithmic complexity of the truck factor calculation. In *PROFES*, pages 119–133.

[Jarczyk et al. 2018] Jarczyk, O. et al. (2018). Surgical teams on GitHub: Modeling performance of GitHub project development processes. *Information and Software Technology*, 100:32–46.

[Li et al. 2017] Li, L. et al. (2017). Predicting software revision outcomes on GitHub using structural holes theory. *Computer Networks*, 114:114–124.

[Ricca et al. 2011] Ricca, F. et al. (2011). On the difficulty of computing the truck factor. In *PROFES*, pages 337–351.

[Young 1988] Young, H. P. (1988). Condorcet's theory of voting. *American Political science review*, 82(4):1231–1244.

[Zazworka et al. 2010] Zazworka, N. et al. (2010). Are developers complying with the process: an XP study. In *ESEM*, pages 14:1–14:10.

Heurísticas para Identificação de Ambiguidade de Autores em Projetos *Open Source*

Talita S. Orfanó [1], Kecia A. M. Ferreira [2], Mariza A. S. Bigonha [1]

Departamento de Ciência da Computação - ICEx/UFMG [1]
Departamento de Computação - CEFET-MG [2]

{talita.orfano, mariza}@dcc.ufmg.br [1], kecia@decom.cefetmg.br [2]

Abstract. *Detection of ambiguity among project developers has been a constant challenge in the study of public repositories. In order to mitigate this problem, we implemented and analyzed five heuristics that aim to identify ambiguities in GitHub projects. These heuristics were originally created to solve the problem in the context of mailing lists. The heuristics that obtained the best results were the Bird, Robles and Canfora.*

Resumo. *A detecção de ambiguidade de desenvolvedores de um projeto tem sido um constante desafio enfrentado no estudo de repositórios públicos. Com a intenção de mitigar este problema, implementamos e analisamos cinco heurísticas que visam identificar ambiguidades em projetos do GitHub. Essas heurísticas foram criadas originalmente para solucionar o problema no contexto de listas de discussões de e-mail. As heurísticas que obtiveram os melhores resultados foram as de Bird, Robles e Canfora.*

1. Introdução

Diante do essencial papel que os repositórios públicos assumiram nos últimos anos, como o GitHub, Bitbucket e GitLab, as análises e estudos acerca de seus dados e características estão constantemente presentes na literatura. Dentre as principais informações coletadas, encontram-se os dados dos contribuidores responsáveis pelas alterações feitas em um ou mais projetos do repositório. Porém, nesse conjunto de dados frequentemente estão presentes contribuidores ambíguos, isto é, usuários que, por utilizarem nomes ou e-mails diferentes em um mesmo projeto são identificados erroneamente como sendo duas ou mais pessoas distintas, e com isso a análise dos estudos é impactada.

A identificação dessa ambiguidade de autores não é uma tarefa fácil, pois não há um padrão universal para a criação de prefixos de e-mail ou para nome definido pelo usuário no repositório. Segundo Wiese et al. [2016] , essa dificuldade na identificação dos contribuidores é uma ameaça importante à validade de estudos que se baseiam nessa informação. Destarte, existem na literatura significativos trabalhos no qual foram desenvolvidas heurísticas para solucionar esse recorrente problema, Bird et al. [2006], Canfora et al. [2011], Robles & Gonzales-Barahona [2005], Goeminne & Mens [2011], Kounters et al. [2012], dentre outros.

O objetivo deste trabalho é analisar de forma comparativa cinco heurísticas de detecção de ambiguidades, a partir de contribuidores de projetos do GitHub. Usamos esse repositório para análise, pois atualmente é o repositório onde estão hospedados o maior número de softwares públicos, dadas as facilidades oferecidas pela API GitHub para extração dos dados, e consequentemente, por ser o mais usado como fonte de dados para estudos científicos.

Para a análise foram coletados dados dos colaboradores dos projetos do GitHub Joda-Time[1] , PowerShell[2] e ElasticSearch[3]. Foram implementadas as heurísticas propostas por Bird et al. [2006], Canfora et al. [2011], Robles & Gonzales-Barahona [2005], Goeminne & Mens [2013] e a Simples de Kounters et al. [2012]. Foi criada uma base de referência para avaliação comparativa acerca do resultado de cada uma das heurísticas.

Os resultados apontam que para projetos de software com até 100 contribuidores, as heurísticas retornam bons resultados, exceto a heurística Simples. Para aqueles de médio ou grande porte, recomenda-se usar as heurísticas de Bird, Robles ou Canfora. Nota-se que quanto maior o número de contribuidores, menor é a precisão das heurísticas.

As principais contribuições desse estudo compreendem: (1) a implementação das heurísticas propostas por Bird et al., Canfora et al., Robles & Gonzales-Barahona, Goeminne & Mens e a Simples de Kounters et al. adaptadas para o contexto do GitHub; (2) avaliação comparativa acerca do resultado de cada uma das heurísticas, tendo como base os dados coletados dos colaboradores de projetos do GitHub; (3) criação de três bases de referência para cada projeto analisado; (4) um algoritmo para validação dos resultados obtidos pelas heurísticas com os dados da base de referência. Acredita-se que essas implementações e o resultado deste estudo contribuem para a comunidade científica e auxiliarão na detecção de ambiguidade de futuros trabalhos que usarem dados dos contribuidores GitHub, garantindo resultados mais assertivos.

Este artigo está organizado da seguinte forma: Seção 2 apresenta um conjunto de trabalhos relacionados a temática abordada; as seções 3 e 4 mostram a metodologia usada na condução do trabalho realizado, apresentando, respectivamente, a coleta dos dados, a implementação das heurísticas e suas análises; Seção 5 expõe os resultados encontrados para cada heurística; Seção 6 discute e analisa de forma comparativa os resultados encontrados e os pontos que ameaçam a validade do trabalho e a Seção 7 sumariza os resultados encontrados e aponta os trabalhos futuros.

2. Trabalhos Relacionados

O processo para identificar ambiguidades de membros e desenvolvedores de um projeto, tem motivado diversos estudos nas últimas décadas. O trabalho de Bird et al. [2006] é um dos mais referenciados na literatura. Eles investigam como ocorre a comunicação e coordenação entre membros de um projeto *Open Source*. Para isso, realizam uma mineração da rede social do projeto Apache HTTP Server, a partir de todas as listas de discussão de e-mail de 1999 a 2006. Dessarte, eles propuseram uma heurística para identificar e tratar a ambiguidade dos participantes dessa lista. A heurística avalia por meio da distância de Levenshtein[4], a similaridade entre nomes e e-mails dos membros da lista de discussão.

Canfora et al. [2011] buscaram estudar as interações sociais entre os contribuidores responsáveis pelo *kernel* de dois sistemas operacionais *Open Source*: FreeBSD e

[1]Joda-Time: `https://github.com/JodaOrg/joda-time`
[2]PowerShell: `https://github.com/PowerShell/PowerShell`
[3]ElasticSearch: `https://github.com/elastic/elasticsearch`
[4]Levenshtein: `http://rosettacode.org/wiki/Levenshtein_distance`

OpenBSD. A coleta foi feita a partir de arquivos de lista de e-mail e repositórios CVS. Para criação da heurística, Canfora et al. basearam-se no estudo apresentado por Bird et al. [2006], porém não utilizaram da distância de Levenshtein. A heurística proposta para avaliação desses projetos compara o nome, o e-mail e suas diferentes derivações.

Robles & Gonzales-Barahona [2005] buscaram reconhecer a identidade dos desenvolvedores por meio da coleta de informações entre diversas fontes de dados em contextos distintos como: lista de *e-mail*, código fonte, repositório de controle de versão, *bug tracking*, dentre outras. A heurística proposta foi aplicada no projeto GNOME. Fontes como servidores GPG e o repositório CVS forneceram nome, *username* e *e-mails* dos desenvolvedores do projeto.

Goeminne & Mens [2011] e, posteriormente, Kounters et al. [2012] propuseram uma heurística simples para detecção de ambiguidade de autores. Essa heurística avalia a similaridade da identidade dos autores, considerando que os elementos em análise, isto é, nome ou *e-mails*, sejam iguais em pelo menos um comprimento mínimo l. Em seu trabalho, Goeminne & Mens [2011] avaliaram de forma comparativa as heurísticas de Bird et al. [2006], Robles & Gonzales-Barahona [2005], a simples e outra heurística proposta pelos autores. Foram avaliados três projetos *Open Source*: Brasero, Evince e Subversion. Foram coletados dados dos repositórios: (1) de código fonte, (2) de listas de discussão e (3) repositórios com o registro dos *bugs* reportados no projeto.

Wiese et al. [2016] comparam seis heurísticas que buscam identificar múltiplos endereços de *e-mail* pertencentes a um mesmo membro, em listas de discussão de *e-mail*. Os autores analisam 150 listas de discussão referentes ao projeto da Apache Software Foundation. Esse estudo também avalia se a janela de tempo e o tamanho da comunidade influenciam no desempenho das heurísticas.

Embora hajam diversos trabalhos que avaliam heurísticas no contexto de lista de discussão, ainda há pouco estudo relacionado com a assertividade dessas heurísticas no contexto exclusivo de repositórios GitHub. Como a demanda pela detecção de ambiguidade é alta nos trabalhos que mineram dados de repositórios públicos, e em geral, os autores não possuem a disponibilidade para avaliar outras fontes de dados, este trabalho de pesquisa busca suprir essa necessidade, via a avaliação das heurísticas propostas por Bird et al., Canfora et al., Robles & Gonzales-Barahona, Goeminne & Mens e Kounters et al., usando projetos presentes no GitHub. As seções 3 e 4 descrevem a coleta de dados a partir de projetos do GitHub, a implementação das heurísticas para a detecção de ambiguidade de contribuidores e o algoritmo usado para sua avaliação.

3. Coleta de Dados

Nessa etapa foram coletados dados de três projetos do GitHub: Elasticsearch, PowerShell e Joda-Time. Esses dados correspondem ao conjunto <nome, *e-mail*> de cada contribuidor do projeto. Neste artigo designamos autor ou contribuidor de um projeto, o usuário responsável por efetuar cada *commit* analisado no repositório. Consideramos também projeto de pequeno porte aquele que contém até 100 contribuidores, médio os que possuem entre 100 e 500 autores e de grande porte, os projetos que possuem acima de 500.

Elasticsearch é um software que permite a busca e análise de um grande conjunto de dados em tempo real. O PowerShell é um *framework* da Microsoft multiplataforma

Tabela 1: Projetos do GitHub selecionados para análise.

Projeto	# de estrelas	# *releases*	# *commits*	# autores
Elasticsearch	26884	198	29220	908
PowerShell	8447	40	5563	162
Joda-Time	3252	54	2052	64

para automação e gerenciamento, incluindo um *shell* de linha de comando. O Joda-Time era a substituição mais utilizada para as classes de data e hora de Java anteriores a versão Java SE 8. A Tabela 1 sumariza algumas das principais informações sobre esses projetos.

Para validação dos resultados obtidos nas heurísticas fez-se necessário a criação de uma base de referência. A compilação dessa base foi feita de forma manual. Para essa construção foram considerados o nome, o *e-mail* e dados obtidos por meio de pesquisas externas ao GitHub. A base de referência foi registrada em um arquivo com extensão .csv onde, em uma mesma linha estão os *e-mails* identificados como sendo de um mesmo autor, de forma que cada linha dessa base corresponda a um autor distinto.

4. Implementação das Heurísticas

Esta seção descreve a implementação das heurísticas de Bird et al., Canfora et al., Robles & Gonzales-Barahona, Goeminne & Mens e a simples heurística proposta por Kounters et al. e Goeminne & Mens, para detecção de ambiguidade de autores de um repositório. Essas heurísticas possuem algumas ações em comum como a análise apenas do prefixo do *e-mail*; a normalização do nome e *e-mail*, retirando acento, letras maiúsculas e outras particularidades de cada heurística; a verificação da igualdade do nome e verificação da igualdade entre os prefixos de *e-mail*. No entanto, como essas heurísticas foram amplamente reconhecidas e estudadas no âmbito de listas de discussão de *e-mails*, especialmente no contexto de projetos *Open Source*, foi necessário uma adaptação de cada uma delas, uma vez que a fonte de dados considerada em nosso trabalho foi unicamente o GitHub, diferentemente do propósito original das heurísticas.

Heurística de Bird et al. [2006] . Esta heurística compara a similaridade entre o primeiro e último nome, o nome completo e o prefixo de *e-mail*. As duas identidades <*nome, e-mail*> são consideradas como pertencentes a um mesmo autor se a similaridade de Levenshtein for maior ou igual ao *threshold* t. Além disso, também é verificado se o prefixo de *e-mail* de um desenvolvedor contém o nome, sobrenome ou derivações desses, semelhantes ao outro desenvolvedor. Foram escolhidos arbitrariamente cinco valores para t: 0,84; 0,87; 0,90; 0,93; 0,96 e 0,99. Para cada um dos softwares analisados foram realizados seis experimentos com essa heurística, um para cada valor de t.

Heurística de Canfora et al. [2011] . Podemos seguimentar essa heurística em dois grandes módulos, o módulo que valida a ambiguidade entre os nomes e suas derivações, e o módulo que compara os *e-mails* e as suas derivações. O primero módulo compara: os nomes ignorando o nome do meio; o nome com as iniciais, comprovando que j k s é igual a *John Kennedy Simith*; apenas o último nome e outras derivações originárias do nome dos contribuidores. Enquanto o segundo módulo compara a igualdade entre os prefixos, o nome extraído a partir do prefixo, como *john.smith* é equivalente a *John*

Smith e outras informações oriundas do prefixo e comparadas com o nome dos autores.

Heurística de Robles & Gonzales-Barahona [2005] . A partir do nome do desenvolvedor, esta heurística cria um conjunto de possíveis prefixos de *e-mail* e a utiliza para comparar com o prefixo de *e-mail* de outro desenvolvedor. Alguns possíveis prefixos são, por exemplo, nome@dominio.com, nome.sobrenome@dominio.com, nome_sobrenome@dominio.com.

Heurística de Goeminne & Mens [2011] . Dois contribuidores são considerados ambíguos se a similaridade de Levenshtein entre os nomes ou prefixos de *e-mail* for maior ou igual a *t*. Também é verificado se o prefixo de *e-mail* contém parte do nome (de tamanho l), para isso o nome é separado por meio dos caracteres '_', '-', ' ', '+', ',' e '.'. Essa heurística se baseia em Robles & Gonzales-Barahona [2005] e cria uma lista de possíveis prefixos de *e-mail*, a partir do nome do contribuidor. Alguns possíveis prefixos são nomesobrenome, nome.sobrenome, nome_sobrenome, nsobrenome, nomeinicialsobrenome, nome-sobrenome, sobrenomenome, sobrenome_nome e assim por diante.

Heurística de Simples - Kounters et al. [2012] e Goeminne & Mens [2011] . Esta heurística verifica se o primeiro rótulo, o nome ou prefixo do *e-mail*, contém parte (de tamanho *l*) do segundo nome e o inverso também. A mesma análise é feita sobre o prefixo dos e-mails. Nessa heurística também foram escolhidos arbitrariamente cinco valores para *l*, variando de 4 a 8. Para cada software em análise obtivemos cinco resultados que serão expostos na Seção 6.

A avaliação das heurísticas foi feita por meio de um algoritmo que compara a base de referência construída com os autores identificados por cada heurística. Esse algoritmo classifica cada comparação entre os autores como sendo *falso-positivo, falso-negativo, verdadeiro-positivo* ou *verdadeiro-negativo*. Considera-se Falso-Positivo (FP) se a heurística afirma que ambos autores têm a mesma identidade, mas a base de referência garante que são pessoas diferentes. Falso-Negativo (FN) ocorre quando a heurística declara que ambos autores possuem identidades diferentes, no entanto a base referência comprova que as identidades pertencem a uma mesma pessoa. Verdadeiro-Positivo (TP) ocorre se a base de referência e a heurística constatam que ambas identidades pertencem ao mesmo contribuidor, enquanto o Verdadeiro-Negativo (TN) atesta o inverso, isto é, tanto a heurística quanto a base concordam que os autores possuem identidades distintas. Esses dados são usados no cálculo do valor de precisão e *recall* de cada uma das heurísticas [Goeminne & Mens 2011]. Desta forma, temos:

$$Precisao = \frac{tp}{tp + fp} (1) \qquad \mathit{Recall} = \frac{tp}{tp + fn} (2)$$

5. Resultados

Nesta seção são apresentados os resultados de cada heurística para os três projetos analisados: Joda-Time, PowerShell e ElasticSearch. A Tabela 2 sumariza os resultados de precisão e *recall* apresentados para cada uma das heurísticas.

Joda-Time. As heurísticas analisadas apresentaram um ótimo resultado de *recall*, conforme ilustra a Tabela 2. Isso ocorreu pois em todas as comparações realizadas pelas heurísticas não foi apontado nenhum resultado falso negativo. Exceto pela heurística simples,

Tabela 2: Precisão e *recall* das heurísticas no Joda-Time, PowerShell e ElasticSearch

	Precisão			*Recall*		
Heurísticas	Joda-Time	PoweShell	ElasticSearch	Joda-Time	PoweShell	ElasticSearch
Bird-84	1	0,32	0,32	1	0,93	0,96
Bird-87	1	0,33	0,33	1	0,93	0,96
Bird-90	1	0,33	0,33	1	0,93	0,95
Bird-93	1	0,33	0,33	1	0,93	0,95
Bird-96	1	0,33	0,33	1	0,93	0,95
Bird-99	1	0,33	0,33	1	0,93	0,95
Canfora	1	0,74	0,32	1	0,93	0,95
Simples-4	0,03	0,004	2,50E-04	1	0,96	1
Simples-5	0,17	0,1	633E-04	1	0,93	0,94
Simples-6	0,21	0,18	0,0038	1	0,93	0,94
Simples-7	0,43	0,29	0,07	1	0,93	0,93
Simples-8	0,43	0,56	0,14	1	0,93	0,92
Robles	1	0,65	0,28	1	0,96	0,98
Goeminne	1	0,58	0,07	1	0,96	0,98

as demais apresentaram 100% de precisão. O gráfico da Figura 1 mostra os resultados de *recall* e precisão, no eixo x e y respectivamente, para cada uma das heurísticas analisadas.

PowerShell. As heurísticas que obtiveram os melhores resultados foram de Canfora e Robles com 74% e 65% de precisão respectivamente. O resultado do *recall* de todas heurísticas é considerado ótimo, variando de 93 a 96%. Figura 2 ilustra esses resultados.

ElasticSearch. Para esse projeto, as heurísticas de Bird, Canfora e Robles foram aquelas com melhor precisão se comparado com as demais, no entanto a precisão foi consideravelmente baixa, atingindo apenas 33% no melhor resultado. A precisão das heurísticas Simples com comprimento *l* de 4, 5 e 6 tendeu a zero, de acordo com a Tabela 2. O valor de *recall* foi ótimo em todas as heurísticas com valores variando de 92 a 100%. Figura 3 apresenta de forma expressiva esses resultados.

6. Discussão

Os resultados deste trabalho de pesquisa mostraram que a precisão tende a reduzir proporcionalmente com o aumento do número de contribuidores no projeto. Enquanto o software Joda-Time considerado como de pequeno porte atingiu uma precisão de 100% em grande parte das heurísticas, Elasticsearch considerado de grande porte conquistou aproximadamente 33% de precisão nos melhores resultados das heurísticas. Os resultados obtidos para o *recall* foram ótimos em todas as heurísticas nos três projetos analisados. Isso mostra que as heurísticas raramente deixam de detectar uma ambiguidade.

Para compreender melhor a razão pela qual a precisão foi tão baixa para o projeto ElasticSearch, examinamos manualmente os resultados gerados por cada heurística e observamos que muitas identidades foram unidas de forma equivocada devido ao prefixo idêntico das mesmas, como é o caso dos prefixos "github", "contact" e "mail", como por exemplo "github@smithsrock.com" e "github@tobigue.de".

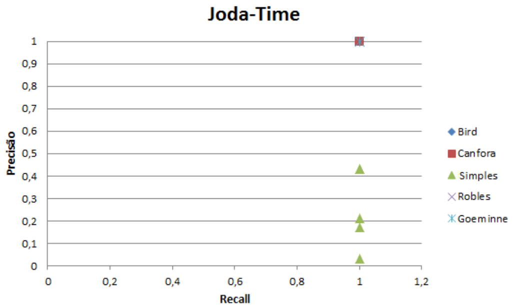


Figura 1: Representação da interação entre desenvolvedores e suas contribuições no Joda-Time

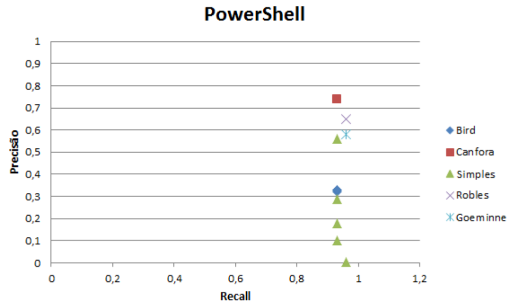


Figura 2: Representação da interação entre desenvolvedores e suas contribuições no PowerShell

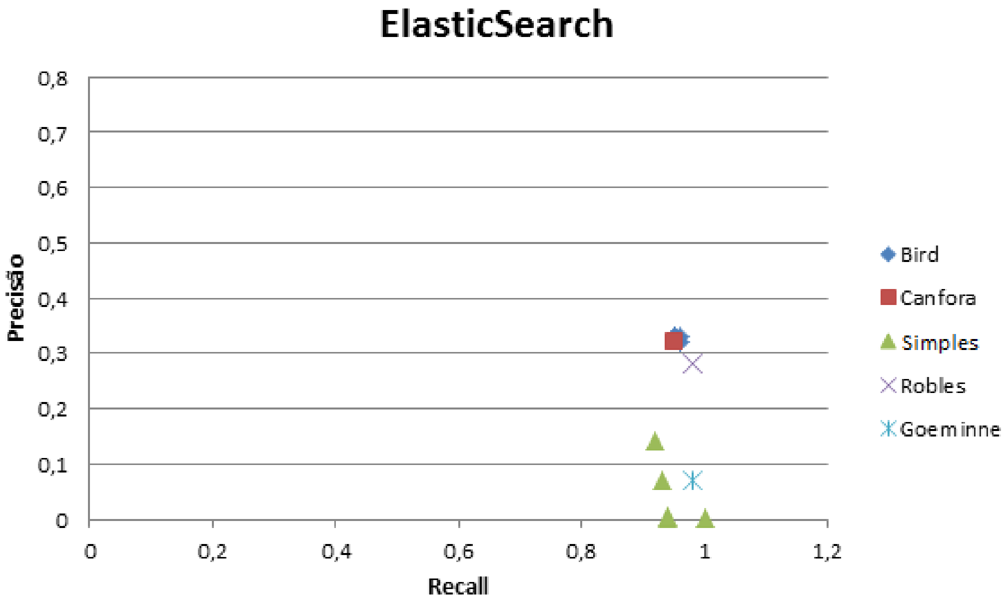


Figura 3: Representação da interação entre desenvolvedores e suas contribuições no Elasticsearch

Resultados apontam que as melhores heurísticas para se utilizar em um projeto de nível médio ou grande são a Bird, Canfora e Robles, enquanto em um projeto pequeno a heurística de Goeminne também apresenta bons resultados. No entanto, devido a baixa precisão faz-se necessário verificar dentre as ambiguidades detectadas pela heurística selecionada, se existem falso-positivo entre as identidades.

Ameaças à Validade. Apesar do minucioso cuidado na criação da base de referência de cada projeto, ela pode conter erros e influenciar no resultado comparativo de cada heurística. Além disso, as heurísticas foram implementadas baseada nas idéias dos artigos onde foram originalmente publicadas e adaptadas para o contexto do GitHub, o que não garante a total integridade das ideias propostas pelos autores. O número de projetos selecionados, bem como as características próprias desses projetos podem, porventura, ter influenciado os resultados obtidos e a análise comparativa realizada posteriormente.

7. Conclusões

A detecção de ambiguidade de autores em um projeto é um importante passo no tratamento dos dados coletados em repositórios como o GitHub. O objetivo do trabalho apresentado neste artigo foi implementar e analisar de forma comparativa cinco heurísticas que detectam ambiguidade de contribuidores. Para isso foram selecionados três projetos do GitHub: Joda-Time, PowerShell e ElasticSearch. Criamos três bases de referência, uma para cada projeto analisado, bem como a implementação de cada uma das heurísticas e de um algoritmo para validação dos resultados obtidos pelas heurísticas com os dados da base de referência.

Os três projetos apresentaram para todas as heurísticas ótimos valores de *recall*, o que nos permite concluir que as heurísticas tendem a detectar todas as ambiguidades do conjunto de dados em análise. No entanto, a precisão variou de acordo com o tamanho do projeto, o Joda-Time que possui poucos autores, apresentou uma precisão de 100%, enquanto os projetos de médio e grande porte PowerShell e ElasticSearch, resultaram em precisões de 74% e 33%, respectivamente.

Como trabalhos futuros, pretendemos: (1) analisar um conjunto maior de projetos do GitHub para obter uma conclusão madura e precisa da melhor heurística a ser usada para cada caso; e (2) implementar uma heurística que diminua o número de falsos-positivos para softwares com um grande número de autores e ambiguidades, para com isso obter uma precisão que transpareça maior confiança para os pesquisadores que necessitam dessas heurísticas.

Referências

Bird, C., Gourley, A., Devanbu, P., Gertz, M., and Swaminathan, A. (2006). Mining email social networks. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 137–143. ACM.

Canfora, G., Cerulo, L., Cimitile, M., and Di Penta, M. (2011). Social interactions around cross-system bug fixings: the case of freebsd and openbsd. In *Proceedings of the 8th working conference on mining software repositories*, pages 143–152. ACM.

Goeminne, M. and Mens, T. (2011). A comparison of identity merge algorithms for software repositories. *Science of Computer Programming*, 78(8):971–986.

Kouters, E., Vasilescu, B., Serebrenik, A., and van den Brand, M. G. (2012). Who's who in gnome: Using lsa to merge software repository identities. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 592–595. IEEE.

Robles, G. and Gonzalez-Barahona, J. M. (2005). Developer identification methods for integrated data from various sources. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–5.

Wiese, I. S., da Silva, J. T., Steinmacher, I., Treude, C., and Gerosa, M. A. (2016). Who is who in the mailing list? comparing six disambiguation heuristics to identify multiple addresses of a participant. In *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*, pages 345–355. IEEE.

Monorepos: A Multivocal Literature Review

Gleison Brito[1], **Ricardo Terra**[2], **Marco Tulio Valente**[1]

[1]Federal University of Minas Gerais, Belo Horizonte, Brazil

[2]Federal University of Lavras, Lavras, Brazil

{gleison.brito,mtov}@dcc.ufmg.br, terra@dcc.ufla.br

Abstract. *Monorepos (Monolithic Repositories) are used by large companies, such as Google and Facebook, and by popular open-source projects, such as Babel and Ember. This study provides an overview on the definition and characteristics of monorepos as well as on their benefits and challenges. Thereupon, we conducted a multivocal literature review on mostly grey literature. Our findings are fourfold. First, monorepos are single repositories that contains multiple projects, related or unrelated, sharing the same dependencies. Second, centralization and standardization are some key characteristics. Third, the main benefits include simplified dependencies, coordination of cross-project changes, and easy refactoring. Fourth, code health, codebase complexity, and tooling investments for both development and execution are considered the main challenges.*

1. Introduction

Monorepos (Monolithic Repositories or Multi-Package Repositories) are commonly described as a single repository containing more than one project, in contrast to the single-repository-per-project model [9]. This model is adopted for several large software companies, including *Facebook*, *Google*, and *Microsoft* [2, 5, 8]. Some popular *open-source* projects also adopt this model to manage their repositories (e.g., Babel and Ember).

The discussion about the adoption of monorepos is an emerging theme in the developer community. The theme is discussed in several forums and blogs, where the adoption of monorepos is either defended or rebutted. There are also much debate about the migration of multiple repositories to a single one, mainly motivated by the adoption of the monorepo model by large companies such as Google and Facebook. However, there is no consensus on the benefits and challenges of this repository model.

In view of such context, this paper provides an overview on monorepos. Thereupon, we conduced a Multivocal Literature Review based on 21 grey literature and two academic papers in order to: (i) undestand the definitions of monorepo, (ii) identify characteristics of monorepos, (iii) investigate benefits of monorepos, and (iv) investigate challenges of monorepos.

Regarding (i), monorepos are usually defined as a single repository that contains multiple projects related or unrelated, but there are two kinds of monorepos: "Monstrous" monorepos, which are commonly used by large companies, such as Google and Facebook, and Projects monorepos, which are used by medium-size open-source projects, such as React and Babel. Regarding (ii), the projects into a Monorepo share the same managing tools and the same version of dependencies, besides all projects are visible to contributors. Regarding (iii), benefits include simplified dependencies, better managing of cross-project changes, easy refactoring, simplified organization, improve overall work culture, better coodination between developers, and better support of build tools to manage the repository. Regarding (iv), challenges include difficulties with code heath, codebase complexity, tooling investiments, loss of version information, build, deploy and test tasks, and migration of many repositories to only one.

The remainder of this paper is organized as follows. Section 2 describes our research metodology. Section 3 reports the results of our multivocal literature review. Section 4 discusses the threats to validity of our study and Section 5 concludes.

2. Research Metodology

In this section, we describe our research methodology. We also provide an overview of the systematic approach used to gather relevant literature.

2.1. Literature Review

After an initial search in the literature to learn more on the topic of monorepos, we could not find a substantial body of academic research on the topic. We therefore decided to conduct a Multivocal Literature Review (MLR), which is based on all accessible literature on a topic [7]. This includes—but is not limited to—blogs, white papers, articles, and academic literature. By using this wide spectrum of literature, the results will give a more broad view at the topic since they include the voices and opinions of academics, practitioners, independent researchers, development firms, and others who have experience on the topic [7].

Garousi et al. [3] emphasize the importance of MLRs in the Software Engineering (SE) field by stating that SE practitioners produces grey literature on a great scale, but most are not published in academic vehicles. Therefore, they argue that not including that literature in systematic reviews implies in researchers missing out important current state-of-the-art practice in SE.

2.2. Research Questions

We conducted this MLR to obtain an understanding of what a monorepo model is, what are the key characteristics of this model, and what are the benefits and challenges of adopting it. In order to achieve the goal, we formulate four research questions:

RQ #1. How does the literature define monorepos?

RQ #2. What are the characteristics of monorepos?

RQ #3. What are the main expected benefits of adopting monorepos?

RQ #4. What are the main expected challenges of adopting monorepos?

2.3. Study Protocol

This section describes the systematic protocol we followed to retrieve the literature used in our study. We describe the databases, the search strategy used to find related literature, the inclusion and exclusion criteria used to find the most relevant literature, and the process in which we catalogued the literature.

Databases. We relied on Google's search engine to find relevant literature:

- **Google Search**[1] to locate grey literature (white papers, blogs, articles, etc.)
- **Google Scholar**[2] to specifically locate academic literature.

We chose Google's search engines instead of more traditional search engines (like Springer Link, ACM Digital Library, IEEE Explore, etc.) because Monorepo is a very new topic and limited academic research is available. We therefore knew before-hand that this literature review would rely mostly on grey literature.

Search Terms. The search string were built following the steps proposed by Brereton et al. [1]:

1. Derive major terms from the research questions by identifying the main concepts.
2. Identify alternative spellings and synonyms for major terms.
3. Check the keywords in any relevant papers we already have.
4. Use the boolean OR to add alternatives spellings and synonyms.
5. Use the boolean AND to link the major terms.

Since the search strings aim to find relevant literature related to the RQs, we defined them as follows:

```
("monorepo" OR "monolithic repository" OR "multi-package repository")
                                   AND
("definition" OR "definitions" OR "characteristic" OR "characteristics"
OR "benefit" OR "benefits" OR "challenge" OR "challenges")
```

Study Selection. After retrieving the results of the initial search, we excluded irrelevant articles using the following inclusion and exclusion criteria:

- Inclusion criteria:
  - Literature that explicitly discuss monorepos;
  - Literature that explicitly discuss the challenges and benefits of monorepos;
  - Literature that discuss the definition of monorepos;
  - Literature published after 2014; and
  - Literature that appears in the first five pages on Google's search.
- Exclusion criteria:
  - Inaccessible literature;
  - Results that Google Search deems to similar to other results; and
  - Vendors tool advertisements.

Search Procedure. As illustrated in Figure 1, we first perform an advanced search in Google Search and Google Scholar. For a better focus on each RQs, we split our search

[1] http://www.google.com/
[2] http://scholar.google.com/

term in four parts. We analyze 20 pages in total, five pages for each part. We therefore applied the inclusion and exclusion criteria, selecting only relevant literature for the primary study. In order to identify and incorporate the most relevant grey literature in our MLR, we again use the guidelines defined by Garousi et al. [4].

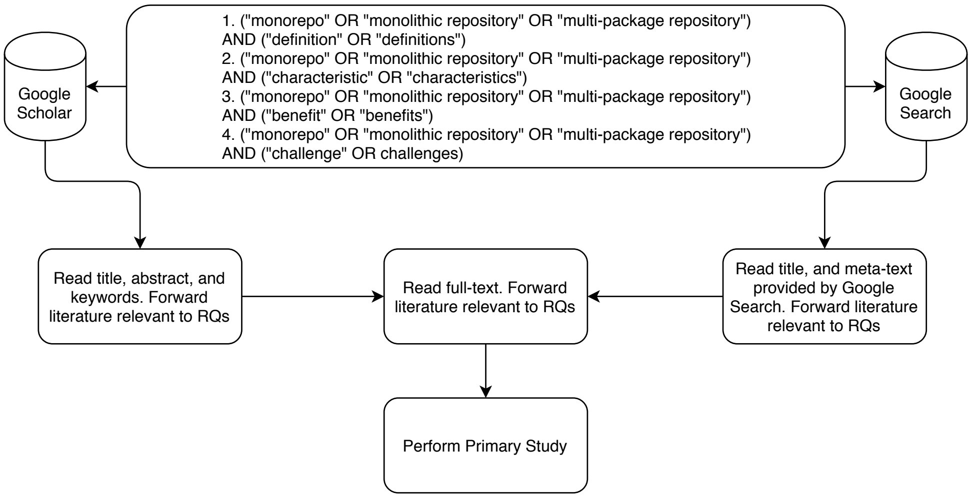


Figure 1. Search process to find relevant literature.

We performed our search procedure in June, 2018. From a total of 255 publications, we applied the inclusion and exclusion criteria to select 23 publications, as properly reported in Table 1.

3. Multivocal Literature Review

Based on the research methodology described in the previous section, this section reports the results of our multivocal literature review.

3.1. Definition of Monorepo (RQ #1)

The most common definition of Monorepo is a single repository that contains multiple projects (publications #4, #11, #13, and #19). These projects can be related or unrelated, but the fact is that they should share the same dependencies. Particularly, publications #11 and #13 define monorepos as a single repository that contains more than one logical project. The projects managed in a Monorepo can depend on each other (such as React and the react-dom package) or they can be completely unrelated (such as the Google search algorithm and Angular).

Monorepos can also be classified as *Monstrous monorepos* or *Project monorepos*, according to publication #13. Monstrous monorepos regards the sheer size to which monorepos at organizations can grow and Project monorepos describes single repositories that are used to manage the core functionality of a project and all of its components. Google and Facebook repositories are examples of Monstruous monorepos, as discussed in publications #12 and #22. Project monorepos are commonly adopted by open-source projects with many modules, such as Babel and Ember. According to publication #14, the monorepo model allows the maintenance of multiple related packages within a single repository.

[3]We have double checked all URLs on July 2nd, 2018.

Table 1. List of selected publications[3]

Literature	#	Publications
Grey	1	Anderson, B. (2017). **Code Repositories and Yak Shaving**. http://iamtherealbill.com/2017/01/repo-yak-shaving/
	2	Belagatti, P. (2016). **Microservices: Mono repo vs. multiple repositories**. https://jaxenter.com/microservices-mono-repo-vs-multiple-repositories-130148.html
	3	Karanth, D. (2016). **Microservices: Pros and Cons of Mono Repos**. https://dzone.com/articles/microservices-pros-and-cons-of-mono-repos
	4	Eberlei, B. (2015). **Monorepos**. https://qafoo.com/talks/15_10_symfony_live_berlin_monorepos.pdf
	5	Long, C. (2017). **Multirepo vs Monorepo**. https://chengl.com/multirepo-vs-monorepo/
	6	Libbey, B. (2017). **Monorepo, Manyrepo, Metarepo**. http://notes.burke.libbey.me/metarepo/
	7	Johnson, N. (2017). **Monorepo**. https://www.yonson.io/post/monorepo/
	8	Farina, M. (2016). **Dangers of Monorepo Projects**. https://dzone.com/articles/dangers-of-monorepo-projects
	9	Das, S. (2017). **Code repository for micro-services: mono repository or multiple repositories**. https://medium.com/@somakdas/code-repository-for-micro-services-mono-repository-or-multiple-repositories-d9ad6a8f6e0e
	10	Pendleton, B. (2017). **Big news in the world of source control**. http://bryanpendleton.blogspot.com/2017/02/big-news-in-world-of-source-control.html
	11	Saase, S. (2015). **Monorepos in Git**. https://developer.atlassian.com/blog/2015/10/monorepos-in-git/
	12	Goode, D. (2014). **Scaling Mercurial at Facebook**. https://code.facebook.com/posts/218678814984400/scaling-mercurial-at-facebook/
	13	Oberlehner, M. (2017). **Monorepos in the Wild**. https://medium.com/@maoberlehner/monorepos-in-the-wild-33c6eb246cb9
	14	Vepsäläinen, J. (2017). **Managing Packages Using a Monorepo**. https://survivejs.com/maintenance/appendices/monorepos/
	15	Seibel, P. (2017). **Repo style wars: mono vs multi**. http://www.gigamonkeys.com/mono-vs-multi/
	16	Luu, D. (2015). **Advantages of monorepos**. https://danluu.com/monorepo/
	17	MacIver, D. (2016). **Why you should use a single repository for all your company's projects**. https://www.drmaciver.com/2016/10/why-you-should-use-a-single-repository-for-all-your-companys-projects/
	18	Fabulich, D. (2017). **We'll Never Know Whether monorepos Are Better**. https://redfin.engineering/well-never-know-whether-monorepos-are-better-2c08ab9324c0
	19	Szorc, G. (2014). **On Monolithic Repositories**. https://gregoryszorc.com/blog/2014/09/09/on-monolithic-repositories/
	20	Beigui, P. (2016). **Mono-Repos @ Google. Are they worth it?**. https://medium.com/@pejvan/monorepos-85e608d43b57
	21	Lucido, A. (2017). **The Journey To Android Monorepo: The History Of Uber Engineering's Android Codebase Organization**. https://eng.uber.com/android-monorepo/
Academic	22	Potvin, R., & Levenberg, J. (2016). **Why Google stores billions of lines of code in a single repository**. Communications of the ACM, 59(7), 78-87.
	23	Jaspan, C. et al. (2018). **Advantages and Disadvantages of a Monolithic Repository - A case study at Google**. 40th International Conference on Software Engineering (ICSE), Software Engineering in Practice (SEIP) Track, 225-234.

3.2. Characteristics of Monorepos (RQ #2)

According to publications #22 and #23, the five main characteristics of monorepos are:

- *Centralization*: Codebase is in a single repository encompassing multiple projects.

- *Standardization*: A shared set of tools govern how engineers interact with the code, including building, testing, browsing, and reviewing code.
- *Visibility*: Code is viewable and searchable by all engineers in the organization.
- *Synchronization*: The development process is trunk-based; engineers should always commit to the head of the repository.
- *Completeness*: Any project in the repository can be built only from dependencies also checked into the repository. Dependencies are unversioned; projects must use whatever version of their dependency at the repository head.

3.3. Benefits of monorepos (RQ #3)

This section provides an overview on the benefits of monorepo model.

- *Simplified dependencies*: As discussed in publications #6 and #16, monorepo model proposes to have one universal version number for all projects. Since atomic cross-project commits are possible, the repository is always in a consistent state. Library versioning is de-emphasized. Instead, a library is expected to maintain a stable API and migrate its callers when this API changes. This depends on being able to make atomic commits.
- *Cross-project changes*: Changing APIs that are used in multiple internal projects is more simply in monorepos than in multiple repositories. According to publications #16 and #17, developers can change an API and all its callers in one commit.
- *Easy refactoring*: According to publications #2, #16 and #17, a well-organized unique repository is likely to have modular code and hence refactoring is likely to be easier in monorepos than in multiple repositories. Restructuring is also easier as everything is neatly in one place and easier to understand.
- *Simplified organization*: In monorepos, projects can be organized and grouped to be more logically consistent, as described in publications #2 and #16.
- *Improved overall work culture*: Monorepos encourage the team unification and hence each member can contribute more specifically towards the goals and objectives of the organization, as discussed in publication #2.
- *Better coordination between developers*: Developers run the entire project on their machine, which helps them understand all services and how they work together. As a result, developers tend to find more bugs locally, before sending pull requests, according to publication #2.
- *Tooling*: In monorepos, all code has a fixed path in a single shared hierarchy, which facilitate building tools that operate on multiple projects, as discussed in publications #2, #6, and #16.

3.4. Challenges of monorepos (RQ #4)

This section discuss some challenges and trade-offs of monorepos.

- *Code health*: In monorepos, according to publication #22, it is easier to add dependencies. However, (i) this reduces the incentive for software developers to produce stable and well-defined APIs; (ii) code cleanup is even more error-prone because it is common for teams to do not think about their dependency graph; and (iii) purposeless dependencies increase project exposure to downstream build breakages, leading to binary size bloating, and creating additional work in building and testing.

- *Codebase complexity*: According publication #14, the main challenge of monorepos is to manage all projects in a single repository. Although the understanding organization of the code in monolithic repositories is easy, it is a complex task to determine where new code should be placed. Besides, publication #7 criticizes monorepos since their high codebase complexity does not necessarily increase productivity.
- *Tooling investments*: A huge repository requires managing tools to scale. Publications #1 discusses the high cost of running these tools.
- *Loss of version information*: Publications #8 argues that it is dangerous to lose version information. Basically, since details of imported libraries can be lost, it may be hard to deal with updates.
- *Build, Test Bloat, and Deploy*: Due to its size, publications #3 and #8 question the cost of building and testing on monorepos, whereas publication #7 questions the cost of deploy.
- *Migration*: Publications #10 introduces “the monorepo problem”. It refers to the fact that migrating of many repositories to only one has a high cost, because it is necessary to modularize all code. This is too critical that publication #21 reports a migration study case.

4. Threats to Validity

To answer the RQs, we investigated different aspects from monorepos to support generalization of our discussions. This research is mostly based on grey literature, which means that most of the material have not been subject to rigorous peer-review, as academic research usually is. However, (i) the inclusion of the grey literature in our review overcame the scarse works available in the digital databases of scientific literature and (ii) we analyzed the grey literature in a systematic way by following the guidelines proposed by Garousi et al. [4].

5. Conclusion

This study presented a Multivocal Literature Review on monorepos based mostly on grey literature. We investigate (i) how monorepos are defined; (ii) what are the characteristics of monorepos; (iii) what are the benefits to adopt monorepos; and (iv) what are the challenges to adopt monorepos.

Regarding (i), monorepos are usually defined as a single repository that contains multiple related or unrelated projects. In this sense there are two kinds of monorepos: Monstrous monorepos, which contains several unrelated projects and millions of lines of code, and Projects monorepos, which contains related components of a specific project.

Regarding (ii), we can enumerate *centralization* since a single repository encompasses multiple projects, *visibility* since everything is visible by all contributors, *synchronization* since the development process is trunk-based, *completeness* since any project can be built using resources available in the repository, and *standardization* since engineers usually share the same set of tools in monorepos.

Regarding (iii), the main benefits include *simplified dependencies* since library versioning is easy, *simplified organization* since projects are organized in a more consistent way, *easy refactoring* since modular repositories foster modular code, *improved* *overall work culture* since monorepos encourage the team unification, *tooling* since a single shared hierarchy facilitate building tools that operate on multiple projects, *better coordination between developers* since developers can easily understand all projects and how they work together, and *better cross-project changes* since an API and its callers can be refactored in a single commit.

Regarding (iv), the main challenges include *codebase complexity* since managing all projects in a single repository is more difficult, *tooling investments* since the cost of running managing tools is large, *code health* since unnecessary dependencies can create additional building and testing work, *loss of version information* since it is dangerous to lose some version information, *build, test, and deploy* since these tasks can take a long time, and *migration* since the cost of migration of many repositories to only one is high.

Several publications compare monorepos with multirepos (#2, #5, #6, #9, #15, and #18). These studies argue that choosing monorepos or multirepo is hard because each model has its own set of principles and practices and its own challenges. Monorepos and multirepos not only have different tooling requirements, but also vary in their engineering culture and philosophy. On the one hand, some companies, such as *Netflix*, value freedom and responsability [6], thus they prefer multirepos. On the other hand, *Google* values consistency and code quality, thus it prefers monorepos.

Ideas for future work include: (i) to conduct surveys with practitioners to expand our understanding on monorepos; (ii) to conduct a study comparing monorepos and multiple repositories models; and (iii) to investigate the adoption of monorepos in open-source projects.

References

[1] Pearl Brereton, Barbara A. Kitchenham, David Budgen, Mark Turner, and Mohamed Khalil. Lessons from applying the systematic literature review process within the software engineering domain. *Journal of Systems and Software*, 80(4):571–583, 2007.

[2] Rain Brian Harrys. The largest git repo on the planet. https://blogs.msdn.microsoft.com/bharry/2017/05/24/the-largest-git-repo-on-the-planet/, 2017. [accessed 15-June-2018].

[3] Vahid Garousi, Michael Felderer, and Mika V. Mäntylä. The need for multivocal literature reviews in software engineering: complementing systematic literature reviews with grey literature. In *20th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, page 26, 2016.

[4] Vahid Garousi, Michael Felderer, and Mika V. Mäntylä. Guidelines for including the grey literature and conducting multivocal literature reviews in software engineering. *arXiv preprint arXiv:1707.02553*, 2017.

[5] Durham Goode. Scaling mercurial at Facebook. https://code.facebook.com/posts/218678814984400/scaling-mercurial-at-facebook/, 2014. [accessed 15-June-2018].

[6] Netflix. Netflix culture. https://jobs.netflix.com/culture, 2009. [accessed 15-June-2018].

[7] Rodney T. Ogawa and Betty Malen. Towards rigor in reviews of multivocal literatures: Applying the exploratory case study method. *Review of Educational Research*, 61(3):265–286, 1991.

[8] Rachel Potvin and Josh Levenberg. Why Google stores billions of lines of code in a single repository. *Communications of the ACM*, 59(7):78–87, 2016.

[9] StackOverflow. Monorepo. https://stackoverflow.com/tags/monorepo/info, 2017. [accessed 15-June-2018].

Minerando Código Comentado

Lucas Grijó[1], Andre Hora[1]

[1] Faculdade de Computação (FACOM)
Universidade Federal de Mato Grosso do Sul (UFMS)

rksgrijo@gmail.com, hora@facom.ufms.br

Abstract. *As software evolves, programmers often comment code snippets as a mean of removing them (a practice known as comment-out code). However, this bad practice may cause problems to software maintainers, such as readability reduction, distraction, and waste of time. In this context, this paper presents an empirical study to evaluate the presence of commented-out code on open source systems. We analyze 100 relevant software systems and 300 releases. As a result, we detect a rate of 4,17% commented-out code. However, we verify that this rate tends to decrease over time. Finally, based on our results, we discuss improvements to software quality tools and development practices.*

Resumo. *Durante a evolução de software, programadores comumente comentam trechos de código como forma de remoção (prática conhecida como comment-out code). Entretanto, essa má prática pode causar problemas para mantenedores de software, como redução de legibilidade, distração e perda de tempo. Nesse contexto, este artigo apresenta um estudo empírico para avaliar a presença de códigos comentado em sistemas open source. Analisa-se 100 sistemas de software relevantes e 300 releases desses projetos. Como resultado, detecta-se que 4,17% dos comentários são compostos por códigos comentado. No entanto, verifica-se que, na mediana, essa taxa tende a diminuir ao longo do tempo. Por fim, com base nos nossos resultados, são discutidas melhorias para ferramentas de qualidade de software e práticas de desenvolvimento.*

1. Introdução

Sistemas de software estão em constante evolução para acomodar necessidades de negócio. Durante o desenvolvimento, programadores utilizam diversos artifícios para lidar com a evolução de software. Uma prática comum quando programadores detectam trechos de código que precisam ser deletados é "removê-los" através de comentários de código. Essa má prática de programação, que deriva do termo em inglês *comment-out code*, se refere ao ato de comentar um trecho de código ao invés de simplesmente removê-lo [Martin 2009]. Por exemplo, o código apresentado na Figura 1 contém trechos funcionais intercalados com trechos comentados.[1]

A adição de código comentado pode causar diversos problemas para mantenedores de software, tais como redução de legibilidade, distração e perda de tempo [Letouzey 2012, Letouzey and Coq 2010, Martin 2009]. De forma geral, essa

[1] Exemplo extraído de https://westminstercollege.edu

Figure 1. Exemplo de código “removido” através de comentário.

prática é muito discutida por profissionais[2], mas, no melhor do nosso conhecimento, ainda não foi devidamente investigada por pesquisadores.

Este artigo apresenta um estudo empírico para avaliar a presença de código comentado em sistemas *open source*. Especificamente, foca-se em responder duas questões de pesquisa: (QP1) Qual a taxa de código comentado atualmente? e (QP2) Como a taxa de código comentado evolui ao longo do tempo? Para responder as questões de pesquisa, analisa-se 300 *releases* fornecidas por 100 sistemas de software hospedados no GitHub. Logo, as principais contribuições desse trabalho são: (1) uma análise em larga escala sobre a presença da má prática de programação código comentado; (2) uma avaliação da ocorrência de código comentado ao longo do tempo; e (3) um conjunto de lições aprendidas através da análise de sistemas *open source*.

2. Por que Comentar Trechos de Código é uma Má Prática?

Diversos problemas podem surgir a partir da utilização de código comentado como forma de remoção. Em seu livro clássico e influente sobre técnicas para escrever código limpo, Clean Code [Martin 2009], Robert Martin sugere: “*Few practices are as odious as commenting-out code. Don’t do this!*”. O autor argumenta que outros mantenedores do código nunca terão a coragem para deletar o código comentado, decaindo a qualidade do software. A empresa de qualidade de software SonarQube[3] também aponta algumas desvantagens da inserção de código comentado:

1. Código comentado é uma distração e pode confundir o programador;
2. Código comentado sempre levanta mais questões do que respostas;
3. Programadores esquecerão rapidamente o quão relevante é o código comentado;
4. Código comentado é uma forma errada de controle de versão, uma vez que ferramentas como Git e SVN são indiscutivelmente a solução apropriada;
5. O simples fato do programador buscar entender a razão por trás do código comentado pode tomar muito tempo.

[2]Por exemplo: https://softwareengineering.stackexchange.com/questions/45378/is-commented-out-code-really-always-bad, http://www.markhneedham.com/blog/2009/01/17/the-danger-of-commenting-out-code, https://blog.sonarsource.com/commented-out-code-eradication-with-sonar

[3]https://www.sonarqube.org

Além desses problemas, é importante notar que o código comentado torna-se defasado ao longo do tempo uma vez que o sistema continua a evoluir normalmente enquanto o trecho de código comentado permanece estático. Por exemplo, código comentado pode referenciar classes e variáveis que foram deletadas ou invocar métodos já renomeados. Portanto, a simples remoção do comentário do código comentado não necessariamente garante o seu correto funcionamento nem sua compilação.

3. Metodologia

3.1. Seleção de Sistemas

Para responder as questões de pesquisa, analisa-se um conjunto de sistemas *open source* relevantes e populares armazenados no repositório GitHub e escritos na linguagem Java. Especificamente, selecionou-se os 100 primeiros projetos Java (ordenados pelo número de estrelas[4]) que atendem aos seguintes critérios: (i) possuir 3 ou mais releases e (ii) ser um sistema real. O primeiro critério foi adotado para realização de análise evolucionária. O segundo critério foi adotado pois o GitHub possui diversos projetos populares que não são sistemas reais, mas sim tutoriais, exemplos de códigos, guias, etc. Para tal, selecionou-se os projetos com página no GitHub em inglês ou português, e identificou-se manualmente se o projeto é um sistema real, como bibliotecas, frameworks, aplicativos móveis, etc.

A Figura 2 apresenta a distribuição do número de estrelas, releases, contribuidores e commits dos 100 sistemas selecionados. As medianas desses indicadores são: 7.450 estrelas, 31,5 releases, 57,5 contribuidores e 1.366 commits, comprovando a popularidade e relevância dos mesmos. Entre esses sistemas, destacam-se: ReactiveX/RxJava, elastic/elasticsearch, square/retrofit, square/okhttp e spring-projects/string-framework, todos com mais de 20.000 estrelas e pelo menos 40 releases.

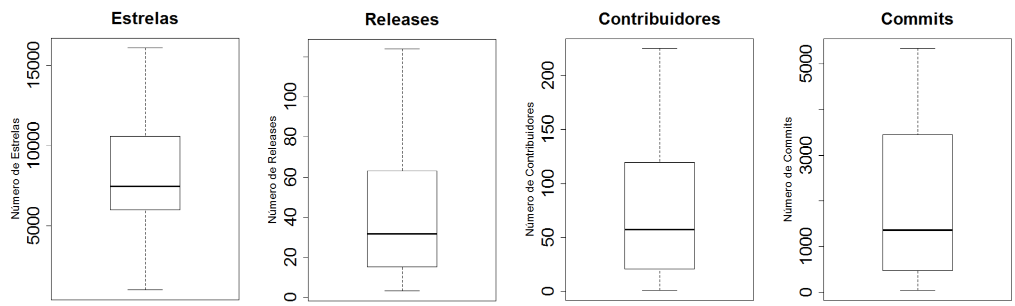


Figure 2. Caracterização dos sistemas analisados.

3.2. Extração de Código Comentado (Commented-out Code)

A abordagem proposta possui duas etapas principais: (1) extração de comentários e (2) detecção de código comentado:

1. Extração de comentários: primeiramente, extraiu-se os comentários dos arquivos Java presentes nos projetos selecionados;
2. Detecção de código comentado: em seguida, foram analisados os comentários para identificar ocorrências de código comentado.

[4]Métrica de popularidade nos projetos GitHub, similar ao *like*.

Extração de Comentários. Apesar de existirem algumas ferramentas capazes de extrair comentários em Java, tais como a Eclipse Java Development Tools (JDT) e outras similares que utilizam bibliotecas de expressões regulares (como a java.util.regex)[5], nenhuma ferramenta encontrada satisfez as nossas necessidades com relação a detecção de todos os tipos de comentários de forma eficiente. A JDT captura apenas um dos três tipos de comentários em Java (ie, Javadoc). Com outras ferramentas, a leitura de comentários suficientemente grandes resultava em um estouro no buffer, devido ao uso excessivo de recursões. Portanto, desenvolveu-se uma ferramenta com base na biblioteca de expressões regulares robusta *pcregrep*[6], para capturar os três tipos de comentários em Java:

- *Comentários multi-linha:* Comentários de uma ou mais linhas. Se inicia com `/*` e termina com `*/`. Exemplo: `/* this is a multi-line comment*/`
- *Comentários Javadoc:* Facilita a criação de documentação automática. Pode ser utilizado em uma ou mais linhas. Se inicia com `/**` e termina com `*/`. Exemplo: `/** this is a Javadoc comment */`
- *Comentários de linha única:* Comentários que terminam no final da linha e são iniciados com `//`. Exemplo: `//this is line comment`

Um desafio ao utilizar expressões regulares para encontrar padrões em códigos-fonte é que strings podem conter qualquer combinação de caracteres. Logo, é possível que uma string contenha um comentário dentro de seus delimitadores. Para evitar esses falsos positivos, tal cenário foi também tratado em nossa implementação.

Acurácia. A detecção de comentários depende da corretude da expressão regular utilizada e sua capacidade de capturar os tipos de comentários. Realizou-se centenas de testes com o auxílio da plataforma web regex 101[7], levando em consideração os mais variados cenários típicos e atípicos, em especial (i) os casos esperados encontrados em dezenas de sistemas reais e (ii) os casos excepcionais listados em *Finding Comments in Source Code Using Regular Expressions*.[8] Desse modo, em nossa avaliação, a ferramenta desenvolvida para detectar comentários Java obteve 100% de precisão e 100% de revocação.

Detecção de Código Comentado. Uma segunda ferramenta foi desenvolvida para a detecção de código comentado. Essa solução utiliza a biblioteca c2nes/javalang[9], que inclui um parser para códigos em Java capaz de tokenizar e reconhecer trechos de códigos bem formados. Nessa etapa, os conteúdos dos comentários extraídos no passo anterior são analisados. Durante a tokenização dos códigos dentro dos comentários, pode-se obter sucesso ou falha. As tokenizações que são bem sucedidas têm a possibilidade de serem códigos válidos, pois possuem uma sintaxe compatível com a linguagem Java (por exemplo: `//private int[] ranks;`). Os trechos que falham durante a tokenização não são códigos válidos (exemplo: `/* this is not a valid code */`).

Diversos casos particulares foram detectados e tratados para evitar falsos positivos. Abaixo são apresentados outros exemplos de códigos comentados detectados:

```
// throw new IllegalArgumentException();
// i = (args.length == 0) ? 1 : Integer.parseInt(args[0]);
```

[5]https://docs.oracle.com/javase/7/docs/api/java/util/regex/package-summary.html
[6]https://www.pcre.org/original/doc/html/pcregrep.html
[7]https://regex101.com
[8]https://blog.ostermiller.org/find-comment
[9]https://github.com/c2nes/javalang

Acurácia. Para analisar a precisão e a revocação da ferramenta desenvolvida, selecionou-se aleatoriamente 2 mil comentários dentre 20 dos 100 sistemas estudados (100 comentários por sistema). Em seguida, analisou-se manualmente esses comentários: 107 continham código comentado e 1.893 continham documentação convencional. Ao analisar essa base com a ferramenta proposta, foi detectado 98 *true positives* (ie, classificado como código comentado e correto), 19 *false positives* (ie, classificado como código comentado e incorreto) e 9 *false negatives* (ie, não classificado como código comentado e incorreto). Desse modo, pode-se computar acurácia da detecção de código comentado: Precisão = TP/(TP+FP) = 98/117 = 83,8%; Revocação = TP/(TP+FN) = 98/107 = 91,6%. Desse modo, a precisão acima de 83% e a revocação acima de 91% mostra que a ferramenta desenvolvida pode ser utilizada com bom nível de confiança.

4. Resultados

QP1 - Qual a taxa de código comentado?

A taxa de código comentado foi calculada a partir da análise das últimas releases dos 100 sistemas selecionados. A Figura 3 (esquerda) apresenta a distribuição do número total de comentários e de código comentado. Na mediana, detectou-se 1.316 comentários por sistema (1o quartil 413 e 3o quartil 5.890). Dentre esses, verificou-se, na mediana, 51 códigos comentados por sistema (1o quartil 14 e 3o quartil 236). A Figura 3 (direita) apresenta a taxa de código comentado por sistema (razão "código comentado" por "total de comentários"): na mediana, 4.17% dos comentários são compostos por códigos comentado (1o quartil 1.93% e 3o quartil 8.31%).

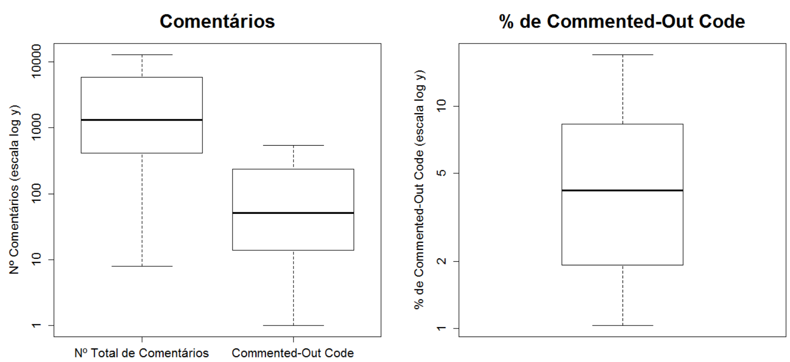


Figure 3. Taxa de código comentado.

Dentre os 100 sistemas, 9 não continham uma única ocorrência de código comentado. Por outro lado, alguns possuem uma alta taxa de código comentado; os 5 sistemas com a maior taxa são: cymcsg/UltimateRecyclerView (28,78%), navasmdc/MaterialDesignLibrary (23,89%), Blankj/AndroidUtilCode (19,93%), libgdx/libgdx (19,19%) e alibaba/fastjson (19,04%).

4.1. QP2 - Como a taxa de código comentado evolui ao longo do tempo?

Para analisar a taxa de código comentado ao longo do tempo, foram coletadas 3 releases de cada sistema: a inicial, uma intermediária e a final (totalizando 300 versões). Especificamente, as releases foram coletadas com suporte do comando `git tag` (que apresenta a lista de releases de um dado projeto), dos quais foram selecionadas a primeira (mais antiga), a intermediária (metade do total de releases) e a última (mais recente). Após a seleção das releases, o comando `git checkout` foi utilizado para baixar a versão correspondente, que, por sua vez, teve seus comentários extraídos e analisados.

A Figura 4 apresenta a distribuição do número total de comentários e de código comentado assim como a taxa de código comentado nas três releases analisadas. As medianas do total de comentários são: 451, 1.064 e 1.316, para as releases iniciais, intermediárias e finais, respectivamente. As medianas de códigos comentados são: 28, 47 e 51. Por fim, as medianas das taxas de códigos comentados são: 6,57%, 4,52% e 4,17%.

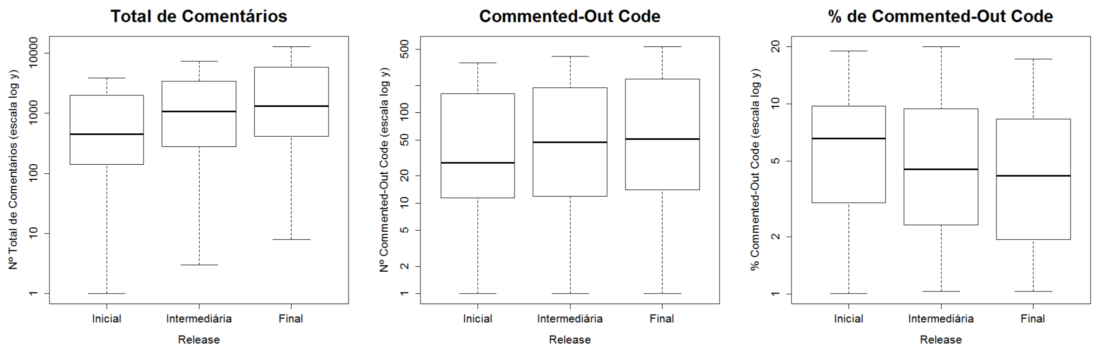


Figure 4. Evolução da taxa de código comentado.

Observa-se algumas tendências no decorrer dos projetos de software analisados. À medida que a quantidade total de comentários aumenta, a quantidade de código comentado code também aumenta; esse resultado é esperado, pois os sistemas tendem a crescer no decorrer do tempo. Entretanto, na mediana, a taxa de código comentado tende a diminuir ao longo do tempo. A mediana da taxa no início dos projetos é de 6,57%, decaindo para 4,17% em suas últimas versões.

Dentre os 100 sistemas analisados, 45 diminuíram suas taxas de código comentado, 46 aumentaram e 9 permaneceram estáveis. Os três sistemas com maior diminuição de código comentado são: iBotPeaches/Apktool (-33,71%), perwendel/spark (-27,95%) e SeleniumHQ/selenium (-24,08%). Por outro lado, os três sistemas com maior aumento de código comentado são: navasmdc/MaterialDesignLibrary (+14,88%), afollestad/material-dialogs (+13,35%) e wasabeef/recyclerview-animators (+13,33%).

5. Discussão

A taxa de código comentado é baixa, mas não deve ser ignorada. 91% dos sistemas analisados possuem código comentado. Portanto, somente 9% dos sistemas estão livres dessa má prática. Na mediana, a taxa de código comentado é de 4,17%. Porém, em alguns casos, essa porcentagem sobe para quase 30%. Logo, é crucial contar com apoio ferramental para identificar e eliminar essa prática. Entretanto, dentre as ferramentas de análise de qualidade do código mais populares do mercado, somente a SonarQube detecta trechos de código comentados. Outras ferramentas, como JArchitect, PMD e Checkstyle, não diferenciam comentários convencionais de código comentado. Desse modo, existe uma escassez de ferramentas para detectar essa má prática.

A taxa de código comentado tende a diminuir ao longo do tempo. De forma geral, a taxa de código comentado tende a diminuir ao longo do tempo, de 6,57% para 4,17%; um decréscimo de 36,53%. Isso indica que as equipes de desenvolvimento zelam pela manutenibilidade do código, particularmente evitando código comentado. Para manter qualidade, portanto, é fundamental que continuem vigilantes e sigam as melhores práticas. Entretanto, em equipes de porte médio ou grande, isso pode se mostrar uma tarefa de difícil gerenciamento e controle, que necessita de automação. Desse modo, sugere-se que a detecção de código comentado seja incluído em processos de integração contínua para que a verificação seja constante.

Em casos particulares, a taxa de código comentado pode aumentar. É importante notar que 45 dos 100 sistemas analisados aumentaram suas taxas de código comentado. Isso é preocupante, pois esses projetos são extremamente populares e relevantes, sendo usados por milhões de desenvolvedores. Além disso, esses sistemas utilizam o controle de versão Git. De fato, um das motivações para comentar código e mantê-los ao longo do tempo é justamente para "lembrar" o trecho de código caso ele seja necessário no futuro, evitando re-implementação [Spinellis 2005]. Portanto, é contraditório que os sistemas analisados incluam essa má prática, pois manter o histórico de estados anteriores é justamente a principal função do controle de versão. Comentar trechos de código pode ser uma forma rápida de realizar testes durante uma seção de programação, mas desenvolvedores não deveriam fazer commits contendo esses trechos.

6. Ameaças à Validade

Detecção de Código Comentado. Neste trabalho foi desenvolvida uma ferramenta para detectar código comentado. Para avaliar a qualidade dessa ferramenta, foi mensurada sua acurácia em detectar corretamente código comentado (ver Seção 3). Em uma avaliação com 2 mil comentários rotulados manualmente, obteve-se precisão de 83,8% e revocação de 91,6%. Desse modo, considera-se o risco dessa ameaça baixo.

Generalização dos Resultados. Este estudo limitou-se à análise de 100 bibliotecas *open source* implementadas em Java. Portanto, os resultados não podem ser generalizados para outras linguagens ou para sistemas comerciais.

7. Trabalhos Relacionados

Apesar das críticas da literatura contra comentar trechos de código [Martin 2009], nosso estudo revelou a presença dessa má prática. No melhor do nosso conhecimento, poucos artigos acadêmicos abordam esse tema, e grande parte de forma secundária ou superficial. Por exemplo, um estudo exploratório sobre estratégias de *backtracking* utilizada por desenvolvedores indica uma prevalência de trechos de código comentados ao invés da deleção [Yoon and Myers 2012]. Além disso, o estudo aponta motivações comuns que levam os desenvolvedores a optar por essa prática. O modelo de qualidade de software SQALE (*Software Quality Assessment Based on Lifecycle Expectations*), que se propõe a avaliar a qualidade de código-fonte, mostra que comentá-los apenas aumenta o esforço de leitura sem agregar qualquer valor ao software [Letouzey 2012, Letouzey and Coq 2010]. Apesar das ferramentas de análise estática terem um papel importante para garantir qualidade, atualmente, somente o SonarQube pode detectar trechos de código comentado [Campbell and Papapetrou 2013]. A pesquisa proposta também se relaciona com estudos sobre violações (*bad smells*) de código. Nesse contexto, diversas abordagens são propostas, por exemplo, para priorizar violações [Kim and Ernst 2007, Allier et al. 2012], avaliar a relevância dessas violações [Araujo et al. 2011] e detectar violações específicas por sistema [Hora et al. 2013, Hora et al. 2015]; tais abordagens, entretanto, não focam na análise de código comentado.

8. Conclusão

Este artigo apresentou um estudo empírico para quantificar a ocorrência de código comentado em 100 sistemas reais e 300 releases. Observou-se que, na mediana, 4,17% dos comentários são compostos por trechos de código e que essa taxa tende a diminuir durante a evolução dos projetos. Entretanto, em casos particulares, a taxa de código comentado é bastante significava, chegando a 30%. Como trabalhos futuros planeja-se (i) aumentar a quantidade de sistemas analisados para melhor caracterizar esse fenômeno, (ii) aumentar a acurácia da ferramenta e (iii) entender melhor quais trechos de código estão sendo comentados (eg, assinaturas, declarações, parâmetros, entre outros).

Agradecimentos: Esta pesquisa é financiada pela UFMS.

References

Allier, S., Anquetil, N., Hora, A., and Ducasse, S. (2012). A framework to compare alert ranking algorithms. In *Working Conference on Reverse Engineering (WCRE)*.

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).

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

Hora, A., Anquetil, N., Ducasse, S., and Valente, M. T. (2013). Mining system specific rules from change patterns. In *Working Conference on Reverse Engineering (WCRE)*.

Hora, A., Anquetil, N., Etien, A., Ducasse, S., and Valente, M. T. (2015). Automatic detection of system-specific conventions unknown to developers. *Journal of Systems and Software*, 109.

Kim, S. and Ernst, M. D. (2007). Which warnings should i fix first? In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

Letouzey, J.-L. (2012). The SQALE method for evaluating technical debt. In *International Workshop on Managing Technical Debt*.

Letouzey, J.-L. and Coq, T. (2010). The sqale analysis model: An analysis model compliant with the representation condition for assessing the quality of software source code. In *International Conference on Advances in System Testing and Validation Lifecycle*.

Martin, R. C. (2009). *Clean code: a handbook of agile software craftsmanship*. Pearson Education.

Spinellis, D. (2005). Version control systems. *IEEE Software*, 22(5).

Yoon, Y. and Myers, B. A. (2012). An exploratory study of backtracking strategies used by developers. In *International Workshop on Co-operative and Human Aspects of Software Engineering*.

Um Estudo Empírico sobre Critérios de Seleção de Repositórios GitHub

Laerte Xavier[1], Jailton Coelho[1], Luciana L. Silva[2]

[1]ASERG Group – Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – Minas Gerais – Brasil

[2]ASERG Group – Instituto Federal de Minas Gerais (IFMG)
Belo Horizonte – Minas Gerais – Brasil

{laertexavier,jailtoncoelho}@dcc.ufmg.br, luciana.lourdes.silva@ifmg.edu.br

Abstract. *Mining GitHub repositories is the first step of a plenty of Software Engineering studies. Based on them, researchers assess metrics, discuss behavior, investigate patterns. However, there is no silver bullet when deciding the criteria for selecting a relevant dataset. Recent studies have characterized researches based on GitHub, while others discuss the number of stars as a measure of popularity. In this study, we perform a systematic mapping of papers published in prestigious conferences in software engineering, with the purpose of investigating (i) the criteria applied to mine repositories; (ii) the characteristics of the selected dataset. To accomplish that, we manually analyze 140 papers published in 2017 at ICSE and FSE, both considered top conferences by CSIndex. As a result, we present a list of five main selection criteria:* POPULARITY, ACTIVITY, INDIVIDUAL CRITERIA, BUGS *and* AGE. *Besides, we show that, in the median, studies involving GitHub datasets analyze 25 repositories, in 5 main programming languages.*

Resumo. *Mineração de repositórios GitHub é o primeiro passo de diversos estudos em Engenharia de Software. Baseado neles, pesquisadores avaliam métricas, discutem comportamentos, investigam padrões. Entretanto, não existe nenhuma "bala de prata" que auxilie na decisão sobre quais critérios utilizar para seleção de um dataset relevante. Neste estudo, realiza-se uma revisão sistemática da literatura de artigos publicados em conferências de alto nível de Engenharia de Software, com o propósito de investigar (i) os critérios aplicados na mineração de repositórios; (ii) as características dos datasets selecionados. Para tanto, são analisados, manualmente, 140 artigos publicados nas conferências ICSE e FSE no ano de 2017, ambas conferências consideradas de alto nível pelo CSIndex. Como resultado, é apresentada uma lista de cinco principais critérios de seleção:* POPULARIDADE, ATIVIDADE, CRITÉRIOS PRÓPRIOS, BUGS *e* IDADE. *Além disso, observou-se que, na mediana, trabalhos envolvendo datasets provenientes do GitHub analisam 25 repositórios, em 5 principais linguagens de programação.*

1. Introdução

Minerar repositórios do GitHub[1] representa, atualmente, um importante passo no desenvolvimento de diversos estudos na área de Engenharia de Software. A plataforma, cri-

[1]https://github.com/

ada com o objetivo de auxiliar no desenvolvimento e armazenamento descentralizado de código aberto, tornou-se também um importante ambiente social, onde criadores, mantenedores e contribuidores interagem tanto em atividades de desenvolvimento, quanto de manutenção e evolução de sistemas. Dentre as principais vantagens de utilizar o GitHub como fonte de dados para trabalhos científicos, destacam-se importantes características da própria plataforma, tais como a disponibilização, via API (*Application Programming Interface*), de dados de repositórios, histórico de mudanças e *report* de *bugs*.

Dessa forma, existem diversos trabalhos, em variados contextos, que sustentam suas conclusões em *datasets* minerados da plataforma. Tratam, por exemplo, do ciclo de vida de repositórios [Coelho et al. 2018], refatoração de código [Silva and Valente 2017], ou *breaking changes* de APIs [Brito et al. 2018]. Entretanto, não existe *bala de prata* a respeito de qual seja a melhor estratégia para obtenção de um *dataset* relevante: quantos sistemas devem ser analisados? Qual linguagem deve ser escolhida? Qual o melhor critério de seleção?

Nesse contexto, existem trabalhos que caracterizam a utilização das *features* disponíveis no GitHub como instrumento para o desenvolvimento de pesquisa científica [Penzenstadler et al. 2014]. No entanto, tais estudos não analisam aspectos relacionados à aplicação dessas *features* na definição de *datasets*. Assim, neste trabalho, foram analisados 140 artigos de Engenharia de Software, publicados nas duas conferências mais relevantes da área (ICSE e FSE), com o objetivo de (i) elicitar os critérios utilizados por esses estudos para minerar repositórios no GitHub; (ii) caracterizar os *datasets* obtidos. Especificamente, o objetivo deste trabalho é o de investigar empiricamente às seguintes questões de pesquisa:

- ***RQ#1.*** Como são selecionados os repositórios GitHub estudados por artigos de Engenharia de Software?
- ***RQ#2.*** Quais as principais características dos *datasets* selecionados?

Dessa forma, as principais contribuições deste estudo são (i) fornecer um panorama de trabalhos com *datasets* oriundos do GitHub, publicados em conferências de prestígio de Engenharia de Software; (ii) prover uma lista de *cinco* critérios utilizados para seleção dos *datasets* destes trabalhos: POPULARIDADE, ATIVIDADE, CRITÉRIOS PRÓPRIOS, BUGS e IDADE; e (iii) descrever as principais características destes *datasets*, em termos de número de repositórios estudados e linguagens de programação.

O restante deste artigo está organizado da seguinte forma: a Seção 2 apresenta a metodologia dos estudos realizados. Na Seção 3 são descritos os resultados obtidos. Na Seção 4 são discutidas as ameaças à validade. Por fim, as conclusões obtidas neste estudo são apresentadas na Seção 5.

2. Metodologia

Com o objetivo de responder às questões de pesquisa propostas, foi desenvolvida uma revisão sistemática da literatura de artigos relevantes da Engenharia de Software. Inicialmente, foram selecionadas conferências de prestígio na área (Seção 2.1). Em seguida, foram analisados, manualmente, os artigos aceitos nas trilhas principais dessas conferências (Seção 2.2), para posterior codificação dos mesmos, conforme descrito na Seção 2.3.

2.1. Seleção de Conferências

Como primeiro passo, foram selecionadas conferências relevantes na área de Engenharia de Software. Para tanto, utilizamos a classificação provida pelo CSINDEX [Valente and Paixao 2018], um sistema de indexação da produção científica brasileira em ciência da computação. Neste sistema, as conferências são classificadas em *top* e *near-to-top*, de acordo com o número de artigos submetidos e o h5-index da conferência. O h5-index é uma métrica autoral que busca medir a produtividade e o impacto (em termos de número citações) de publicações científicas. Dessa forma, de acordo com o CSINDEX, são consideradas *top*, as conferências tais que: *(i) o número de submissões seja maior que 180; e (ii) o h5-index da conferência seja maior que 40.*

A Tabela 1 apresenta as *cinco* conferências mais relevantes em Engenharia de Software, de acordo com o CSINDEX, ordenadas pelo critério de julgamento dos desenvolvedores da ferramenta. Dentre elas, duas conferências são classificadas como *top*: ICSE (*International Conference on Software Engineering*), com 415 submissões em 2017 e h5-index igual a 68; e FSE (*Symposium on the Foundations of Software Engineering*), com 295 submissões no mesmo ano e h5-index, 43. Dessa forma, foram selecionadas para estudo estas top-conferências, no ano de 2017 (período de análise da ferramenta).

Tabela 1. A cinco conferências mais relevantes em Engenharia de Software, de acordo com o CSINDEX. Em negrito aquelas classificadas como *top* pelo sistema

Conferência	#Submissões	#Aceitos	Taxa de Aceitação	h5-index
ICSE	**415**	**68**	**16,40%**	**68**
FSE	**295**	**72**	**24,40%**	**43**
ASE	314	65	20,70%	31
MSR	121	37	30,60%	39
ISSTA	118	31	26,30%	31

2.2. Filtragem de Artigos

Após a definição do primeiro critério de seleção, que resultou em 140 artigos aceitos nas trilhas principais (*Research Track*) do ICSE e do FSE em 2017, o segundo passo deste estudo foi a análise manual de cada um destes trabalhos. Dessa forma, com o objetivo de filtrar aqueles que utilizaram *datasets* provenientes do GitHub, foram analisados os *abstracts* e as seções de *study design* de cada um deles. Assim, identificaram-se 19 artigos (13,57% do total analisados) cujos *datasets* foram minerados da plataforma. Destes, 11 artigos foram publicados no FSE e 8 no ICSE, os quais foram todos utilizados nas análises deste trabalho. A Tabela 2 sumariza estes resultados.

Tabela 2. Filtragem de artigos cujos *datasets* foram minerados do GitHub

Conferência	#Aceitos	#GitHub
FSE	72	11 (15,27%)
ICSE	68	8 (11,76%)
Total	**140**	**19 (13,57%)**

2.3. Análise dos Artigos

Por fim, após a seleção dos 19 artigos cujos *datasets* foram minerados do GitHub, a análise de cada um destes estudos se deu de forma manual, com o objetivo de:

- Codificar temas relacionados ao critério utilizado na filtragem dos repositórios analisados (RQ#1);
- Identificar a *linguagem de programação* dos repositórios analisados (RQ#2);
- Identificar a *quantidade* de repositórios selecionados (i.e., o tamanho dos *datasets* estudados) (RQ#2).

3. Resultados

RQ#1. Como são selecionados os repositórios GitHub estudados por artigos de Engenharia de Software?

Com o objetivo de responder a esta questão de pesquisa, os *abstracts* e as seções de *study design* (ou equivalentes) foram analisadas em cada um dos 19 artigos selecionados nesta revisão. Para cada um deles foram atribuídos códigos, ou critérios, que permitissem o agrupamento desses estudos. Assim, *cinco* critérios de seleção principais foram elicitados: POPULARIDADE, ATIVIDADE, CRITÉRIOS PRÓPRIOS, BUGS e IDADE. A Tabela 3 detalha cada um deles, bem como apresenta a quantidade de artigos que os aplica.

Tabela 3. Critérios de seleção de repositórios no GitHub

Critério	Descrição	#Ocorrências
POPULARIDADE	Nº de estrelas ou *forks*	10
ATIVIDADE	Qtde. de *commits* e *pushs*	3
CRITÉRIOS PRÓPRIOS	Particularidades do estudo	3
BUGS	Qtde. de *bugs* reportados	2
IDADE	Data de criação	1

POPULARIDADE. O critério de seleção mais recorrente entre os estudos de Engenharia de Software está relacionado à quantidade de pessoas que interagem com os repositórios. Dessa forma, os autores consideram relevante selecionar para o *dataset* dos seus estudos repositórios que possuem, de alguma forma, impacto dentro da comunidade do GitHub. Neste critério, foram observados três abordagens diferentes: seleção por número de estrelas (5 artigos), por notoriedade na literatura (3 artigos) e por número de *forks* (2 artigos).

Conforme discutido por [Borges et al. 2016], o número de estrelas é de fato utilizado como critério de seleção de repositórios em estudos de Engenharia de Software [Coelho and Valente 2017, Floyd et al. 2017, Long et al. 2017, Bocić and Bultan 2017, Xiong et al. 2017]. Nestes casos, os autores estão interessados em selecionar repositórios que possuem maior reconhecimento dentro da própria plataforma do GitHub, uma vez que as estrelas funcionam como ferramenta de qualificação. Outra abordagem recorrente é o de selecionar repositórios famosos na literatura e hospedados na plataforma (e.g., JUNIT-TEAM/JUNIT4 e FACEBOOK/REACT) [Ma et al. 2017, Khatchadourian and Masuhara 2017, Padhye and Sen 2017]. Nestes casos, os repositórios funcionam como *subjects* de experimentos voltados para avaliação de ferramentas ou novas abordagens propostas pelos autores. Por fim, o número de *forks* também é utilizado como critério de seleção de repositórios populares [Brown et al. 2017, Hellendoorn and Devanbu 2017], uma vez que mensuram a quantidade de desenvolvedores trabalhando paralelamente em diferentes instâncias daquele código.

ATIVIDADE. O segundo critério mais recorrente de seleção de repositórios refere-se às atividades registradas no repositório [Labuschagne et al. 2017, Garbervetsky et al. 2017, Wittern et al. 2017]. Neste caso, são utilizados como filtro métricas tais como *número de pushs*, *número de commits* e, num caso bastante peculiar, *número de requisições na rede*. Nestes *datasets*, é relevante observar que os estudos possuem o foco em analisar o comportamento dos desenvolvedores de sistemas *open-source*, bem como as interações que acontecem entre eles. Isso requer que os repositórios sejam ativos para fornecer dados relevantes. Assim, não são raras as definições de limiares que sirvam como base para essas filtragens e que delimitem as escolhas dos repositórios em análise (e.g., pelo menos um *commit* no último mês). Por fim, percebe-se também que este critério é comumente associado a outros, como o de POPULARIDADE, por exemplo, provendo um refinamento maior aos *datasets* estudados [Coelho and Valente 2017].

CRITÉRIOS PRÓPRIOS. Em três dos artigos analisados foi observada a adoção de critérios bastante específicos, relacionados à natureza dos experimentos desenvolvidos. Por exemplo, em [Abdalkareem et al. 2017] o *dataset* selecionado respeitou o seguinte critério definido pelos autores:

"We choose non-forked applications with more than 100 commits and more than 2 developers."

Neste caso, além de um critério de ATIVIDADE, relacionado à quantidade de *commits*, os autores decidiram filtrar os repositórios com pelo menos 2 desenvolvedores. O objetivo é obter uma base de contribuidores relevantes para o survey desenvolvido no estudo. Decisões particulares como esta são encontradas também em [Sadeghi et al. 2017, Linares-Vásquez et al. 2017].

RQ#2. Quais as principais características dos *datasets* selecionados?

Nesta questão de pesquisa são analisadas as principais características dos *datasets* encontrados nos 19 artigos revisados. Para tanto, durante a fase de análise (Seção 2.3), observou-se a linguagem de programação dos repositórios selecionados, bem como a quantidade de sistemas que reunem. Destaca-se que tais características são as mais relevantes neste contexto, uma vez que são frequentemente utilizadas para caracterização de *datasets* [Penzenstadler et al. 2014].

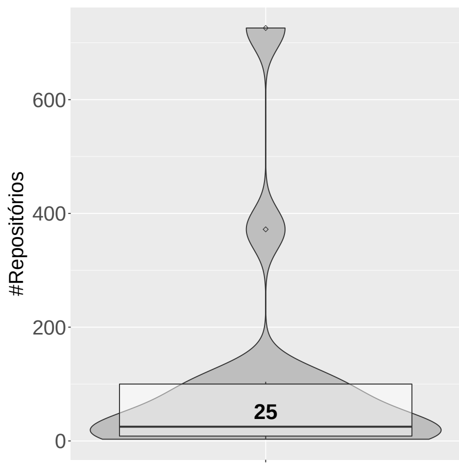


Figura 1. Quantidade de repositórios analisados por artigo

A Figura 1 apresenta a distribuição da quantidade de repositórios selecionados por artigo. Observa-se que o primeiro quartil é igual a 8.5; a mediana, 25; e o terceiro quartil, 100 repositórios. Nesse contexto, destacam-se como *outliers* estudos em larga escala, tais como [Linares-Vásquez et al. 2017, Long et al. 2017], que avaliam 726 e 372 repositórios.

A Tabela 4 descreve as linguagens de programação dos datasets avaliados, a quantidade de artigos que os estudam e o total de repositórios que reunem. Nesse caso, observa-se que JAVA é a linguagem estudada no maior número de artigos, oito no total, enquanto que JAVASCRIPT é a linguagem com maior número de repositórios avaliados (mais de 38K). Essa discrepância se dá pela existência de um *outlier* entre os *datasets* estudados: em [Abdalkareem et al. 2017] foram avaliados 38.807 repositórios JAVASCRIPT, com o objetivo de entender as motivações dos desenvolvedores de sistemas nesta linguagem utilizarem pacotes triviais da mesma.

Tabela 4. Principais linguagens selecionadas nos *datasets*

Linguagem	#Artigos	#Repositórios
JAVA	8	14.567
JAVASCRIPT	5	38.807
C	2	150
PYTHON	1	7
RAILS	1	25
TODAS	2	107

Além disso, observa-se que os estudos que avaliam repositórios utilizando o critério de POPULARIDADE, filtrados pela notoriedade da literatura [Ma et al. 2017, Khatchadourian and Masuhara 2017, Padhye and Sen 2017], possuem os menores *datasets* observados, com 5, 5 e 7 repositórios, respectivamente. Isto pode ser explicado pelo caráter subjetivo da seleção, umas vez que envolvem conhecimento da literatura e não são minerados automaticamente via API. Tais datasets reúnem repositórios de linguagens tais como JAVA e PYTHON, respectivamente.

Por fim, observa-se que em dois casos [Coelho and Valente 2017, Garbervetsky et al. 2017] não houve pre-definição da linguagem de programação dos repositórios minerados. Nestes casos, os *datasets* foram filtrados utilizando critérios convenientes (POPULARIDADE e ATIVIDADE, respectivamente), sem estabelecer a de linguagem de programação de interesse. Entretanto, a análise mais aprofundada destes artigos revela que, apesar de terem minerado repositórios sem pre-definição das suas linguagens de programação, seus *datasets* reunem, majoritariamente, sistemas escritos em JAVA, JAVASCRIPT e OBJECTIVE-C.

4. Ameaças à Validade

Validade de Conclusão. As conclusões apresentadas neste estudo são ameaçadas pelo tamanho da amostra de artigos selecionados. É importante destacar que uma amostra de apenas 19 artigos pode não ser representativa o suficiente para que se estabeleça uma relação de causa e efeito verdadeira. Entretanto, com o objetivo de diminuir esta ameaça, foram selecionados artigos de alto impacto, publicadas em conferências de prestígios, das mais diversas áreas da Engenharia de Software.

Validade Externa. A ameaça à validade externa diz respeito à capacidade de generalização dos resultados apresentados neste estudo. Como podemos observar, as conclusões obtidas neste trabalho dizem respeito à revisão de um conjunto de 19 artigos, publicados em apenas duas conferências de Engenharia de Software. Dessa forma, a lista de critérios apresentadas não pode ser generalizada para todos os trabalhos da área. Entretanto, para mitigar esta ameaça foram escolhidas conferências com publicações de alto nível, com alta taxa de submissões e alto número de citações.

Validade Interna. Além de ter seus resultados restritos a um pequeno conjunto de artigos, podemos destacar como ameaça à validade interna deste trabalho a avaliação manual realizada sobre os estudos revisados. Apesar de todo rigor de protocolo e execução desprendidos na fase de avaliação, destaca-se que ela foi executada por apenas um autor deste artigo, e sua natureza subjetiva pode representar um risco para as conclusões apresentadas. Entretanto, os resultados obtidos foram confrontados com outros trabalhos da literatura [Penzenstadler et al. 2014, Borges et al. 2016], destacando a conformidade dos critérios apresentados.

5. Conclusão

Neste trabalho foram analisados 19 artigos publicados nas conferências ICSE e FSE no ano de 2017, com a finalidade de investigar em detalhes como se dá o processo de seleção de repositórios do GitHub para estudos em Engenharia de Software. Dessa forma, cinco principais critérios de seleção foram elicitados: POPULARIDADE, ATIVIDADE, CRITÉRIOS PRÓPRIOS, BUGS e IDADE. Além disso, observou-se que os *datasets* selecionados possuem, na mediana, 25 repositórios, escritos nas linguagens JAVA, JAVASCRIPT e C, principalmente. Por fim, observou-se também que a linguagem com maior número de repositórios analisados é JAVASCRIPT.

Referências

Abdalkareem, R., Nourry, O., Wehaibi, S., Mujahid, S., and Shihab, E. (2017). Why do developers use trivial packages? an empirical case study on npm. In *11th Joint Meeting on Foundations of Software Engineering (FSE)*, pages 385–395.

Bocić, I. and Bultan, T. (2017). Symbolic model extraction for web application verification. In *39th International Conference on Software Engineering (ICSE)*, pages 724–734.

Borges, H., Hora, A., and Valente, M. T. (2016). Understanding the factors that impact the popularity of GitHub repositories. In *32nd IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 334–344.

Brito, A., Xavier, L., Hora, A., and Valente, M. T. (2018). Why and how Java developers break APIs. In *25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 255–265.

Brown, D. B., Vaughn, M., Liblit, B., and Reps, T. (2017). The care and feeding of wild-caught mutants. In *11th Joint Meeting on Foundations of Software Engineering (FSE)*, pages 511–522.

Coelho, J. and Valente, M. T. (2017). Why modern open source projects fail. In *11th Joint Meeting on Foundations of Software Engineering (FSE)*, pages 186–196.

Coelho, J., Valente, M. T., Silva, L. L., and Hora, A. (2018). Why we engage in FLOSS: Answers from core developers. In *11th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 1–8.

Floyd, B., Santander, T., and Weimer, W. (2017). Decoding the representation of code in the brain: An fmri study of code review and expertise. In *39th International Conference on Software Engineering (ICSE)*, pages 175–186.

Garbervetsky, D., Zoppi, E., and Livshits, B. (2017). Toward full elasticity in distributed static analysis: The case of callgraph analysis. In *11th Joint Meeting on Foundations of Software Engineering (FSE)*, pages 442–453.

Hellendoorn, V. J. and Devanbu, P. (2017). Are deep neural networks the best choice for modeling source code? In *11th Joint Meeting on Foundations of Software Engineering (FSE)*, pages 763–773.

Khatchadourian, R. and Masuhara, H. (2017). Automated refactoring of legacy Java software to default methods. In *39th International Conference on Software Engineering (ICSE)*, pages 82–93.

Labuschagne, A., Inozemtseva, L., and Holmes, R. (2017). Measuring the cost of regression testing in practice: A study of Java projects using continuous integration. In *11th Joint Meeting on Foundations of Software Engineering (FSE)*, pages 821–830.

Linares-Vásquez, M., Bavota, G., Tufano, M., Moran, K., Penta, M. D., Vendome, C., Bernal-Cárdenas, C., and Poshyvanyk, D. (2017). Enabling mutation testing for Android apps. In *11th Joint Meeting on Foundations of Software Engineering (FSE)*, pages 233–244.

Long, F., Amidon, P., and Rinard, M. (2017). Automatic inference of code transforms for patch generation. In *11th Joint Meeting on Foundations of Software Engineering (FSE)*, pages 727–739.

Ma, W., Chen, L., Zhang, X., Zhou, Y., and Xu, B. (2017). How do developers fix cross-project correlated bugs?: A case study on the GitHub scientific python ecosystem. In *39th International Conference on Software Engineering (ICSE)*, pages 381–392.

Padhye, R. and Sen, K. (2017). Travioli: A dynamic analysis for detecting data-structure traversals. In *39th International Conference on Software Engineering (ICSE)*, pages 473–483.

Penzenstadler, B., Raturi, A., Richardson, D., Calero, C., Femmer, H., and Franch, X. (2014). Systematic mapping study on software engineering for sustainability (se4s). In *18th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 1–14.

Sadeghi, A., Jabbarvand, R., and Malek, S. (2017). Patdroid: Permission-aware gui testing of Android. In *11th Joint Meeting on Foundations of Software Engineering (FSE)*, pages 220–232.

Silva, D. and Valente, M. T. (2017). RefDiff: Detecting refactorings in version histories. In *14th International Conference on Mining Software Repositories (MSR)*, pages 1–11.

Valente, M. T. and Paixao, K. (2018). CSIndexbr: Exploring the Brazilian scientific production in Computer Science. *arXiv*, abs/1807.09266.

Wittern, E., Ying, A. T. T., Zheng, Y., Dolby, J., and Laredo, J. A. (2017). Statically checking web API requests in JavaScript. In *39th International Conference on Software Engineering (ICSE)*, pages 244–254.

Xiong, Y., Wang, J., Yan, R., Zhang, J., Han, S., Huang, G., and Zhang, L. (2017). Precise condition synthesis for program repair. In *39th International Conference on Software Engineering (ICSE)*, pages 416–426.

GitHub REST API vs GHTorrent vs GitHub Archive: A Comparative Study

Thaís Mombach[1]**,Marco Tulio Valente**[1]

[1]Department of Computer Science – UFMG
Belo Horizonte – MG – Brazil

{thaismombach,mtov}@dcc.ufmg.br

Abstract. *GitHub is a popular platform for source code storage. This fact along with the API provided by the system contributes to make GitHub widely used on Software Engineering research. This paper presents and compares three systems that provide GitHub data: GitHub REST API, GitHub Archive and GHTorrent. For that, we rely on three queries where each system is used to collect the desired data. The goal of this paper is to provide a basic understanding about each system, their differences and in which case each one should be used. Our ultimate goal is to help researchers when choosing the best system to collect data required to the their research.*

1. Introduction

Nowadays, GitHub is the most popular platform for source code storage and sharing, with more than 87 million repositories and 30 million users.[1] GitHub is used by important Open Source Software (OSS) to store and share their code, and also to organize the contributions from developers around the world. This fact along with the friendly API that GitHub provides contribute to the platform being commonly used on Software Engineering research [Cosentino et al. 2017]. Particularly, GitHub provides an official REST API that allows access to information about projects through HTTP requests. Due to the large interest on GitHub for research, other platforms and datasets that collect data from the platform were create, such as GitHub Archive[2] and GHTorrent [Gousios and Spinellis 2012]. Essentially, these systems collect data from GitHub and make them available for further analysis.

GitHub REST API[3] is the official GitHub API that returns results in JSON format. This API provides data about repositories, users, issues, pull requests, among others. GitHub Archive[2] is a project created by Ilya Grigorik to collect GitHub data, so users can easily retrieve and use this data on their work. The system collects GitHub events and make them available in hourly archives that can be downloaded by HTTP client. Although GitHub Archive does not provide direct data about repositories and users, they can be obtained through Git events. The last project is GHTorrent [Gousios and Spinellis 2012] created by Georgios Gousios. This project provides an off-line mirror of GitHub data, which is retrieved by collecting events. The data is available in structured and unstructured formats. In the first case, the data is organized in tables on MySQL; in the latest one, the raw events data is stored in MongoDB.

[1]https://github.com/search
[2]https://www.gharchive.org/
[3]https://developer.github.com/v3/

The main purpose of this work is to compare these three projects in order to provide a basic understanding on how they work, when they should be used and what are the main differences among them. This paper is organized in six remaining sections. Section 2 includes a more detailed presentation of the three projects studied in this paper. Section 3 shows examples of queries that can be performed using these projects. In Section 4, we provide a comparative analysis of the three studied projects. Section 5 presents threat to validity. Section 6 presents related work, and Section 7 concludes the study.

2. Studied Systems

GitHub Rest API and GraphQL APIs provides access to data about hosted projects. However, this paper focus on the REST API, due to its popularity.

In order to retrieve data using the REST API, the requests should be made to *https://api.github.com/* specifying the desired data. For example, we can use *https://api.github.com/repo/owner/repo-name* to retrieve data about a repository called *owner/repo-name*. In fact, there is a large amount of data that can be retrieved using GitHub REST API, but the number of requests that can be performed is limited. This limit depends on the authentication provided by the user, including user and password, token, or key/secret. Currently, the rate limit for authenticated and unauthenticated requests is 5,000 and 60 requests per hour, respectively. However, under the REST API there is also a Search API with a different rate limit, 30 requests per minute for authenticated requests and 10 for unauthenticated.

GitHub Archive is a project created by Ilya Gregorik to store public events from GitHub, since February, 2011. Currently, the system extracts data from GitHub Events API, which provides more than 20 different types of events. The extracted data is available in hourly archives that can be downloaded using an HTTP client. Another option is using Google Big Query, where the data is stored in tables organized by year, month, and day. These tables have fields with basic information about the repository, the actor of the event, and a JSON object for the payload. Using Google Big Query, the data can be easily processed, since there is no need to download it; however, it is only possible to process 1 TB of data for free. In this paper, all examples are based on Google Big Query.

GHTorrent is a project created and maintained by Georgios Gousios. It provides an off-line mirror of GitHub data, which is available in two formats: structured data and raw data about events. The events are collected using GitHub Events API, and complemented with further requests to GitHub REST API to provide a structured representation. GHTorrent provides three ways to access its data: MySQL or MongoDB dumps, web service provided by GHTorrrent when accessing MySQL or MongoDB latest dumps, or Google Big Query when accessing from MySQL. In this paper, we focus on GHTorrent using MySQL. In this configuration, the tables represent projects, users, organizations, commits, pull requests, issues and others. Moreover, GHTorrent also provides geolocation data (city, state and country) of GitHub users, based on the location provided on their profile.

3. Comparative Queries

In this section, we provide examples of concrete queries using the studied systems.

3.1. Query #1: Top-1,000 Projects Number of Stars

GitHub REST API: The following URL returns a list of repositories sorted by number of stars in a descendant order.[4]

```
https://api.github.com/search/repositories?q=stars:1..*&sort=stars&order=desc
&page=1&per_page=100&access_token=xxxx
```

This URL has the following parts: (a) *https://api.github.com/* identifies requests to GitHub REST API; (b) *search* directs the request to the search API; (c) *repositories* defines that the search is for repositories; (d) *q=stars:1..** filters repositories according to the number of stars specified on a range; (e) *&sort=stars* defines by which field the result is sorted: stars, forks or update; (f) *&order=desc* specifies whether the result is in descendant or ascendant order; (g) *&page=1&per_page=100* specifies the desired page and the number of items per page (100 items, in this example).

GitHub Archive: Using this system, the following query returns a list of repositories sorted by number of stars in a descendant order.

```
1 SELECT repo.name, count(DISTINCT actor.id) AS num_stars
2 FROM (TABLE_QUERY([githubarchive:month], 'REGEXP_MATCH(table_id
    , r"^201\d0\d")'))
3 WHERE type = 'WatchEvent'
4 GROUP BY repo.name
5 ORDER BY num_stars DESC
6 LIMIT 1000;
```

To understand this query, it is important to mention that GitHub Archive data is organized in tables by year, month, and by day; essentially, these tables store the events when they happened. There is no table for the current year, but there are tables for the current month and current day. In this way, the previous query relies on *month* tables (line 2) to search for *WatchEvents* (line 3), which are the events triggered when a user stars a repository. For each repository (line 4), the number of distinct authors of *WatchEvents* are counted to obtain the total number of stars (line 1). The final result is sorted in descendant order (line 5) and limited to 1,000 repositories.

GHTorrent: In order to obtain the top-1,000 projects by number of stars, *projects* and *watchers* tables are used to construct the following query.

[4] Stars are often used as proxy for the popularity of GitHub software [Borges et al. 2016b] and therefore frequently used when selecting systems for empirical studies in software engineering.

```sql
SELECT p.id, p.name, p.url, w.num_stars
FROM
(SELECT id, name, url FROM projects WHERE deleted = 0) p
INNER JOIN
(SELECT repo_id, COUNT(DISTINCT user_id) AS num_stars FROM
    watchers GROUP BY repo_id) w
ON p.id = w.repo_id
ORDER BY w.num_watchers DESC
LIMIT 1000;
```

As mentioned, this query uses two tables: *projects* (which stores information about GitHub repositories) and *watchers* (which store information about stars given by a user to a repository). On *watchers*, users and repositories are identified by an *id* (computed by GHTorrent) and each row represents a star. Therefore, to calculate the number of stars the repositories are grouped and the number of different users on the *watchers* table are counted. However, GHTorrent also stores information about deleted repositories; therefore the *projects* table is used to identify only active projects. The result of *watchers* and *projects* tables are combined, sorted in descendant order by number of stars and limited to the first 1,000 results.

3.2. Query #2: Number of Commits per Contributor of a Given Repository

GitHub REST API: This URL returns the number of commits per contributor of a repository called *owner/repo_name*.

```
http://api.github.com/repos/owner/repo_name/contributors?page=1&per_page=100
&access_token=xxxx
```

The previous URL has some parts that are different from the ones in Query #1, as follows: (1) *repos* directs the request to the repository endpoint of Github REST API; (2) *owner/repo_name* is the repository name, (3) *contributors* define that the desired data is about repository contributors. The answer is a list of contributors of *owner/repo_name* in descendant order by number of commits along with the total number of pages containing the answer.

GitHub Archive: The following query searches for *push events* triggered when a user pushes to *owner/repo* (line 3). For each contributor (line 4) the number of commits is computed adding the number of commits on each *PushEvent* (line 1).

```sql
1 SELECT actor.login, SUM(INTEGER(JSON_EXTRACT(payload, '$.size')
    )) AS num_commits
2 FROM (TABLE_QUERY([githubarchive:month], 'REGEXP_MATCH(table_id
    , r"^201\d0\d")'))
3 WHERE type = 'PushEvent' AND repo.name = 'owner/repo'
4 GROUP BY actor.login;
```

GHTorrent: In this system, the *projects* table provides the *id* of *owner/repo*. This *id* is used to filter commits using *project_commits* table (line 2) and *commits* table (line 3). For each commit author (line 4) his/her number of commits is calculated (line 1). Finally, in order to obtain the contributor's name (instead of its *id*), the result is combined with the *user* table.

```
1 SELECT c.author_id, count(c.id) AS num_commits
2 FROM (SELECT * FROM project_commits WHERE project_id = (SELECT
    id FROM projects WHERE url = 'http://api.github.com/xxxx/
    xxxx') pc
3 INNER JOIN commits c ON pc.commit_id = c.id
4 GROUP BY c.author_id;
```

3.3. Query #3: Languages of Top-1,000 Projects

GitHub REST API: Essentially, for each repository returned by Query #1, the following URL returns its languages:

```
http://api.github.com/repos/owner/repository/languages&access_token=xxxx
```

This URL also refers to the repository API as in Query #2, but it differs on the *language* component, which requests the different languages used in a repository.

GitHub Archive: Using this system, it is not possible to answer this query because there is no event that lists all languages used in a repository.

GHTorrent: In the following query, the result from Query #1 (line 2) is combined with *project_language* table that contains the languages used on each project (line 3). Then, for each repository, the query returns its distinct languages (line 1).

```
1 SELECT DISTINCT ptop.url, plang.language
2 FROM (SELECT p.id, p.url FROM (SELECT id FROM projects WHERE
    deleted = 0) p INNER JOIN (SELECT repo_id, COUNT(DISTINCT
    user_id) AS num_stars FROM watchers GROUP BY repo_id) w ON p
    .id = w.repo_id ORDER BY w.num_stars DESC LIMIT 1000) ptop
3 INNER JOIN project_languages plang ON plang.project_id = ptop.
    id;
```

4. Analysis

The first aspect that must be taken into account is since when the data is required in a query. When using the official GitHub REST API, the information available covers all GitHub history, but this does not happen when using GitHub Archive and GHTorrent. GitHub Archive collects events since 02/12/2011. Therefore, even though it has a large amount of data, it does not contains the whole history of events from GitHub. The same happens to GHTorrent, which provides information in its MySQL database since 2012.

	GitHub REST API V3	GitHub Archive	GHTorrent
Data Since	-	02/12/2011	2012
Data From	-	GitHub REST API Events	GitHub REST API
Data Type	GitHub Data	GitHub Events	Structured GitHub Data
Data Update	Live	Hourly	Monthly

Table 1. Analysis of GitHub REST API V3, GitHub Archive, and GHTorrent.

In other words, GitHub Archive and GHTorrent aims to store part of the data available on GitHub and to make it easier for researchers to access this data without a rate limit. GitHub Archive stores data from events that occurs on GitHub organized in hourly files accordingly to the time they happened. GHTorrent provides structured data as MySQL dumps, which are monthly released. Therefore, an advantage of using GitHub Archive or GHTorrent is that it is possible to analyze how the data has evolved, and also to replicate a study with the same data. By contrast, as a disadvantage, the data is not always up to date, like on GitHub REST API. All these mentioned aspects are summarized on Table 1.

Particularly, in Query #1 it is more accurate to calculate the current top-1,000 repositories by number of stars using GitHub REST API, because the information is up to date and takes into account the whole GitHub history. By contrast, the result might not be the same for GHTorrent and GitHub Archive, since the data considered by such systems does not contain all history and therefore might not reflect the current status of GitHub. In addition, the query's complexity is lower for GitHub REST API than for GHTorrent and GitHub Achive. The main reason is that usually GHTorrent and GitHub Archive tables have to be combined to calculate the result.

For Query #2, depending on the selected repository the result might not be affected by the fact that GHTorrent and GitHub Archive do not provide up to date data from GitHub. The complexity of using GitHub Archive derives from the need to process all *month* tables available, if there is not time frame specified for the query. For GHTorrent, it is the need to retrieve the commits from all GitHub repositories. As a result, although the result might be almost the same for the three systems, using GHTorrent and GitHub Archive implies processing more data.

Regarding Query #3, we showed that it is not possible to use GitHub Archive to collect this information since there is no event that provides the required data (i.e., all programming languages used by a given repository). Using GitHub REST API, this query is expensive because for each project on top-1,000 list an additional query needs to be performed to retrieve all the languages used on such projects, on a total of 1,010 requests (10 to retrieve the top-1,000 list, assuming pages with the result of 100 project, plus 1,000 requests to obtain the languages of each project). Finally, when using GHTorrent, besides accessing the tables to calculate the top-1,000 projects, it is also necessary to access the table where the language for each project is stored.

5. Threats to Validity

In this paper, we compared three systems that can be used to retrieve data from GitHub. To the best of our knowledge, these are the principal systems to this purpose. However, we acknowledge that GitHub provides a new API, based on GraphQL technology, which can assume a key role in the future. Furthermore, we assessed the systems using only three queries. However, these queries were carefully selected based on our experience with the systems. Although limited in size, we claim these queries illustrate the pros and cons of using each system. Finally, our evaluation is qualitative by nature. In future work, we plan to conduct a quantitative evaluation of these systems, comparing for example runtime performance and, more importantly, precision and recall of the items returned by each studied system.

6. Related Studies

GitHub data has been widely used on Software Engineering Research; as a result, it is important to investigate the reliability of this data and the potential threats that might appear when mining GitHub. [Kalliamvakou et al. 2014] investigated this potential problems and proposed a set of recommendations to avoid the threats. [Cosentino et al. 2016] studied how researchers are mining GitHub, analyzing the empirical methods employed, used datasets and their limitations; regarding the datasets, the authors paper pointed out that GHTorrent is the most used dataset; however, GitHub API is also widely used.

A similar work to this paper presents a tutorial about how to analyze data from GitHub in order to avoid misleading results using GHTorrent as data source [Gousios and Spinellis 2017]. Another work investigates how GHTorrent dataset is used on researches and how much data is processed [Borges et al. 2016a]. However, to the best of our knowledge this is the first work that presents and compares the different sources of GitHub data.

7. Conclusion

The objective of this study was to provide a brief explanation and use scenarios for three solutions that can be used to mining GitHub data: GitHub REST API, GitHub Archive and GHTorrent. For each system was presented an explanation about their behavior, and available data. GitHub provides a REST API that provides through HTTP requests GitHub data as JSON objects. GitHub Archive collects GitHub events since 2011 and make them available in hourly files that can be downloaded for further analysis. The last one, GHTorrent, also access GitHub events to create dumps of raw and structured data, for MongoDB and MySQL databases, respectively. All this information are relevant when choosing a data source for a research, and it can impact on its result.

It was also presented three different comparative queries that were constructed using the presented systems. For each query it was discussed the positive and negative points of using this systems. The main goal of this paper was to provide an analysis of this different solutions that can be used to mining GitHub data system in order to help researches when choosing the one that best suits their research.

Based on our findings, we derive two practical recommendations to researchers:

- GitHub REST API is more appropriate when it is important to retrieve the most recent and up to date data (e.g., about number of stars).
- GHTorrent and GitHub Archive is more appropriate when it is important to retrieve historical data about projects (e.g., time series with number of stars per month).

As future work, we plan to work in four fronts: (a) include a comparison with GitHub GraphQL API; (b) extend our set of comparative queries; (c) evaluate the runtime performance of each system; (d) evaluate the precision and recall of GHTorrent and GitHub Archive, using the results provided by the official REST API as ground truth.

Acknowledgments

Our research is supported by CAPES, FAPEMIG, and CNPq.

References

Borges, H., Coelho, J., Carvalho, P., Fernandes, M., and Valente, M. T. (2016a). Como pesquisadores usam o dataset GHTorrent? In *5th Brazilian Workshop on Software Visualization, Evolution and Maintenance (VEM)*, pages 1–8.

Borges, H., Hora, A., and Valente, M. T. (2016b). Understanding the factors that impact the popularity of GitHub repositories. In *32nd IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 334–344.

Cosentino, V., Izquierdo, J. L. C., and Cabot, J. (2016). Findings from GitHub: methods, datasets and limitations. In *13th Working Conference on Mining Software Repositories (MSR)*, pages 137–141.

Cosentino, V., Izquierdo, J. L. C., and Cabot, J. (2017). A systematic mapping study of software development with GitHub. *IEEE Access*, 5:7173–7192.

Gousios, G. and Spinellis, D. (2012). GHTorrent: Github's data from a firehose. In *9th Working Conference on Mining Software Repositories (MSR)*, pages 12–21.

Gousios, G. and Spinellis, D. (2017). Mining software engineering data from GitHub. In *39th International Conference on Software Engineering Companion (ICSE)*, pages 501–502.

Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M., and Damian, D. (2014). The promises and perils of mining GitHub. In *11th Working Conference on Mining Software Repositories (MSR)*, pages 92–101.

Microservices in Practice: A Survey Study

Markos Viggiato[1], Ricardo Terra[2], Henrique Rocha[3], Marco Tulio Valente[1], Eduardo Figueiredo[1]

[1]Dept. of Computer Science, Federal University of Minas Gerais (UFMG)
Belo Horizonte – MG – Brazil

[2]Dept. of Computer Science, Federal University of Lavras (UFLA)
Lavras – MG – Brazil

[3]Inria Lille - Nord Europe
Villeneuve D'ascq, France

{markosviggiato, mtov, figueiredo}@dcc.ufmg.br, terra@dcc.ufla.br, henrique.rocha@inria.fr

Abstract. *Microservices architectures have become largely popular in the last years. However, we still lack empirical evidence about the use of microservices and the practices followed by practitioners. Thereupon, in this paper, we report the results of a survey with 122 professionals who work with microservices. We report how the industry is using this architectural style and whether the perception of practitioners regarding the advantages and challenges of microservices is according to the literature.*

1. Introduction

Microservices have become largely popular in the last years together with the spread of DevOps practices and containers technologies, such as Kubernetes and Docker [Pahl 2015]. We can see a significant increase in the use of microservices architectural style since 2014 [Klock et al. 2017], which can be verified in the service-oriented software industry where the usage of microservices has been far superior when compared to other software architecture models [Alshuqayran et al. 2016].

Microservices are autonomous components that isolate fine-grained business capabilities. Furthermore, a microservice usually runs on its own process and communicates using standardized interfaces and lightweight protocols [Fowler and Lewis 2017, Hassan et al. 2017]. In practice, microservices are widely used by large Web companies, such as Netflix, LinkedIn, and Amazon, which can be motivated by the benefits that microservices bring, e.g., the reduced time to put a new feature in operation [Alshuqayran et al. 2016].

There are many benefits of using microservices, such as technology diversity in a single system, better scalability, increase productivity, and ease of deployment [Alshuqayran et al. 2016, Newman 2015]. Consequently, these benefits may improve software maintainability [Alshuqayran et al. 2016]. However, microservices also have their drawbacks. Usually, the services that compose the software are part of a distributed setting. Therefore, microservices could complicate some tasks such as finding a service within the network, managing the security, executing transactions, and optimizing the communication between services [Alshuqayran et al. 2016, Yu et al. 2016].

Shedding light on microservices usage in practice is important for many reasons. We can perceive the advantages that motivate practitioners and the most important challenges faced when developing software under this architecture. This information can support decision-making about migrating systems to microservices or even start to develop an entire application under this architectural style. It can also aid software developers to understand and follow the best practices, making the microservices usage more effective. In addition, it may support the adoption of practices in software domains by practitioners and the developers' perception of the software quality [Mori et al. 2018, Oliveira et al. 2018, Guimaraes et al. 2013].

Nevertheless, there are no studies that investigate how microservices are used in practice. To the best of our knowledge, existing works consist of systematic mapping studies, which summarize the progress of microservices technology so far [Alshuqayran et al. 2016, Pahl and Jamshidi 2016]. By contrast, in this paper, we propose to look at microservices from a practical perspective, i.e., with the aim of understanding and reveal how practitioners are in fact using microservices. More specifically, we describe the results of a survey designed to reveal the usage of microservices in practice. First, we conducted a mapping study to identify and highlight potential advantages and challenges faced by professionals who work with microservices. Next, we surveyed developers about the findings of the mapping study, aiming at verifying whether the use of microservices in the industry is according to the best recommendations mentioned in the literature and whether the advantages and challenges found in the mapping study are in fact what practitioners face.

2. Microservices

Microservices are an architectural style in which the process of software development is done by using autonomous components that isolate fine-grained business functionalities and communicate one with other through standardized interfaces [Hassan et al. 2017]. Due to an extensive use in web and cloud-based applications, we can observe a migration of some companies from the monolith architecture to microservices since the latter brings many benefits such as self-manageable (decentralized governance) and lightweight components [Aderaldo et al. 2017].

The purpose of microservices is to use autonomous units that are isolated one from another and coordinate them into a distributed infrastructure by a lightweight container technology, such as Docker. Usually, the adoption of this architectural model implies also in adopting agile practice, such as DevOps, which reduces the time between implementing a change in the system and transferring this change to the production environment [Aderaldo et al. 2017].

The isolation of business functionalities is highly recommended when using microservices, and allows independent development and deployment of each microservice. Moreover, the isolation also optimizes the autonomy and the replaceability of the services. Indeed, a microservice architectural style brings many benefits for developers but is also comes with many challenges. In Section 3.1, we present a detailed description of the advantages and challenges of working with microservices.

3. Study Design

This study included two phases, a mapping study (Section 3.1), and a survey (Section 3.2), as described next.

3.1. Mapping Study

Initially, we performed a mapping study to collect information about microservices from blogs and articles, as well as from more traditional literature, including books and papers. Mapping studies are particularly recommended for understanding emerging fields or technologies [Wohlin et al. 2012], which is certainly the case of microservices. In order to retrieve documents about microservices, we used three specific search strings on Google: *microservices architecture*, *good features of microservices architecture*, and *bad features/parts of microservices architecture*. Our intention was to gather documents regarding all aspects of microservices, such as documents proposing general terms and definitions (using the first string), and documents with the best characteristics and the main challenges faced by developers (using the second and third strings). After analyzing the retrieved documents, we identified five papers and one book from the scientific literature. Furthermore, we also considered 15 relevant documents from well-known experienced practitioners, including ten articles from websites and five documents from blogs. It is important to note that microservices have become popular in recent years, therefore there is still not a large number of relevant documents available.

The first author of this paper carefully read all the 21 relevant documents in order to extract their recurrent topics and themes. Thereupon, we classified the topics into advantages and challenges faced by developers when already using microservices architectures.

Advantages. In a system composed of multiple microservices, developers have the possibility of using many different technologies [Newman 2015]. This **technology diversity** is a very common characteristic in applications using microservices and it allows the use of the right tool for the right job. Moreover, the heterogeneity of technologies allows the addition of new technologies during development or maintenance in a more efficient way. For example, this is suitable for web applications where we can observe a constant and fast change in development environments and frameworks [Ramos et al. 2016]. Another benefit often associated to microservices is the possibility of deploying a given service independently from the others. This **independent deployment** could lead to a faster implementation of new features [Newman 2015]. **Scalability** is also mentioned as an advantage of microservices since it can be achieved on demand, scaling only the service that contains a given functionality. It is also possible to replicate specific services, instead of the entire system. Finally, **maintainability** is often reported as a benefit since developers can modify or replace a service without impacting the entire application.

Challenges. It is often reported that microservices demand distributed data management and hence **distributed transactions**, which makes their implementation much more complex. Other studies [Pahl and Jamshidi 2016, Aderaldo et al. 2017] report that automated tests are extremely important in microservices, especially **integration tests**, which can be more complex and time-consuming. In addition, when the tests fail, it can be harder to determine which functionality has been broken [Newman 2015]. **Service faults** are also cited as a challenge when using microservices since the identification of a fault in a distributed setting is much harder than in a monolithic one. Finally, developers often mention that **Remote Procedure Calls** (RPC) are expensive and take much longer than local calls, which means that RPC may become a challenge when developing applications under microservices.

3.2. Survey Design

We designed a survey aiming at confirming (or not) the general characteristics, advantages, and challenges associated with microservices, as indicated by our mapping study. The survey has 14 questions and it is divided into three sections. The first section is about the background of the participants, including questions about experience with software development and with microservices as well as about the size of applications/number of services that they already worked with. The second section is related to definitions and trade-offs of using microservices. For instance, it includes questions about the ideal size of a microservice, and advantages and problems faced by developers when using this technology. Finally, the third section is composed of open-ended questions about the technologies used by developers when implementing microservices applications. Our intention is to identify the most popular programming languages and technologies used under microservices architecture since the literature states that many technologies can be used in microservices systems.

To find participants, we implemented an algorithm to search for microservices developers in the Stack Overflow community. We identified and retrieved nicknames from users who own questions or answers containing tag *microservices*. In order to collect the email, we matched the Stack Overflow nickname with the equivalent nickname at GitHub. In addition, we promoted the survey in many communities about microservices and cloud-based development, including a Google group[1], a Google Plus community[2], and a Reddit community[3]. The survey remained open between June and July 2017, and we obtained 122 complete responses.

4. Survey Results

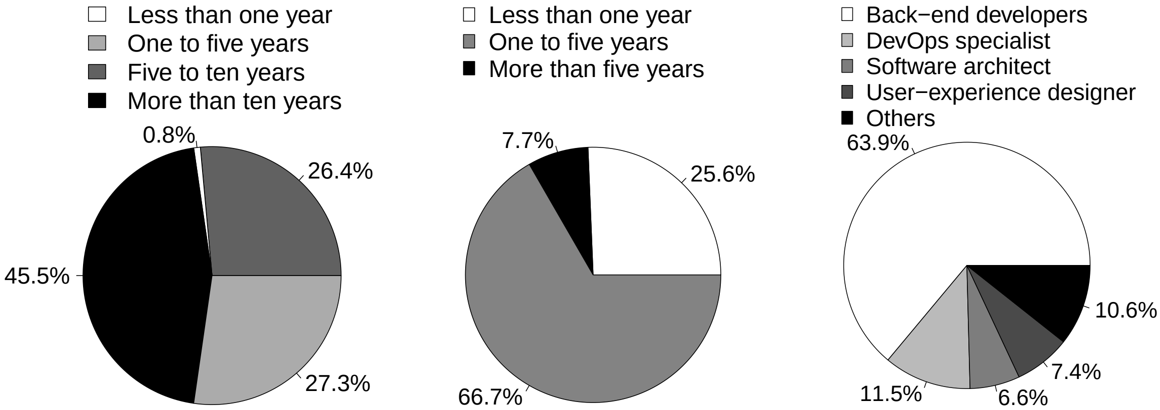
In this section, we describe the results obtained from the survey by presenting the participants background experience (Section 4.1), the popular languages and technologies (Section 4.2), the perceived advantages and challenges (Section 4.3), and the participants' feedback (Section 4.4).

4.1. Participants Background

As presented in Figure 1, almost 72% of the participants have at least five years of experience with software development. Furthermore, about 74% of them have more than one year of experience with microservices; more specifically, almost 67% have one to five years of experience. In addition, approximately 64% of the respondents are back-end developers while only 11.5% are DevOps.

Although not graphically illustrated, we also collected data regarding the size of the applications. 70.5% of the respondents worked with monolithic-based systems larger

[1] https://groups.google.com/forum/#!forum/microservices
[2] https://plus.google.com/communities/112442985624053749478
[3] https://www.reddit.com/r/microservices/



(a) Development experience (b) Microservices experience (c) Participant's role

Figure 1. Participants' background.

than 50 KLOC, and about 54% worked with microservices-based applications larger than 10 KLOC. These numbers confirm that most survey participants are not novice in the microservices field.

Considering that microservices are an emerging field, and based on the professional background of the respondents, we claim the participants have sufficient knowledge on the subject to answer the survey questions.

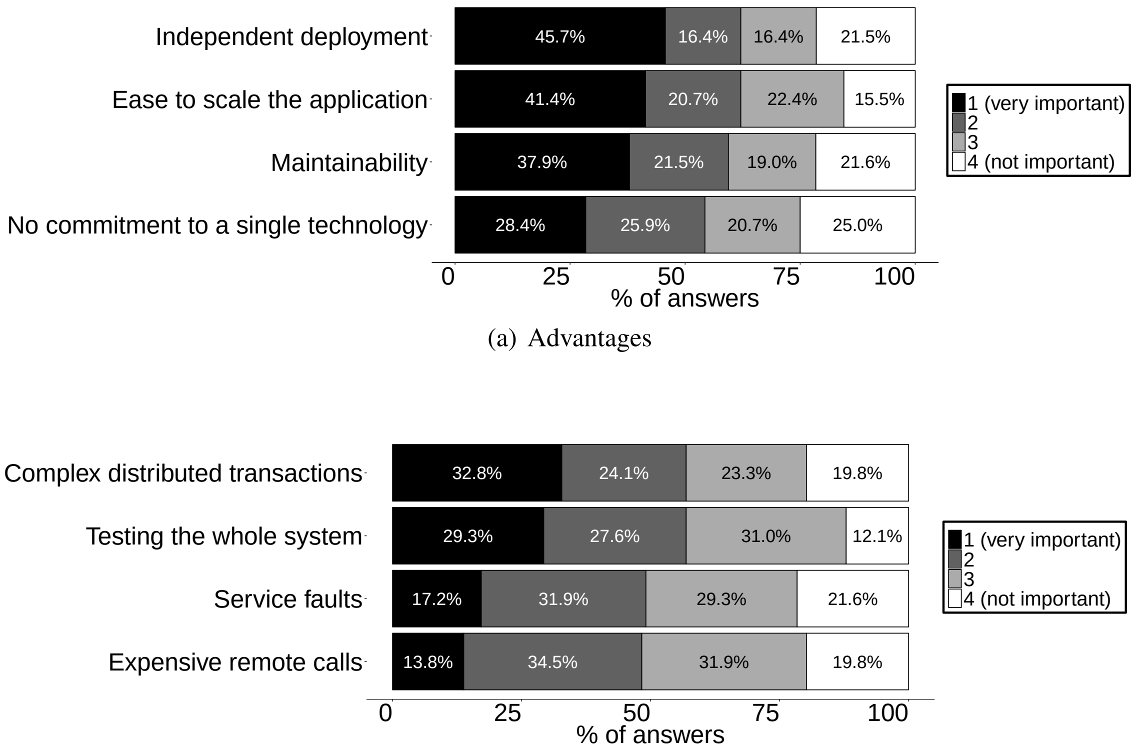
4.2. Most Popular Programming Languages and Technologies

Aiming at characterizing microservices applications, we asked the survey participants about the languages and technologies they usually use in their projects. We found that four programming languages are largely used: Java (33%), JavaScript through Node.js (18%), C# (12%), and PHP (8%). The answers also mention other 14 programming languages, which indicate the flexibility of microservices-based applications regarding programming languages. Regarding the most common DBMS, the results include Postgres (30%), MySQL (25%), MongoDB (20%), SQL Server (12%), and Oracle (9%). Finally, regarding the communication protocols, 62% of the participants declared they use REST over HTTP.

4.3. Advantages and Challenges of Microservices

Figure 2a presents the percentage of participants' answers about four advantages usually associated to microservices, in a scale from 1 (very important) to 4 (not important at all). For **independent deployment**, we can see the largest difference from score 1 to the others, which reveals a higher agreement rate for this feature when compared to the others. Yet, for the other three advantages, we can verify that more than 50% of the respondents chose scores 1 or 2, indicating these characteristics are in fact relevant when working with microservices.

From the mapping study, we also identified four main challenges faced by developers when working with microservices. In Figure 2b, we can see the responses of the survey's participants regarding these challenges. The participants agree that **complex**



(a) Advantages

(b) Challenges

Figure 2. Microservices advantages and challenges

distributed transactions are a very important challenge. In fact, this challenge has the highest percentage for score 1 (32.8%). We can also observe that about 57% and 49% consider **testing the whole system** and **service faults**, respectively, as important challenges (scores 1 and 2). In contrast, challenge **expensive remote calls** is very important to only 13.8% of the participants. Interestingly, practitioners disagree with the literature in this last challenge since they do not find expensive remote calls a very important challenge in microservices development. In a nutshell, developers should pay special attention to distributed transactions, testing of the whole system and service faults, since these may become big problems in the system.

4.4. Feedback from Participants

We also sent another email to the developers directly contacted to answer our survey. In this follow-up message, we described the major survey results with the aim of receiving their impressions about our study and findings. We also intended to verify whether they agree or not with our results. We received nine answers; all of them with a positive feedback. In general, developers answered that our paper provides a good overview of the microservice practice. For instance, two developers commented this is *really interesting* and *good paper*. Another developer highlighted that microservices are not a "holy grail" and that monolithic can also have small and separated modules, even in distributed settings.

5. Threats to Validity

Some threats may affect the validity of our findings. First, the survey participants may not represent the entire population of microservices practitioners. To mitigate this risk, we put efforts in promoting the survey in many different communities to include professionals from different software ecosystems. Second, the term *DevOps* (which is one of the options of the survey question about the participant's role) might not be common in some contexts. For example, in small organizations, DevOps tasks such as development and delivery process automation may fall on senior developers and architects. Third, although it would be desirable for our survey the analysis of larger (w.r.t. size), stable (w.r.t. age), and specific branches of industry applications, we argue that our survey brings a broad overview since it is based on the expertise of 122 developers who work with heterogeneous microservices-based applications.

6. Related Work

Most of the research in microservices restrict their study to a specific domain, such as business [Yu et al. 2016]. There are also researches investigating microservices by performing a systematic mapping study. For instance, a study summarized the progress of studies about microservices until 2016, and identified the gaps and requirements [Alshuqayran et al. 2016]. Other study taxonomically classified and compared studies of this architectural style and their application in the cloud [Pahl and Jamshidi 2016]. Our study follows a different route as we look at the microservices from a practical perspective. We aimed to understand and indicate how the software development industry is, in fact, using this popular architecture, and how practitioners perceive the advantages and challenges of microservices.

7. Conclusion

The findings of this paper indicate that practitioners usually follow the best practices for microservices reported in the literature. We also confirmed the benefits provided by microservices, such as **independent deployment**, **ease to scale the applications**, **maintainability**, and **no commitment to a single technology stack**. Last but not least, we also confirmed the challenges developers may face, such as **complex distributed transactions**, **testing the whole system**, and **service faults**.

However, we also found some important topics that are in disagreement. First, professionals usually work as back-end developers (64%), instead of as **DevOps** specialists in cross-functional teams. Second, according to the mapping study, **expensive remote calls** is one of the challenges that developers face when working with microservices. However, about 52% of the respondents declared that it is not an important or it is a little important issue in their systems.

As future work, we intend to conduct interviews with microservices professionals to confirm our results and to better understand if and why practitioners do not follow some best practices. We also plan to perform an industrial-scale case study with companies that adopt microservices to monitor real software developers developing microservices-based projects in order to report the problems they face and the solutions they apply.

Acknowledgements

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

References

[Aderaldo et al. 2017] Aderaldo, C. M., Mendonça, N. C., Pahl, C., and Jamshidi, P. (2017). Benchmark Requirements for Microservices Architecture Research. In *1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE)*, pages 8–13.

[Alshuqayran et al. 2016] Alshuqayran, N., Ali, N., and Evans, R. (2016). A Systematic Mapping Study in Microservice Architecture. In *9th International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 44–51.

[Fowler and Lewis 2017] Fowler, M. and Lewis, J. (March 25, 2014. [Online; accessed April, 2017]). *Microservices*. https://martinfowler.com/articles/microservices.html.

[Guimaraes et al. 2013] Guimaraes, E., Garcia, A., Figueiredo, E., and Cai, Y. (2013). Prioritizing software anomalies with software metrics and architecture blueprints: a controlled experiment. In *Proceedings of the 5th International Workshop on Modeling in Software Engineering*, pages 82–88. IEEE Press.

[Hassan et al. 2017] Hassan, S., Ali, N., and Bahsoon, R. (2017). Microservice ambients: An architectural meta-modelling approach for microservice granularity. In *IEEE International Conference on Software Architecture (ICSA)*, pages 1–10.

[Klock et al. 2017] Klock, S., Werf, J. M. E. M. V. D., Guelen, J. P., and Jansen, S. (2017). Workload-based clustering of coherent feature sets in microservice architectures. In *IEEE International Conference on Software Architecture (ICSA)*, pages 11–20.

[Mori et al. 2018] Mori, A., Vale, G., Viggiato, M., Oliveira, J., Figueiredo, E., Cirilo, E., Jamshidi, P., and Kastner, C. (2018). Evaluating domain-specific metric thresholds: an empirical study. In *International Conference on Technical Debt (TechDebt)*.

[Newman 2015] Newman, S. (2015). *Building Microservices*. O'Reilly Media, Inc.

[Oliveira et al. 2018] Oliveira, J., Viggiato, M., Santos, M., Figueiredo, E., and Marques-Neto, H. (2018). An empirical study on the impact of android code smells on resource usage. In *International Conference on Software Engineering & Knowledge Engineering (SEKE)*.

[Pahl 2015] Pahl, C. (2015). Containerization and the PaaS Cloud. *IEEE Cloud Computing*, 2(3):24–31.

[Pahl and Jamshidi 2016] Pahl, C. and Jamshidi, P. (2016). Microservices: A systematic mapping study. In *6th International Conference on Cloud Computing and Services Science (CLOSER)*, pages 137–146.

[Ramos et al. 2016] Ramos, M., Valente, M. T., Terra, R., and Santos, G. (2016). AngularJS in the wild: A survey with 460 developers. In *7th International Workshop on Evaluation and Usability of Programming Languages and Tools*, pages 9–16.

[Wohlin et al. 2012] Wohlin, C., Runeson, P., Hst, M., Ohlsson, M. C., Regnell, B., and Wessln, A. (2012). *Experimentation in Software Engineering*. Springer.

[Yu et al. 2016] Yu, Y., Silveira, H., and Sundaram, M. (2016). A microservice based reference architecture model in the context of enterprise architecture. In *1st Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*, pages 1856–1860.

Um Método para Detectar Similaridade entre Sistemas baseado em Decisões de Design: um Estudo Preliminar

Marcos Dósea[1,2], **Cláudio Sant'Anna**[1]

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

[2]Departamento de Sistemas de Informação
Universidade Federal de Sergipe – Itabaiana, SE – Brazil

`dosea@ufs.br, santanna@dcc.ufba.br`

Abstract. *Stable systems are commonly used as design model to develop, maintain, or assure the quality of other systems developed based on similar design decisions. However, finding applications developed with similar design decisions may not be a trivial task. This work proposes a new method to automatically calculate the similarity between systems. The method calculates the similarity using an abstract representation that relies on design roles identified in each system. We conducted an exploratory study applying the proposed method on fifteen real-world systems from three distinct system domains. The proposed method was able to identify a high similarity between systems developed based on similar design decisions.*

Resumo. *Sistemas estáveis são geralmente utilizados como modelo de design para desenvolver, manter ou garantir a qualidade de outras sistemas desenvolvidos com decisões de design similares. Entretanto, encontrar sistemas desenvolvidos com decisões de design similares pode não ser uma tarefa trivial. Este trabalho propõe um método automático para calcular a similaridade entre sistemas a partir de uma representação abstrata baseada nos papéis de design identificados em cada sistema. Foi conduzido um estudo exploratório aplicando o método proposto sobre quinze sistemas do mundo real de três domínios de sistemas distintos. O método proposto foi capaz de identificar alta similaridade entre os sistemas desenvolvidos com decisões de design semelhantes.*

1. Introdução

Sistemas estáveis e com qualidade reconhecida são geralmente utilizados como modelo de *design* para desenvolver, manter e garantir a qualidade de outros sistemas desenvolvidos com decisões de *design* similares. Por exemplo, considerando um sistema desenvolvido com estilo arquitetural em camadas e utilizando o *framework* Hibernate[1], é comum que o desenvolvedor busque soluções de *design* em outros sistemas desenvolvidos com decisões de *design* similares, ou seja, com estilo arquitetural, *frameworks* e bibliotecas similares [Singer et al. 2010]. Identificar sistemas desenvolvidos com decisões de *design* similares também pode auxiliar na criação de *benchmarks* usados para extrair valores limiares de métricas que levam em consideração o contexto de *design* das classes dos sistemas [Dósea et al. 2018].

[1]http://hibernate.org/orm/

A similaridade entre trechos de código, usada principalmente para busca de exemplos código e detecção de clones, é um tópico bastante explorado na literatura [Holmes and Murphy 2005, Roy et al. 2009, Wu et al. 2010]. Entretanto, identificar a similaridade de porções de código de maior granuralidade, como componentes ou o próprio sistema são tópicos ainda pouco explorados. A busca por sistemas desenvolvidos com decisões de *design* similares pode ser ainda mais complexa quando precisa ser realizada em grandes repositórios públicos de sistemas, por exemplo, o GitHub[2], ou mesmo repositórios corporativos com uma diversidade de sistemas disponíveis.

Nesse contexto, este trabalho propõe um método para calcular a similiaridade entre sistemas a partir de uma representação abstrata, criada para cada sistema, que se baseia nos papéis de *design* identificados. O papel de *design* de uma classe é um conjunto de responsabilidades assumidas pela classe para se encaixar em uma comunidade, como, por exemplo, um *framework* ou uma arquitetura corporativa [Wirfs-Brock and McKean 2003]. Em sistemas orientados a objetos, a arquitetura referência normalmente associa os papéis de *design* a uma ou mais classes através de herança, implementação de interfaces ou anotações. Essa arquitetura referência pode utilizar papéis pré-definidos em um *framework*, por exemplo, classes associadas aos papel de *design Action* no Struts. Mas também pode definir seus próprios papéis, por exemplo, criar a interface *IRepository* para associar o papel de *design Persistence*, responsável pela persistência. O método proposto baseia-se na hipótese de que sistemas implementados com os mesmos papéis de *design* devem possuir alta similaridade entre si.

Além de definir um novo método para o cálculo da similaridade, foi conduzido um estudo exploratório aplicando o método proposto para calcular a similaridade entre 15 sistemas de três domínios distintos. Foi considerado que dois sistemas pertencem a um mesmo domínio quando eles executam sobre a mesma plataforma. Os resultados mostraram que o método foi capaz de indentificar a similaridade no *design* entre a maioria dos sistemas do mesmo domínio. Mas o valor pode se tornar mais representativo se outras decisões de *design*, por exemplo, bibliotecas utilizadas, forem consideradas no cálculo.

O restante do artigo está organizado como descrito a seguir: A Seção 2 apresenta os trabalhos relacionados. A Seção 3 apresenta o método proposto para calcular a similaridade entre sistemas. A Seção 4 descreve o configuração do estudo exploratório realizado para avaliar a proposta. A Seção 5 apresenta os resultados e a Seção 6 discute as ameaças à validade. Finalmente, a Seção 7 apresenta a conclusão e os trabalhos futuros.

2. Trabalhos Relacionados

Poucos trabalhos consideram a similaridade entre porções maiores de código-fonte. Tibermacine et al. [Tibermacine et al. 2014] propõem uma abordagem para medir a similaridade de web services através de suas interfaces WSDL. O objetivo é encontrar o melhor substituto para um Web Service quando ele falhar. Al-msie'deen et al. [Al-msie'deen et al. 2013] propõem uma abordagem para minerar *features* através do cálculo da similaridade léxica e estrutural. A abordagem proposta também baseia-se em uma similiaridade estrutural dos papéis de *design* identificados. Entretanto, o método proposto utiliza um nível de abstração mais alto que permite considerar o sistema completo.

[2]https://github.com/

Nagappan et al. [Nagappan et al. 2013] propõem uma técnica para avaliar a cobertura de uma amostra de sistemas para que sejam representativos para realização de um experimento. Propõem uma função de similaridade de sistemas baseada em dimensões numéricas (ex: número de desenvolvedores e linhas de código) e dimensões categóricas (ex: principal linguagem de programação e domínio). O método proposto propõe uma nova dimensão numérica de similaridade baseada nos papéis de *design* identificados, não se limitando a informações genéricas sobre o sistema.

3. Método para Calcular a Similaridade entre Sistemas

O objetivo do método é encontrar sistemas desenvolvidos com decisões de *design* similares. O método calcula a similiaridade entre sistemas a partir de uma representação abstrata, criada para cada sistema, baseada nos papéis de *design* identificados e no percentual de linhas de código associadas a cada papel de *design*. Decisões de *design*, por exemplo, estilo arquitetural, *frameworks* e bibliotecas utilizados, acabam impacatando na representação abstrata obtida para cada sistema. O método é executado em quatro passos:

Passo 1) Identificar os papéis de *design* das classes de cada sistema: Para execução desse passo foi utilizada a ferramenta DesignRoleMiner [Dósea et al. 2018] que propõe uma heurística para identificar os papéis de *design* de cada classe do sistema. A heurística utiliza informações estruturais sobre herança, anotações e implementação de interfaces para associar um papel de *design* a cada classe. Para uniformizar o nome dos papéis de *design*, também é utilizada uma tabela de tokens que ao serem encontrados associam um papel de *design* padronizado. Por exemplo, uma classe que implementa a interface IRepository será associada ao papel de *design Persistence* porque 'Repository' é um dos tokens referentes ao papel de *design* padronizado *Persistence*. Caso esse token não existisse na tabela, a classe seria associada a um papel de *design* com o mesmo nome da interface. A heurística obteve alta precisão quando avaliada com sistemas corporativos.

Passo 2) Desconsiderar papéis de *design* que não descrevem o *design* do sistema: Quando a heurística não consegue identificar o papel de *design* da classe, um papel de *design* genérico, denominado *Undefined*, é associado. Muitos sistemas, independente do domínio, acabam tendo classes associadas ao papel *Undefined*. Esse papel foi desconsiderado porque se ele fosse utilizado na criação da representação abstrata do sistema, adicionaria algum valor de similaridade entre a maioria dos sistemas. Também foi desconsiderado o papel de *design Test*, associado a classes responsáveis pela execução de testes no código, porque essas classes não fazem parte do *design* do sistema.

Passo 3) Criar a representação abstrata do sistema: A representação abstrata proposta descreve cada sistema como um vetor que considera os papéis de *design* identificados no sistema e percentual de código-fonte associado a cada papel. Por exemplo, se a metade do número de linhas de código dos papéis de *design* considerados estão associadas ao papel de *design Service*, então foi considerado o peso de 50% para esse papel no vetor. O mesmo cálculo é realizado para os demais papéis a serem considerados. Definir a representatividade do papel de *design* a partir do percentual de linhas de código associado ao papel permite comparar sistemas com tamanhos distintos. Adicionalmente, pretende-se avaliar se esse percentual é suficiente para representar outras decisões de *design*. Por exemplo, o uso de diferentes bibliotecas e estilo de codificação para implementar o papel de *design Persistence* em sistemas distintos pode influenciar no percentual de código associado a esse papel em cada sistema.

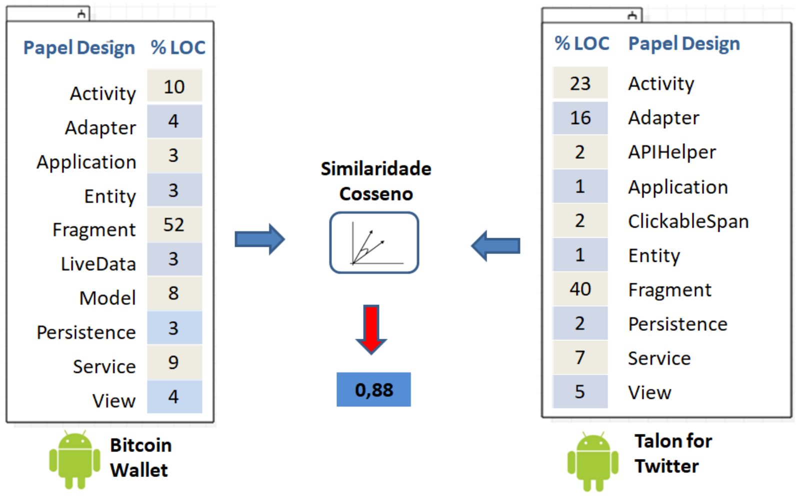


Figura 1. Cálculo da Similaridade entre Sistemas Bitcoin Wallet e Talon for Twitter

Passo 4) Calcular a similaridade das representações abstratas dos sistemas: Foi usada a medida de similaridade cosseno para comparar os dois vetores que representam o *design* de cada sistema. O valor resultante varia de 0 a 1. Quanto mais próximo de 1 mais similares são os sistemas. A medida cosseno foi utilizada por ser uma das mais eficientes funções de similaridade utilizadas em aplicações de processamento de linguagem natural e recuperação da informação [Wilkinson and Hingston 1991]. A função calcula a relevância dos tokens através do cálculo do cosseno entre dois vetores. No método proposto, foi considerado que os tokens são os papéis de *design* e a representatividade (peso) desses papéis foi definida a partir do percentual de linhas de código associado.

A Figura 1 ilustra esse processo com os sistemas Android Bitcoin Wallet e Talon for Twitter, apresentados na Seção 4. Cada sistema é representado de forma abstrata pelos papéis de *design* identificados (passo 1) e o percentual de código-fonte associado a cada papel (passo 3). A representação eliminou os papéis de *design Test* e *Undefined* (Passo 2). A partir dessas representações foi realizado o cálculo da similaridade usando a função cosseno (passo 4). O resultado (0,88) é muito próximo do valor 1, indicando que os dois sistemas possuem alta similaridade nas decisões de *design*.

4. Configuração do Estudo

O objetivo do estudo exploratório foi avaliar se o método para o cálculo de similaridade entre sistemas, proposto na Seção 3, é efetivo para encontrar sistemas desenvolvidos com decisões de *design* similares. Foram formuladas as seguintes questões de pesquisa:

QP1: *Sistemas pertencentes ao mesmo domínio possuem alta similaridade?*

QP2: *A representação abstrata do* design *do sistema proposta foi suficiente para detectar a similaridade entre sistemas?*

A primeira questão de pesquisa visa determinar se sistemas do mesmo domínio sempre irão possuir alta similaridade. Na segunda questão de pesquisa pretende-se verificar se a representação abstrata proposta para descrever o *design* do sistema foi suficiente para detectar sistemas desenvolvidos com decisões de *design* similares.

Sistemas Alvo: Foram selecionados 15 sistemas do repositório GitHub, já utili-

Tabela 1. Características dos Sistemas Alvo

Domínio	Sistema	Descrição	#classes	#métodos	#LOC	#releases	#Papéis de Design	#Data Commit
Aplicações Android	Bitcoin Wallet	Pagamentos em Bitcoin	111	1407	25K	138	9	2016-10-07
	Exoplayer	Media player	429	4532	84K	63	27	2016-10-06
	K-9 Mail	Cliente de Email	488	6913	109K	344	23	2016-04-13
	SMS Backup+	Backup de SMS	118	921	12K	78	10	2016-07-08
	Talon for Twitter	Cliente Twitter	312	3391	83K	1	12	2016-04-03
Plug-ins Eclipse	Activiti-Designer	Editor BPMN	704	3768	74K	19	51	2016-08-18
	AngularJS Eclipse	Editor do AngularJS	106	653	12K	19	14	2016-07-03
	Arduino IDE for Eclipse	IDE para Arduino	124	1180	24K	5	12	2016-04-04
	Drools and jBPM	IDE para Drools e jBPM	796	6936	107K	76	54	2016-12-09
	SonarLint Eclipse	Avalia qualidade do código	200	1178	18K	45	20	2016-12-12
Aplicações Web	BigBlueButton	Aprendizado on-line	1537	11377	176K	20	56	2016-10-18
	OpenMRS	Registro de pacientes	1066	12195	210K	115	37	2016-10-12
	Heritrix	Portal para web-crawling	567	4950	90K	2	34	2016-07-21
	Qalingo	Sistema E-commerce	956	13836	147K	4	18	2016-09-25
	LibrePlan	Gerenciamento de Projetos	1541	23278	282K	32	35	2016-11-09

zados em outros estudos [Dósea et al. 2018]. Os sistemas foram selecionados usando as strings de busca: "Eclipse plugin language:java", "android language:java" e "Web language:java". Adicionalmente, foram excluídos *frameworks* e bibliotecas porque elas raramente compartilham decisões de *design* com outros sistemas.

A Tabela 1 apresenta as características dos quinze sistemas utilizados. As colunas #classes e #métodos exibem o número de classes e métodos de cada sistema. Os sistemas possuem entre 12 e 282 mil linhas de código (coluna #LOC). A coluna #releases mostra o número de releases estáveis disponível em cada sistema. A última coluna mostra a data do *commit* correspondente ao código fonte utilizado no estudo. Finalmente, a coluna #papéis de *design* mostra o número de papéis de *design* identificados em cada sistema. Por exemplo, o sistema Android SMS Backup+ possui 10 papéis de *design* identificados. O website [3] contém detalhes sobre os papéis de *design* identificados em cada sistema.

Procedimentos do Estudo: Para responder a primeira questão de pesquisa (QP1) foi realizado o cálculo da similaridade entre os 15 sistemas estudados, totalizando 105 valores de similaridade. Em seguida, para cada sistema, foi identificado os quatro sistemas mais similares, ou seja, com os maiores valores na medida proposta de similaridade. Finalmente, o código-fonte desses quatro sistemas mais similares foi analisado para justificar o valor de similaridade encontrado. Para responder a segunda questão de pesquisa (QP2) também foi analisado o código-fonte dos quatro sistemas mais similares com o objetivo de identificar se a utilização dos papéis de *design* e percentual de linhas de código associado a cada papel foi suficiente para representar outras decisões de *design* do código, por exemplo, bibliotecas e estilo de codificação utilizado.

5. Resultados e Discussão

A Tabela 2 apresenta os valores de similaridade, usando o método proposto na Seção 3. Cada valor da tabela corresponde ao cálculo da similaridade entre os sistemas que constam na linha e na coluna. Por exemplo, a similaridade entre os sistemas K9 Mail e Bitcoin Wallet é igual a 0,41. Os valores em negrito destacam os quatro sistemas mais similares ao sistema que consta na linha. Por exemplo, os quatro sistemas mais similares ao sistema LibrePlan são OpenMRS, Heritrix, Qalingo e SMS Backup.

QP1: *Sistemas pertencentes ao mesmo domínio possuem alta similaridade?*

A hipótese inicial era que sistemas pertencentes ao mesmo domínio sempre teriam alta similaridade entre si devido a alta probabilidade de usarem os mesmos papéis

[3]https://sites.google.com/site/designdecisions2018/data/rq1

Tabela 2. Similaridade entre Sistemas

	BigBlueButton	OpenMRS	Heritrix	Qalingo	LibrePlan	Bitcoin	K9 Mail	Exoplayer	SMS Backup	Talon	Activiti	AngularJS	Arduino	DroolsJBPM	Sonarlint
BigBlueButton	1,00	0,17	0,23	0,28	0,19	0,08	0,41	0,33	0,28	0,11	0,31	**0,56**	**0,48**	**0,51**	**0,56**
OpenMRS	0,17	1,00	0,10	**0,37**	**0,34**	0,10	0,24	**0,29**	**0,31**	0,08	0,06	0,07	0,07	0,13	0,08
Heritrix	**0,23**	0,10	1,00	**0,48**	**0,28**	0,03	**0,15**	0,10	0,10	0,01	0,06	0,05	0,05	0,07	0,05
Qalingo	**0,28**	**0,37**	**0,48**	1,00	**0,64**	0,11	0,18	0,15	0,18	0,06	0,14	0,01	0,03	0,08	0,03
LibrePlan	0,19	**0,34**	**0,28**	**0,64**	1,00	0,11	0,16	0,15	**0,21**	0,06	0,06	0,02	0,03	0,08	0,03
Bitcoin	0,08	0,10	0,03	0,11	**0,11**	1,00	**0,41**	0,08	**0,16**	**0,88**	0,04	0,07	0,06	0,08	0,08
K9 Mail	**0,41**	0,24	0,15	0,18	0,16	**0,41**	1,00	0,04	**0,65**	**0,53**	0,15	0,33	0,25	0,32	0,33
Exoplayer	**0,33**	0,29	0,1	0,15	0,15	0,08	0,04	1,00	0,28	0,10	0,15	**0,34**	0,24	**0,36**	**0,35**
SMS Backup	**0,28**	**0,31**	0,10	0,18	0,21	0,16	**0,65**	**0,28**	1,00	0,03	0,03	0,03	0,03	0,09	0,04
Talon	**0,11**	0,08	0,01	0,06	0,06	**0,88**	**0,53**	0,10	**0,30**	1,00	0,03	0,08	0,07	0,08	0,10
Acitiviti	0,31	0,06	0,06	0,14	0,06	0,04	0,15	0,15	0,03	0,03	1,00	**0,42**	**0,38**	**0,47**	**0,40**
AngularJS	**0,56**	0,07	0,05	0,01	0,02	0,07	0,33	0,34	0,03	0,08	0,42	1,00	**0,77**	**0,74**	**0,89**
Arduino	**0,48**	0,07	0,05	0,03	0,03	0,06	0,25	0,24	0,03	0,07	0,38	**0,77**	1,00	**0,62**	**0,76**
DroolsJBPM	**0,51**	0,13	0,07	0,07	0,08	0,08	0,32	0,36	0,09	0,08	0,47	**0,74**	**0,62**	1,00	**0,74**
Sonarlint	**0,56**	0,08	0,05	0,01	0,03	0,08	0,33	0,35	0,04	0,10	0,4	**0,89**	**0,76**	**0,74**	1,00

de *design*. Os resultados obtidos e posterior confirmação através da análise do código-fonte, mostraram que nem sempre isso ocorre. Alguns sistemas, pertencentes ao mesmo domínio, tiveram baixo valor de similaridade devido ao uso de papéis de *design* muito diferentes dos geralmente utilizados pelos sistemas deste domínio. Adicionalmente, sistemas que não associam o papel de *design* das classses através de mecanismos de herança, interface ou anotações também geraram uma representação abstrata do *design* do sistema pouco representativa para o cálculo da similaridade.

No domínio dos sistemas Web, o sistema Qalingo obteve maior similaridade com os outros quatro sistemas do mesmo domínio. Já os sistemas Libreplan e Heritrix obtiveram maior similaridade com três sistemas do domínio, e o sistema OpenMRS com apenas dois sistemas do domínio Web. *Service*, *Entity*, *Component* e *Controller* são papéis de *design* geralmente encontrados e representativos em sistemas desse domínio. Entretanto, alguns sistemas obtiveram maior similaridade com sistemas do domínio Android devido a alguns papéis comuns aos dois domínios. Por exemplo, o sistema Android SMS Backup foi o quarto sistema mais similar ao sistema Web LibrePlan. *Persistence* e *Service* foram papéis de *design* comuns aos dois sistemas e que influenciaram no cálculo da similaridade. Entretanto, no sistema LibrePlan, *Persistence* é implementado com o Hibernate, enquanto que, no sistema Android SMS Backup, são usadas bibliotecas do Android para persistência de mensagens SMS. Ou seja, o mesmo papel (*Persistence*) é implementado com decisões de *design* distintas nos dois sistemas. Finalmente o sistema BigBlueButton acabou não tendo alta similaridade nem com os sistemas do domínio Web nem com os do domínio Android. Analisando o código desse sistema, notou-se que 45% de suas classes não tiveram um papel de *design* associado (*Undefined*), criando uma representação abstrata do sistema pouco representativa dentro do domínio Web.

No domínio dos sistemas Android, os sistemas Bitcoin, K9 Mail, Talon for Twitter tiveram três sistemas com mais alta similaridade pertencentes ao mesmo domínio. O sistema SMS Backup obteve alta similaridade com dois sistemas do mesmo domínio. Apenas o sistema Exoplayer não obteve alta similaridade com nenhum sistema Android. Assim como aconteceu com o sistema BigBlueButton, o sistema Exoplayer teve 56% das classes associadas aos papéis de *design Undefined* e *Test*, que não são considerados para representação abstrata do sistema usada no cálculo da similaridade. Adicionalmente, os papéis de *design Activity*, *Fragment*, *Service*, comuns nos sistemas do domínio Android, não existiam ou eram pouco representativos no sistema Exoplayer. Dessa forma, o cálculo da similaridade refletiu o distanciamento do *design* do Exoplayer dos sistemas do seu domínio.

Finalmente, os sistemas do domínio de plug-ins para Eclipse foram os que obtiveram os maiores valores de similaridade entre si. Esses valores são explicados pela menor variação dos papéis de *design* que podem ser utilizados. Em todos os plug-ins são bem representativos papéis de *design* como *Action*, *Dialog*, *Plugin* e *View*. Entretanto, o *design* do sistema Web BigblueButton, discutido anteriormente, acabou tendo maior similaridade com alguns plug-ins do Eclipse devido a identificação de alguns papéis de *design* comuns ao domínio dos plug-ins para Eclipse (ex. *View*, *Action* e *Dialog*). Os papéis de *design* são implementados com bibliotecas distintas nos dois domínios, porém, como o método não considera isso no cálculo, houve alta similaridade entre esses sistemas.

Em resumo, a medida de similaridade proposta conseguiu refletir a similaridade de *design* normalmente encontrada em sistemas do mesmo domínio. Também foi capaz de identificar quando o *design* do sistema se distanciava das decisões de *design* comuns ao sistemas do mesmo domínio, por exemplo, ao usar papéis de *design* diferentes.

QP2: *A representação abstrata do* design *do sistema proposta foi suficiente para detectar a similaridade entre sistemas?*

A representação proposta foi capaz de identificar a similaridade de decisões de *design* entre maioria dos sistemas pertencentes ao mesmo domínio. Entretanto, em alguns casos ocorreu alta similaridade entre sistemas pertencentes a domínios distintos. Por exemplo, o sistema Web OpenMRS obteve alta similaridade com dois sistemas pertencentes ao domínio Android (Exoplayer e SMS Backup) devido ao uso comum de alguns papéis de *design*. Por exemplo, papéis de *design* como *Persistence*, *Service*, *Entity* e *View* são comuns aos dois domínios, mas normalmente são implementados com decisões de *design* distintas em cada domínio. Ou seja, nesses casos o método encontrou alta similaridade entre sistemas que, na realidade, são implementados com outras decisões de *design* distintas, apesar de terem conjuntos de papéis de *design* semelhantes. Trabalhos futuros podem avaliar se considerar outras decisões de *design*, por exemplo, as bibliotecas usadas por cada papel de *design*, trariam melhorias para o cálculo da similaridade.

6. Ameaças à Validade

Nesta seção são discutidas as ameaças à validade do estudo preliminar conduzido e as ações que foram tomadas para minimizá-las.

Validade do Construto: Uma possível ameaça é que os sistemas escolhidos priorizem a similaridade de sistemas do mesmo domínio. Para diminuir esse viés, foram aplicados critérios bem definidos para selecionar os sistemas do repositório GitHub. Outra ameaça foi a escolha da medida de similaridade cosseno. Apesar de ser uma das medidas de similaridade mais utilizadas, esse estudo foi exploratório e trabalhos futuros podem avaliar outras medidas de similaridade.

Validade Interna: A principal ameaça é em relação a ferramenta para identificar os papéis de *design* utilizados para criação da representação abstrata do sistema. Entretanto, a ferramenta já foi utilizada em outros estudos identificando com boa precisão a maioria dos papéis de *design* dos sistemas [Dósea et al. 2018]. Apesar do papel *Undefined* não ser considerado na representação abstrata proposta, futuras melhorias na identificação dos papéis de *design* podem trazer melhorias no cálculo de similaridade.

Validade Externa: Os resultados obtidos são válidos para os quinze sistemas avaliados e três domínios. Não é sugerido a generalização desses resultados para outros sistemas ou domínios.

7. Conclusão

Neste trabalho foi apresentado um método para detectar a similaridade entre sistemas. Sistemas similares podem ser utilizados como modelo de *design* para o desenvolvimento e manutenção de funcionalidades e para criação de benchmarks usados na extração de valores limiares para métricas. Foi realizada uma avaliação com 15 sistemas de três domínios distintos e os resultados mostraram que o método proposto foi capaz de encontrar a maioria dos sistemas desenvolvidos com decisões de *design* similares.

Como trabalhos futuros, pretende-se evoluir a medida de similaridade proposta para considerar as bibliotecas utilizadas por cada papel de *design*. Percebeu-se alguns casos onde o mesmo papel de *design*, implementado com bibliotecas distintas, aumentou equivocadamente o valor de similaridade. Adicionalmente, pretende-se conduzir estudos que avaliem se a medida proposta pode ser utilizada para detectar o distanciamento do *design* planejado durante o processo de desenvolvimento de uma aplicação.

Agradecimentos. Esse trabalho é apoiado pelo CNPq (projeto 312153/2016-3).

Referências

Al-msie'deen, R., Seriai, A. D., Huchard, M., Urtado, C., and Vauttier, S. (2013). Mining features from the object-oriented source code of software variants by combining lexical and structural similarity. In *Int. Conf. on Inf. Reuse Integration (IRI)*, pages 586–593.

Dósea, M., Sant'Anna, C., and da Silva, B. C. (2018). How do design decisions affect the distribution of software metrics? In *Proceedings of the 26th Conference on Program Comprehension (ICPC)*, pages 74–85.

Holmes, R. and Murphy, G. C. (2005). Using structural context to recommend source code examples. In *Int. Conf. on Software Engineering (ICSE)*, pages 117–125.

Nagappan, M., Zimmermann, T., and Bird, C. (2013). Diversity in software engineering research. In *Foundations of Software Engineering (FSE)*, pages 466–476.

Roy, C. K., Cordy, J. R., and Koschke, R. (2009). Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74(7):470–495.

Singer, J., Lethbridge, T., Vinson, N., and Anquetil, N. (2010). An examination of software engineering work practices. In *CASCON First Decade High Impact Papers (CASCON)*, pages 174–188.

Tibermacine, O., Tibermacine, C., and Cherif, F. (2014). A Practical Approach to the Measurement of Similarity between WSDL-based Web Services. *Revue des Nouvelles Technologies de l'Information*, pages 3–18.

Wilkinson, R. and Hingston, P. (1991). Using the cosine measure in a neural network for document retrieval. In *Int. ACM SIGIR Conf. on Res. and Dev. in Inf. Retrieval (SIGIR)*, pages 202–210, New York, NY, USA. ACM.

Wirfs-Brock, R. and McKean, A. (2003). *Object design: roles, responsibilities, and collaborations*. Addison-Wesley Professional.

Wu, Y. C., Mar, L. W., and Jiau, H. C. (2010). Codocent: Support api usage with code example and api documentation. In *Int. Conf. Soft. Eng. Adv.(ICSEA)*, pages 135–140.

Comparando Técnicas de Extração de Valores Limiares para Métricas: Um Estudo Preliminar com Desenvolvedores Web

Raphael Lima[1], **Marcos Dósea**[1,2], **Claudio Sant'Anna**[1]

[1]Departamento de Ciência da Computação –
Universidade Federal da Bahia – Salvador, BA – Brasil

[2]Departamento de Sistemas de Informação –
Universidade Federal de Sergipe – Itabaiana, SE – Brasil

raphael.lima@ufba.br, santanna@dcc.ufba.br, dosea@ufs.br

Abstract. *The use of source code metrics still faces challenges. One of the main challenges is defining thresholds values that make sense. There are a number of techniques for defining these values. However, there is a lack of studies that evaluate the effectiveness of these techniques. This study aims to evaluate the accuracy of five techniques in the definition of threshold values for the detection of design problems. To do this, we conducted an empirical study to evaluate whether developers agree with design problems detected with the use of thresholds, defined by each of the five techniques, for four source code metrics.*

Resumo. *O uso de métricas de código-fonte ainda enfrenta desafios. Um dos principais desafios é a definição de valores limiares que façam sentido. Existe uma série de técnicas para definição de valores limiares. No entanto, existem poucos estudos que avaliem a efetividade destas técnicas. Este estudo tem por objetivo avaliar a acurácia de cinco técnicas usadas para definição dos valores limiares usados para detecção de problemas de design. Para isso, nós conduzimos um estudo empírico para avaliar a percepção dos desenvolvedores sobre problemas de design detectados com os valores limiares obtidos em cinco técnicas, para quatro métricas de código-fonte.*

1. Introdução

A utilização de métricas para a avaliação da qualidade do código-fonte ainda enfrenta alguns desafios [Kamei and Shihab 2016]. Uma das principais dificuldades é a definição dos valores limiares para indicar possíveis problemas de *design* no código-fonte. Por exemplo, Lanza & Marinescu [2006] propõem um valor limiar de 20 linhas de código para classificar um método como longo. Porém, esse valor pode não ser adequado para todos os sistemas. Fatores contextuais como a linguagem de programação e o domínio da aplicação podem influenciar nessa definição [Zhang et al. 2013].

Várias técnicas foram propostas para extrair valores limiares [Alves et al. 2010, Ferreira et al. 2012, Vale and Figueiredo 2015]. Entretanto, os valores limiares obtidos por essas técnicas são genéricos e não consideram informações do contexto da classe ou do sistema que será avaliado. Por exemplo, considerando um sistema desenvolvido em três camadas (*View, Business e Persistence*), será que classes/métodos da camada *Business*, que concentram a complexidade da aplicação, deveriam ser avaliados com os mesmos valores limiares das classes das camadas *View* e *Persistence*? Trabalhos recentes vêm propondo técnicas para a identificação do contexto de *design* ou arquitetural da classe que permitiriam definir valores limiares que considerem essa informação de contexto [Aniche et al. 2016, Dósea et al. 2018].

Entretanto, apesar da variedade de técnicas, existem poucos estudos que avaliam a percepção dos desenvolvedores sobre os valores limiares. Também não há clareza sobre a percepção dos desenvolvedores sobre a utilização do contexto de *design* da classe na avaliação dos problemas de *design*. Neste trabalho conduzimos um estudo empírico para avaliar a percepção de desenvolvedores Web sobre valores limiares obtidos a partir de cinco técnicas de identificação de valores limiares. Os resultados preliminares indicam que a utilização de valores limiares que consideram alguma informação de contexto podem auxiliar a reduzir o número de falsos positivos para o desenvolvedor.

2. Trabalhos Relacionados

Poucos estudos avaliam a percepção dos desenvolvedores sobre valores limiares. Palomba et al. [2014] avaliam três sistemas Java buscando 12 *bad smells*. O estudo mostra que *bad smells* podem ou não representar um problema de acordo com o contexto do sistema. Oliveira et al. [2015] propõem uma técnica para extração de valores limiares baseada em *benchmark* de sistemas e validam os valores obitdos com desenvolvedores. Os resultados indicaram que aplicações de boa qualidade respeitam os valores limiares. Nosso estudo também avaliou problemas de *design* com desenvolvedores experientes, mas a avaliação não foi limitada a problemas sugeridos por uma única técnica, uma vez que comparamos cinco técnicas distintas de extração de valores limiares.

Vale *et al.* [2018] avaliam a utilização de limiares obtidos a partir de três técnicas de identificação de valores limiares para detectar *God Class* e *Lazy Class* em uma linha de produto de software. Os problemas no sistema foram previamente sugeridos por especialistas. Como resultado, a técnica proposta pelos próprios autores apresenta pequena vantagem em relação às outras técnicas avaliadas. Nosso estudo também avalia as mesmas técnicas avaliadas por Vale *et al.* [2018], exceto a técnica de Lanza & Marinescu [2006], por ela considerar que as métricas têm distribuição normal, o que raramente ocorre. Adicionalmente, o estudo proposto incluiu novas técnicas e utilizou sistemas Web reais com desenvolvedores experientes para avaliar os problemas de *design*.

3. Configuração do Estudo

O objetivo deste estudo foi investigar a percepção de desenvolvedores Web sobre problemas de *design* apontados a partir de valores limiares obtidos com a aplicação de cinco técnicas que extraem valores limiares a partir de *benchmark* de sistemas. Foram formuladas as seguintes questões de pesquisa:

QP1: *Qual técnica propõe valores limiares que melhor refletem a percepção dos desenvolvedores sobre os problemas de design?*

QP2: *Qual a percepção dos desenvolvedores sobre a utilização do papel de design da classe para identificar problemas de design?*

A primeira questão tem como objetivo avaliar de forma quantitativa a percepeção dos desenvolvedores sobre os problemas de *design* identificados, isto é, os métodos cujos valores de métricas ultrapassaram os valores limiares. A segunda questão de pesquisa pretende avaliar de forma qualitativa a percepção dos desenvovedores sobre o papel de *design* da classe na identificação dos problemas de *design*.

Técnicas para Identificar Valores Limiares: Selecionamos técnicas que não consideram que a distribuição dos valores das métricas seguem distribuição normal, visto que é muito difícil que isso ocorra. Para evitar viés na configuração de parâmetros, também consideramos técnicas que podem ser completamente automatizadas. Baseados nesses critérios, foram selecionadas as técnicas de Alves *et al.* [2010] e Vale & Figueiredo [2015] que geram um valor limiar genérico para cada métrica. Também avaliamos uma variação da técnica de Alves *et al.* [2010], proposta por Dósea *et al.* [2016]. Esta variação consiste em usar sistemas de referência para compor o *benchmark*. Sistemas de referência são desenvolvidos com regras de *design* similares ao sistema que será avaliado e são considerados referência de qualidade de código. Por exemplo, para este estudo, os sistemas de referência foram selecionados por membros experientes da equipe.

A quarta técnica avaliada foi a proposta por Aniche *et al.* [2016] que também se baseia na técnica de Alves *et al.* [2010], mas define valores limiares específicos para as métricas de acordo com o papel arquitetural das classes definidos a partir de *frameworks* pré-definidos (Ex: Android, Spring). Por exemplo, uma classe associada ao papel arquitetural *Controller* deve ser avaliada por valores limiares específicos identificados para esse papel. Finalmente, avaliamos uma nova técnica sugerida por Dósea *et al.* [2018], que também usa Alves *et al.* [2010] e *benchmarks* compostos por sistemas de referência, mas identifica valores limiares específicos de acordo com o papel de *design* da classe. O papel de *design* é uma extensão do conceito de papel arquitetural e permite definir o papel de classes não vinculadas a um *framework* pré-definido [Dósea et al. 2018].

Sistemas Alvo: Foram utilizados dois sistemas Web, desenvolvidos na linguagem Java, que pertencem a Superintendência de Tecnologia da Informação (STI) da Universidade Federal da Bahia (UFBA). Estudos empíricos com foco em avaliar percepção de problemas de *design* por desenvolvedores também não utilizam muitos sistemas [Palomba et al. 2014, Vale et al. 2018]. A seleção foi feita por conveniência, priorizando sistemas cujos desenvolvedores mais experientes estivessem disponíveis para a entrevista. Foram utilizados o Sistema Integrado de Serviços e Usuários (SIUS) e o Sistema de Avalização Docente/Discente (SIAV). O SIUS possui 76 classes, 844 métodos e 8545 linhas de código. O SIAV possui 53 classes, 525 métodos e 5277 linhas de código.

Métricas Analisadas: Foram utilizadas quatro métricas de método, selecionadas pela facilidade de cálculo manual pelos desenvolvedores sem o auxílio de ferramentas, característica importante para facilitar a percepção do problema de *design*. A primeira foi a métrica Complexidade Ciclomática de McCabe (CC) [McCabe 1976] que conta o número de desvios em cada método. A segunda foi a métrica de Número de Parâmetros do Método (NMP) [Fowler and Beck 1999] que conta o número de parâmetros de cada método. A terceira foi a métrica de Linhas de Código (LOC) [Lanza and Marinescu 2006] que conta o número de comandos executáveis em cada método, excluindo comentários e linhas em branco. A última foi a métrica que mede o Acoplamento Eferente (EC) [Martin and Lippman 2000] que conta o número de classes internas e externas ao sistema invocadas em cada método para chamar um método ou acessar um atributo.

Procedimentos do Estudo: A primeira etapa do estudo foi selecionar os sistemas para compor os *benchmarks*. As técnicas de Alves *et al.* [2010], Vale *et al.* [2015] e Aniche *et al.* [2016] usaram o mesmo *benchmark* composto por 15 sistemas Web reais, desenvolvidos na linguagem Java e selecionados no repositório Github em 11/05/2018. Os sistemas possuem pelo menos 50 estrelas e no mínimo uma atualização desde 01/01/2018. A busca usando esses critérios retornou 272 projetos. Dessa lista, foram excluídas bibliotecas e *frameworks*, sistemas usados como exemplos de implementação, sistemas sem nenhuma *release* e com menos de 100 classes. Esses critérios tiveram como objetivo selecionar sistemas Web reais. Para as outras duas técnicas, baseadas em sistemas de referência [Dósea et al. 2016], foram criados dois *benchmarks*, um para cada sistema avaliado, devido as diferenças nas regras de *design*. Por exemplo, um dos sistemas usava Java Server Faces e o outro Struts para exibição dos dados. Os sistemas de cada *benchmarks* foram sugeridos por um grupo de três desenvolvedores experientes do próprio setor de TI. Os valores limiares foram gerados através da ferramenta DesignRoleMiner [Dósea et al. 2018] que disponibiliza a implementação das cinco técnicas. Adicionalmente, a ferramenta disponibiliza o cálculo de métricas de código, e gera uma lista de métodos com problemas de *design* indicados a partir dos valores limiares obtidos com as cinco técnicas.

Na segunda etapa, o objetivo foi selecionar os métodos, cujo código-fonte seria avaliado pelos desenvolvedores em busca de problemas de *design*. Em um estudo piloto, percebeu-se que avaliar todos os métodos de cada sistema inviabilizaria o estudo, devido ao tempo necessário e a fadiga demonstrada pelos desenvolvedores. Percebeu-se que métodos com todas as métricas abaixo dos valores limiares não eram identificados como problemáticos. Portanto, apenas métodos com uma das métricas acima de algum dos valores limiares definidos por uma das cinco técnicas foram considerados para avaliação. Também foi percebido que métodos associados ao mesmo papel de *design* e com valores de métricas similares obtinham as mesmas avaliações dos desenvolvedores. Por exemplo, se um método do papel de *design Persistence* com 50 linhas de código era avaliado como longo, outro método com 53 linhas de código também era considerado longo. Portanto, para evitar repetição de avaliações e diminuir a quantidade de métodos avaliados, foram selecionados, por papel de *design* e métrica, uma amostra de 10% dos métodos, de forma que não tivéssemos mais de um método com valores iguais ou muito próximos.

Na terceira etapa, foram realizadas entrevistas com um desenvolvedor de cada sistema. Cada desenvolvedor possuía pelo menos cinco anos trabalhando no projeto. Sem saber que os métodos avaliados foram indicados como problemáticos por uma das cinco técnicas estudadas, foi solicitado que o desenvolvedor avaliasse os métodos em busca dos quatro problemas de *design*: método longo, método com muitos parâmetros, método com muitos desvios em seu código e método muito acoplado a outras classes. Dessa forma, para cada método avaliado, o desenvolvedor poderia dizer que não haveria problema ou indicar um ou mais dos quatro problemas de *design*. Para analisar os dados quantitativos obitdos para as cinco técnicas foi realizada o cálculo das medidas de *recall*, precisão e medida-F.

4. Ameaças à Validade

Esta seção apresenta ameaças à validade do estudo e ações tomadas para minimizá-las.

Validade do Construto: Uma possível ameaça é que os *benchmark*s utilizados gerem valores limiares que prejudiquem os resultados de algum método avaliado. Para diminuir esse viés, aplicamos critérios bem definidos para selecionar sistemas Web reais do repositório Github e montamos uma equipe de especialistas para decidir quais sistemas iriam compor os *benchmark*s usados pelos métodos baseados em sistemas referência de *design* e qualidade. Sobre a medição da percepção dos desenvolvedores, uma possível ameaça era que os desenvolvedores fossem sugestionados para os problemas de *design*. Para evitar esse problema, o protocolo[1] define que o desenvolvedor deve ser informado que os métodos avaliados poderiam ou não ter alguns dos quatro problemas de *design*.

Validade Interna: A principal ameaça é o número de desenvolvedores entrevistados. Mas o estudo é preliminar e também não é comum em outros estudos similares uma grande quantidade de entrevistados. Para minimizar essa ameaça selecionamos o desenvolvedor mais experiente de cada equipe, mas pretendemos em trabalhos futuros entrevistar outros desenvolvedores da mesma equipe e incluir novos projetos.

Validade Externa: Os resultados obtidos são válidos para os dois sistemas avaliados e para as quatro métricas estudadas. Não sugerimos que sejam generalizados.

5. Resultados

A Tabela 1 apresenta os valores limiares obtidos pelas cinco técnicas para os dois sistemas avaliados. Os valores definidos pelas técnicas de Alves *et al.* [2010], Vale & Figueiredo [2015] e Aniche *et al.* [2016] são iguais para os dois sistemas, uma vez que essas técnicas não recomendam a utilização de *benchmarks* específicos para sistemas diferentes. Para as técnicas de Dósea *et al.* [2016] e Dósea *et al.* [2018] foram criados *benchmarks* distintos, detalhados na Seção 3, para avaliação dos dois sistemas. Adicionalmente, as técnicas de Aniche *et al.* [2016] e Dósea *et al.* [2018] definem valores limiares específicos de acordo com o papel arquitetural e papel de *design* da classe respectivamente.

Tabela 1. Valores limiares propostos pelas técnicas

	SIUS					SIAV				
Técnicas	Papel de Design	LOC	CC	Eferente	NOP	Papel de Design	LOC	CC	Eferente	NOP
Alves [2010]	Geral	85	16	10	3	Geral	85	16	10	3
Aniche [2016]	Entity	8	1	2	0	Entity	8	1	2	0
	Controller	28	4	8	0	Controller	28	4	8	0
	Persistence	41	4	7	3	Persistence	41	4	7	3
	Service	39	7	12	4	Service	39	7	12	4
Vale [2015]	Geral	18	3	6	1	Geral	18	3	6	1
Dósea [2016]	Geral	50	11	13	1	Geral	98	16	16	3
Dósea [2018]	Authenticate	43	9	12	2	Authenticator	43	5	11	3
	Adapter	15	2	5	2	Action	92	14	23	3
	Validate	47	18	6	2	Exception	43	5	11	3
	Controller	45	8	14	3	Persistence	43	6	11	3
	Entity	47	13	12	2	Entity	43	5	11	3
	Service	31	8	12	6	Decorator	43	5	11	3
	Model	15	2	5	2	AbstractBO	55	18	11	3
	View	15	2	5	2	ValidatorForm	43	5	11	3

Para calcular a precisão e o *recall* nós usamos os seguintes conceitos: Verdadeiros positivos (TP) são os métodos apontados como problemáticos por uma técnica (valor da métrica maior que o valor limiar) e o desenvolvedor concordou que deveria haver uma refatoração devido ao mesmo problema; Falsos positivos (FP) são os métodos apontados como problemáticos por uma técnica e o desenvolvedor discordou que deveriam ser refatorados; Verdadeiros negativos (TN) são os métodos que não foram apontados como problemáticos por uma técnica e o desenvolvedor concordou em não refatorar; Falsos negativos (FN) são os métodos que não foram apontados como problemáticos por uma

[1]Protocolo e dados dos resultados disponíveis no site: https://sites.google.com/view/vem-2018

técnica e o desenvolvedor concordou que deveriam ser refatorados. Para calcular *Recall* foi usada a fórmula TP / (TP + FN). Para a precisão a fórmula utilizada foi TP / (TP + FP). Para calcular a medida-F foi usada a fórmula (2*precisão**recall*)/(precisão+*recall*).

QP1: *Qual método propõe valores limiares que melhor refletem a percepção dos desenvolvedores sobre os problemas de design?*

As Tabelas 2 e 3 apresentam os resultados das medidas de Precisão, *Recall* e medida-F de cada técnica em cada métrica analisada sobre os dois sistemas avaliados. Para o cálculo destas medidas, foram analisadas as respostas dos desenvolvedores acerca dos métodos apontados com possíveis problemas de *design*. Inicialmente, percebe-se que para métrica de número de linhas de código (LOC), a técnica que apresenta uma maior precisão, em ambos os sistemas, foi a de Alves *et al.* [2010] seguida pelas técnicas de Dósea *et al.* [2016] e Dósea *et al.* [2018], que apresentaram valores maiores de *recall*. A técnica de Vale & Figueiredo [2015] apresentou valores altos de *recall*, mas baixa precisão, por ter definido valores limiares muito baixos que acabam incrementando os falsos positivos. Se analisarmos os valores da medida-F, podemos perceber que as três técnicas Alves *et al.* [2010], Dósea *et al.* [2016] e Dósea *et al.* [2018] obtiveram os melhores resultados para a métrica LOC.

Tabela 2. Resultados das Análises do Sistema SIUS

Técnicas	LOC			CC			Eferente			NOP		
	Precisão	Recall	F	Precisão	Recall	F	Precisão	Recall	F	Precisão	Recall	F
Alves [2010]	1,00	0,33	0,50	0,00	0,00	—	0,03	1,00	0,05	0,07	1,00	0,13
Aniche [2016]	0,17	0,33	0,22	0,40	0,33	0,13	0,02	1,00	0,04	0,02	1,00	0,04
Vale [2015]	0,08	1,00	0,15	0,30	1,00	0,46	0,01	1,00	0,03	0,06	1,00	0,04
Dósea [2016]	1,00	0,33	0,50	0,00	0,00	—	0,17	1,00	0,29	0,07	1,00	0,13
Dósea [2018]	0,50	0,33	0,40	0,75	0,50	0,60	1,00	1,00	1,00	0,07	1,00	0,13

Tabela 3. Resultados das Análises do Sistema SIAV

Técnicas	LOC			CC			Eferente			NOP		
	Precisão	Recall	F	Precisão	Recall	F	Precisão	Recall	F	Precisão	Recall	F
Alves [2010]	1,00	0,50	0,67	1,00	0,33	0,21	0,67	0,50	0,57	—	—	—
Aniche [2016]	0,25	1,00	0,40	0,13	0,67	0,67	0,44	1,00	0,62	0,00	—	—
Vale [2015]	0,10	1,00	0,17	0,14	1,00	0,25	0,36	1,00	0,53	0,00	—	—
Dósea [2016]	0,50	1,00	0,67	0,50	0,33	0,40	0,67	0,50	0,57	0,00	—	—
Dósea [2018]	0,50	1,00	0,67	0,67	0,67	0,67	0,67	0,50	0,57	—	—	—

Para a métrica de complexidade ciclomática (CC), percebe-se que a técnica com os melhores resultados para precisão, *recall* e medida-F nos dois sistemas foi a de Dósea *et al.* [2018]. As outras técnicas apresentaram variações entre os sistemas. As técnicas de Alves *et al.* [2010] e Dósea *et al.* [2016] não obtiveram resultados no sistema SIUS e tiveram resultados regulares no SIAV. As técnicas de Aniche *et al.* [2016] e Vale & Figueiredo [2015] apresentaram baixa precisão, principalmente no SIAV.

Para a métrica de Acoplamento Eferente, verificamos que para o sistema SIUS, todas as técnicas apresentaram 100% de *recall*, mas em termos de precisão a técnica de Dósea *et al.* [2018] foi a que apresentou os melhores resultados. Para o SIAV, as técnicas de Alves *et al.* [2010], Dósea *et al.* [2016] e Dósea *et al.* [2018] novamente obtiveram bons resultados. Aniche *et al.* [2016] e Vale & Figueiredo [2015] continuaram apresentando valores altos de *recall*, mas baixa precisão. Analisando as medidas-F, pode-se perceber que a técnica de Dósea *et al.* [2018] foi a que obteve os melhores resultados.

Para a métrica de número de parâmetros (NOP), todas as técnicas apresentaram resultados ruins. Para o sistema SIUS, todas apresentaram um *recall* de 100%, mas taxas muito baixas de precisão. Os piores valores foram os da técnica de Aniche *et al.* [2016]. Para o SIAV, não foi possível gerar os resultados devido à falta de verdadeiros positivos (TP) e falsos negativos (FN), isto é, não existiram métodos com número de parâmetros acima dos limiares para que os desenvolvedores concordassem ou discordassem. Como as técnicas praticamente não apontaram métodos para serem avaliados no SIAV, a análise foi prejudicada para esta métrica.

Após as análises individuais dos resultados em cada métrica, podemos definir que o método que propôs os valores limiares que mais se aproximaram da percepção dos desenvolvedores sobre os problemas de *design* foi a técnica de Dósea *et al.* [2018]. As técnicas de Dósea *et al.* [2016] e Alves *et al.* [2010] também obtiveram alguns bons resultados, principalmente para a métrica LOC. Os piores resultados ficaram com as técnicas de Aniche *et al.* [2016] e Vale & Figueiredo [2015] que apresentaram valores baixos de precisão para praticamente todas as métricas analisadas, apesar dos valores altos de *recall*.

QP2: *Qual a percepção dos desenvolvedores sobre a utilização do papel de design da classe para identificar problemas de design?*

Para fazer uma análise qualitativa dos resultados, foram realizadas algumas perguntas (disponíveis no protocolo) aos desenvolvedores com o objetivo de capturar a percepção dos desenvolvedores sobre a influência do papel de *design* das classes na avaliação dos problemas de *design*. No sistema SIAV, percebemos na Tabela 1, que os valores limiares obtidos com a técnica de Dósea *et al.* [2018] para o papel de *design Action* nas métricas LOC, CC e Eferente foram mais do que o dobro dos limiares encontrados para os outros papéis de *design*. Para a métrica de acoplamento, o desenvolvedor do SIAV explica que *"É comum que sejam encontrados métodos acoplados em camadas Business e nas Actions. Na Action, geralmente são feitas muitas chamadas a business ou a services, no qual são chamadas algumas regras do sistema"*. Em relação a métrica de linhas de código, ele explica que *"É comum que as Actions sejam longas, pois a Action é responsável por receber dados da interface, fazer consultas em outras camadas (como service ou business) para então tratar o que for necessário e depois retornar um direcionamento (foward) ao sistema."* Os relatos exemplificam situações nas quais o desenvolvedor parece considerar o papel de *design* da classe para fazer a avaliação da qualidade do método. Ou seja, o papel de *design* da classe é utilizado pelos desenvolvedores como informação de contexto para definir o valor limiar a ser considerado na avaliação.

Outra descoberta foi a possibilidade de problemas de *design* identificados em métodos associados ao mesmo papel de *design* poderem ser estendidos para outros métodos associados ao mesmo papel de *design* e com métricas similares. O desenvolvedor do SIUS disse que *"Se um método de MBEAN é longo ou complexo, e ele faz o que precisa ser feito para realizar suas tarefas, então outros métodos de MBEAN também terão esses problemas."*.

6. Conclusão e Trabalhos Futuros

Neste trabalho foi conduzido um estudo empírico para avaliar a percepção dos desenvolvedores Web sobre valores limiares obtidos a partir de cinco técnicas de identificação de valores limiares baseados em *benchmark* de sistemas. Os resultados preliminares indicam que a utilização de valores limiares que consideram alguma informação de contexto podem auxiliar a reduzir o número de falsos positivos para o desenvolvedor. Como trabalhos futuros, pretende-se estender o estudo para outros sistemas maiores, com a participação de outros desenvolvedores e empresas diferentes.

Agradecimentos. Esse trabalho é apoiado pelo CNPq (projeto 312153/2016-3).

Referências

Alves, T. L., Ypma, C., and Visser, J. (2010). Deriving metric thresholds from benchmark data. In *Int'l. Conf. on Sofware Maintenance. (ICSM)*, pages 1–10.

Aniche, M., Treude, C., Zaidman, A., Deursen, A. V., and Gerosa, M. A. (2016). Satt: Tailoring code metric thresholds for different software architectures. In *Int. Working Conf. on Source Code Analysis and Manipulation (SCAM)*, pages 41–50.

Dósea, M., Sant'Anna, C., and da Silva, B. C. (2018). How do design decisions affect the distribution of software metrics? In *Proceedings of the 26th Conference on Program Comprehension (ICPC)*, pages 74–85.

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

Ferreira, K. A., Bigonha, M. A., Bigonha, R. S., Mendes, L. F., and Almeida, H. C. (2012). Identifying thresholds for object-oriented software metrics. *Journal of Systems and Software.*, 85(2):244–257.

Fowler, M. and Beck, K. (1999). *Refactoring: improving the design of existing code*. Addison-Wesley Professional, Boston, MA, USA.

Kamei, Y. and Shihab, E. (2016). Defect prediction: Accomplishments and future challenges. In *Int'l. Conf. on Soft. Analysis, Evolution and Reengineering*, pages 33–45.

Lanza, M. and Marinescu, R. (2006). *Object-Oriented Metrics in Practice*. Springer, Berlin, Heidelberg.

Martin, R. and Lippman, S. (2000). *More C++ Gems*. SIGS Reference Library. Cambridge University Press.

McCabe, T. J. (1976). A complexity measure. In *Proc. of the 2nd Int't Conf. on Software Engineering*, Los Alamitos, CA, USA. IEEE Computer Society Press.

Oliveira, P., Valente, M. T., Bergel, A., and Serebrenik, A. (2015). Validating metric thresholds with developers: An early result. In *Int'l. Conf. on Software Maintenance and Evolution (ICSME), Bremen, Germany*, pages 546–550.

Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., and De Lucia, A. (2014). Do they really smell bad? a study on developers' perception of bad code smells. In *Int'l Conf. on Software Maintenance and Evolution (ICSME)*, pages 101–110. IEEE.

Vale, G., Fernandes, E., and Figueiredo, E. (2018). On the proposal and evaluation of a benchmark-based threshold derivation method. *Software Quality Journal*, pages 1–32.

Vale, G. A. D. and Figueiredo, E. M. L. (2015). A method to derive metric thresholds for software product lines. In *2015 29th Braz. Symp. on Soft. Eng.*, pages 110–119.

Zhang, F., Mockus, A., Zou, Y., Khomh, F., and Hassan, A. E. (2013). How does context affect the distribution of software maintainability metrics? In *IEEE Int'l. Conf. on Software Maintenance (ICSM)*, pages 350–359.

Um Estudo Empírico sobre o Impacto dos Pré-processamentos e Normalizações no Cálculo do Acoplamento Conceitual

Paulo Batista da Costa[1], **Igor Wiese**[1], **Igor Steinmacher**[1], **Reginaldo Ré**[1]

[1] Universidade Tecnológica Federal do Paraná - UTFPR
Campus Campo Mourão - PR.

pauloc@alunos.utfpr.edu.br,{igor,igorfs,reginaldo.re}@utfpr.edu.br

Resumo. *O acoplamento é uma das propriedades fundamentais com mais influência sobre a manutenção e evolução do software. Dentre as medidas de acoplamento está o acoplamento conceitual. Na literatura é possível encontrar diversos modelos para cálculo do acoplamento conceitual que combinam pré-processamentos (uso de camelcase, lematização, etc), com diferentes normalizações, tais como, TF-IDF (term frequency–inverse document frequency e o LSI (Latent Semantic Indexing). Este estudo comparou os diferentes modelos de obtenção do acoplamento conceitual entre arquivos de código-fonte. Descobriu-se que a normalização LSI retorna valores mais altos de similaridade entre os artefatos e que os pré-processamentos influenciam de forma diferente as normalizações.*

1. Introdução

A obtenção de similaridade semântica entre arquivos de código-fonte resulta em uma métrica chamada ”acoplamento conceitual”. Essa métrica determina uma relação de interdependência entre arquivos de código-fonte em virtude dos conceitos empregados no desenvolvimento de um projeto de software e é usada com propósitos diferentes, por exemplo, para recomendação de refatoração [Bavota et al. 2011], melhorar a modularidade [Bavota et al. 2010], recomendação de mudanças conjuntas entre artefatos [Kagdi et al. 2013] e uso em técnicas para predizer a ocorrência de defeitos [Kagdi et al. 2013].

Há várias formas de calcular o acoplamento conceitual. Dentre as variações possíveis estão o uso ou descarte do comentário de código-fonte, a remoção ou permanência de termos em relação a frequência que ocorrem, a realização ou não de lematização (do inglês, *stemming*) [Marcus et al. 2008] e o uso de métodos de normalização distintos como o TF-IDF e LSI [Garnier and Garcia 2016, Kagdi et al. 2013]. Um exemplo deste impacto poderia ser observado no uso do *CamelCase*. Esse pré-processamento implica na escrita de palavras compostas, onde cada palavra é iniciada com maiúsculas e unidas sem espaços. Este padrão é utilizado em diversas linguagens de programação como `C++`, `Java`, `Python` e `Ruby` principalmente nas definições de classes e objetos, entretanto, por falta de adoção do padrão um mesmo conceito pode ter sido escrito com alguma variação (*GetCar*, *get_Car*, *getcar*, *Get_Car*) que implicaria no cálculo do acoplamento conceitual.

Apesar da existência de diferentes modelos de cálculo do acoplamento conceitual, não foram encontrados estudos que os comparem em diferentes linguagens de programação. Portanto, o objetivo deste trabalho consiste em comparar como os pré-processamentos e normalizações combinados em diferentes modelos impactam o valor do acoplamento conceitual obtido. Para a realização deste estudo foram selecionados 61 projetos de software livre, hospedados no `github`, sendo que de cada projeto foram coletadas de 3 a 6 versões. Os projetos selecionados são desenvolvidos em `C++`, `Java`, `Javascript`, `Python` e `Ruby`. Deles foram extraídos as relações de acoplamento conceitual nos arquivos de código-fonte e foram comparadas por meio de testes estatísticos.

Este trabalho esta organizado da seguinte forma. Na seção 2 serão apresentados os conceitos e os trabalhos relacionados. Na seção 3, a metodologia executada. Na seção 4 são apresentados os resultados e a discussão. A seção 5 apresenta as conclusões.

2. Referencial Teórico

2.1. Conceitos

Na Tabela 1 estão descritos os conceitos relacionados ao estudo dos modelos de cálculo do acoplamento conceitual. Na primeira coluna, Conceitos, é apresentado o conceito em si, na segunda coluna, Definição, é apresentada uma breve descrição a respeito do conceito e, na terceira coluna, Tipo, é indicado se o conceito é um pré-processamento (PP) ou uma normalização (NM). Além dos conceitos mostrados na tabela, é importante salientar que o termo **modelo** refere-se a combinação de um conjunto de pré-processamentos (PP) combinado com uma forma de normalização (NM). Portanto, o modelo é o que dá a origem a uma forma de cálculo de acoplamento conceitual.

2.2. Trabalhos Relacionados

[Garnier and Garcia 2016], [Antoniol et al. 2000] e [Safeer et al. 2010] utilizaram a normalização TF-IDF para calcular o acoplamento conceitual. Entretanto, eles diferem nos pré-processamentos usados e, principalmente, no propósito (predição de falhas, mapeamento do código e documentação e clusterização de artefatos). Ao contrário, nosso trabalho pretende estudar o cálculo das métricas ao invés de sua aplicação para algum propósito específico.

[Kagdi et al. 2013], [Marcus 2004] e [Lucia et al. 2007] utilizaram o modelo de normalização LSI e também diferentes pré-processamentos. Novamente, os trabalhos usaram o modelo de normalização para propósitos específicos, enquanto [Kagdi et al. 2013] e [Marcus 2004] usaram os seus modelos para prever mudanças conjuntas, [Lucia et al. 2007] construiu uma ferramenta para rastrear artefatos.

3. Metodologia

3.1. Questões de pesquisa

O objetivo desta pesquisa consiste em analisar se o modelo de cálculo do acoplamento conceitual impacta os valores de acoplamento conceitual. As seguintes questões de pesquisa foram investigadas neste trabalho.

QP1: A variação dos parâmetros de pré-processamento influenciam a técnica de normalização usada no modelo de cálculo do acoplamento conceitual?

QP2: É possível determinar o melhor modelo (pré-processamento + técnica de normalização) para cálculo do acoplamento conceitual?

Tabela 1. Definições de Conceitos.

Conceito	Definição	Tipo
Token	Representação de um termo presente em arquivos de código-fonte.	PP
CamelCase	Composição de termos pela junção de duas ou mais palavras com o uso de letras sensíveis ao caso (ou *underscore*).	PP
Tokenização	Processo que divide o código-fonte em um cojunto de termos.	PP
Remoção de quartil	Remover os termos cuja frequência esteja localizada num determinado quartil. Um quartil é a quarta parte de uma amostra, sendo que uma amostra possui quatro quartis [Hyndman and Fan 1996]. Remover o primeiro quartil corresponde a remoção dos 25% termos menos frequentes. Remover o terceiro, corresponde a remoção de 75% dos termos menos frequentes .	PP
Lematização (do inglês, *stemming*)	Reduzir um termo à sua base (forma não flexionada). Exemplo: *Development* ao ser lematizado se torna *Develop*.	PP
TF-IDF	Normalização de frequência dos termos, cujos índices de frequência são proporcionais ao número de aparições de um termo em um arquivo e é condicionado ao número de vezes em que aparece em todos os arquivos da amostra. O índice normalizado para cada termo está condicionado ao número de vezes em que ele ocorre no arquivo e do número total de vezes em que ele ocorre no projeto (todos os arquivos), sendo que quanto mais vezes ele ocorrer no arquivo e no projeto maior será o seu valor normalizado.	NM
LSI	Normalização baseada no princípio que palavras que ocorrem no mesmo contexto (mesmo arquivo) possuem semântica semelhantes. Isso possibilita afirmar a semelhança conceitual entre termos, mesmo que os termos não sejam o mesmo, mas que tenha ocorram no mesmo contexto. O valor do termo normalizado será maior em virtude do número de vezes que ele ocorre no arquivo, além disso ele considera termos que aparecem conjuntamente como similares.	NM
Ingênuo	Utilização das frequências dos termos para o cálculo do acoplamento conceitual sem aplicar qualquer normalização	NM

3.2. Coleta de Dados

A primeira etapa de execução deste estudo é a coleta das versões de alguns software de código aberto. Foram obtidos 61 projetos de software, dos quais 10 são desenvolvidos em `C++`, 11 em `Java`, 12 em `Javascript`, 16 em `Python` e 12 em `Ruby`. Ao todo foram analisadas 299 versões, sendo 49 em `C++`, 50 em `Java`, 55 em `Javascript`, 78 em `Python` e 60 em `Ruby`, sendo que foram analisadas de 3 a 6 versões *major* mais recentes por projeto. As versões *minor* ou *beta* foram descartadas uma vez que ela apresentam pouca variação de código, logo as similaridades entre os arquivos computados pela métrica de acoplamento conceitual deve ser irrelevante. De cada uma das versões dos projetos selecionados foram coletados o código fonte usando o comando `git clone (url do projeto)`.

Nossa amostra final é diversa. Todos os projetos estão hospedados no `github` e foram selecionados porque são ativos, que pode ser observado pela quantidade de *forks* e pela popularidade indicada por meio da quantidade de estrelas, e representam diferentes domínios de aplicação (*frameworks*, ferramentas, etc). Os projetos foram selecionados aleatoriamente dentre os que possuem maiores números de *forks* e estrelas. Dentre eles destacamos Agera Google, Flask, Electron, Angular e a linguagem Ruby.

3.3. Formas de cálculo de Acoplamento Conceitual

Após a coleta das versões, foram executados todos os modelos de obtenção de acoplamento conceitual comparados neste estudo. A Figura 1 representa o fluxo da construção dos modelos de obtenção de valores de acoplamento conceitual e expõem as possíveis variações desse processo. De acordo com a Figura 1, após a leitura dos arquivos de código-fonte, é realizado o processo de *tokenização*, no passo 2. No passo 3, é realizado a remoção de palavras reservadas da linguagem a partir de uma lista. Também são removidos caracteres especiais e números. No passo 4, todo *token* cujo nome é formado por menos do que três caracteres é removido, pois em inglês (idioma no qual os identificadores estão descritos) geralmente as palavras que representam algum conceito possuem três caracteres ou mais. Por exemplo, um identificador de variável com o nome representado pelo *token* 'a' não remete a nenhum conceito, portanto é eliminado nesta etapa.

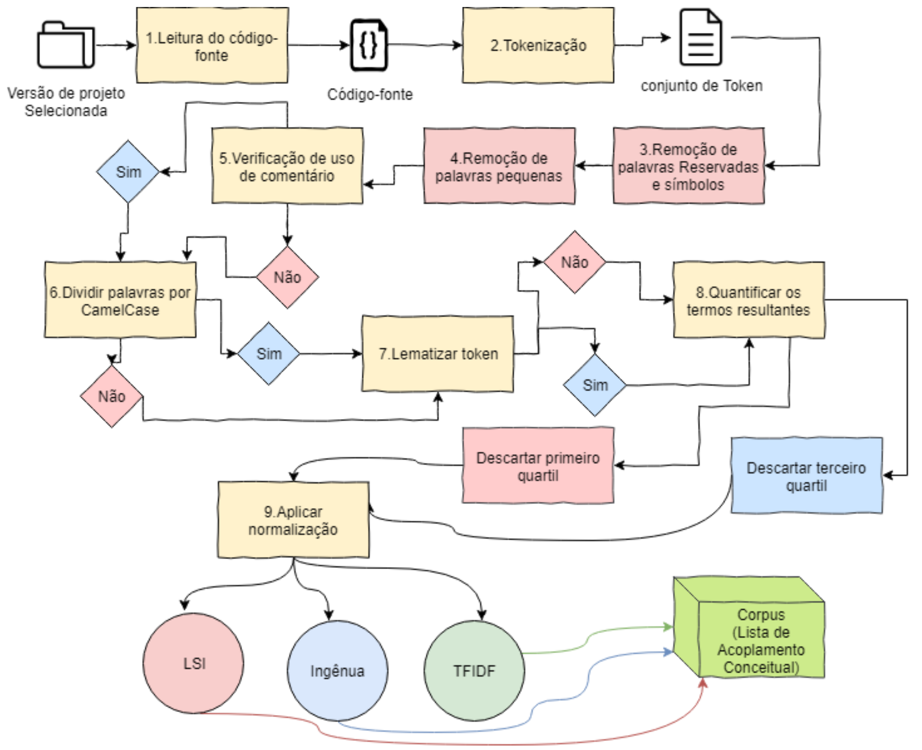


Figura 1. Construção do Corpus para Obtenção de Acoplamento Conceitual

A partir do passo 5 são construídos os modelos comparados neste trabalho. Conforme visto na seção de conceitos, especificamente na Tabela 1, nós comparamos neste trabalho 3 técnicas de normalização (LSI, TF-IDF e ingênuo) com oito combinações de pré-processamento (remoção de comentários de código-fonte, usar comentário de código-fonte, dividir os termos em virtude de *CamelCase*, não dividir os termos por *CamelCase*, lematizar os termos, não lematizar os termos, remoção do primeiro quartil dos termos menos frequentes em arquivos de código-fonte e remoção do terceiro quartil dos termos menos frequentes em arquivos de código-fonte).

É importante mencionar que no passo 8 os termos são quantificados de acordo com a frequência em que ocorrem nos documentos de código-fonte. A partir deste passo, todo documento de código-fonte é representado por um vetor cuja as posições possuem o número de ocorrência para cada termo presente no código-fonte. Em seguida, há a remoção de termos menos frequentes que variam entre o primeiro e o terceiro quartil. Essa remoção é feita para remover os termos menos significantes (termos menos frequentes possuem conceitos menos relevantes para o arquivo), e com ela há a possibilidade de 16 variações do processo de obtenção do acoplamento conceitual.

No passo 9, o último, cada vetor passa pelo processo de normalização, gerando uma matriz onde as linhas e colunas são as combinações par-a-par entre todos os arquivos de código fonte de uma versão com os seus respectivos valores de acoplamento conceitual. Apesar de haverem 16 variações de conjuntos de pré-processamento, para este estudo foram escolhidos 6 configurações apresentadas na tabela 2. Essas configurações foram escolhidas a partir da literatura.

Tabela 2. Combinações de Pré-processamento utilizados neste estudo

Utilizar Comentário	Dividir por *CamelCase*	Lematizar	Remover quartil
sim	sim	sim	remover 3º quartil
sim	sim	não	remover 1º quartil
sim	não	não	remover 3º quartil
não	sim	sim	remover 3º quartil
não	não	sim	remover 1º quartil
não	não	não	remover 1º quartil

3.4. Análise dos Resultados

Primeiramente foi verificado se existe diferença estatística entre as distribuições dos valores de acoplamento conceitual obtidos para pares de arquivos de código fonte usando um modelo de obtenção. Essa verificação foi realizada por meio da aplicação do teste de hipótese *Mann–Whitney U test*. Esse teste foi aplicado para verificar se a distribuição desses valores é diferente para um α = 0.05, α representa o nível de significância. Se o valor-*p* resultante da execução do for menor do que o α, rejeita-se a *hipótese nula*, que neste contexto, pressupõe que as distribuições dos valores de acoplamento gerados entre os pares de arquivos de uma determinada versão são idênticas para os dois modelos comparados.

Após verificar se existe diferença entre as distribuições dos valores de acoplamento conceitual, será aplicado o teste *Cohen's D* (tamanho de efeito), cuja finalidade é dimensionar o tamanho da diferença verificada entre as distribuições avaliadas. Para interpretar o valor do *Cohen's D* obtido, usaremos a escala que indica: ($\delta < 0.1$) insignificante; ($0.1 \leq \delta < 0.2$) muito pequeno; ($0.2 \leq \delta < 0.5$) pequeno; ($0.5 \leq \delta < 0.8$) médio; ($0.8 \leq \delta < 1.20$) grande;($1.20 \leq \delta < 2.0$) muito grande; e, ($\delta \geq 2.0$) enorme. São considerados os melhores modelos aqueles que apresentam o tamanho de efeito maior quando são comparados par a par.

4. Resultados

Foram realizadas 45747 comparações entre diferentes modelos de acoplamento conceitual para responder as questões de pesquisa. Esse número de comparações resulta da comparação par a par de 18 modelos de obtenção de acoplamento conceitual para as 299 versões de software. Para cada versão foram realizadas 153 comparações. Os resultados das duas questões de pesquisa investigadas neste trabalho são apresentados à seguir.

4.1. QP1: A variação dos parâmetros de pré-processamento influencia a técnica de normalização usada no modelo de cálculo do acoplamento conceitual?

Para responder essa questão de pesquisa, primeiramente foram comparados os valores de acoplamento conceitual obtidos entre todos os pares de arquivos de código fonte encontrados em cada versão de um projeto. Nesta QP, as normalizações foram fixadas, variando somente os pré-processamentos listados na Tabela 2. Não foram realizadas variações quanto a ordem da aplicação dos pré-processamentos do texto do código-fonte, fato que será estudado em trabalhos futuros.

De acordo com a Tabela 3, identificou-se para as linguagens `Javascript`, `Python` e `Ruby`, que o melhor conjunto de pré-processamentos é: usar comentário de código-fonte, dividir termos por *CamelCase*, não lematizar, remover termos menos frequentes com valores inferiores ao primeiro quartil. O mesmo conjunto de pré-processamento é o melhor para as linguagens `C++` e `Java` combinadas com as normalizações TF-IDF e ingênua.

No entanto, o mesmo não ocorreu para as linguagens `C++` e `Java` com o uso do LSI. Neste caso, o melhor conjunto de pré-processamento é formado pelo uso de comentário de código, não divisão de termos por *CamelCase*, não utilização da lematização e remoção do terceiro quartil de termos menos frequentes. Dessa forma, há uma maior remoção de termos menos frequentes nos arquivos de código-fonte, além do fato de que os termos permanecem inalterados em virtude de *CamelCase*.

Tabela 3. Melhores pré-processamentos para cada normalização por linguagem

Linguagem	Pré-processamento	Normalização
`Python`, `Javascript`, `Ruby`	Uso de comentário, divisão por *CamelCase*, Sem Lematização, remover primeiro quartil de termos menos frequentes	LSI,TF-IDF,Ingênua
`C++`, `Java`	Uso de comentário, divisão por *CamelCase*, Sem Lematização, remover primeiro quartil de termos menos frequentes	TF-IDF,Ingênua
`C++`, `Java`	Uso de comentário, não divide por *CamelCase*, Sem Lematização, remover terceiro quartil de termos menos frequentes	LSI

Ao comparar os modelos com normalização LSI entre si, verificou-se que em 76,94% das comparações, os valores de acoplamento conceitual tinham um tamanho de efeito muito pequeno. Isso significa dizer que a maioria dos modelos com normalização LSI retorna valores muito semelhantes de acoplamento conceitual, independentemente da linguagem de programação analisada. O mesmo não aconteceu com os modelos TF-IDF e ingênuo. No caso do TF-IDF, 60,28% das comparações retornaram um tamanho de efeito muito pequeno. No modelo ingênuo, o pré-processamento tem uma influência maior, pois 53,93% das comparações apresentaram tamanho de feito muito pequeno.

Dessa forma, é possível afirmar que a variação dos parâmetros de pré-processamento de texto de código-fonte causam diferenças entre os valores de acoplamento conceitual, especialmente quando a normalização usada é o TF-IDF e o modelo ingênuo. Nota-se que para as linguagens `C++` e `Java` os conjuntos dos melhores pré-processamentos em virtude das normalizações são os mesmos entre si e diferentes do que ocorre nas demais linguagens deste estudo. Isto possivelmente está relacionado as diferentes similaridade de sintaxe do código-fonte escrito em `C++` e `Java`.

4.2. QP2: É possível determinar o melhor modelo (pré-processamento + técnica de normalização) para cálculo do acoplamento conceitual?

Para responder esta questão de pesquisa foram comparados todos os modelos construídos entre si em cada versão de cada linguagem.

De acordo com a Tabela 4, pode-se observar que as linguagens `C++` e `Java` possuem a melhor combinação de pré-processamento e normalização em comum. O mesmo pode ser dito entre as linguagens `Javascript`, `Python` e `Ruby`. Em todas as linguagens os modelos baseados em LSI obtiveram o melhor desempenho. Do total de comparações, os melhores modelos obtiveram em 27,55% dos casos uma vantagem com tamanho de efeito muito pequeno, 19,8% com tamanho de efeito pequeno, 14,91% com tamanho de efeito grande e 35,45% com tamanho de efeito muito grande.

Tabela 4. Melhores combinações de pré-processamento e normalização por linguagem

Linguagem	Pré-processamento	Normalização
`C++`, `Java`	Uso de comentário de código-fonte, não dividir em virtude de *CamelCase*, não lematizar termos, remoção do terceiro quartil de termos menos frequentes.	LSI
`Javascript`, `Python`, `Ruby`	Uso de comentário de código-fonte, dividir em virtude de *CamelCase*, não lematizar termos, remoção do primeiro quartil de termos menos frequentes	LSI

Em termos práticos, em relação a linguagem `C++`, o melhor modelo de acoplamento conceitual apresentou maiores similaridades entre os arquivos de código fonte em 6 projetos (27 versões). No caso da linguagem `Java`, o melhor modelo foi superior em 10 projetos (41 versões). Para a linguagem `Javascript`, foram 7 projetos (30 versões). Na linguagem `Python`, o melhor modelo foi superior em 10 projetos (43 versões). Por último, a linguagem `Ruby` apresentou seu melhor modelo com maiores valores de acoplamento em 8 projetos (33 versões).

Conclui-se que de uma forma geral, usar a normalização baseada no LSI fornece valores maiores da similaridade de acoplamento conceitual entre os arquivos de código fonte. Já o pré-processamento depende da linguagem do projeto que se deseja analisar.

5. Conclusões

Este trabalho estudou o impacto da combinação de diferentes pré-processamentos e técnicas de normalização usadas na literatura. Além disso, ao invés de realizar estudos somente aplicando o acoplamento conceitual com arquivos de código fonte da linguagem Java e C++, foram avaliadas três novas linguagens (ruby, python e Javascript).

Os resultados indicam que os pré-processamentos influenciam o cálculo do acoplamento conceitual, e que o LSI é a melhor normalização (RQ2) - independente da linguagem - para se calcular o acoplamento conceitual. Verificou-se também que as linguagens `C++` e `Java` possuem pré-processamentos comuns, assim como as linguagens `Javascript`, `Python` e `Ruby` e que os pré-processamentos influenciam a obtenção do acoplamento conceitual independentemente da normalização usada (RQ1).

Em trabalhos futuros, serão investigadas questões derivadas deste estudo, tais como: A ordem da aplicação dos pré-processamentos afetam os valores de acoplamento conceitual obtidos? Por quais razões certas combinações de pré-processamento são melhores do que outras de acordo com a linguagem, domínio do projeto e características do código-fonte? Além disso, os melhores modelos de obtenção de acoplamento conceitual encontrados neste estudos serão utilizados em tarefas de predição de mudanças conjuntas e predição de defeitos para se comparar os impactos na prática.

Referências

Antoniol, G., Canfora, G., Casazza, G., and Lucia, A. D. (2000). Information retrieval models for recovering traceability links between code and documentation. In *Proceedings 2000 International Conference on Software Maintenance*, pages 40–49.

Bavota, G., De Lucia, A., Marcus, A., and Oliveto, R. (2010). Software re-modularization based on structural and semantic metrics. In *Reverse Engineering (WCRE), 2010 17th Working Conference on*, pages 195–204. IEEE.

Bavota, G., De Lucia, A., and Oliveto, R. (2011). Identifying extract class refactoring opportunities using structural and semantic cohesion measures. *Journal of Systems and Software*, 84(3):397–414.

Garnier, M. and Garcia, A. (2016). On the evaluation of structured information retrieval-based bug localization on 20 c# projects. In *Proceedings of the 30th Brazilian Symposium on Software Engineering*, pages 123–132. ACM.

Hyndman, R. J. and Fan, Y. (1996). Sample quantiles in statistical packages. *The American Statistician*, 50(4):361–365.

Kagdi, H., Gethers, M., and Poshyvanyk, D. (2013). Integrating conceptual and logical couplings for change impact analysis in software. *Empirical Software Engineering*, 18(5):933–969.

Lucia, A. D., Fasano, F., Oliveto, R., and Tortora, G. (2007). Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(4):13.

Marcus, A. (2004). Semantic driven program analysis. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 469–473. IEEE.

Marcus, A., Poshyvanyk, D., and Ferenc, R. (2008). Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Transactions on Software Engineering*, 34(2):287–300.

Safeer, Y., Mustafa, A., and Ali, A. N. (2010). Clustering unstructured data (flat files)-an implementation in text mining tool. *arXiv preprint arXiv:1007.4324*.

Violação de padrões de uso de APIs em sistemas configuráveis

Bruno Mecca[1], Diogo Boaventura[1], Bruno B. P. Cafeo[1], Elder Cirilo[2]

[1] Faculdade de Computação – Universidade Federal de Mato Grosso do Sul (UFMS)
Cidade Universitária – 79070-900 – Campo Grande – MS

[2]Departamento de Ciência da Computação – Universidade Federal de São João del-Rei (UFSJ)
Campus Tancredo de Almeida Neves (CTAN) – 36301-360 – São João del-Rei – MG

`{bruno.mecca,diogo.b.fonseca}@aluno.ufms.br,`
`cafeo@facom.ufms.br, elder@ufsj.edu.br`

Resumo. *API (Application Programming Interface) tem sido vastamente adotada no desenvolvimento de software. Com isso, é comum que funções de API sejam utilizadas em diferentes contextos, assim como também frequentemente aplicadas de maneira conjunta de modo que seus usos seguem algumas regras ou padrões de uso. No entanto, não se sabe se violações nos padrões de uso ocorrem em sistemas implementados com diretivas de pré-processamento. Dessa forma, o objetivo deste estudo é verificar se ocorrem violações de padrões de uso em sistemas configuráveis devido a diretivas de pré-processamento. Para isso, foram extraídos padrões de uso de quatro sistemas configuráveis de código aberto: totalizando 253.589 linhas de código analisadas e mais de 9.848 commits. Após a identificação dos padrões, foram analisadas possíveis violações de uso. Como principais contribuições, pode-se destacar um primeiro estudo que mostra que violações de padrões de uso de APIs podem ocorrer devido à diretivas de pré-processamento.*

1. Introdução

API (*Application Programming Interface*) é uma importante forma de reúso que tem sido vastamente adotada no desenvolvimento de software [Zhong et al. 2009]. É comum que uma função de API seja utilizada em diferentes contextos ao longo do ciclo de vida de software, independentemente se a API é pública (ex., *Java Development Kit* e *C Standard Library*) ou privada (funções internas do código). Algumas funções de API são frequentemente utilizadas de maneira conjunta e seus usos seguem algumas regras que o desenvolvedor aplica para a implementação de funcionalidades. A combinação de funções de API que geralmente são invocadas conjuntamente é chamada de padrão de uso [Zhong et al. 2009]. Exemplos ilustrativos de padrões de uso de APIs citados em diversos estudos da área são: *lock()* e *unlock()*, *open()* e *close()*, e *start()* e *stop()*.

Sistemas configuráveis, assim como a maioria dos sistemas atuais, também fazem uso de APIs. Sistemas configuráveis compartilham um núcleo comum e também possuem diferentes funcionalidades em suas configurações resultantes [Apel et al. 2013]. Quando consideramos sistemas configuráveis implementados em linguagem C, desenvolvedores comumente utilizam a técnica de pré-processamento para anotar a implementação correspondente à variabilidade (ou opção de configuração) devido à sua flexibilidade e expressividade [Apel and Kästner 2009]. O pré-processador identifica o código que deve ser compilado ou não baseado em diretivas de pré-processamento, tais como `#ifdef`. Desenvolvedores usam as diretivas de pré-processamento para envolver desde estruturas inteiras de código (ex., funções) até uma simples variável.

Vários estudos criticam o uso de pré-processamento devido aos seus efeitos negativos no entendimento de código, manutenibilidade e propensão a erros [Ernst et al. 2002, Medeiros et al. 2015]. Além disso, diversos trabalhos exploram a identificação de padrões de uso de APIs [Li and Zhou 2005, Wang and Godfrey 2013]. No entanto, não existem trabalhos que explorem padrões de uso no contexto de sistemas configuráveis implementados com pré-processadores. Mais especificamente, não se sabe, devido à complexidade inerente de códigos com diretivas de pré-processamento, se padrões de uso de APIs podem ser violados.

Este trabalho tem por objetivo verificar violações de padrões de uso de APIs (internas e externas) em sistemas configuráveis implementados com diretivas de pré-processamento. Para isso, foram extraídos padrões de uso utilizando uma adaptação das técnicas [Li and Zhou 2005]. Após a identificação dos padrões, buscou-se possíveis violações de uso devido às diretivas de pré-processamento. Quatro sistemas configuráveis de código aberto foram analisados, totalizando 253,589 linhas de código e mais de 9,848 commits. Como principais contribuições, pode-se destacar uma lista de padrões de uso de APIs para os sistemas analisados, bem como um primeiro estudo que mostra que violações de padrões de uso de APIs podem ocorrer devido ao uso de diretivas de pré-processamento.

2. Problema Motivacional

Nessa seção são apresentados os conceitos básicos (Seção 2.1) para o entendimento do problema abordado no estudo exploratório (Seção 2.2).

2.1. Fundamentação Teórica

O pré-processamento é usado em muitos sistemas para suportar a implementação de variabilidades. Desenvolvedores comumente utilizam diretivas de pré-processamento, tais como `#ifdef` e `#endif`, para anotar blocos de código-fonte como condicionais. O pré-processador identifica o código que deve ser compilado ou não com base em diretivas de pré-processamento que envolvem estruturas de código associando-as a uma variabilidade. Na Figura 1 observam-se fragmentos de código do sistema *Hexchat* cercados por diretivas de pré-processamento. As instruções `#ifdef` e `#endif` definem as fronteiras de parte da implementação da variabilidade e indicam ao pré-processador o que incluir no processo de compilação. Dessa forma, as instruções entre as linhas 03 e 04 só serão compiladas se a variabilidade `USE_PLUGIN` estiver definida. Por fim, o código da linha 09 só será compilado se a variabilidade `USE_DBUS` estiver definida.

```
01.     static void irc_init (session *sess){
02.        #ifdef USE_PLUGIN
03.           if (!arg_skip_plugins)
04.              plugin_auto_load (sess); /* autoload ~/.xchat *.so */
05.        #endif
06.
07.        #ifdef USE_DBUS
08.           plugin_add (sess, NULL, NULL, dbus_plugin_init, NULL, NULL, FALSE);
09.        #endif
10.
11.        snprintf (buf, sizeof (buf), "%s/%s", get_xdir_fs (), "startup.txt");
12.        load_perform_file (sess, buf);
13.     }
```

Figura 1. Violação de padrão de uso na presença de variabilidades.

2.2. Descrição do Problema

Ao longo da implementação de sistemas configuráveis, desenvolvedores comumente utilizam APIs para fazer uso de funcionalidades já implementadas internamente ou em bibliotecas externas. Muitas vezes restrições implícitas podem existir quando se faz uso de funções de APIs. Isto é, o desenvolvedor deve obedecer tais restrições, comumente chamadas de padrões de uso de APIs, para garantir a execução adequada das funcionalidades desejadas no sistema. Padrões de uso de APIs geralmente são caracterizadas por pares de chamadas de funções de APIs que são utilizadas de maneira conjunta [Li and Zhou 2005]. Na Figura 1 podemos notar as chamadas das funções *plugin_auto_load* e *plugin_add*. As funções de API *plugin_auto_load* e *plugin_add* são consideradas padrões de uso, pois comumente estão relacionadas no código-fonte para a implementação de uma ou mais funcionalidades.

Pré-processadores são populares por conta da flexibilidade e simplicidade de uso, além de estarem incorporados em muitas linguagens de programação (ex.: C, C++, Java Micro Edition). No entanto, vários estudos apontam para uma crescente complexidade e ofuscamento de código, o chamado *#ifdef hell* [Apel and Kästner 2009]. Tal situação pode levar à uma dificuldade de entendimento e altos custos de manutenção [Medeiros et al. 2015]. Nesse contexto, esse trabalho supõe que padrões de uso de APIs podem ser violados devido à complexidade inerente de código com diretivas de pré-processamento. Na Figura 1 podemos notar que as chamadas da função de API *plugin_auto_load* ocorre no contexto da variabilidade `USE_PLUGIN` e a chamada da função de API *plugin_add* ocorre no contexto da variabilidade `USE_DBUS`. Caso uma das variabilidades não seja selecionada para compor a configuração final do software, podemos ter um problema devido à violação de um padrão de uso. Ou seja, não haverá violação do padrão de uso apenas se uma configuração contiver ambos `USE_PLUGIN` e `USE_DBUS` na configuração final.

3. Estudo Exploratório

Esta seção descreve o estudo conduzido nesse artigo. Primeiramente é apresentado o projeto do estudo e, posteriormente, os resultados e discussões.

3.1. Projeto do Estudo

Para avaliar as violações de padrões de uso de APIs foram considerados quatro sistemas configuráveis de código aberto implementados em linguagem C com diretivas de pré-processamento. Os sistemas, disponíveis em repositórios Git, são de diferentes domínios, tamanho e idade. A seleção desse conjunto baseou-se em trabalhos anteriores que exploram sistemas configuráveis [Medeiros et al. 2017]. A Tabela 1 apresenta os sistemas configuráveis analisados, bem como o número de commits, a idade em anos, o número médio de linhas de código, o número médio de funções, o número médio de chamadas de funções ao longo dos anos e o número médio de variabilidades.

Para realizar o estudo, foram executados três principais atividades: (i) mineração e extração de dados, (ii) identificação de padrões de uso de APIs e (iii) análise de possíveis violações de padrões de uso. Cada uma das atividades é detalhada a seguir.

Mineração e extração de dados. Inicialmente analisou-se o repositório dos projetos selecionados com o propósito de extrair chamadas de funções de API (internas e externas). Para isso, implementou-se uma ferramenta que recupera cada *commit* armazenado no repositório. Posteriormente a ferramenta cria uma Árvore Sintática Abstrata (AST) de

Tabela 1. Conjunto de sistemas analisados

Sistema	# Commits	Idade	LOC	Funções	Chamadas de Funções	Variabilidades
Hexchat	3380	8	69988	1672,43	6101,33	17
Lighttpd	2568	13	53284	726,35	3568,23	31
Mpsolve	1674	5	46967	574,52	4092,55	10
OpenVPN	2226	11	83350	500,51	1240,03	17

cada arquivo em cada *commit* e preserva todas as informações de variabilidade, tendo em cada nó da AST uma macro associada à diretiva de pré-processamento. Dessa forma, identificaram-se as chamadas de funções e as variabilidades em que ocorreram tais chamadas.

Identificação de padrões de uso de APIs. Para identificar os pares de funções com a mesma frequência de chamadas, ou seja, padrões de uso não-ordenado de funções do tipo *push-pop*, aplicou-se o algoritmo APRIORI. O algoritmo prediz regras de associação entre itens genéricos baseado na ocorrência dos itens em uma base de ocorrências. Utilizou-se como base de ocorrência o histórico de chamadas das funções para cada commit analisado. Considerou-se como padrões relevantes aqueles com alto nível de confidência e baixo suporte, conforme apresentado na Figura 2.

Análise de possíveis violações de padrões de uso. Com os padrões de uso mais relevantes, verificou-se os locais das chamadas das funções dos padrões identificados. É importante analisar o local das chamadas, pois caso as funções de um padrão sejam chamadas em variabilidades diferentes, há a possibilidade de se gerar configurações do sistema em que o padrão seja violado. Dessa forma, para cada par, foram armazenadas as variabilidades em que ocorreram as chamadas. Caso ao menos uma variabilidade fosse diferente entre o par, considerou-se como uma possível violação do padrão de uso. Além disso, uma análise manual foi feita nas violações detectadas para evitar possíveis ocasionalidades de padrões e de violações.

3.2. Resultados

A análise de possíveis violações de padrões de uso resultou na descoberta de 105 possíveis cenários de violações nos sistemas configuráveis. A Tabela 2 apresenta para cada sistema configurável, o número de padrões onde pelo menos uma de suas funções ocorre no

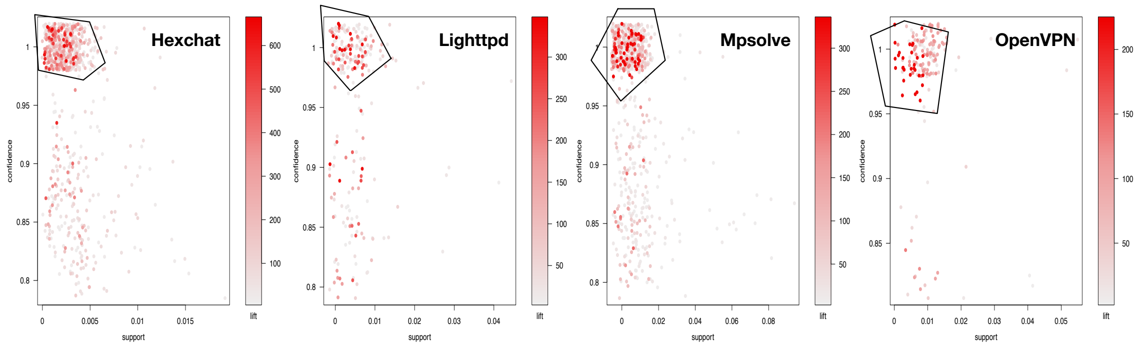


Figura 2. Valores de suporte e confianças aplicados para balancear a relevância dos padrões em cada sistema configurável.

contexto de variabilidades (ou seja, chamadas em código de variabilidade) e o número total de padrões identificados entre parentesis; o número variabilidades com padrões de uso de APIs e o número entre parentesis é o número de variabilidades de cada sistema; a porcentagem de padrões em variabilidades; e a porcentagem de variabilidades que possuem funções que fazem parte de padrões de uso.

Pelos dados apresentados, percebe-se que apenas um pequeno número de padrões de uso de API ocorrem no contexto de variabilidades (aprox. 7,3%). Ou seja, a maioria dos padrões de uso de APIs acontecem em código que não possui diretivas de pré-processamento associadas. Esse é uma observação interessante, pois a maioria dos padrões de uso não estão relacionados com variabilidades. Um contraponto é o sistema *OpenVPN* que possui 26,2% dos padrões de uso identificados ocorrendo no contexto de variabilidades. É interessante observar que o *OpenVPN* é um dos sistemas com o menor número variabilidades (Tabela 1). Dessa forma, acredita-se que existirá uma grande variação no número de padrões relacionados à variabilidades dependendo de diversos fatores do sistema e, por consequência, uma variação no possível número de violação de padrões devido ao uso de diretivas de pré-processamento. Uma análise mais extensa com vários sistemas de diferentes domínios pode ajudar a entender a relação entre número de variabilidades, trechos de código com compilação condicional e padrões de uso de APIs.

Um outro ponto a ser destacado é o alto número de variabilidades com chamadas à funções que fazem parte de algum padrão de uso de API com relação ao número total de variabilidades. No geral, padrões de uso ocorrem em 58,3% das variabilidades. Quanto maior o número de variabilidades com chamadas à funções participantes de padrões de API, maior a chance de violação de padrões. Por meio de uma análise manual e mais detalhada por alguns padrões, acredita-se que este resultado é correlacionado ao grau de espalhamento da implementação das variabilidades. Ou seja, quanto maior o espalhamento na implementação das variabilidades, maior a chance de envolver a chamada de alguma função participante de um padrão. Como consequência, acredita-se que haja uma maior chance de violação de padrões de API devido ao aumento no número de possíveis produtos gerados pelo sistema configurável e pelo maior ofuscamento de código devido ao espalhamento.

Tabela 2. Dados extraídos dos sistemas analisados.

Sistema	# Padrões em variabilidades	# Variabilidades com padrões	% de padrões em variabilidades	% de variabilidades com padrões
Hexchat	18 (509)	3 (17)	3,5	17,6
Lighttpd	41 (253)	24 (31)	16,2	77,4
Mpsolve	13 (551)	5 (10)	2,3	50,0
OpenVPN	33 (126)	15 (17)	26,2	88,2
Média	26,2 (331,2)	11,7 (18,7)	12,0	58,3

Complementarmente, em análise manual realizada sob cada possível cenário de violação identificado, observou-se que algumas violações se confirmam e permanecem no código por um longo número de commits. Por exemplo, observou-se uma possível violação no padrão de uso (*lua_islightuserdata* ⇒ *lua_touserdata*) no commit `fe6b720`[1]. Esta mesma violação permaneceu no código até o commit `4184c38`[2], o que corresponde a um espaço de tempo de 120 meses. No geral, em média, as possíveis violações de

[1]github.com/lighttpd/lighttpd1.4/commit/fe6b720
[2]github.com/lighttpd/lighttpd1.4/commit/4184c38

padrões que ocorrem no contexto de variabilidades permaneceram no código por 454 commits.

O estudo revela um relevante problema no desenvolvimento de sistemas configuráveis ainda não explorado. A partir desse problema podemos ressaltar as seguintes implicações imediatas que podem ser alvo de estudos futuros:

- **Identificação de padrões de uso de APIs em sistemas configuráveis:** Vários trabalhos abordam a identificação de padrões de uso de APIs. No entanto, não existem trabalhos que explorem essa identificação em sistemas configuráveis implementados com diretivas de pré-processamento. Apesar de uma leve tendência nos dados encontrados, notam-se variações em alguns sistemas da tendência comum. Acredita-se que a diferença na codificação de sistemas configuráveis implique em diferenças nos padrões de APIs identificados. Ademais, uma identificação de padrões de APIs internas se faz necessário devido ao impacto de uma violação de um padrão em diversas configurações.
- **Detecção de violação de padrões de uso de APIs:** Inúmeras configurações de um sistema podem conter violações no uso de APIs. Com isso, surge a necessidade pelo desenvolvimento de novas técnicas para evitar violações no uso de APIs. No contexto de sistemas configuráveis, isso se torna um desafio haja vista a explosão no número de possíveis configurações a medida que o número de variabilidades aumenta com a evolução natural do sistema. É necessário explorar técnicas de análises estática e dinâmica para auxiliar na detecção de violação de padrões de uso de APIs já conhecidos de maneira escalável.
- **Análise das razões da violação e de eventuais correções de violações:** Pelos dados extraídos no estudo, notou-se uma grande quantidade de commits entre a violação de alguns padrões e sua eventual correção. Foge do escopo deste trabalho, mas é percebido que algumas das violações parecem ocorrer devido à existência das diretivas de pré-processamento. O desenvolvedor não consegue abstrair no momento da codificação quais as possíveis combinações de configurações e quais as possíveis violações que ocorreriam em cada combinação. Um outro ponto interessante para ser explorado é a análise das correções realizadas para evitar as violações. Tanto a análise da causa quanto a correção realizada são de suma importância para auxiliar o desenvolvedor na codificação e para a criação de ferramentas de apoio.

4. Ameaças à validade

Existem pelo menos três ameaças à validade dos nossos resultados. Primeiro, não pode-se afirmar que nossas conclusões são mantidas quando se consideram outras APIs externas, outros domínios de aplicações (por exemplo, sistemas que não são configuráveis) ou outras linguagens de programação. Para minimizar tais riscos argumenta-se que os sistemas utilizam diversas APIs externas, além de um número considerável de APIs internas. Os sistemas configuráveis são de diferentes domínios e de diferentes tamanhos. Por fim, utiliza-se a linguagem C com pré-processadores que é considerada a mais empregada para a implementação de variabilidades em sistemas industriais [Apel and Kästner 2009]. Segundo, a escolha dos valores de suporte e confiança foram subjetivas, já que comumente não existem valores recomendados para o contexto do estudo abordado neste trabalho. Para controlar essa ameaça foram testados vários valores de suporte e confiança para balancear cobertura e representatividade dos padrões encontrados. Terceiro, a extração de padrões de uso de APIs e a análise de violações dos padrões foram feitas nos mesmos sistemas. Dessa forma, acredita-se que algumas violações de padrões podem ter sido removidas ao longo do desenvolvimento. Ou seja, o número de violações de padrões pode ser maior que o detectado nesse estudo. Para minimizar essa ameaça, foram analisados todos os commits ao longo da evolução e identificadas possíveis violações corrigidas ao longo da evolução dos sistemas.

5. Trabalhos relacionados

APIs e padrões de uso. Recentemente diferentes aspectos sobre uso de APIs ganharam atenção. Alguns estudos exploram os obstáculos encontrados no uso de APIs [Wang and Godfrey 2013]. Para isso a maioria dos trabalhos utiliza sites de Q&A (ex., Stack Overflow) como base para o entendimento de problemas encontrados com o uso de APIs. A maioria dos trabalhos, que está mais relacionada com este trabalho, foca na mineração de padrões de uso de APIs [Li and Zhou 2005, Borges and Valente 2015]. Tais trabalhos adotam diferentes categorias de padrões de uso, diferentes técnicas para inferência de padrões e diferentes formas de avaliar os padrões. As categorias que mais se destacam são: temporais, não-ordenadas e ordenadas. Diferentemente destes trabalhos, não focamos na melhoria do uso de APIs e nem na proposta de um nova técnica para detecção de padrões de uso de APIs. Neste trabalhos utilizamos uma variação do padrão não-ordenado *push-pop* para a detecção de padrões de uso. Além disso, nenhum dos trabalhos anteriores utiliza sistema configuráveis como foco principal do trabalho.

Sistemas configuráveis. Desenvolvedores devem considerar múltiplas configurações quando executam testes ou realizam análise estática em sistemas configuráveis. Como o espaço de configuração geralmente cresce exponencialmente com o número de variabilidades, várias abordagens são propostas para analisar problemas específicos em sistemas configuráveis [Tartler et al. 2012b, Thüm et al. 2014]. Além disso, diferentes técnicas são utilizadas para realizar a análise considerando variabilidades (do inglês, *variability-aware analysis*) como, por exemplo, *t-wise* [Perrouin et al. 2010], *statement-coverage* [Tartler et al. 2012a] e *one-disabled* [Abal et al. 2014]. Neste trabalho, não focamos na proposta de técnicas ou melhorias de abordagens que tratam a explosão combinatorial de configurações em sistemas configuráveis. Além disso, nenhum desses trabalhos abordam problemas nos quais variabilidades afetam os padrões de uso de APIs. Neste trabalho abordamos violações de padrões de uso nas inúmeras configurações possíveis de um sistema configurável.

6. Conclusões

Este artigo apresentou um estudo sobre violações de padrões de uso de APIs em sistemas configuráveis implementados com diretivas de pré-processamento. O estudo considerou a análise de quatros sistemas configuráveis de código aberto: totalizando 253.589 linhas de código analisadas e mais de 9.848 commits. Os resultados do estudo indicam que a suspeita inicial se confirma: a complexidade inerente de códigos com diretivas de pré-processamento pode levar os desenvolvedores a violar os padrões de uso de APIs, muito provavelmente de maneira inconsciente. Neste contexto, entende-se que é necessária uma identificação de padrões de uso de APIs no contexto de sistema, pois ferramentas de auxílio já existentes não estão sendo utilizadas ou não são suficientes para apoiar desenvolvedores de sistemas configuráveis. Um outro ponto importante é o desenvolvimento de abordagens que possibilitem a detecção de possíveis violações em sistemas configuráveis de maneira escalável. Por fim, é necessário um maior entendimento sobre as causas das eventuais inserções de violações (relacionadas ou não a diretivas de pré-processamento), bem como as soluções dadas pelos desenvolvedores para corrigir eventuais violações.

Agradecimentos

O autor Bruno Mecca agradece o apoio do PIBIC/UFMS para a execução deste trabalho.

Referências

Abal, I., Brabrand, C., and Wasowski, A. (2014). 42 variability bugs in the linux kernel: A qualitative analysis. In *29th ACM/IEEE International Conference on Automated Software Engineering (ASE)*, pages 421–432.

Apel, S., Batory, D., Kästner, C., and Saake, G. (2013). *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer Publishing Company.

Apel, S. and Kästner, C. (2009). Virtual separation of concerns - a second chance for preprocessors. *Journal of Object Technology*, 8(6):59–78.

Borges, H. S. and Valente, M. T. (2015). Mining usage patterns for the android api. *PeerJ Computer Science*, 1:1–12.

Ernst, M. D., Badros, G. J., and Notkin, D. (2002). An empirical analysis of c preprocessor use. *IEEE Trans. Softw. Eng.*, 28(12):1146–1170.

Li, Z. and Zhou, Y. (2005). Pr-miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *13th International Symposium on Foundations of Software Engineering (FSE)*, pages 306–315.

Medeiros, F., Kästner, C., Ribeiro, M., Nadi, S., and Gheyi, R. (2015). The Love/Hate Relationship with the C Preprocessor: An Interview Study. In *29th European Conference on Object-Oriented Programming (ECOOP 2015)*, pages 495–518.

Medeiros, F., Ribeiro, M., Gheyi, R., Apel, S., Kastner, C., Ferreira, B., Carvalho, L., and Fonseca, B. (2017). Discipline matters: Refactoring of preprocessor directives in the #ifdef hell. *Transactions on Software Engineering*, 44(5):453–469.

Perrouin, G., Sen, S., Klein, J., Baudry, B., and l. Traon, Y. (2010). Automated and scalable t-wise test case generation strategies for software product lines. In *3rd International Conference on Software Testing, Verification and Validation (ICST)*, pages 459–468.

Tartler, R., Lohmann, D., Dietrich, C., Egger, C., and Sincero, J. (2012a). Configuration coverage in the analysis of large-scale system software. *SIGOPS Oper. Syst. Rev.*, 45(3):10–14.

Tartler, R., Sincero, J., Dietrich, C., Schröder-Preikschat, W., and Lohmann, D. (2012b). Revealing and repairing configuration inconsistencies in large-scale system software. *International Journal on Software Tools for Technology Transfer*, 14(5):531–551.

Thüm, T., Apel, S., Kästner, C., Schaefer, I., and Saake, G. (2014). A classification and survey of analysis strategies for software product lines. *ACM Comput. Surv.*, 47(1):1–45.

Wang, W. and Godfrey, M. W. (2013). Detecting API usage obstacles: A study of ios and Android developer questions. In *10th Working Conference on Mining Software Repositories (MSR)*, pages 61–64.

Zhong, H., Xie, T., Zhang, L., Pei, J., and Mei, H. (2009). Mapo: Mining and recommending API usage patterns. In *23rd European Conference on Object-Oriented Programming (ECOOP)*, pages 318–343.

Avaliação da Frequência de Mudanças em Dependências entre Variabilidades em Sistemas Configuráveis

Raiza A. Oliveira[1], Bruno Mecca[1], Bruno B. P. Cafeo[1], André Hora[1]

[1]Faculdade de Computação
Universidade Federal de Mato Grosso do Sul (UFMS)
Caixa Postal 43017-6221 – Campo Grande – MS – Brasil

{raiza.a.oliveira,bruno.mecca}@aluno.ufms.br,{cafeo,hora}@facom.ufms.br

Abstract. *Configurable systems enable mass customization by exploiting similarities between members of the program family. Approaches and tools to support source code maintenance in the presence of features dependencies are often proposed. However, few studies deal with the evolution of dependencies. This way, this work analyzes how often dependencies changes in terms of release and commit and what these changes are. The results show a considerable frequency of changes to each commit and many changes to the code that implements the dependencies.*

Resumo. *Sistemas configuráveis possibilitam a customização em massa, explorando semelhanças entre os membros da família de software. Abordagens e ferramentas para apoiar a manutenção de código na presença de dependências entre variabilidades são frequentemente propostas. No entanto, poucos trabalhos abordam sobre a evolução das dependências. Dessa forma, esse trabalho analisa a frequência de mudanças das dependências em termos de release e commit e quais são essas mudanças. Os resultados mostram uma frequência considerável de mudanças a cada commit e muitas mudanças no código que implementa as dependências.*

1. Introdução

Sistemas de software vêm se tornando configuráveis para atender demandas constantes em segmentos de mercado distintos [Clements and Northrop 2002]. Opções de configurações, também conhecidas como variabilidades [Apel et al. 2013], envolvem desde pequenos trechos de código até módulos completos. Sistemas que se utilizam de variabilidades para implementar tais funcionalidades são chamados de sistemas configuráveis [Apel et al. 2013].

Em nível de código-fonte, as variabilidades podem se relacionar por meio do compartilhamento de elementos de programa, como funções e variáveis. Por exemplo, uma variável definida em uma variabilidade, e usada em outra. Essas relações são chamadas de dependências entre variabilidades [Cafeo et al. 2016a]. Pode-se dizer que uma dependência entre variabilidades ocorre sempre que um ou mais elementos de programa dentro dos limites de uma variabilidade dependem de elementos externos a essa variabilidade.

Em um cenário de evolução de software, mudanças ocorrem para acomodar novas funcionalidades, corrigir erros, entre outros. Em sistemas configuráveis, as dependências dificultam a manutenção, pois o desenvolvedor precisa garantir que uma mudança não afetará a consistência das variabilidades já existentes. Para isso, baseia-se nas dependências. O problema é que, no código-fonte, variabilidades geralmente não estão contidas em um único módulo, e sim espalhadas em fragmentos de código [Cafeo et al. 2016a]. Dessa forma, desenvolvedores devem analisar minuciosamente o código-fonte para realizar mudanças.

Diversos trabalhos discutem soluções de apoio à manutenção em sistemas configuráveis na presença de dependências entre variabilidades [Cafeo et al. 2016b, Ribeiro et al. 2014, Schröter et al. 2014]. No entanto, ainda faltam estudos que analisem a frequência com que mudanças em dependências ocorrem ao longo da evolução. Mais especificamente, não se sabe a frequência com que tais mudanças ocorrem. É importante saber essa frequência, para avaliar a quantidade de informação perdida ao se analisar *releases* ao invés de *commits*. Para tal, este trabalho busca responder as seguintes questões de pesquisa:

QP1: Mudanças em dependências devem ser analisadas em termos de *releases* ou *commits*?

QP2: Qual a categoria (incluída, excluída ou preservada) de mudanças na implementação de dependências entre variabilidades ocorre com mais frequência?

Para responder tais questões, realiza-se uma análise de todo o histórico de evolução de 6 sistemas configuráveis hospedados no GitHub[1], em termos de *releases* e *commits*. Dessa forma, as principais contribuições deste trabalho são prover: *(i)* dados sobre dependências entre variabilidades que revelam frequência e intensidade de mudanças na evolução de sistemas configuráveis para guiar trabalhos futuros, e *(ii)* bases para o desenvolvimento de ferramentas que apoiem a manutenção de sistemas configuráveis na presença de dependências entre variabilidades, e *(iii)* uma estratégia de identificação de dependências entre variabilidades e mudanças de granularidade fina em tais dependências baseada na ferramenta TypeChef[2].

O restante deste artigo está organizado conforme descrito a seguir. Na Seção 2 são apresentados os conceitos básicos abordados nesse artigo. Na Seção 3 é descrita a metodologia proposta. Na Seção 4 são apresentados os resultados obtidos. As implicações dos resultados são apresentadas na Seção 5. Na Seção 6 são apresentados os riscos à validade deste trabalho. Os trabalhos relacionados são apresentados na Seção 7. Por fim, na Seção 8 apresenta-se as conclusões deste trabalho e os possíveis desdobramentos dos resultados em outros estudos.

2. Fundamentação Teórica

Sistemas configuráveis são usados em diferentes domínios de aplicação, por proverem flexibilidade ao software. Por exemplo, um sistema operacional para dispositivo móvel que compartilha um conjunto comum de funcionalidades (ex. calendário), mas varia em outras (ex. resolução de tela) dependendo de vários fatores, tais como hardware utilizado ou preferências de usuário. As configurações do sistema são formadas por meio da combinação de um núcleo comum com funcionalidades específicas, chamadas de variabilidades [Apel et al. 2013].

Segundo Apel et al., sistemas configuráveis são comumente implementados por meio de compilação condicional [Apel et al. 2013][3]. Nessa abordagem o pré-processador

[1]https://github.com/

[2]https://github.com/ckaestne/TypeChef

[3]Está fora do escopo deste trabalho avaliar outras abordagens de implementação como, por exemplo,

identifica o código que deve ser compilado ou não com base em diretivas de pré-processamento (ex. #ifdef) que envolvem estruturas de código associando-as à uma variabilidade. Portanto, no nível de código-fonte, uma variabilidade é um conjunto de elementos de programa (i.e., variáveis ou funções) cercados por diretivas de pré-processamento [Cafeo et al. 2016a, Kästner et al. 2008].

Observa-se na Figura 1 um fragmento de código extraído do Hexchat[4], em que é possível observar exemplos de implementação de variabilidades. As instruções `#ifdef` e `#endif` definem as fronteiras de partes da implementação das variabilidades e indicam ao pré-processador o que incluir no processo de compilação. Dessa forma, as instruções entre as Linhas 2 e 15 (arquivo hexchat.c) só serão compiladas se a variabilidade `USE_LIBPROXY` estiver definida.

```
1  #ifdef !def(USE_LIBPROXY)
2  static void irc_init (session *sess){
3  #ifdef USE_PLUGIN
4      if (!arg_skip_plugins)
5          plugin_auto_load (sess);    /* autoload ~/.xchat *.so */
6  #endif
7  #ifdef USE_DBUS
8      plugin_add (sess, NULL, NULL, dbus_plugin_init, NULL, NULL,
9  FALSE);
10 #endif
11     snprintf (buf, sizeof (buf), "%s/%s", get_xdir_fs (), "startup.txt");
12     load_perform_file (sess, buf);
13 }
14 . . .
15 #endif
```

Arquivo hexchat.c

```
1  #ifdef !def(WIN32)
2  void gtkutil_file_req (const char *title, void *callback, void
3  *userdata,  char *filter, char *extensions, int flags){
4      struct file_req *freq;
5      GtkWidget *dialog;
6      GtkFileFilter *filefilter;
7      extern char *get_xdir_fs (void);
8      char *token;
9      . . .
10     freq->userdata = userdata;
11     freq->title = g_locale_from_utf8 (title, -1, 0, 0, 0);
12     . . .
13     thread_start (freq->th, win32_thread, freq);
14 . . .
15 #endif
```

Arquivo gtkutils.c

Figura 1. Fragmento de código contendo dependência entre variabilidades

Variabilidades comumente se relacionam para realizar tarefas específicas de um sistema configurável [Apel et al. 2013]. Neste trabalho, seguindo a definição de trabalhos da área [Cafeo et al. 2016a, Ribeiro et al. 2011, Rodrigues et al. 2016], considera-se que sempre que um ou mais elementos de programa definidos em uma variabilidade se relacionem com elementos de programa de outra variabilidade configura-se uma dependência entre variabilidades. Ou seja, uma dependência entre duas variabilidades pode ser composta por diversos relacionamentos entre elementos de programas de ambas as variabilidades.

Na Figura 1, pode-se observar uma dependência entre as variabilidades `USE_LIBPROXY` (arquivo hexchat.c) e `WIN32` (arquivo gtktils.c). A dependência é configurada por meio de apenas um par de elementos de programa (cada um pertencente a uma variabilidade). Mais especificamente, a função `get_xdir_fs` definida na variabilidade `WIN32` (Linha 7) é utilizada na variabilidade `USE_LIBPROXY` (Linha 11) configurando uma dependência.

3. Metodologia

Neste trabalho são analisados 6 sistemas configuráveis de código aberto implementados em linguagem C, abrangendo diferentes domínios de aplicação. Tal seleção, baseou-se em trabalhos anteriores [Cafeo et al. 2016b, Rodrigues et al. 2016]. Observa-se na Tabela 1 a lista dos sistemas analisados, bem como o número de *releases* e *commits*, idade do sistema, número total de linhas de código considerando todas as *releases*, a média de *commits* entre cada *release*, a média de dias entre cada *commit* e cada *release*.

abordagens composicionais.

[4]https://github.com/hexchat/hexchat

Tabela 1. Conjunto de sistemas analisados

Sistema	# *Releases*	# *Commits*	Idade (anos)	LOC (Total)	Média de *Commits* por *Release*	Intervalo entre *Commits* (dias)	Intervalo entre *Releases* (dias)
Curl	178	23202	18	13457614	130,3	0,29	38
GZip	8	439	25	98362	54,9	18,65	1023
Hexchat	17	3380	8	1329498	54,9	0,87	172
Lighttpd	56	2568	13	4704730	198,8	1,86	85
Libssh	33	3222	10	1225425	97,6	1,13	111
OpenVPN	76	2226	11	9904685	29,3	1,88	55

Para cada sistema, analisou-se todo o histórico disponível no repositório[5], totalizando mais de 30 milhões de linhas de código distribuídos em 368 *releases* e mais de 35 mil *commits*. O procedimento de coleta de dados foi dividido em duas fases. Primeiramente foram coletados os dados referentes às *releases* dos sistemas e, em seguida, coletou-se os dados referentes aos *commits*. Cada uma dessas fases possuem 4 atividades, sendo:

Minerar repositórios de software. A primeira atividade foi recuperar o repositório dos projetos selecionados para analisar as mudanças históricas nas dependências entre variabilidades. Para isso, implementou-se uma ferramenta que recupera cada *release* (ou *commit*) do projeto por meio da identificação de *tags* e *logs*.

Identificar dependências entre variabilidades. Após recuperar o código-fonte de cada versão do projeto, identificou-se as dependências entre variabilidades. Para tal, implementou-se uma ferramenta baseada no TypeChef [Kästner et al. 2011], que identifica dependências entre variabilidades a partir da Árvore Sintática Abstrata (AST) gerada pelo TypeChef, que contém a declaração e definição dos elementos de programa.

Detectar mudanças. Para detectar as mudanças nas dependências entre variabilidades, armazenou-se dados de duas *releases* (ou *commits*) subsequentes para compará-las. Tal comparação possibilitou identificar as dependências entre variabilidades incluídas, excluídas e preservadas. Dentre as dependências preservadas, buscou-se identificar quais sofreram algum tipo de alteração estrutural.

Comparar *releases* e *commits*. Após a extração de dados referentes aos *commits* e *releases* dos sistemas, realizou-se uma comparação para identificar em qual caso as alterações na implementação das dependências são mais frequentes. Na Figura 2 apresenta-se uma ilustração do método utilizado.

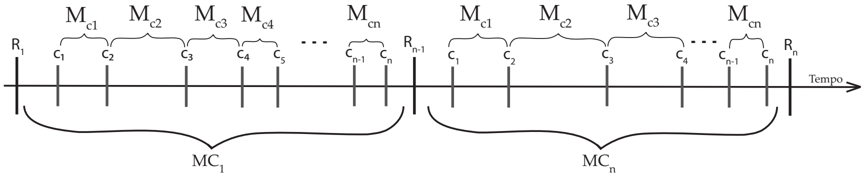


Figura 2. Em que R_i indica a *release*, c_i indica o *commit*, M_{ci} indica a quantidade de mudanças entre dois *commits* subsequentes, e MC_i indica $\sum_{i=1}^{n} M_{ci}$

Inicialmente contabilizou-se o total de mudanças em *commits* subsequentes (M_{ci}), em seguida calculou-se a quantidade de mudanças acumuladas em *commits* subsequentes (MC_i) de acordo com a fórmula: $\sum_{i=1}^{n} M_{ci}$, em que n é a quantidade de

[5]Considerou-se o repositório hospedado no GitHub até o dia 15 de junho de 2018.

commits entre duas *releases* subsequentes. Por fim, fez-se a comparação entre o resultado obtido (MC_i) e a quantidade de mudanças em *releases* (R_i) no mesmo período de tempo.

4. Resultados

Nessa seção são apresentados os resultados do estudo. Na Seção 4.1 são apresentadas as mudanças em dependências em termos de *commit* e em termos de *release*. Na Seção 4.2 são apresentados dados descritivos sobre as mudanças que ocorrem nas dependências.

4.1. Commit ou release?

Na Figura 3 pode-se observar a distribuição da quantidade de mudanças ao longo da evolução dos sistemas. Observa-se que a quantidade de mudanças em *commits* tende a aumentar mais rapidamente ao longo da vida do sistema do que em *releases*.

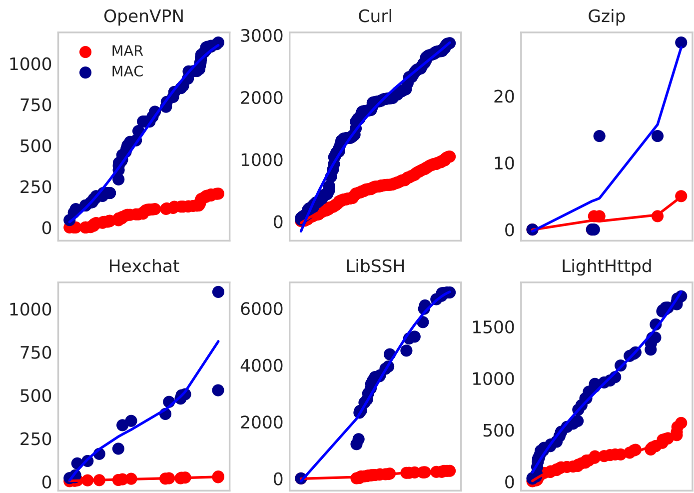


Figura 3. Mudanças acumuladas em *releases* (MAR) e em *commits* (MAC)

Os dados apresentados na Tabela 2 contemplam mudanças de uma forma geral, ou seja, estão contabilizadas inclusões de novas dependências, exclusões e dependências preservadas durante uma evolução, que sofreram algum tipo de alteração estrutural. Observa-se que as mudanças em *commits* são mais comuns que em *releases*, chegando a ser até 1604% maior no mesmo intervalo de tempo.

Tabela 2. Quantidade média de mudanças ao longo da evolução dos sistemas

	Curl	Gzip	Hexchat	LighHttpd	LibSSH	OpenVPN
Média de mudanças em commits	13,40	6,50	108,65	27,30	220,82	12,17
Média de mudanças em releases	10,32	5,17	6,38	8,76	14,07	5,25

Ao se considerar somente *releases* pode-se perder algumas mudanças. Por exemplo, ao comparar as *releases* 1.4.11 e 1.4.12 do sistema LigthHttpd[6] foram contabilizadas 11 mudanças em dependências. Entre essas *releases* há um total de 69 *commits*, que acumulam um total de 27 mudanças em dependências. Isso ocorre porquê uma mesma dependência pode sofrer diversas alterações em seus elementos, e quando considera-se apenas *releases* tais alterações podem ser ignoradas.

[6]https://github.com/lighttpd/lighttpd1.4

4.2. Tipos de mudança

Embora mudanças em dependências entre variabilidades ocorram em maior número em termos de *commits*, a distribuição dos tipos dessas mudanças ocorre de forma diferente em *releases* e *commits*. Na Figura 4 observa-se a taxa de cada tipo de mudança em termos de *commits* e *releases* ao longo da evolução dos sistemas. As exclusões e inclusões de novas dependências correspondem a maior parte das mudanças em termos de *releases*. No entanto, as mudanças em dependências preservadas, ou seja, que não são novas e nem excluídas durante uma evolução, mas que sofrem algum tipo de alteração estrutural é maior em termos de *commits*.

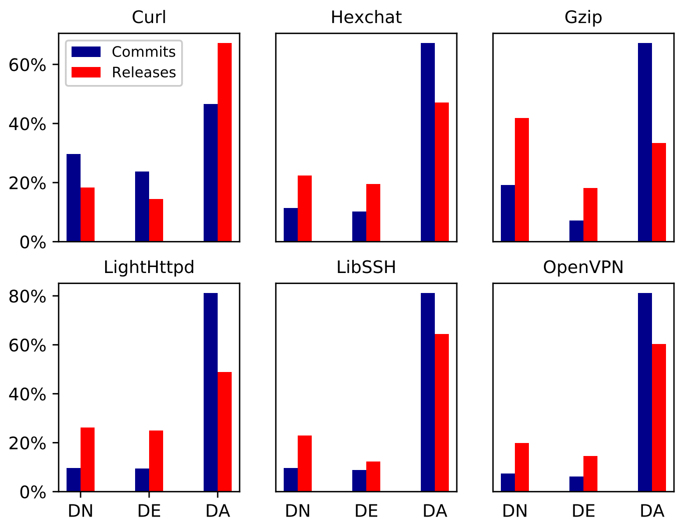


Figura 4. Taxa de dependências novas (DN), dependências excluídas (DE) e dependências preservadas alteradas (DA) nos sistemas analisados

Com exceção do sistema Curl[7], nos demais sistemas a taxa de dependências preservadas e alteradas é maior em *commits* do que em *releases*. Em média, cerca de 73% das mudanças em *commits* pertencem a essa classe, contra cerca de 55% em *releases*. A média de dependências novas e excluídas ao longo da evolução dos sistemas mostra-se menor em *commits*. Cerca de 15% das mudanças em *commits* são referentes a inclusão de novas dependências, contra aproximadamente 26% em *releases*. As mudanças que se referem a exclusão de dependências em *commits* é cerca de 11%, em *releases* essa quantidade é de aproximadamente 18% em média.

5. Implicações

A partir dos resultados obtidos, identificou-se que mudanças nas dependências são mais frequentes quando analisa-se *commits*. Ressalta-se que mudanças de granularidade grossa (inclusão e exclusão de dependências) correspondem a maior parte das mudanças nas dependências em termos de *releases*. Em termos de *commits*, as mudanças de granularidade fina correspondem a maior parte das mudanças. Esse resultado indica que analisar *commits* mostra-se uma melhor opção quando se busca identificar mudanças de granularidade fina.

[7]https://curl.haxx.se/

Dependências entre variabilidades tendem a ser preservadas ao longo da evolução dos sistemas. Dependências com mudanças constantes tendem a impactar mais o código, e portanto merecem mais atenção nas manutenções. Dessa forma, nota-se a necessidade de ferramentas para: *(i)* análise de impacto no código; *(ii)* predição de erros; e *(iii)* análise de degradação arquitetural que pode indicar se as alterações estão violando a arquitetura planejada, ou analisar como uma eventual degradação evoluiu.

6. Ameaças à Validade

Validade Interna. Neste trabalho foram exploradas as alterações nos elementos de programa (funções e variáveis) por meio de análise estática do histórico de evolução do código. Não foram consideradas outros elementos de programa que poderiam participar da implementação de dependências, bem como técnicas de análise dinâmica, casos de teste ou técnicas mais custosas, como fluxo de dados.

Validade Externa. Os sistemas configuráveis podem não ser representativos, tal ameaça foi reduzida avaliando sistemas configuráveis de diferentes domínios de aplicação, amplamente utilizados e avaliados em pesquisas anteriores [Medeiros et al. 2016, Rodrigues et al. 2016].

Validade de Construção. Neste estudo analisou-se sistemas configuráveis escritos em C que utilizam compilação condicional para implementar variabilidades. Tal mecanismo pode aumentar o número de dependências quando comparado a outras técnicas de implementação. Argumenta-se que esse é um dos mecanismos mais utilizados para implementar sistemas configuráveis [Ribeiro et al. 2014].

Validade de Conclusão. Os sistemas escolhidos vieram de diferentes domínios de aplicação. Existe o risco de que a variação devido a diferenças individuais seja maior do que a decorrente do tratamento. Tal variação ajuda a promover a validade externa do estudo, melhorando a capacidade de generalizar os resultados do experimento.

7. Trabalhos Relacionados

Estudos empíricos. Diversos trabalhos tentam compreender o fenômeno relacionado à dependências entre variabilidades em sistemas configuráveis por meio de estudos empíricos [Cafeo et al. 2016a, Ribeiro et al. 2011, Rodrigues et al. 2016]. Esses trabalhos analisam: a quantidade de dependências; métricas que caracterizam as dependências; e possíveis impactos negativos que tais dependências podem causar em atributos de manutenção. Apesar desses trabalhos confirmarem a importância das dependências na manutenção de sistemas configuráveis, nenhum deles exploram a frequência, ou se tais dependências são alteradas.

Apoio à manutenção. Nos últimos anos surgiram trabalhos propondo abordagens e ferramentas para o apoio à manutenção de código-fonte na presença de dependências entre variabilidades [Cafeo et al. 2016b, Schröter et al. 2014]. Tais trabalhos analisam como a implementação de dependências pode afetar a manutenção e propõem formas de apoiá-la. Tais trabalhos não focam especificamente em evolução. Este estudo busca oferecer informações à respeito da frequência com que dependências sofrem alterações ao longo da evolução do sistema em termos de *releases* e *commits*.

8. Conclusões

A análise realizada neste trabalho caracterizou a frequência com que dependências entre variabilidades sofrem alterações ao longo da evolução de sistemas configuráveis, abordando somente *releases* e *commits*. Tal caracterização se deu por meio da análise histórica de 6 sistemas implementados em C com diretivas de pré-processamento. Verificou-se que dependências são mais propensas a mudanças em *commits* do que em *releases*. Em trabalhos futuros essa análise poderá considerar janelas de tempo independentes de *releases* e/ou *commits*.

Mudanças em dependências preservadas são predominantes em termos de *commits*. Em termos de *releases*, a maior parte das mudanças correspondem a inclusão e exclusão de dependências. Tais resultados apontam que estudos que envolvem a investigação de mudanças de granularidade fina em dependências entre variabilidades devem considerar a análise em termos de *commits*.

Agradecimentos: Este trabalho é parcialmente financiado pelo PIBIC/UFMS edital nº 61 e pelo CNPq (processo 133822/2017-6).

Referências

Apel, S., Batory, D., Kästner, C., and Saake, G. (2013). *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer Publishing Company.

Cafeo, B. B. P., Cirilo, E., Garcia, A., Dantas, F., and Lee, J. (2016a). Feature dependencies as change propagators. *Information Software Technology*, 69:37–49.

Cafeo, B. B. P., Hunsen, C., Garcia, A., Apel, S., and Lee, J. (2016b). Segregating feature interfaces to support software product line maintenance. In *15th International Conference on Modularity (ICM)*, pages 1–12.

Clements, P. and Northrop, L. (2002). *Software product lines*. Addison-Wesley.

Kästner, C., Apel, S., and Kuhlemann, M. (2008). Granularity in software product lines. In *30th International Conference on Software Engineering (ICSE)*, pages 311–320.

Kästner, C., Giarrusso, P. G., Rendel, T., Erdweg, S., Ostermann, K., and Berger, T. (2011). Variability-aware parsing in the presence of lexical macros and conditional compilation. In *International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 805–824.

Medeiros, F., Kästner, C., Ribeiro, M., Gheyi, R., and Apel, S. (2016). A comparison of 10 sampling algorithms for configurable systems. In *38th International Conference on Software Engineering (ICSE)*, pages 643–654.

Ribeiro, M., Borba, P., and Kästner, C. (2014). Feature maintenance with emergent interfaces. In *36th International Conference on Software Engineering (ICSE)*, pages 989–1000.

Ribeiro, M., Queiroz, F., Borba, P., Tolêdo, T., Brabrand, C., and Soares, S. (2011). On the impact of feature dependencies when maintaining preprocessor-based software product lines. In *10th International Conference on Generative Programming and Component Engineering (GPCE)*, pages 23–32.

Rodrigues, I., Ribeiro, M., Medeiros, F., Borba, P., Fonseca, B., and Gheyi, R. (2016). Assessing fine-grained feature dependencies. *Information Software Technology*, 78:27–52.

Schröter, R., Siegmund, N., Thüm, T., and Saake, G. (2014). Feature-context interfaces: Tailored programming interfaces for software product lines. In *18th International Software Product Line Conference (SPLC)*, pages 102–111.

VMAG 3D – An approach for supporting the comprehension of software system models using motion control in a multiuser 3D visualization environment

Sergio H. M. B. B. Antunes, Claudia S. C. Rodrigues and Cláudia M. L. Werner

COPPE/UFRJ - Universidade Federal do Rio de Janeiro (UFRJ)
Caixa Postal 68.511 – CEP 21.945-970 – Rio de Janeiro – RJ – Brasil

{shbento,susie,werner}@cos.ufrj.br

Abstract. *This paper presents an approach for Model Visualization Assisted by Gestures in 3D, called VMAG 3D, to support the three-dimensional visualization of system models. It is inspired by the VisAr3D approach, a teaching and learning environment that provides the exploration and interaction of UML models with the use of 3D visualization, and aims to support the visualization of computer systems models, using gesture control, favoring a better usability and opening the possibility for a greater accessibility, while encouraging collaboration and communication among users. A prototype based on this approach was developed and a study was done to evaluate its usability, finding positive evidences from user experience.*

1. Introduction

System modeling is fundamental in learning and understanding of computing systems [Cian *et al.* 2017]. However, as technology progresses, these systems become increasingly larger and more complex, involving a greater number of components and working with a larger amount of data. This, in turn, makes the modeling such systems more demanding, having to represent much more elements and relationships, not only requiring a large amount of documentation, but also the means to organize it [Hamunen 2016]. This increase in the amount of data being presented leads to more difficulties when viewing and comprehending said models and the systems they represent. In order to support the comprehension of system models, applications have been developed to provide support, usually by offering different methods of visualizing the models and filtering the data shown, reducing sensorial overload [Chi 2000].

Applications that support the comprehension of abstract concepts like system modeling, be it through visualization or other means like setting frameworks, have shown to benefit from non-conventional approaches, such as Virtual Reality [Rodrigues 2016] or Multimodal Interfaces [Narayan 2017], to name a few. These approaches in particular offer more natural forms of interaction, visualization and collaboration among multiple users [Cohen *et al.* 2015], including allowing different forms of interaction at the same time [Greenwald *et al.* 2017]. With these, it's possible to support the visualization of software and its components, from different viewpoints, in a more natural manner, favoring its comprehension.

This paper presents the VMAG 3D Approach: Visualization of system Models Assisted by Gestures in 3D. Based on the VisAR3D approach described in [Rodrigues

2016], VMAG 3D has as its main objective to support the understanding of systems models with a large number of elements. This is done by providing the user with the ability to interact with diagrams displayed on the computer screen. A Kinect sensor is used to track the gestures of users, allowing a non-traditional way of interaction, as well as increasing the user's interest in the subject provided by the novelty of motion controls. Audio capturing also allows students to recall their observations about the diagrams, encouraging communication between them.

This paper is organized in 5 sections. Besides this introductory section, Section 2 presents some fundamental concepts that are relevant to the approach. Section 3 describes in detail the VMAG 3D approach, and Section 4 describes the evaluation of usability conducted. Section 5 ends this paper with the conclusion.

2. Fundamental Concepts

2.1. Virtual Reality

Virtual Reality (VR) can be defined as an "advanced user interface" for accessing applications running on the computer, allowing real-time visualization, movement and user interaction in three-dimensional computer generated environments [Kirner & Siscoutto 2007].

In order to allow this interaction, a system that employs VR needs to use various types of devices that provide the user with means to interact and visualize the virtual environment [Huang *et al.* 2010]. An example of such devices would be Motion Capture sensors, like the Kinect sensor developed by Microsoft [Alves *et al.* 2012], which particularly allows the capture of sound, as well as tracking up to 6 users simultaneously.

Due to the possibility of immersion and interaction with an environment that can be adapted to several situations, many of which simulate risk situations while keeping the user safe, VR has found its use for applications in several areas [Sherman & Craig 2002], such as education and entertainment, among others.

2.2. Multimodal Interfaces

Multimodal can be understood as having "multiple modes" or "multiple modalities" of interaction, with each mode being associated with at least one of the senses of perception: hearing, sight, touch, smell and taste [Inacio Júnior 2007]. Thus, one can understand multimodal interfaces as interaction interfaces between one or more users and a computer system that allows a varied number of data inputs [Reeves *et al.* 2004].

One of the advantages of using multimodal interfaces is user adaptability: the user can choose the method that is most practical in a given situation, such as operating a telephone by voice commands [Laput *et al.* 2013] and switch to touchscreen controls when convenient. Another important aspect of systems that use multimodal interfaces is the potential for greater accessibility [Oviatt & Cohen 2000].

However, when developing applications that use this technology, it is important to take into consideration how information is presented to the user, in order to avoid overloading information [Moreno & Mayer 2002]. Other points that require special care during development are the necessity of system feedback to users, data consistency, and adaptability [Reeves *et al.* 2004].

2.3. Collaboration

Collaboration can be defined as an activity where members of the same group work together and mutually support each other to achieve a common goal. This collaborative work occurs without the existence of hierarchical forms of division of tasks. There are no "managers" or "administrators" [Carlone & Webb 2006].

A development environment can be said to be a Collaborative Environment if it provides means for its members to interact with one another, favoring communication and coordination of activities, which can be done with the aid of specialized software supporting group work, a Groupware [Kan *et al.* 2001].

2.4. VisAr3D

VisAr3D - 3D Software Architecture Visualization approach represents a teaching and learning environment that uses Virtual Reality and Augmented Reality (AR) technologies to provide the exploration and interaction of UML models through 3D visualization. This approach was proposed in [Rodrigues 2012] and several of its concepts serve as the basis for the VMAG 3D approach.

Its main objectives are: support students' development and participation in complex projects, reduce the distance between theory and practice, support the assimilation of knowledge and skills, be attractive to students, simplicity and ease of use, the 3D diagrams used look similar to the original 2D models, visual enhancement and hide details and information when not requested.

The VisAr3D architecture is divided into three modules: an Architectural Module, where diagrams are created, documented, and exported in XMI[1] format; an Augmented Reality Module, which recognizes the 2D projection of the diagram and allows access to the related XMI file; and a Virtual Reality Module, responsible for automatically displaying a 3D model based on a 2D projection.

3. VMAG 3D

VMAG 3D - Visualization of system Models Assisted by Gestures in 3D - approach offers a way to encourage and assist computer students in understanding complex models in a collaborative way. In order to offer a different form of control, it uses a Kinect sensor to capture gestures and audio, offering a multimodal interface for interaction. An application of the same name was developed in order to better showcase this approach, as well as to allow a study of its usability, which will be detailed in the next section.

The main objectives of this approach are: supporting the comprehension of System Models, encourage collaborative work within System Modelling and promote greater student interest in that area.

---

[1] http://www.omg.org/spec/XMI/

To achieve these objectives, the following functionalities were developed: Exploring the models using either gestures or the mouse, in a 3D environment; Exhibition of multiple models, allowing a better understanding of components of a system and its interactions; Data Filtering, reducing the amount of data that are displayed, allowing users to select what they want to see; Audio recordings, which can store user commentaries regarding specific models; System Feedback, which helps users to better understand details of the application's operation; Multimodal Interface, allowing users to choose their favorite method of control; and collaboration support, using the Kinect's capability to track up to 6 people, simultaneously.

The application was developed using Unity3D[2], with its coding made using the C# language. Data regarding the models are stored in XMI files, which can be created and edited using third-party software, such as Spark's Enterprise Architect[3].

The application's interface (Figure 1) allows the users to select 1 out of 7 data filters, showing the element's name, which diagrams contain these elements, their attributes, their operators, any documentation related to that element, in which package they are contained, and the author's name, respectively. Once the desired filter is selected through the arrow buttons, the user just needs to hover over the desired element with either the mouse cursor, or the hand cursors that are tracked to each user's right hand. The top-right button displays and hides the Audio Interface. It is possible to see a list of messages already recorded associated with the diagram, if any, and new messages are automatically added to the list.

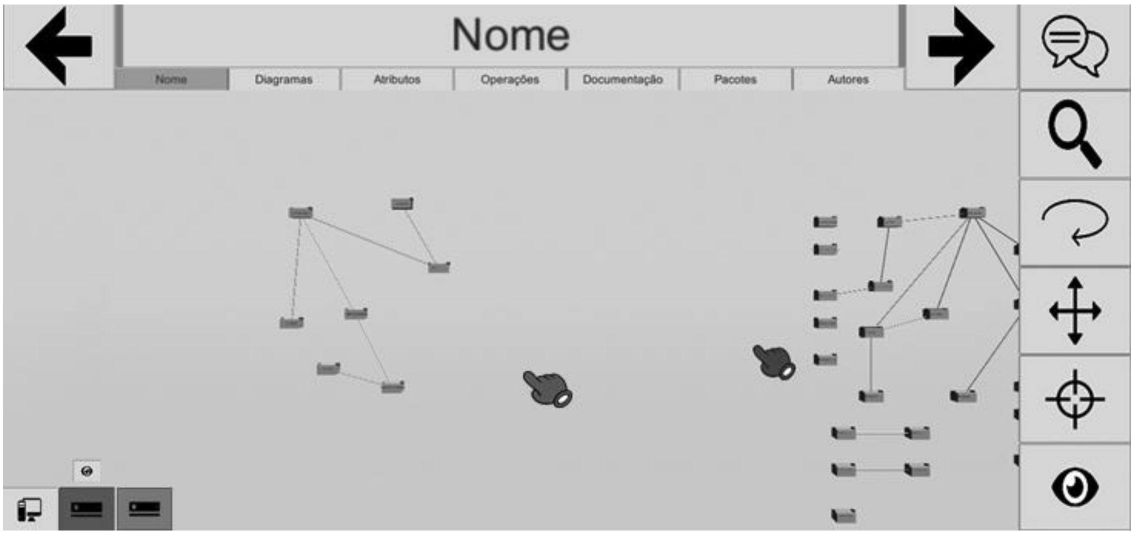


Figure 1. VMAG 3D's Interface

Also on the right side of the main interface, below the Audio Interface button, there are five buttons used to control de visualization. They allow the users to, respectively, from top to bottom: zoom in and out, orbit around a point in the view, move the view perpendicularly to the direction that the view is pointing at, reset the view to its original position, and request control of the view. These controls allow users

[2] https://unity3d.com/

[3] http://sparxsystems.com/products/ea/

to explore complex diagrams, by giving the possibility to view those in more detail, focusing in specific parts, and so on.

Given that multiple users can operate the application at the same time, in order to avoid conflicts for controlling the view, only one user can operate it at a given time. Any user can request control of the view by interacting with the Request Control button (at the bottom). If no user already has the control, he/she will be able to interact with the other four buttons and move the view around. To release control, all that is required is to interact with the Request Control button again.

Since keeping track of which user is controlling the view can become somewhat hectic, a display located at the bottom-left corner of the screen shows which users are being tracked, as well as which one, if any, has control over the view. These icons match the colors of the cursors, for the users being tracked by the Kinect sensor: Blue, Red, Green, Yellow, Purple and Orange.

4. Evaluation

According to [Juristo & Moreno 2010], Experimental Software Engineering states that the validity of any knowledge must be evaluated so that it can be considered scientific. With this in mind, a study to evaluate the application of VMAG 3D tool was planned and conducted in July and August 2017, respectively.

Following GQM – Goal/Question/Metric [Basili *et al.* 1994], the objective of the evaluation was to analyze the use of the VMAG 3D prototype, for the purpose of characterizing, with respect to usability, satisfaction and user perception in using gesture control when operating a visualization tool of three-dimensional UML models, from the point of view of researchers, in the context of execution of tasks by students, similar to those applied in a Systems Modeling discipline, using a system with many modeling elements. The study placed participants as users of the application, who should operate it to obtain data within a model. All participants were exposed to the same model and the same tasks in order to maintain a standardization of the results obtained. This model was based on the Odyssey tool model [Werner *et al.* 1999], presenting four diagrams belonging to the tool, and containing 91 classes.

Bearing in mind the cooperative nature of VMAG 3D, some of the tasks were conducted in pairs, with one researcher serving as the participant's assistant, being careful not to influence the outcome. It was decided to assign eight tasks, out of which five were done by the participant alone, and three done as a team.

A pilot study was performed with a master's degree student in Software Engineering. This pilot helped to adjust some of the questions for the evaluation. After that, ten participants were selected, through convenience by the indication of third parties and advertisements spread throughout the Technology Center building, at the Federal University of Rio de Janeiro (UFRJ). All participants were students, but of distinct academic levels (ranging from graduation, master's degree and doctorate's degree) of areas related to computing and with some level of knowledge about UML models.

Each participant received basic training through a presentation, showing the features and controls of VMAG 3D. This training was followed by a brief interval for the user to freely interact with the tool, in order to avoid the possibility of any kind of bias caused by the lack of familiarity with the provided controls. These activities lasted 10 minutes, approximately. After the training, each participant received the tasks which involved using the tool to identify specific information within the model and make use of the tool's functionalities. Some examples of those tasks would be making an audio recording stating what was the attribute of a specific class, or finding which class was part of another diagram. With the completion of all tasks, each participant received the evaluation questionnaire. Each run took 60 minutes, approximately.

4.1. Results

All 10 participants managed to accomplish the tasks and answer the questions correctly. Regarding their experience using VMAG3D, only 10% of the volunteers said they had difficulties operating it. Participants also stated that the simplified models helped visualize the diagrams and better comprehend the relationship between classes.

Participants who stated that they were comfortable interacting with the interface (40%) were also those who affirmed being more familiar with gesture controls, which implies that the source of this difficulty is not exclusively the application itself, but the level of the user’s experience with gesture controls also played a part on it. This is reinforced by some comments from the participants. Furthermore, while some users had trouble interacting with the interface, the Kinect’s capture seemed to work properly for more wide and open movements, which according to users, provided a better experience for exploring the diagram.

Furthermore, 90% of the participants affirmed that the tool was adequate to multiple users at the same time, with 60% of users verbally expressing the ease in doing the tasks in a team, in comparison with the individual tasks. It is also important to consider that both types of tasks had the same type of format in order to avoid bias.

Comparing users' satisfaction with the two control methods, gesture control versus mouse and keyboard, it is possible to note that there was a greater affinity for mouse control. This can be justified by the fact that controlling with the mouse is more common and, by being more familiar, presents less difficulty. However, it is important to note that more than 80% of users reported being "Satisfied" or "Very Satisfied" with gesture controls, with one volunteer commenting: "It seems to be an interesting tool for viewing diagrams. Zoom and camera movement are very useful for this".

5. Conclusions

The purpose of the VMAG 3D approach was to allow the user to manipulate 3D visualization of models of a software system by gestures, facilitating the understanding of the model and the user's learning, as well as encouraging other important practices for the understanding of models: collaboration and communication between users.

The evaluation showed that users could use the prototype to view the data and acquire the information contained therein and that its operation occurred in a comfortable way, with little training required and little difficulty in use.

The main contributions of this work as a whole are: insertion of multimodal interface in the VisAr3D approach, exploration of the possibility of using audio recordings as a way to encourage communication between users, and evaluation of the contribution of 3D visualization by more than one user.

From a critical analysis of the approach, as well as some comments made during the evaluation of its implementation, some limitations could be identified. Among them, those related to decisions taken during the development of the approach are: it is currently restricted to the understanding of UML models, the Kinect sensor has considerable serious accuracy problems. Other limitations pertain to the evaluation performed, like the amount of participants used and the difficulty of some of the tasks.

This approach opens up new research perspectives that can be explored in future works, such as: improving the interaction between the user and the interface, promoting a greater interaction among users and encouraging collaboration, inserting other forms of application control, and conducting a new evaluation.

References

Alves, R. de S., de Araujo, J. O. A., Madeiro, F. (2012) "AlfabetoKinect: An application to aid in the literacy of children with the use of Kinect", Simpósio Brasileiro de Informática na Educação, pp. 1-5, Rio de Janeiro, Brazil (in Portuguese).

Basili, V. R., Caldeira, G., Rombach, D. (1994). "Goal Question Metric Approach" Encyclopedia of Software Engineering, pp 528–532.

Carlone, H.B., Webb, S.M. (2006) "On (not) overcoming our history of hierarchy: Complexities of university/school collaboration", Sci. Ed., Vol. 90, pp.544–568.

Chi, E.H. (2000) "A taxonomy of visualization techniques using the data state reference model", IEEE Symposium on Information Visualization 2000, INFOVIS 2000, p. 69.

Cian, E., Dasgupta, S., Hof, A.F., Sluisveld, M.A.E., Kohler, J., Pfluger, B., Vuuren, D.P. (2017) "Actors, Decision-Making, and Institutions in Quantitative System Modelling", FEEM Working Paper, Available at: https://ssrn.com/abstract=3038695

Cohen, P.R., Kaiser, E.C., Buchanan, M.C., Lind, S., Corrigan, M.J., Wesson, R.M. (2015) "Sketch-Thru-Plan: a multimodal interface for command and control", Communications of the ACM CACM, Vol. 58, Ed. 4, pp. 56-65.

Greenwald, S., Kulik, A., Kunert, A., Beck, S., Frohlich, B., Cobb, S., Parsons, S., Newbutt, N., Gouveia, C., Cook, C., Snyder, A., Payne, S., Holland, J., Buessing, S., Fields, G., Corning, W., Lee, V., Xia, L., Maes, P. (2017) "Technology and applications for collaborative learning in virtual reality", 12th International Conference on Computer Supported Collaborative Learning (CSCL), pp. 1-8, Pennsylvania, United States.

Hamunen, J. (2016) "Challenges in Adopting a Devops Approach to Software Development and Operations", Master's Degree thesis, Alto University School of Business, Finland.

Huang, X., Acero, A., Chelba, C., Deng, L., Duchene, D., Goodgman, J., Hon, H., Jacoby, D., Jiang, L., Loynd, R., Mahajan, M., Mau, P., Meredith, S., Mughal, S., Neto, S., Plumpe, M., Wang, K., Wang, Y. (2000) "MiPaD: A Next Generation PDA

Prototype", Sixth International Conference on Spoken Language Processing, pp.33-36, Beijing, China.

Inacio Júnior, V. R. (2007) "A framework for developing multimodal interfaces in ubiquitous computing applications", Master's degree Dissertation, Instituto de Ciências Matemáticas e de Computação, ICMC, São Paulo, Brazil (in Portuguese).

Juristo, N., Moreno, A. M. (2013) "Basics of software engineering experimentation", Springer Science & Business Media.

Kan, H.Y., Duffy, V.G., Su, C.J. (2001) "An Internet virtual reality collaborative environment for effective product design", Computers in Industry, Vol. 45, June, pp. 197–213.

Kirner, C., Siscoutto, R. (2007) "Virtual and Augmented Reality: Concepts, Design and Applications", IX Symposium on Virtual and Augmented Reality, Petrópolis, Brazil, pp. 2-21 (in Portuguese).

Laput, G. P., Dontcheva, M., Wilensky, G., Chang, W., Agarwala, A., Linder, J., Adar, E. (2013) "PixelTone: a multimodal interface for image editing", Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 2185-2194, Paris, France.

Moreno, R., Mayer, R.E. (2002) "Learning Science in Virtual Reality Multimedia Environments: Role of Methods and Media", Journal of Educational Psychology, Vol. 94, No. 3, pp. 598–610.

Narayan, N. (2017) "MiNT: MULTIMODAL iNTERACTION FOR MODELING AND MODEL REFACTORING", Technische Universität München, München, Germany.

Oviatt, S., Cohen, P. (2000) "MULTIMODAL INTERFACES THAT PROCESS WHAT COMES NATURALLY", COMMUNICATIONS OF THE ACM, March, Vol. 43, Ed. 3, pp. 45-53.

Reeves, L.M., Lai, J., Larson, J.A., Oviatt, S., Balaji, T.S., Buisine, S., Collings, P., Cohen, P., Kraal, B., Martin, J., Mctear, M., Raman, T., Stanney, K.M., Su, H., Wang, Q. (2004) "GUIDELINES FOR MULTIMODAL User Interface Design", COMMUNICATIONS OF THE ACM, January, Vol. 47, Ed. 1, pp. 57-59.

Rodrigues, C. S. C. (2012) "VisAr3D - Uma Abordagem Baseada em Tecnologias Emergentes 3D para o Apoio à Compreensão de Modelos UML", Doctorate Thesis, Rio de Janeiro: UFRJ/COPPE.

Rodrigues, C. S. C., Werner, C. M. L., Landau, L. (2016) "VisAr3D: an innovative 3D visualization of UML models", Proceedings of the 38th International Conference on Software Engineering Companion, pp. 451-460.

Sherman, W. R., Craig, A. B. (2002) "Understanding virtual reality: Interface, application, and design", Elsevier.

Werner, C., Mattoso, M., Braga, R. et al. (1999) "Odyssey: A Reuse Environment based on Domain Models1", Caderno de Ferramentas do XIII Simpósio Brasileiro de Engenharia de Software (XIII SBES), pp.17-20, Florianópolis, Brazil (in Portuguese). Avaliable at http://reuse.cos.ufrj.br/site/pt/index.php

Development and Maintenance of Model-Oriented Software with Visualization – Exploratory and Experimental Study

Thiago Gottardi[1], **Rosana T. Vaccare Braga**[1]

[1]Institute of Mathematics and Computer Sciences – University of São Paulo
P.O. Box 668 – São Carlos – São Paulo – Brazil

{gottardi,rtvb}@icmc.usp.br

Abstract. *Several development methods employ models throughout the software life-cycle, such as: Unified Process, Model-Driven Engineering, Model-Oriented Programming and Models at Run-time. In previous works, we have studied how to generalize concepts from these methods into a generic Model-Oriented (MO) Software paradigm. In this paper, we present new visualization tools and an exploratory study to discuss new opportunities that arise from the MO software concepts, e.g., representing code and run-time data as diagrams. We also include a preliminary experiment where developers employ diagrams created by these tools to perform maintenance tasks. As results, the participants managed to perform the tasks correctly. We conclude that diagrams can be used to inspect the Model-Oriented Software and show how object diagrams can be made useful according to agile modeling principles. The experiment also indicates that visualization tools could be used for MO software maintenance.*

1. Introduction

Modeling languages and tools allow to represent software artifacts and to improve their comprehension by humans [Ambler, 2002; Brambilla et al., 2017]. The availability of Unified Modeling Language (UML) modeling tools has fostered the software development according to the Unified Processes (UP) [Larman, 2005]. Indeed, these processes include instructions to developers on how different kinds of UML diagrams could be employed. In the UP life-cycle, object diagrams can be used by the development team to express the software dynamic behavior. However, agile practitioners use models on the design phase, while object diagrams are seen as unnecessarily complex [Ambler, 2002].

Model-Driven Engineering (MDE) allows to create software using models as the main development artifact [Brambilla et al., 2017]. Model-Oriented Programming (MOP) is a software programming paradigm that was created by employing MDE concepts into a new programming language. It allows to further tighten the gap between code and modeling, by using a model-oriented programming language e.g., Umple. It allows programmers to write code without losing semantics from the design models, as well as rapid prototyping thanks to round-trip generation tools Badreddin et al. [2012]. Models at Run-time systems are systems that employ software models during run-time. These systems are being tested as a solution for self-adaptive and self-aware systems that are capable of self-healing and automatic integration [Aßmann et al., 2014].

After tools for Model-Driven Engineering (MDE) were developed, e.g., Eclipse Modeling Framework[1] (EMF), opportunities appeared for studying how modeling tools

[1]https://eclipse.org/modeling/emf/

could be employed for software development besides their original intent. After conducting a secondary study on this context [Gottardi and Braga, 2018], we have identified the opportunity of further extending MO Programming with Models at Run-Time concepts into a broader software development paradigm that we call Model-Oriented (MO) Software. In this paradigm, abstractions are generalized, virtually eliminating the gap between: **(1)** code and models; **(2)** run-time objects and models.

In this paper, we explore how this gap reduction allows developers to visualize and manipulate code and data by using modeling tools interactively during development and execution. We include an exploratory study along with a preliminary experimental study on MO software maintenance by using diagrams generated by these tools. The contributions of these studies show how MO software development could foster constant visualization for classes and objects. In this manner, UML models can be used to inspect the software and finally make UML object diagrams more useful for development.

This paper is structured as follows: In Section 2, related works are cited and compared to ours. Section 3 contains MO concepts and discusses how they were employed to support visualization as an exploratory study. In Section 4, we present an experimental study to verify if the diagrams generated by one of the visualization tools is useful during MO software maintenance tasks. Finally, Section 5 contains the conclusions and prospections for future works.

2. Related Works

Modeling tools have been developed for supporting Model-Based and Model-Driven software development practices. Instances of tools for model comparison and modeling tool construction frameworks have been reported in the literature [Brambilla et al., 2017]. We have also conducted a secondary study with the intent of listing related tools and approaches [Gottardi and Braga, 2018].

Since this work is related to Model-Oriented Programming, it is worth mentioning the set of tools created for Umple coding, which allows programmers to transform code into UML models and back at any time throughout development cycle without semantic loss. Following the creation of Umple, Badreddin et al. [2012] have also conducted an empirical study on the comprehension of UML, Umple and Java. Afterwards, they discussed how Java had the worst comprehension ratios among these languages. Their study is the most related work presented in this section. In comparison, in our work, we propose to create a broader view of model-orientation that is not tied to programming languages, making it platform independent while complying to existing modeling tools. Instead of evaluating comprehensibility of artifacts, we conducted a study that involved coding activities with the intent of verifying if there are benefits of employing the visualization tools in practice. In our work, we are also interested on visualizing dynamic models, showing objects from the run-time application, which is similar to Models@Run.time approaches [Aßmann et al., 2014], e.g., Kevoree[2].

The difference between our study and the related studies is that we are focusing on studying how MO software could be visualized dynamically and statically. The MO principles allow us to generate model representations of the software, which can be visualized

[2] http://kevoree.org/

as diagrams, texts or other artifacts for improving human comprehension. Therefore, in this paper we discuss a exploratory study on how MO principles could be supported by visualization tools. We also present an experimental study assessing how this visualization could help programmers to write code for MO software.

3. Model-Oriented Software Development and Visualization

Model-Oriented Software is a hypothetical software category that blends principles from both Model-Oriented Programming and Models at Run-time, along with an MDE method for building software. From the perspective of developers, a MO system can be developed by using modeling tools. It includes optional support for round-trip software engineering, while avoiding semantic loss, thanks to its MOP properties. From the perspective of users, a MO system is configured by using models and runs by using models thanks to its Models at Run-time properties [Gottardi, 2018].

MO software was planned as a platform-independent definition for software. This was considered to avoid forcing developers to use a new language or platform. Therefore, it was intended to provide a set of requirements that would fit existing languages, libraries, frameworks and tools. This condition makes MO software more generic than MO programming, which depends on programming language constructions.

MO software handles models at run-time and includes reflection support. This allows the system to be configured by these models and to store data to serve as parametric configuration of the system. The state of the systems can also be stored as a model. Furthermore, this generalization empowers the developers to utilize modeling tools to edit and visualize the configuration and the execution. It is arguable that this support for visualization could also be beneficial to end-users who wish to understand the data of the system, i.e., not only developers.

As the data is stored as models, the transmission of data among modules of the system could also be carried by models. This suggests that the system could be built by using a Domain Specific Language (DSL) [Brambilla et al., 2017] that has been optimized to properly represent the domain concepts of the MO software application. Models can be alternatively represented in a different concrete syntax, bridging the gap between the data that is handled by the machines and the humans who wish to read this as graphical or textual information. When comparing MO software to MDE, the main difference to the end-users is that the final software developed with MDE methods may be completely unrelated to modeling techniques, while the resulting MO software is always based on models because it employs models at run-time.

Table 1. Basic Comparison Between MDE and MO Software

Level	Name	MDE	MO Software
M3	Meta-metamodel	Language for Metamodeling	Language for Metamodeling
M2	Metamodel	Language Specification	System Design
M1	Model	Development Artifact	Data File
M0	Code	Real World Objects	Program or Model Interpreter
M-1	Data Instance	–	Models in Memory

Table 1 compares MDE versus MO basic concepts. While M3 level is common, other levels are shifted. For example, while in MDE level M2 refers to metamodels, in

MO it is used as the system design. The design model of a MO software is mapped to a metamodel, which can be visualized graphically as a diagram at any moment throughout the life-cycle. This differs from how meta-levels are described within the MDE literature [Brambilla et al., 2017]. This allows MO to employ models (M1 and M-1) as data. Therefore, data can be visualized as diagrams or exported to model files to be manipulated by modeling tools. MO software also employs model interpretation besides code generation (M0) to increase flexibility, as in Models at Run-time applications.

3.1. MO Software Visualization: A Retail System Example

In this section, an example of an online retail system is described to illustrate MO systems. This system was developed as a web services system composed by a server and a client. The client manages the cart of the customer, while the server stores the product information, collects the final cart managed by the client and executes the final checkout.

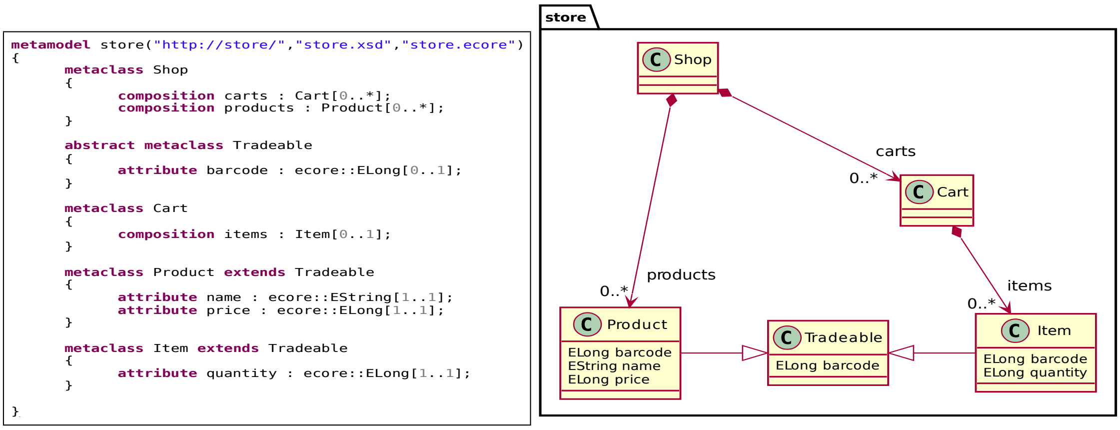


Figure 1. Cashier Application Static Development Models

As presented in Figure 1, the design of the data types is represented as a metamodel, i.e., the static portion of the software that should not change at run-time. The figure (left side) contains a textual representation using a definition language created within this project. Since this language is similar to KM3 (part of EMF), describing its constructs is not the focus of this paper. The figure also includes a class diagram representation (right side), which is generated by the tools created within our project.

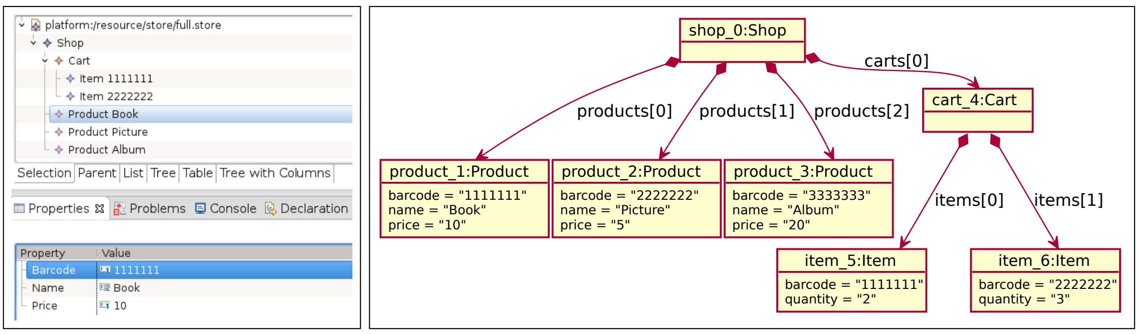


Figure 2. Cashier Application Dynamic Execution Models

The classes are divided into Shop (which stores the cart data and product data); Carts (which stores the items selected by customers); Products (which are the registry of products sold by the shop); and Items (which are the quantifier objects of products); *Tradeable* is the generalization for the "barcode" attribute. Figure 2 contains the dynamic portion of the sample MO software. The run-time data is represented graphically using model editor tools (left) and diagrams (right).

3.2. Tool Set for Model-Oriented Development and Visualization

A set of cooperating tools has been developed for assisting developers who wish to build MO software systems. All referenced tools are functional[3]. We have also developed diagram generators for class and object models. As MO software has both its static and dynamic representation mapped to models, these tools allow developers to inspect the software at any time during development or run-time. The class diagram generator can also be used in conjunction with the round-trip strategy of metamodel creation using both ECore and the proposed definition language, allowing quick visualization. When using different versions of the design model, it is possible to generate a diagram that represents the changes between them. For example, according to the design represented in in Figure 3, changes were made to add a "Customer" class (highlighted). These diagrams have been employed in the experimental study described in Section 4.

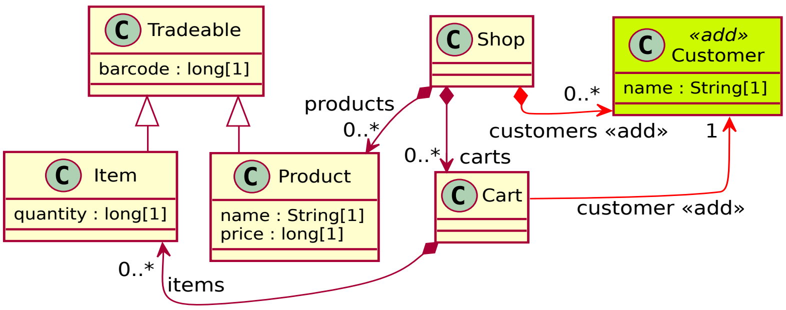


Figure 3. Cashier Application Static Maintenance Diagram

The object diagram generator is an advanced tool capable of dynamically decoding the metamodel. It interprets the also dynamic model instance from the MO software run-time data and then generates an object diagram for any MO software. This tool reads the metamodels and parses their definitions to seek the classes and their relationships that are referenced by the model instance dynamically. This was necessary to provide a tool that is completely metamodel independent, i.e., it can generate object diagrams from any MO run-time data. Our tools are compatible to EMF and also make use of PlantUML[4] to output the object diagram into image files (e.g., Figure 2).

It is also worth mentioning other development tools that generate MO software code. Since ECore is a metamodel language devised as part of EMF, we have developed tools to transform our metamodel definition language into ECore metamodels and back (round-trip method). The resulting ECore artifact is created to represent the metamodel specified by the definition language code. This metamodel is intended to be used as the data types of the MO system. It is eventually converted to data structures in programming language and XSD to represent the XML structure. This XSD is used to represent the

[3]http://tiny.cc/mo-tools
[4]http://plantuml.com/

model instance handling when using the XMI (XML format) standard[5]. The XSD generation tool creates an XSD instance without using ECore as intermediate language. Code generation tools for Java and C++ were also developed. They generate model parsing and handling modules to assist the development of MO software that is fully compatible to standard XMI model editors.

4. Experimental Study on Model-Oriented Software Maintenance

This section presents an experimental study conducted to evaluate the usage of a design diagram as instruction for the maintenance task of MO software systems. This study also allows to compare the efforts of using Java versus the proposed textual language for these maintenance tasks. It is expected that the outcome of this study would elucidate if the diagram generated by our tool is sufficient for instructing the developer. The comparison allows to identify which method takes less effort. Experiment packing is available[6].

4.1. Method

In this experiment, the maintenance of portions of a MO software is considered using different development methods. The experiment was applied in the context of graduate students properly trained in software development activities. The study was carried out from the perspective of the researchers, however, it was also intended to cover the expectations from software project managers.

The object of this study is the effort to perform a maintenance task on MO software while using the provided diagrams for instructions. The object was treated with two different development methods. While both treatments employed the same diagrams, they varied in the text language used for coding. The first method is based on writing Java code, while the second method involves our custom metamodel definition language (discussed in Section 3). The participants received the instructions in print, therefore, they could not use the computer to search or copy the content of the diagrams in the instructions. The diagrams within the maintenance task instructions were generated by our MO static visualization tool (example in Figure 3) for four different applications (different names for classes and relationships but in equal quantities).

The first dependent variable captures the time for the Java treatment, while the second variable captures the time for our custom metamodel definition language. Both variables indicate the time taken to complete the given task correctly. Participants had a maximum of 30 minutes for each task and were reserved the right to quit if they believe that they would not be able to finish.

4.2. Results

Next, we present the raw data collected from the study executions. The elapsed time data for completing each implementation task is presented on Table 2. The raw timings data was collected automatically in seconds (with millisecond precision), however, the data is presented as minutes and seconds on this table for readability.

Each row of the table represents a participant. The table allows to compare the data of Java and modeling tasks in pairs: data from the first task using Java coding (Java 1)

[5] `https://www.omg.org/spec/XMI/About-XMI/`
[6] `http://tiny.cc/mo-maintenance-pack`

/ data from the same task using the proposed modeling tool (Model 1); and data for the second task using Java and the proposed modeling tool (Java 2 & Model 2).

It is important to remind that all tasks were designed to have the same complexity, however, the experience gain from participants could help them to complete the second task faster. Finally, the last column indicates the group of the participant (Group): Group 1 participants performed the task in order Java 1, Model 1, Java 2, Model 2, while Group 2 participants performed the task in order Model 1, Java 1, Model 2, Java 2.

Table 2. Maintenance Task Timings

Participant	Java 1	Model 1	Java 2	Model 2	Group
1	04m59s	03m39s	01m08s	00m33s	1
2	28m18s	02m00s	27m57s	02m48s	1
3	08m51s	15m19s	28m12s	02m25s	1
4	05m02s	01m26s	02m14s	01m33s	2
5	09m41s	04m13s	12m09s	03m16s	2

Statistical T-Test was calculated for treatment comparison. The computed t-value is -2.341229, while the p-value is 0.04393179. Therefore, the probability of the Null hypothesis being true is below 5%, thus favoring the alternate hypothesis. As the estimated mean is negative, it is suggested that the modeling tool usage was beneficial for the tasks. Besides this comparison, all participants managed to finish the tasks during the allotted time, suggesting that the provided diagrams (e.g., Figure 3) were sufficient for their tasks.

4.3. Threats to Validity

Internal validity: The different levels of knowledge of the participants could have compromised the data. To mitigate this threat, the participants were thoroughly trained before the study tasks. Different computers and configurations could have affected the recorded logs. However, participants worked by using the computers with the same make and model in the same room and at the same time.

Validity by construction: The participants' expectations could have affected the results. To mitigate this threat, we have collected as much data as possible and asked the participants to perform as natural as possible. Also, we have concealed the objective of the experiment to avoid them from actively affecting their data towards a specific result.

External validity: It is possible that the exercises were not accurate for the real world. The experimental systems had simple requirements and their scalability was not evaluated. To mitigate this threat, we designed the exercises based on functional software inspired by real world applications.

Conclusion validity: The data collection and measurements precision could have affected the study. To mitigate this threat, all data was captured automatically as soon as the participants concluded each activity in order to allow better precision; Since we have a small population, we applied T-Tests to analyze the experiment data statistically to avoid the issues with low statistic power. Besides that, we are working on larger scale experiments and applications for the proposed methods.

5. Conclusions and Future Works

In this paper, we presented how Model-Oriented software could be visualized thanks to its concepts that map both the code and data into models. The employed tools allow to visualize the models as text and diagrams. These tools have been explained as part of an exploratory study. This exploratory study also included an example of MO software, which was used as a basis for a experimental study. In this experiment, the participants were asked to perform a maintenance task on MO software. The results indicate that every participant could finish within the allotted time, i.e., all 20 maintenance task attempts were successful. According to statistical testing, the proposed textual language also increased their productivity. Despite its preliminary status, results suggest that the diagrams created by our tools can instruct the developers to carry their task. Therefore, we conclude that the presented studies are references for further studies and tool developments.

We are working on new tools for MO development and visualization. This includes new compilers, code generators and diagram editors. Among tools for class declarations, new tools on verification and testing are under development. It is also worth mentioning a development tool for MO software that employs the object models for code generation. In conjunction to the existing class and object model visualization, this tool allows to automatize data handling and instantiation, which is being considered as part of a new experimental study on MO software testing and debugging. We also intend to verify the impact of version control on the artifacts of MO software, analyze their impact on large scale projects as well as the efforts of teaching this paradigm to students.

Acknowledgements

This work is derivative from projects funded by CAPES (DS-8428398/D, BEX 3482-15-4) and FAPESP (2016/05129-0). Special thanks to Lilian Passos Scatalon, whose suggestions have been incorporated to the presented studies. Many thanks to study participants.

References

Ambler, S. (2002). *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process*. Wiley.

Aßmann, U., Götz, S., Jézéquel, J.-M., Morin, B., and Trapp, M. (2014). *A Reference Architecture and Roadmap for Models@run.time Systems*, pages 1–18. Springer International Publishing, Cham, Switzerland.

Badreddin, O., Forward, A., and Lethbridge, T. C. (2012). Model oriented programming: An empirical study of comprehension. In *Proceedings*, CASCON '12, pages 73–86, Riverton, NJ, USA. IBM Corp.

Brambilla, M., Cabot, J., Wimmer, M., and Baresi, L. (2017). *Model-Driven Software Engineering in Practice: Second Edition*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers.

Gottardi, T. (2018). *A proposal for the evolution of model-driven software engineering*. PhD thesis, ICMC-USP. Advisor: Prof. Dr. Rosana T. V. Braga, 297 pages.

Gottardi, T. and Braga, R. T. V. (2018). Understanding the successes and challenges of model-driven software engineering - a comprehensive systematic mapping. In *Proceedings of CLEI 2018*. IEEE.

Larman, C. (2005). *Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and the Unified Process*. Object-Oriented Design (Computer Science). Prentice Hall, 3rd edition.

Uma Análise da Produção Científica Brasileira em Conferências de Manutenção e Evolução de Software

Klérisson Paixão[1], Marcelo de A. Maia[1], Marco Tulio Valente[2]

[1] Faculdade de Computação, Universidade Federal de Uberlândia

[2]Departamento de Ciência da Computação, Universidade Federal de Minas Gerais

{klerisson, marcelo.maia}@ufu.br, mtov@dcc.ufmg.br

Abstract. *The Brazilian scientific community of Software Maintenance and Evolution presents effective participation in several international scientific conferences of the highest quality standard. This article aims to analyze the scientific papers published in selected conferences over the last 5 years. The information gathered allows us to reach the following conclusions: (i) Brazilian researchers participate in almost all major conferences in the area; (ii) the leading institutions on publication ranking in this area are: UFMG, UFPE, and UFRN; (iii) Six are the central themes addressed in the publications.*

Resumo. *A comunidade científica brasileira de Manutenção e Evolução de Software apresenta efetiva participação em várias conferências científicas internacionais do mais alto padrão de qualidade. Este artigo tem o objetivo de analisar os artigos científicos publicados em conferências selecionadas nos últimos 5 anos. As informações levantadas permitem chegar às seguintes conclusões: (i) a produção brasileira apresenta participação em quase todas as principais conferências da área; (ii) as instituições que lideram o ranking de publicações nesta área são: UFMG, UFPE e UFRN; (iii) Seis foram os temas centrais tratados nas publicações.*

1. Introdução

Já com mais de três décadas de existência a comunidade científica brasileira de Engenharia de Software contabiliza significativos avanços na produção de trabalhos de alta qualidade [Neto et al. 2013]. Em especial, a comunidade de Manutenção e Evolução de Software demonstra efetiva participação em várias conferências científicas internacionais do mais alto padrão de qualidade. Contudo, não há na literatura um mapa que retrate os avanços dessa comunidade.

Análise bibliométrica é uma técnica comum para analisar publicações científicas. Um dos objetivos dessa análise é extrair introspecções sobre uma área e/ou comunidade de pesquisa. Por exemplo, tal análise ajuda a entender aspectos sobre a disseminação do conhecimento criado [Vasilescu et al. 2014], produtividade de autores e grupos [Garousi and Varma 2010], como está a evolução da área [Hoonlor et al. 2013] e tendências no interesse dos pesquisadores [Mathew et al. 2018].

Assim, neste artigo apresenta-se um estudo bibliométrico e de mineração de publicações científicas para retratar a produção brasileira em Manutenção e Evolução de Software. Este primeiro estudo se restringe a analisar publicações de conferências que, tipicamente, são o principal veículo de disseminação de artigos em Ciência da Computação [Meyer et al. 2009]. Além disso, são analisados os últimos 5 anos em cada conferência para mapear os avanços mais recentes. Especificamente, são propostas três questões de pesquisa:

QP#1 Qual a produção brasileira nas conferências selecionadas?
QP#2 Quais são os grupos brasileiros que publicaram nas conferências selecionadas?
QP#3 Quais os principais temas pesquisados?

Para responder a essas questões, foram analisados 96 artigos científicos publicados em conferências internacionais selecionadas. Os resultados mostram que: (i) a produção brasileira apresenta participação em quase todas as principais conferências da área; (ii) as instituições que lideram o ranking de publicações nesta área são: UFMG, UFPE e UFRN. Em parceria com outras instituições, pesquisadores dessas três universidades são responsáveis por mais de 40% da produção científica; (iii) Seis foram os temas mais frequentes tratados nas publicações brasileiras.

2. Método de Pesquisa

2.1. Seleção de Conferências

Várias conferências sobre Engenharia de Software são organizadas todos anos, sendo que em grande parte Manutenção de Software é um tema recorrente nessas conferências. No entanto, neste estudo o foco está em conferências com níveis de seletividade elevados para publicação de artigos.

Sendo assim, optou-se por analisar as conferências listadas no sistema CSINDEXBR [Valente and Paixão 2018]. Tal sistema indexa artigos completos de autores brasileiros publicados na trilha principal de conferências selecionadas em Ciência da Computação. Para uma conferência se integrar a base do CSINDEXBR, basicamente, três critérios são exigidos: (i) número de submissão de artigos maior que 100; (ii) taxa de aceitação menor que 30%; (iii) índice *h5* maior que 20[1]. Embora haja exceções a estes critérios no CSINDEXBR, neste estudo foram consideradas todas as conferências listadas na área de Engenharia de Software, totalizando 16 conferências. Na Figura 1 são apresentadas as quantidades de artigos aceitos e rejeitados por conferência em 2017.

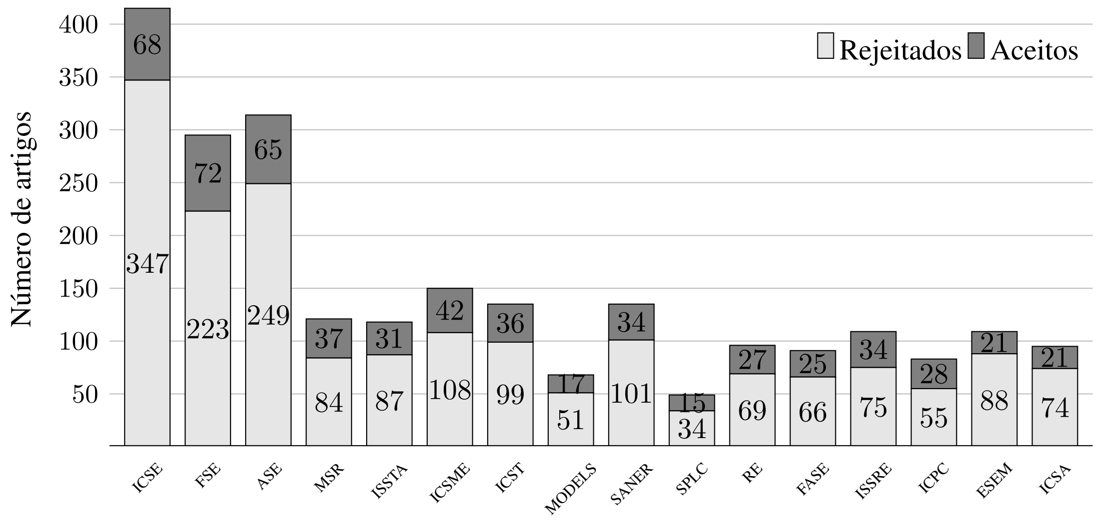


Figura 1. Estatísticas gerais das conferências selecionadas no ano de 2017.

2.2. Seleção de Artigos

O CSINDEXBR provê a lista de artigos publicados em cada uma das conferências. A seguir são descritos os passos para selecionar os artigos relacionados a Manutenção de Software:

[1] *h5* é o maior número *h* tal que *h* artigos publicados nos últimos 5 anos tenha pelo menos *h* citações.

1. Os artigos publicados de 2013 a 2017 foram todos selecionados. Os artigos de 2018 foram descartados, pois apenas uma das 16 conferências teve seus artigos indexados ainda.
2. Leitura de títulos e resumos foram feitas pelos autores deste trabalho separadamente e cada qual julgou a aderência dos artigos à sub-área de Manutenção.
3. Os respectivos julgamentos foram debatidos conjuntamente entre os autores para se chegar a um consenso.
4. Nos casos em que não houve consenso prevaleceu o julgamento maioritário.

Por fim, 62 artigos dos 96 artigos publicados pela comunidade brasileira entre 2013-2017 nas 16 conferências foram selecionados[2].

2.3. Análise dos Artigos

Para responder as questões de pesquisa foi conduzida uma análise exploratória de dados de modo a resumir as características principais das conferências e produção brasileira.

Para identificar os principais comunidades de pesquisa, além de afiliação de cada autor, foram feitas análises de redes de coautoria. Uma rede de coautoria é composta de nós e arestas, onde os nós são os autores dos artigos e as arestas indicam que os autores relacionados publicaram em conjunto. Especificamente, são apresentados três aspectos da rede: (i) grau de conectividade de autores que corresponde ao número de arestas incidentes ou o quanto colabora com os outros autores; (ii) centralidade de intermediação *(betweeness)* que indica a importância de um autor na interligação de outros autores da rede; (iii) identificação de comunidades (subgrafos) dentro da rede de coautoria.

Por fim, análise de tópicos foi utilizada para caracterizar os principais temas dos artigos publicados. A definição do número de tópicos, neste caso seis, foi feita com base na métrica de coerência de tópicos. Conforme a Figura 2, observa-se que até seis tópicos o nível de coerência estava se elevando, mas começa a decair com mais tópicos. De fato, com 10 tópicos o nível de coerência se eleva novamente, no entanto, conforme Figuras 3(a) e 3(b), com 10 tópicos ocorrem maiores interseções entre tópicos o que não é ideal para categorizar os artigos.

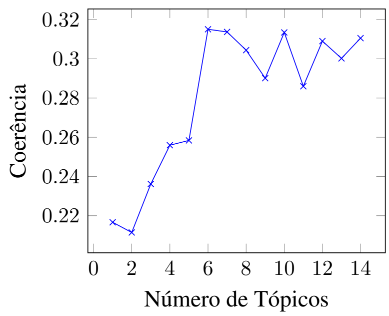


Figura 2. Coerência

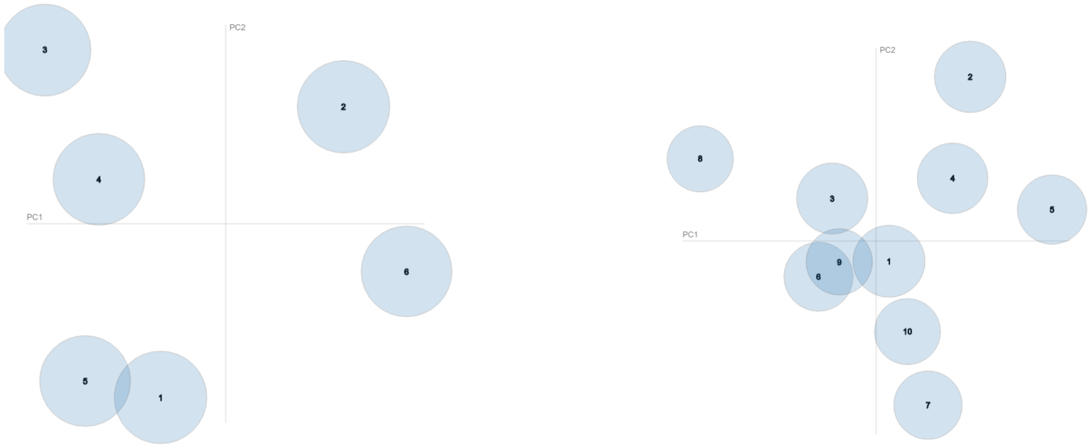
3. Resultados

QP#1 Qual a produção brasileira nas conferências selecionadas?

Na Figura 4 são apresentados a quantidade total de artigos com autores brasileiros nas conferências selecionadas, bem como a quantidade selecionada para este estudo. Observa-se que as publicações brasileiras estão presentes em quase todas as principais conferências sobre Engenharia de Software. Apenas na conferência ISSTA não houve publicações brasileiras nos últimos 5 anos.

Outro ponto de destaque é a quantidade de artigos aderentes a sub-área de Manutenção e Evolução de Software. Até mesmo conferências predominantemente de outras sub-áreas como ICST (Teste de Software) e ESEM (Engenharia de Software Experimental) publicam trabalhos sobre Manutenção de Software. Por outro lado, nas conferências ICSA, MODELS e RE não encontramos trabalhos sobre manutenção.

[2]https://zenodo.org/record/1303392



(a) Seis tópicos (b) 10 tópicos

Figura 3. Inter-distância entre tópicos.

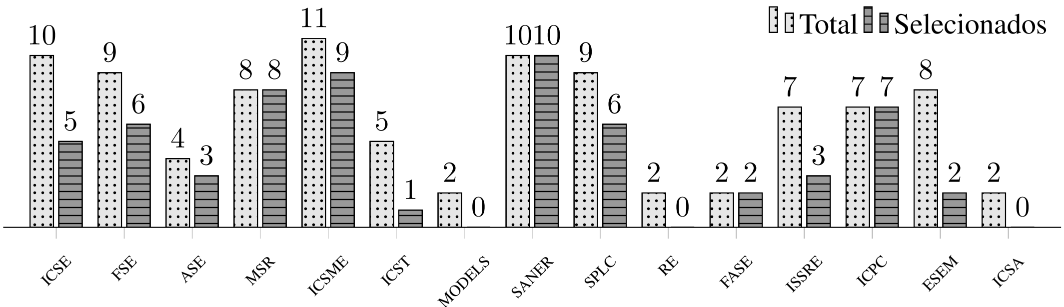


Figura 4. Distribuição de artigos por conferências publicados entre 2013–2017

QP#2 Quais são os grupos brasileiros que publicaram nas conferências selecionadas?

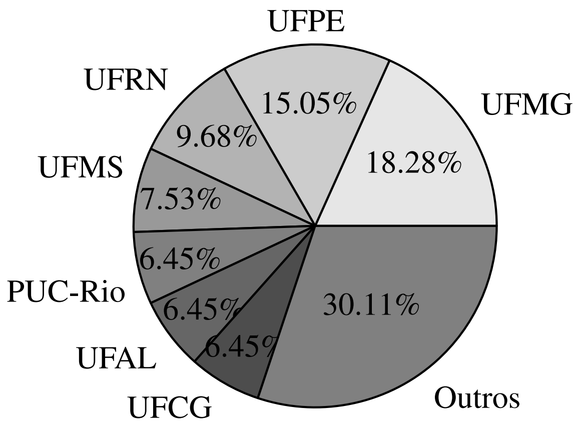


Figura 5. Percentual de artigos publicados por instituição.

A Figura 5 apresenta a quantidade percentual de publicações das instituições brasileiras. Observa-se que pouco mais de um terço dos trabalhos sobre Manutenção e Evolução de Software são oriundos ou feitos em parceria com pesquisadores da UFMG e UFPE. Além disso, nota-se que sete instituições concentram dois terços dos trabalhos publicados. A última terça parte tem participação de mais 12 instituições. Embora haja uma concentração em poucas instituições, nota-se que os trabalhos foram feitos em grande parte em colaboração com outras instituições. Por exemplo, dos 17 artigos da UFMG 10 tiveram participação de outras instituições brasileiras (i.e., CEFET/MG, PUC-Minas, PUC-Rio, UFLA, UFMS e UFU). No caso da UFPE, 6 artigos envolveram pesquisadores de outras 4 instituições (i.e., UFAL, UFCG, UNB, UFPA), do total de 16.

Para aprofundar no aspecto de colaboração entre pesquisadores a Figura 6 apresenta a rede de coautoria. Perceba que nesta rede autores estrangeiros estão presentes. No total a rede contém 158 pesquisadores e 448 relações entre os pesquisadores. O grau médio de conectividade é de 5,67, ou seja, na média cada pesquisador publicou trabalhos com outros 5 pesquisadores. No entanto, o maior grupo conectado tem diâmetro igual a 12. A densidade da rede, ou seja, a proporção de arestas diretas existentes em relação a quantidade total de possíveis arestas, é de apenas 3%. As Tabelas 1(a) e 1(b) apresentam o grau de conectividade dos 10 pesquisadores mais conectados e a centralidade de intermediação dos 10 pesquisadores que unem grandes sub-grupos, respectivamente.

Tabela 1. Conectividade e centralidade dos pesquisadores.

(a)

Pesquisador	#
Alessandro Garcia (PUC-Rio)	33
Marco Túlio Valente (UFMG)	27
Rohit Gheyi (UFCG)	23
Márcio Ribeiro (UFAL)	20
Marcelo d'Amorim (UFPE)	18
Fernando Castor (UFPE)	14
Eiji Adachi Barbosa (UFRN)	12
Paulo Borba (UFPE)	12
Baldoino Fonseca (UFAL)	11
Leonardo da S. Sousa (PUC-Rio)	11

(b)

Pesquisador	#
Rodrigo Bonifácio (UNB)	0.25
Rohit Gheyi (UFCG)	0.22
Roberta Coelho (UFRN)	0.21
Marco Tulio Valente (UFMG)	0.20
Alessandro Garcia (PUC-Rio)	0.20
Paulo Borba (UFPE)	0.19
Marco Aurélio Gerosa (IME-USP)	0.15
Márcio Ribeiro (UFAL)	0.13
Gustavo Pinto (UFPA)	0.13
Leopoldo Teixeira (UFPE)	0.08

Uma forma comum para se extrair comunidades de uma rede é por meio da métrica de modularidade. Neste trabalho, o número de comunidades é dado pelo maior número de partições em um dendrograma gerado pelo algoritmo de *Louvain* [Blondel et al. 2008]. Uma comunidade possui alta conectividade entre os próprios membros da comunidade, mas baixa conectividade fora da comunidade. A Tabela 2 apresenta as comunidades de maior afinidade entre os pesquisadores brasileiros. No total foram encontradas 11 comunidades.

QP#3 Quais os principais temas pesquisados?

Nessa última questão de pesquisa analisa-se os temas publicados pelos pesquisadores brasileiros. A Figura 7 congrega os 6 tópicos com as 10 palavras mais frequentes para cada tópico. Ainda na mesma figura o número de artigos classificados automaticamente é apresentado por tópico. Observa-se dois tópicos com maior número de artigos, Figuras 7(a) e 7(b), nos quais pode-se inferir que são artigos relacionados ao tema Linha de Produtos de Software e Refatoração de Código, respectivamente. Em seguida, tem-se dois tópicos que tratam predominantemente sobre desenvolvedores, mas um com relação a *bugs* e códigos aberto e o outro tópico com relação a transformações e mudança de códigos, Figuras 7(c) e 7(d). Por fim, tem-se os tópicos com menor quantidade de artigos, Figuras 7(e) e 7(f), que tratam de análises de consumo energético e impacto de mudanças.

4. Ameaças à Validade

Validade Externa: Este estudo limitou-se a análise de 92 trabalhos publicados em conferências monitoradas pelo CSINDEXBR. Portanto, os resultados reportados não podem ser generalizados para outras fontes de dados.

Validade Interna: Dentre os aspectos que podem afetar os resultados apresentados, destaca-se o método para seleção dos trabalhos analisados. A fim de minimizar essa ameaça, a seleção foi feita separadamente por cada autor deste trabalho em seguida foi feita uma etapa de discussão sobre cada julgamento conjuntamente.

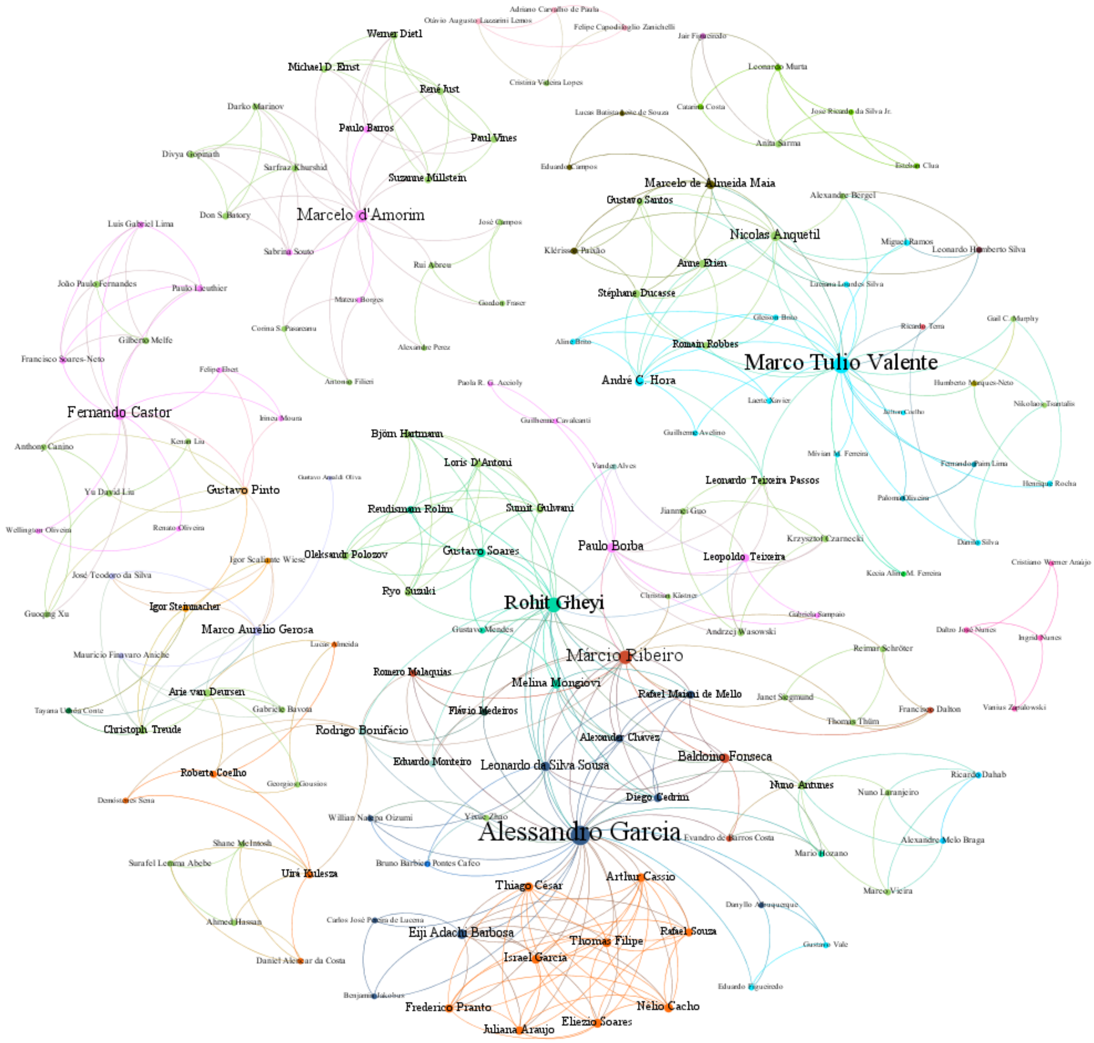
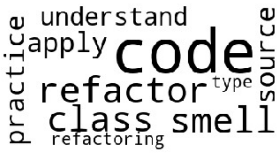


Figura 6. Rede de coautoria dos pesquisadores brasileiros e estrangeiros.

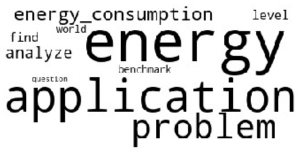
(a) 12 artigos



(b) 12 artigos

(c) 10 artigos

(d) 11 artigos



(e) 9 artigos

(f) 8 artigos

Figura 7. Tópicos e distribuição de artigos.

5. Trabalhos Relacionados

Alguns trabalhos analisam características de conferências onde artigos científicos são publicados. Em Engenharia de Sofware, Vasilescu et al. analisaram 10 anos de 11 conferências [Vasilescu et al. 2014]. Introduziram métricas que quantificam a representatividade do comitê de programa, prestígio científico, abertura para novos autores entre outros aspectos. Com foco em um dos principais periódicos em Engenharia de Software (i.e.,

Tabela 2. Sub-grupos que possuem alta conectividade entre membros

#	Pesquisadores
1	Adriano C. de Paula, Felipe C. Zanichelli, Otávio Augusto L. Lemos
2	Daniel A. da Costa, Demóstenes Sena, Gustavo A. Oliva, Igor S. Wiese, Igor Steinmacher, José T. da Silva, Lucas Almeida, Marco Aurélio Gerosa, Mauricio F. Aniche, Roberta Coelho, Tayana U. Conte, Uirá Kulesza
3	Alessandro Garcia, Arthur Cassio, Benjamin Jakobus , Bruno B. P. Cafeo, Carlos José P. de Lucena, Danyllo Albuquerque, Eduardo Figueiredo, Eiji A. Barbosa, Eliezio Soares, Frederico Pranto, Gustavo Vale, Israel García, Juliana Araujo, Nélio Cacho, Rafael Souza, Thiago César, Thomas Filipe, Willian Nalepa Oizumi
4	Alexander Chávez, Baldoino Fonseca, Diego Cedrim, Eduardo Monteiro, Flávio Medeiros, Francisco Dalton, Gustavo Mendes, Gustavo Soares, Leonardo da Silva Sousa, Márcio Ribeiro, Rafael Maiani de Mello, Reudismam Rolim, Rodrigo Bonifácio, Rohit Gheyi, Romero Malaquias
5	Aline Brito, André C. Hora, Danilo Silva, Eduardo Campos, Fernando P. Lima, Gleison Brito, Guilherme Avelino, Gustavo Santos, Henrique Rocha, Humberto Marques-Neto, Jailton Coelho, Kecia Aline M. Ferreira, Klérisson Paixão, Laerte Xavier, Leonardo H. Silva, Lucas B. L. de Souza, Luciana L. Silva, Marcelo de A. Maia, Marco Tulio Valente, Miguel Ramos, Mívian M. Ferreira, Paloma Oliveira, Ricardo Terra
6	Alexandre Melo Braga, Evandro de B. Costa, Mario Hozano, Ricardo Dahab
7	Marcelo d'Amorim, Mateus Borges, Paulo Barros, Sabrina Souto
8	Gabriela Sampaio, Guilherme Cavalcanti, Leopoldo Teixeira, Paola R. G. Accioly, Paulo Borba, Vander Alves
9	Anita Sarma, Catarina Costa, Esteban Clua, Jair Figueiredo, Jose Ricardo da Silva Jr., Leonardo Murta
10	Felipe Ebert, Fernando Castor, Francisco Soares-Neto, Gilberto Melfe, Gustavo Pinto, Irineu Moura, João Paulo Fernandes, Luis Gabriel Lima, Paulo Lieuthier, Renato Oliveira, Wellington Oliveira
11	Cristiano Werner Araújo, Daltro José Nunes, Ingrid Nunes, Vanius Zapalowski

Transactions on Sofware Engineering), Hamadicharef analisou o número de publicações, distribuição de citações e redes de colaboração [Hamadicharef 2011]. Ainda reporta um estudo sobre ocorrência de palavras chaves nos títulos publicados para mostrar tendências naquele periódico. Mathew et al. usou análise de tópicos para sumarizar numerosas quantidades de artigos científicos em Engenharia de Software [Mathew et al. 2018]. Os resultados mostram que análise de tópicos tem potencial para detectar tendências em uma comunidade.

Alguns trabalhos investigaram a produção científica da comunidade brasileira em Ciência da Computação. Costa et al. analisou redes de coautoria de artigos publicados em 30 edições do Simpósio Brasileiro de Banco de Dados [Costa de Lima et al. 2017]. Carianha e Neto exploraram 5 edições do Simpósio Brasileiro de Qualidade de Software e reportaram análises demográficas sobre aquela comunidade, bem como os principais tópicos publicados no evento [Carianha and Neto 2013]. Leite et al. reportam análises quantitativas de 5 edições do Simpósio Brasileiro de Engenharia de Software (SBES) e concluíram que Teste de Software foi a área com maior número de publicações naquela comunidade [Leite et al. 2011]. Mais dois trabalhos investigam publicações do SBES. O primeiro reporta uma análise qualitativa de 126 artigos com o objetivo de identificar falhas em estudos experimentais [Monteiro et al. 2017]. Por último, 25 edições do SBES foram mapeadas identificando os principais grupos de pesquisa, sub-áreas, distribuição das publicações entre outros aspectos [Neto et al. 2013].

6. Conclusão e Trabalhos Futuros

Este trabalho apresentou um estudo sobre a produção científica brasileira em Manutenção e Evolução de Software em conferências considerados de alto padrão de qualidade. Para isso, foram analisados 96 artigos que foram publicados nestas conferências selecionadas. As informações levantadas nos permitiram chegar às seguintes conclusões: (i) a produção brasileira apresenta participação em quase todas as principais conferências da área; (ii) as instituições que lideram o ranking de publicações nesta área são: UFMG, UFPE e UFRN. Em parceria com outras instituições pesquisadores dessas três universidades são responsáveis por mais 40% da produção científica; (iii) Seis foram os temas centrais tratados nas publicações. Como trabalhos futuros, pretende-se estender este estudo para outras áreas e verificar as semelhanças e diferenças entre comunidades.

Referências

Blondel, V. D., Guillaume, J.-L., Lambiotte, R., and Lefebvre, E. (2008). Fast unfolding of communities in large networks. *J. Stat. Mech. Theory Exp.*, 2008(10):P10008.

Carianha, M. S. and Neto, C. R. L. (2013). Uma análise dos últimos 5 anos do Simpósio Brasileiro de Qualidade de Software. In *Proc. SBQS*, pages 153 – 164.

Costa de Lima, L. H., Penha, G., de Alencar Rocha, L. M., Moro, M. M., Couto da Silva, A. P., Laender, A. H. F., and M. de Oliveira, J. P. (2017). The collaboration network of the Brazilian Symposium on Databases. *J. Braz. Comput. Soc.*, 23(1):10.

Garousi, V. and Varma, T. (2010). A bibliometric assessment of canadian software engineering scholars and institutions (1996-2006). *Comp. Inform. Sci.*, 3(2):19.

Hamadicharef, B. (2011). Scientometric study of the ieee transactions on software engineering 1980-2010. In *Proc. CACS*, pages 101–106.

Hoonlor, A., Szymanski, B. K., and Zaki, M. J. (2013). Trends in Computer Science research. *Commun. ACM*, 56(10):74–83.

Leite, J., Batista, T., and Leite, L. (2011). Software Engineering research in Brazil: An analysis of the last five editions of SBES. In *Proc. SBES*, pages 24–29.

Mathew, G., Agrawal, A., and Menzies, T. (2018). Trends in topics at SE conferences (1993-2013). *arXiv*, abs/1608.08100.

Meyer, B., Choppy, C., Staunstrup, J., and van Leeuwen, J. (2009). Viewpoint: Research evaluation for computer science. *Commun. ACM*, 52(4):31–34.

Monteiro, D., Gadelha, R., Alencar, T., Neves, B., Yeltsin, I., Gomes, T., and Cortés, M. (2017). An analysis of the empirical Software Engineering over the last 10 editions of Brazilian Software Engineering Symposium. In *Proc. SBES*, pages 44–53.

Neto, P. A. d. M. S., Gomes, J. S., Almeida, E. S. d., Leite, J. C., Batista, T. V., and Leite, L. (2013). 25 years of Software Engineering in Brazil: Beyond an insider's view. *Journal of Systems and Software*, 86(4):872 – 889.

Valente, M. T. and Paixão, K. (2018). CSIndexbr: Exploring the Brazilian scientific production in Computer Science. *arXiv*, abs/1807.09266.

Vasilescu, B., Serebrenik, A., Mens, T., van den Brand, M. G., and Pek, E. (2014). How healthy are Software Engineering conferences? *Sci. Comput. Program.*, 89:251 – 272.

An Infrastructure for Software Release Analysis through Provenance Graphs

Felipe Curty[1], Troy Kohwalter[1], Vanessa Braganholo[1], Leonardo Murta[1]

[1]Instituto de Computação – Universidade Federal Fluminense (UFF)

felipecrp@id.uff.br, {tkohwalter,vanessa,leomurta}@ic.uff.br

Abstract. *Nowadays, quickly evolving and delivering software through a continuous delivery process is a competitive advantage and a way to keep software updated in response to the frequent changes in customers' requirements. However, the faster the software release cycle, the more challenging to track software evolution. In this paper, we propose Releasy, a tool that aims at supporting projects that use continuous delivery by generating and reporting their release provenance. The provenance generated by Releasy allows graphical visualization of the software evolution and supports queries to discover implicit information, such as the implemented features of each release and the involved developers. We also show in this paper a preliminary evaluation of Releasy in action, generating the changelog of an open source project with the provenance collected by our tool.*

1. Introduction

Continuous delivery aims at producing high-quality software in short release cycles and enables organizations to quickly, reliably, and efficiently evolve software, which may become an advantage over competitors (Chen, 2015). Ideally, continuous delivery enables software to be released whenever it is needed – it could be weekly, daily, or even after each commit (Neely and Stolt, 2013).

When continuous delivery is in place, tracking each release becomes a challenge due to the increasing amount of releases. Accurately knowing which issues were delivered in a specific release, the sequence of commits that were performed to implement such issues, and the developers that were in charge of such implementation is paramount for management and maintenance. The inability to change a software due to the lack of visibility over its releases may lead to losing business opportunities (Bennett and Rajlich, 2000). Hence, tracking software evolution is important, but becomes difficult in continuous delivery since the software can be released often.

Related work (Amorim Pereira and Schots, 2011; Bhattacharya et al., 2012; D'Ambros et al., 2008; De Nies et al., 2013; German and Hindle, 2006) has investigated software evolution through the use of software trails, which are the information left behind by contributors during the development process. However, only D'Ambros *et al.* (2008) handle release information. Nevertheless it is not suited to detect the issues implemented in each release and, therefore, cannot be used to generate the software changelog.

In this paper, we propose Releasy, a tool that collects provenance data from releases by parsing the software version control and issue tracking systems. Provenance describes the history of an artifact, comprising all information about the origin and derivations of that artifact (Groth and Moreau, 2013). For a given release, our tool can identify: (1) the previous release, (2) the features that were delivered in the release under analysis, (3) the commits that implemented each feature, and (4) the developers that authored the commits. Releasy can also export the collected provenance using the PROV-N notation (Groth and Moreau, 2013), which enables using the release provenance data for graphical visualizations and queries in any of the existing provenance visualization tools.

Releasy can be handy for both the practitioners and researchers. We envision the use of Releasy in a daily basis by practitioners that want to automate the generation of the public changelog of a project, to find the developers that are the most suited to fix bugs in a particular feature, and to understand the differences between releases. On the other hand, we also envision the use of Releasy by researchers that need to mine software archives and want to integrate their own mined data with release information. One could easily programmatically access this information and integrate it with other data because Releasy provides all data output in the PROV-N notation.

We used Releasy to track the releases of the *Homebrew* open source project as a method to evaluate our approach. We observed from the obtained results that our tool is capable of reconstructing the software changelog and showing accurate information about who contribute to each feature.

This paper comprises five sections besides this introduction. In Section 2, we describe our approach to collect release provenance. In Section 3, we describe Releasy, detailing how it works. In Section 4, we evaluate Releasy using the *Homebrew* open source project. In Section 5, we contrast and compare Releasy with some related work. Finally, in Section 6, we conclude the paper, discussing some future work related to our research.

2. Release Provenance

Our approach collects software trails from version control and issue tracking systems to build a provenance graph that helps understand the software releases. As shown in Figure 1, Releasy manages information about developers, commits, tags, and issues. The white boxes represent information gathered from the version control system, i.e., developers, commits, and tags; the gray box represents information gathered from the issue tracking system, i.e., issues; and the relationship between commits and issues is inferred by our approach. We formally define the analyzed project as $p = (D, C, I, T)$, where D is the set of developers, C is the set of commits, I is the set of issues, and T is the set of tags.

The commits represent every change made to the software and are organized in a directed acyclic graph. Figure 2 shows an example of such graph, where the commits are represented by rounded boxes (e.g., c_{01cf}); tags are represented by pointing boxes (e.g., $t_{3.0.15}$) and denote important software states in its history, such as releases; issues are represented by diamonds (e.g., i_1) and denote the change requests addressed by commits; and arrows represent the parent relationship between two commits, that is, the parent commit that is reachable from its children (e.g., c_{a20c} is reachable from c_{bc12} and c_{ak4s}).

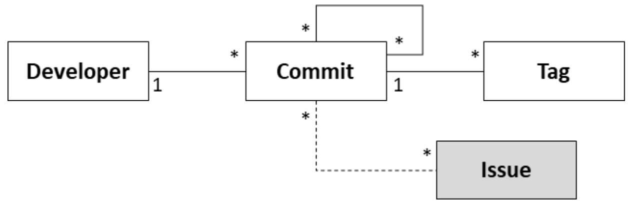


Figure 1: Release provenance metamodel with version control system information in white and issue tracking system information in gray.

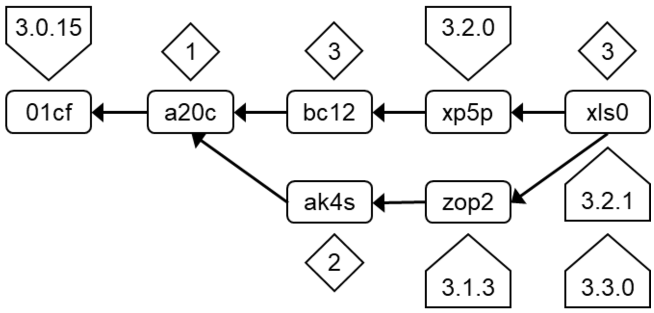


Figure 2. A release provenance graph with commits (rounded boxes), issues (diamonds), and releases (numbered pointing boxes).

We define each commit as $c_i = (P_i, a_i, I_i, T_i)$, where $P_i \subseteq C$ is the set of parents of the commit c_i, $a_i \in D$ is the author of the commit, $I_i \in I$ is the set of issues addressed by this commit, and $T_i \in T$ is the set of tags that point to the commit. From a given commit c_i, we form its history H_i including the commit itself and all the reachable commits, recursively. This is defined as $H_i = \{c_i\} \cup \bigcup_{c_k \in P_i} H_k$. In our example, shown in Figure 2, the history of c_{bc12} is $H_{bc12} = \{c_{bc12}\} \cup H_{a20c} = \{c_{bc12}, c_{a20c}, c_{01cf}\}$ and the history of c_{xls0} is $H_{xls0} = \{c_{xls0}\} \cup H_{xp5p} \cup H_{zop2} = \{c_{xls0}, c_{xp5p}, c_{bc12}, c_{a20c}, c_{01cf}, c_{zop2}, c_{ak4s}\}$.

Our approach uses tags to represent releases (we provide more details in Section 3.3). Thus, we define the commit history CH_j of tag t_j as $CH_j = \{c \in C \middle| c \in H_i \wedge t_j \in T_i\}$. We can also define all issue history IH_{j} of tag t_j as $IH_{\mathrm{j}} = \bigcup_{\mathrm{c_i} \in CH_{\mathrm{j}}} \mathrm{I_i}$, e.g., $IH_{3.2.1} = \{i_1, i_2, i_3\}$. Then, we can compare two tags in terms of released commits, e.g., $CH_{3.2.0} \backslash CH_{3.0.15} = \{c_{xp5p}, c_{bc12}, c_{a20c}\}$ or in terms of released issues, e.g., $IH_{3.2.0} \backslash IH_{3.0.15} = \{i_1, i_3\}$, where \ denotes set difference.

However, we are also interested in detecting commits and issues released by a tag t_j that were not released by previous tags. Therefore, we define the tag history $TH_j = \{t_k \in T \middle| CH_k \subset CH_j\}$, which represents the set of tags reachable from the tag t_j, e.g., $TH_{3.2.1} = \{t_{3.20}, t_{3.1.3}, t_{3.015}\}$. Then, we define the commits released by the tag t_j that are not reachable by any previous tag as $CR_j = CH_j \setminus \bigcup_{t_k \in TH_j} CH_k$, e.g., $CR_{3.2.1} = \{c_{xls0}\}$. We can also define the issues released exclusively by the tag t_j as $IR_{\mathrm{j}} = \bigcup_{\mathrm{c_i} \in CR_{\mathrm{j}}} \mathrm{I_i}$, e.g., $IR_{3.2.1} =$

$\{i_3\}$. Finally, we can identify the issues that were reworked across releases, e.g., $IR_{3.2.1} \cap IR_{3.2.0} = \{i_3\}$.

3. Releasy

We developed Releasy – a python tool with the primary goal to parse software trails and collect provenance from software releases. Releasy currently supports Git repositories and GitHub Issues. It offers the following features:

1. Project overview – we show the last release, the total number of releases, the number of commits, the number of developers, the number of issues related to commits, and enumerate the developers (Figure 3).
2. Release information – we show information about a specific release, including the reachable releases (the base releases), the date the release was made, the number of commits, the developers that contributed to the release, and the implemented issues (classified as features or bugfixes) (Figure 4).
3. PROV-N export – we export the release provenance graph according to the PROV-N notation, so other tools can import the data and provide additional queries and graphical visualization of the release (Figure 5).

Users can adopt Releasy by clonning the Git repository of the target project and configuring the tool with the issue tracking system URL.

3.1. Extracting Information from Git and GitHub

Releasy parses the Git repository using the `git log --reverse --all` command, which provides the full history of the software project, from the first to the last commit. Currently, it tracks the following information: hash (id), message, author, committer names and e-mails, and tags. With this information, Releasy populates the white boxes of the metamodel shown in Figure 1.

It also parses the GitHub Issues repository through its open REST API, which provides information about the issues related to the software. Currently, Releasy tracks the following information: id, author, subject, dates of creation and closure, and labels. With this information, it populates the gray box of the metamodel shown in Figure 1.

3.2. Handling Issues

While commits store all the software changes, it does not natively provide information about the implemented issues. Fortunately, many projects usually refer to the issue ID in the commit message. On projects with this behavior, it is possible to link the commit with a specific issue. For example, the message “Implements feature #1” says that this commit is related to issue with ID 1. Thus, we can associate commits and issues in our metamodel (Figure 1) by searching for the following regular expression in the commit message: “`^.*#([0-9]+).*`”.

In general, issues can represent any change request, such as features and bugfixes. Though there is no direct way to identify which is the goal of an issue, we try to infer it by examining its labels on the issue tracker. Currently, we check the labels assigned to each issue and if the regular expression "`^bug.*$`" matches, we consider the issue a bugfix. Otherwise, we consider it a feature. If the project uses a different convention, then the regular expression can be adapted.

3.3. Handling Releases

Our approach uses tags to identify releases, which is a common pattern among software projects. However, some tags may not represent releases, i.e., waypoints to enable developers to recover a given development stage quickly. Our approach uses name convention to identify the tags related to releases. We use the regular expression "`^v?[0-9]+\.[0-9]+\.[0-9]+(-.+)?$`" to match releases. This pattern is complient with the semantic versioning[1] notation, e.g., 1.0.0 and 1.1.0-beta. Although this release name convention is vastly adopted, if the project uses a different convention, the regular expression can be adapted.

3.4. Data Export

Releasy can export the release provenance graph according to the PROV-N notation (Groth and Moreau, 2013). Through PROV-N, other tools can provide additional features based on the release information, such as queries and graph visualization. An example of a graph visualization is shown in Figure 5 using Prov Viewer (Kohwalter et al., 2016), which is a graph-based visualization tool for interactive exploration of provenance data.

4. Evaluation

We used Releasy to generate release information of *Homebrew*[2], an open source software to manage package installation in MacOS. It enables users to install and uninstall software packages on their operation system.

First, we generated the project overview, shown in Figure 3. Releasy discovered 86 releases on *Homebrew*. The last release is indicated by tag $t_{1.6.7}$. Also, the project has 15,806 commits, 694 developers, and 2,282 issues linked in the commits. In fact, the project has more issues in its issue tracking system, but Releasy only shows those that are linked to commits. Finally, Releasy lists all the project developers.

```
Project Overview
 - 1.6.7 is the last of 86 releases
 - 15806 commits made by 694 developers
 - 2282 issues linked

Developers
 - Max Howell <...@...hylblue.com>
 - Adam Vandenberg <...@...il.com>
 [...]
```

Figure 3. Project Overview

Then, we performed queries for specific releases. Figure 4 shows information about the release tagged as $t_{1.6.5}$. Releasy shows that the release was created on 05/25/2018 and was

[1] https://semver.org/

[2] https://github.com/Homebrew/brew

based on the release tagged as $t_{1.6.4}$. It also identifies 10 developers that contributed to that release. Then, Releasy listed all the 16 issues implemented in that release.

```
Information about release 1.6.5
Based on: 1.6.4
Date: 2018-05-25 18:31:19+02:00
Commits: 79
Authors:
  - Markus Reiter <...@...ermark.us>
  - Gautham Goli <...@...l.com>
  [...]

Issues:
  - 4210: Update Homebrew-Cask references.
  - 4209: Reset `repo_var` so it actually is re-computed.
  - 4195: Activate Homebrew-Cask tap migration.Based on: 4.0.0-migration 3.3.7
  [...]
```

Figure 4. Release 4.0.0 Information

Finally, we exported a release to PROV-N and then generated the graphical visualization with Prov Viewer. We chose to present part of the release tagged as $t_{1.5.12}$ because it would better fit in the paper as an example. The result is shown in Figure 5. The release was based on the released tagged as $t_{1.5.11}$. We represented three commits: $\{c_{a264}\}$ was developed by a_{buck} and $\{c_{2876}, c_{3cd5}\}$ were developed by a_{me}. The commits $\{c_{a264}, c_{2876}\}$ where delivered directly to the release and the commit $\{c_{3cd5}\}$ implemented issue i_{3821}, which was delivered to the release.

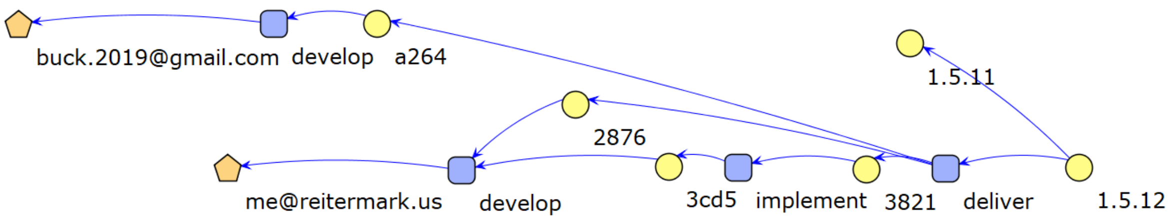


Figure 5. Release provenance graph of release 1.5.12 of *Homebrew* visualized through Prov Viewer

5. Related Work

During our research, we found other work that aims at supporting developers to track and understand software evolution through software trails. Table 1 compares them using the following information: supported version control system (VCS) and issue tracking system (ITS); and capability to handle release information (Rel.), to export provenance (Prov.), and generate software visualization (Vis.).

D'Ambros et al. (2008) is the only related work that handles release information. Their approach can analyze software evolution and collect metrics, such as the major developer and change impact. Thus, using these metrics, they can compare releases. However, they do not provide support for identifying issues released in a particular release, nor compare releases in terms of issues. Therefore, they are unable to build the software changelog. Nonetheless, in the future, their ideas can be integrated with our tool.

Table 1. Related work of Releasy, comparing capabilities to generate PROV, build information about releases, and generate software visualization.

Authors	Summary	VCS	ITS	Rel.	Prov.	Vis.
German and Hindle (2006)	softChange is a tool to summarize, browse, and visualize the evolution of a project through its software trails.	CVS	Bugzilla	-	-	Yes
D'Ambros *et al.* (2008)	Present an approach populating data into a Release History Database (RHDB).	CVS	Bugzilla	Yes	-	Yes
Pereira and Schots (2011)	GraphVCS is a tool that enables visualization and searches through software history.	SVN	-	-	-	Yes
Bhattacharya *et al.* (2012)	Present an approach to characterize software evolution through graphs.	? [3]		-	-	-
De Nies *et al.* (2013)	Git2PROV is a tool to export Git history provenance in one of the following notations PROV-JSON, PROV-N, PROV-O, and SVG.	Git	-	-	Yes	Yes
Releasy		Git	GitHub Issues	Yes	Yes	Yes

6. Conclusion and Future Work

In this paper we presented an approach to collect software trails and generate the release provenance graph, which enables detection of the software releases, its issues, and its developers. Thus, our approach helps developers understanding the software evolution concerned to releases, which is essential in continuous delivery projects, i.e., projects that can release at any time.

We developed Releasy[4], a tool that implements our approach and enables querying information about releases. Besides that, the tool allows developers to export the collected provenance in the PROV-N notation, which enables this information to be handled by other tools compliant to PROV-N. In this paper, we use the exported provenance to generate a graphical visualization of the release in Prov Viewer.

We intend to use the exported information generated by Releasy to allow queries that exploit the inference capabilities of PROV. This would enable developers to write their own queries that traverse the commit history and infer the required information. This could significantly increase the comprehension capability of developers and end users regarding the software release history.

Finally, we have performed a preliminary evaluation on Releasy. The results suggest that Releasy can be used to track software evolution and build the software changelog. However, our tool still depends on patterns, e.g., the issue id on commit messages, which could be a problem in projects that developers do not follow such patterns.

[3] The authors did not specify the supported version control system and issue tracking system.

[4] Available at https://github.com/gems-uff/releasy.

In the future, we plan to evaluate Releasy in other projects by contrasting its output with other manually generated software changelogs. This way, we can check whether the quality of Releasy output is enough to replace the manual generation of release changelogs.

References

Amorim Pereira, T., Schots, M., 2011. GraphVCS: Uma Abordagem para a Visualização e Compreensão de Repositórios de Controle de Versão.

Bennett, K.H., Rajlich, V.T., 2000. Software Maintenance and Evolution: A Roadmap, in: Proceedings of the Conference on The Future of Software Engineering, ICSE '00. ACM, New York, NY, USA, pp. 73–87.

Bhattacharya, P., Iliofotou, M., Neamtiu, I., Faloutsos, M., 2012. Graph-based analysis and prediction for software evolution, in: 34th International Conference on Software Engineering (ICSE). pp. 419–429.

Chen, L., 2015. Continuous Delivery: Huge Benefits, but Challenges Too. IEEE Softw. 32, 50–54.

D'Ambros, M., Gall, H., Lanza, M., Pinzger, M., 2008. Analysing Software Repositories to Understand Software Evolution, in: Software Evolution. Springer, Berlin, Heidelberg, pp. 37–67.

De Nies, T., Magliacane, S., Verborgh, R., Coppens, S., Groth, P., Mannens, E., Van De Walle, R., 2013. Git2PROV: Exposing Version Control System Content As W3C PROV, in: Proceedings of the 12th International Semantic Web Conference, ISWC-PD '13. CEUR-WS.org, Aachen, Germany, pp. 125–128.

German, D.M., Hindle, A., 2006. Visualizing the evolution of software using softchange. Int. J. Softw. Eng. Knowl. Eng. 16, 5–21.

Groth, P., Moreau, L., 2013. PROV-Overview [WWW Document]. URL https://www.w3.org/TR/prov-overview/.

Kohwalter, T., Oliveira, T., Freire, J., Clua, E., Murta, L., 2016. Prov Viewer: A Graph-Based Visualization Tool for Interactive Exploration of Provenance Data, in: Provenance and Annotation of Data and Processes, Lecture Notes in Computer Science. Springer, Cham, pp. 71–82.

Neely, S., Stolt, S., 2013. Continuous Delivery? Easy! Just Change Everything (Well, Maybe It Is Not That Easy), in: 2013 Agile Conference. Presented at the 2013 Agile Conference, pp. 121–128.

Uma técnica para a quantificação do esforço de merge

Tayane Silva Fernandes de Moura, Leonardo Gresta Paulino Murta

Universidade Federa Fluminense (UFF) – Instituto de Computação
Av. Gal. Milton Tavares de Souza, s/nº São Domingos - Niterói – RJ CEP: 24210-346

tayanemoura@id.uff.br, leomurta@ic.uff.br

Abstract. *Developers that use version control systems can work in parallel with other developers and merge their versions afterwards. Sometimes these merges fail, demanding manual intervention to resolve conflicts. Some studies aim at analyzing the merges failures, however, there is a lack of tool support in the literature to measure the merge effort, jeopardizing such kind of analyses. In this article, we propose a technique and its companion tool for analyzing Git repositories and providing metrics related to the merge effort. We evaluated our tool over five projects, showing that rework and wasted work happens in, approximately, 10% to 30% of the projects. Moreover, the average of actions of these efforts is almost the same.*

Resumo. *Com o uso de sistemas de controle de versão os desenvolvedores podem trabalhar em paralelo e, ao final, integrar (merge) suas contribuições. Algumas vezes esses merges falham e é necessário a intervenção de um desenvolvedor para resolver os conflitos. Existem estudos que buscam analisar essas falhas de merge e, para isso, necessitam quantificar o esforço envolvido. Esse artigo propõe uma técnica, implementada em uma ferramenta, capaz de analisar repositórios Git e fornecer métricas relacionadas ao esforço de merge. A ferramenta proposta foi avaliada em cinco projetos e pudemos observar que a incidência de retrabalho e de trabalho desperdiçado acontece em, aproximadamente, 10% a 30% dos commits de merge. Além disso, a quantidade média de ações desses esforços é muito parecida.*

1. Introdução

Sistemas de Controle de Versão (SCV) são ferramentas que controlam mudanças nos artefatos de software, denominados Itens de Configuração (IC) [Grinter 1995]. Além disso, os SCV permitem acesso aos repositórios em que os artefatos e seus históricos estão guardados e também proporcionam mecanismos para bloquear, comparar e combinar (*merge*) diferentes versões do mesmo IC, o que possibilita o desenvolvimento paralelo de um software. Quando é utilizada a política otimista de controle de concorrência, dois artefatos podem ser alterados simultaneamente, o que muitas vezes pode agilizar o desenvolvimento. Ao final das alterações, as duas versões são combinadas (i.e., *merge*), gerando-se uma nova versão do IC [Prudêncio et al. 2012]. Estudos passados indicam que 10% a 20% dos *merges* falham [Brun et al. 2011; Kasi and Sarma 2013] e quando isso ocorre os desenvolvedores precisam fazer manualmente o *merge* dessas versões.

Existem estudos que buscam analisar os conflitos de *merge*. Accioly et al. [2017] analisaram os conflitos de *merge* e elaboraram, através do estudo de mudanças de código, um catálogo de padrões de conflitos. Santos et al. [2012] também fizeram um estudo em torno de características de *merges*. Nesse estudo, os autores apresentam uma abordagem para a extração de métricas que visam prever se o *merge* de ramos será difícil. Existem ainda outros estudos que visam entender a natureza dos conflitos de *merge* [Kasi and Sarma 2013; Menezes 2016; Yuzuki et al. 2015]. Todos esses estudos se beneficiariam de uma técnica capaz de quantificar o esforço envolvido nos merges.

Alguns trabalhos [Cavalcanti et al. 2017; Mehdi et al. 2014; Prudêncio et al. 2012; Santos and Murta 2012] fazem uso de métricas de esforço de *merge* nas suas análises, porém, como o foco desses trabalhos não é na extração de métricas de esforço, a descrição da técnica de extração adotada é usualmente superficial e a implementação dessa técnica raramente está disponível de forma desacoplada para uso por outros pesquisadores. Assim, a cada pesquisa no tema, os pesquisadores se veem obrigados a reimplementar seus próprios coletores de métricas de *merge*. Como efeito colateral, há um elevado grau de retrabalho e pouca certeza que os resultados dos estudos possam ser comparados, já que cada implementação atua numa granularidade distinta e segue uma técnica distinta. Como consequência da pouca transparência sobre o funcionamento de cada uma dessas técnicas, a interpretação e a reprodução dos resultados são prejudicadas.

Nesse artigo, propomos uma técnica, e a sua implementação em uma ferramenta, para apoiar pesquisadores nos estudos sobre o esforço de *merge*. A partir da URL de um repositório Git, a ferramenta é capaz de identificar todos os *commits* de *merge* do repositório e calcular o trabalho extra necessário para efetuar cada *merge* (código que pertence ao *commit* de *merge* mas não veio dos ramos), assim como o retrabalho (código duplicado nos ramos) e o trabalho desperdiçado (código que pertence aos ramos e que não foi incorporado no *merge*). Cada uma dessas métricas é calculada de forma absoluta e normalizada, atendendo assim às várias necessidades de uso.

Para avaliar a ferramenta fizemos uma análise em cinco projetos open-source e concluímos que a incidência de retrabalho e de trabalho desperdiçado acontece em aproximadamente 10% a 30% dos *commits* de *merge*. Além disso, a quantidade média de ações desses esforços é muito parecida.

O restante do artigo está organizado em outras cinco seções. Na Seção 2, detalhamos a técnica proposta, explicando o processo de merge e a extração das métricas de esforço. Na Seção 3, exploramos a implementação da nossa ferramenta. Na Seção 4, apresentamos os resultados obtidos na avaliação da ferramenta. Na Seção 5, apontamos os trabalhos relacionados. Por fim, na Seção 6, expomos nossa conclusão e trabalhos futuros.

2. Esforço de Merge

Em um SCV, dois desenvolvedores podem trabalhar em paralelo no mesmo IC e depois combinar suas versões. Para ilustrar esse processo, criamos o cenário da Figura 1. Inicialmente, o desenvolvedor cria o ramo *dev* a partir do *commit base* C3 e realiza algumas modificações no código, gerando os *commits* C4, C6 e C7. Enquanto isso, um outro desenvolvedor continua trabalhando no ramo *master,* produzindo os *commits* C5 e C8. Ao final das modificações é feito o *merge* entre os ramos, gerando o *commit de merge* C9, a partir dos *commits pais* C7 e C8.

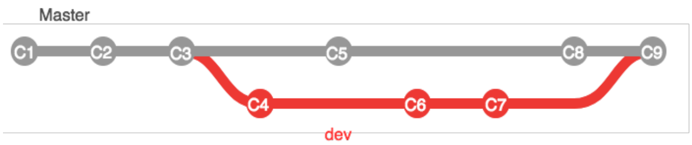


Figura 1: Exemplo de merge entre ramos

Quando o *merge* falha, o desenvolvedor precisa resolver o conflito manualmente. Quando o *merge* manual é necessário, o desenvolvedor deve escolher entre uma das duas versões, concatenar as duas versões em alguma ordem, combinar as duas versões de alguma forma ou escrever um novo código [Menezes 2016].

Um estudo [Menezes 2016] mostrou que 75% dos conflitos ocorridos nos repositórios estudados foram resolvidos escolhendo-se uma das duas versões. Quando o desenvolvedor opta por uma versão em detrimento da outra, existe um *trabalho desperdiçado* [Prudêncio et al. 2012], já que houve a criação de código em um dos ramos que não foi incorporado no *merge*. Na Figura 2 podemos ver que o trabalho desperdiçado é o código de uma ação que foi realizada, mas que não foi integrada durante o *merge*. No mesmo estudo, Menezes [2016] concluiu que 13% dos conflitos foram solucionados com código novo, ou seja, demandando um esforço adicional durante o *merge*. Denominamos esse esforço adicional como *trabalho extra*, que consiste nas ações que foram integradas ao código, mas que não foram realizadas anteriormente ao merge. Além disso, quantificamos também o *retrabalho*, que acontece quando existe interseção entre as ações criadas em paralelo, como na Figura 2.

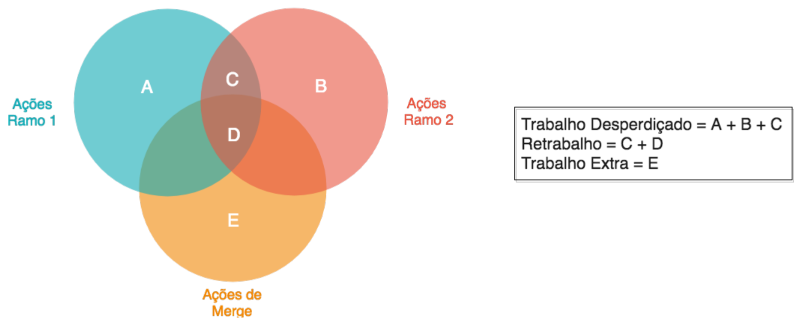


Figura 2: Tipos de métricas

Para representar cada conjunto das ações, utilizamos *multisets* [Knuth 1998], tendo em vista que pode haver repetições de ações nos conjuntos. Cada *multiset* contém as ações que representam as adições ou remoções de linhas de código. Os *multisets* são compatíveis com as operações usuais de conjuntos, como diferença, interseção e união, além de adicionar uma operação de soma. Por exemplo, supondo $A = \{a, a, b, c, c\}$ e $B = \{a, b, b, c, c\}$, temos $A \backslash B = \{a\}$, $B \backslash A = \{b\}$, $A \cap B = \{a, b, c, c\}$, $A \cup B = \{a, a, b, b, c, c\}$ e $A + B = \{a, a, a, b, b, b, c, c, c, c\}$.

Com o intuito de extrair as métricas, é necessário primeiro fazer a análise dos *commits* do repositório a procura dos *commits* de *merge*. Para isso, verificamos cada um dos *commits* em busca daqueles que têm dois pais. Como merges com mais de dois pais, denominados *octopus*, não podem ter conflitos e edições manuais por definição, esses casos teriam zero esforço e são ignorados. A partir de um *commit* de *merge* denominado $commit_{merge}$, são obtidos os seus *commits* pais e, a partir deles, é identificado o *commit* em que eles derivaram, denominado $commit_{base}$. Diante dessas informações, são realizadas três operações de *diff* para obter as referidas ações. O primeiro *diff* é realizado entre $commit_{base}$ e $commit_{merge}$, obtendo-se assim as ações que foram de fato incorporadas no *merge*. Definimos formalmente as ações de *merge* como:

$$actions_{merge} = diff(commit_{base}, commit_{merge})$$

Em seguida, é realizado o *diff* entre $commit_{base}$ e $commit_{parent1}$, visando obter as ações que foram feitas no ramo 1. Logo, as ações do ramo 1 são formalmente definidas como:

$$actions_{branch1} = diff(commit_{base}, commit_{parent1})$$

Por fim, é feito o *diff* entre $commit_{base}$ e $commit_{parent2}$, retornando as ações feitas no ramo 2. Portanto, as ações do ramo 2 são definidas formalmente como:

$$actions_{branch2} = diff(commit_{base}, commit_{parent2})$$

A partir dessas ações, é possível identificar as ações que representam retrabalho, trabalho desperdiçado e trabalho extra. Para o cálculo do retrabalho é feita a interseção entre as ações feitas nos ramos. O retrabalho é definido formalmente como:

$$actions_{rework} = actions_{branch1} \cap actions_{branch2}$$

A fim de determinar o trabalho desperdiçado e o trabalho extra é necessário primeiro identificar as ações que foram feitas em algum dos ramos. Portanto, definimos as ações dos ramos formalmente como:

$$actions_{branches} = actions_{branch1} + actions_{branch2}$$

Para calcular o trabalho desperdiçado, usamos o complemento relativo das ações de *merge* nas ações dos ramos, formalmente definido como:

$$actions_{wasted} = actions_{branches} \setminus actions_{merge}$$

Por fim, para determinar o trabalho extra usamos o complemento relativo das ações dos ramos nas ações de merge, formalmente definido como:

$$actions_{extra} = actions_{merge} \setminus actions_{branches}$$

Desta forma, é possível obter as métricas relativas a essas ações tanto de forma absoluta quanto de forma normalizada, conforme descrito a seguir:

$$rework = |actions_{rework}|$$

$$rework_{normalized} = rework \;/\; |actions_{branch1} \cup actions_{branch2}|$$

$$wasted = |actions_{wasted}|$$

$$wasted_{normalized} = wasted \;/\; |actions_{branches}|$$

$$extra = |actions_{extra}|$$

$$extra_{normalized} = extra \;/\; |actions_{merge}|$$

3. Implementação

Os conceitos discutidos na seção 2 foram implementados em uma ferramenta, que permite a coleta automática das métricas anteriormente apresentadas. A ferramenta foi desenvolvida em Python 3.6 e está disponível em https://github.com/gems-uff/merge-effort. O usuário pode utilizar a ferramenta de duas formas: via API ou através de um interpretador de linha de comando. Para ilustrar a utilização da ferramenta nessa seção, utilizamos o interpretador de linha de comando.

O usuário pode analisar tanto um repositório remoto, por exemplo, disponível no GitHub, quanto um repositório local, que já esteja em seu computador. Caso o usuário não utilize um repositório local, a ferramenta clonará o repositório temporariamente para fazer as análises. A ferramenta também disponibiliza a opção do usuário determinar uma lista de *commits* a serem analisados. Por default, caso o usuário não informe os *commits*, a ferramenta analisará todos os *commits* daquele repositório. Além disso, é possível informar se deseja obter as métricas absolutas ou normalizadas.

Criamos o exemplo da Figura 3 (a) para ilustrar o processo do reconhecimento dos esforços explicados na Seção 2. Esse exemplo faz uso de uma função fatorial intencionalmente simples, visando facilitar o entendimento da técnica proposta. A Figura 3 (b) apresenta o resultado dos *diffs* discutidos na Seção 2. Através do exemplo obtemos os seguintes esforços:

$$actions_{rework} = \{\text{"- if i<1:", "- fat(4)", "+ fat\_4= fat(4)", "+ print(fat\_4)"}\}$$

$$actions_{wasted} = \left\{\begin{array}{c}\text{"+ fat\_4= fat(4)", "+ fat\_4= fat(4)", "- if i< 1:",}\\ \text{"- fat(4)", "+ print(fat\_4)", "+ if i<= 1"}\end{array}\right\}$$

$$actions_{extra} = \{\text{"-def fat(i): ", "+ def fat\_iterativo(i):", "+ fat= fat\_iterativo(4)"}\}$$

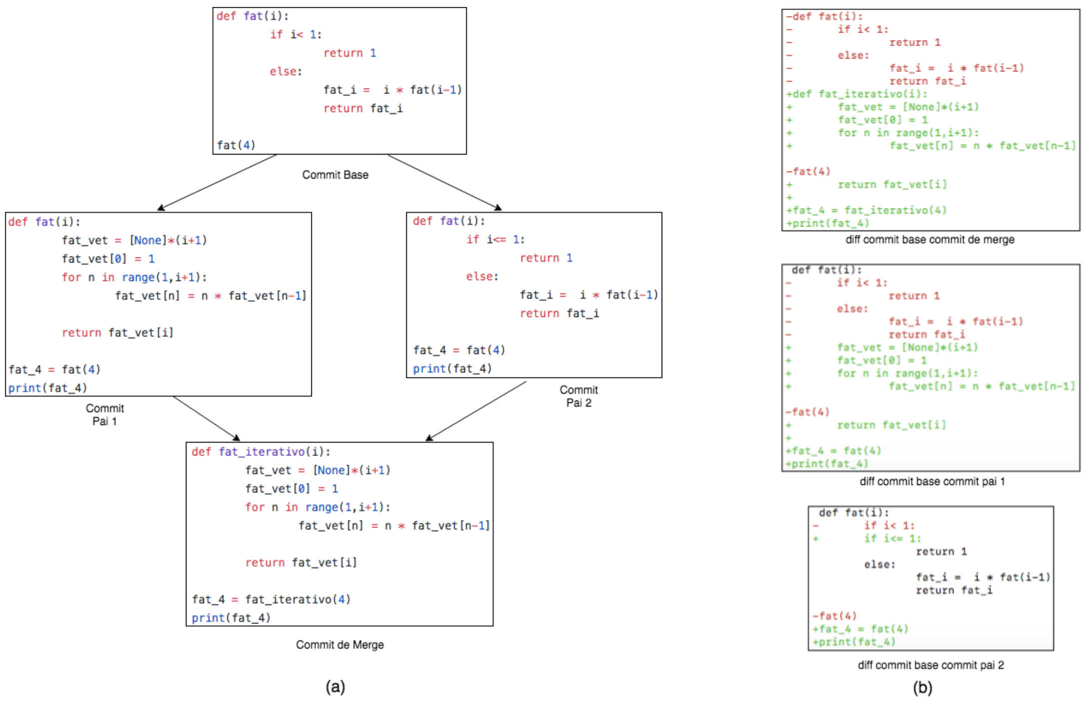


Figura 3 Exemplo de merge e seus diffs

Após todos os cálculos, a ferramenta retorna os valores de retrabalho, trabalho desperdiçado e trabalho extra. Para o exemplo, a saída da ferramenta é: 14 ações realizadas pelo ramo 1, 5 ações realizadas pelo ramo 2, 16 ações realizadas no *merge*, 4 ações de retrabalho, 6 ações de trabalho desperdiçado e 3 ações de trabalho extra.

4. Avaliação

Com o propósito de avaliar a utilidade da nossa ferramenta analisamos cinco repositórios *open-source* obtidos no GitHub. Utilizamos os seguintes critérios para a escolha dos projetos: ter pelo menos 1.000 *commits* e pelo menos 10 contribuidores. A Tabela 1 contém os cinco projetos selecionados, o número total de *commits*, o número total de contribuidores, a linguagem e a quantidade de *commits* de *merge*.

Tabela 1 Projetos Selecionados

Projeto	Commits	Contribuidores	Linguagem	Commits de Merge
Sapos	1.344	11	Ruby	153
Voldemort	4.262	61	Java	495
DBAAS	5.079	18	Python	677
Jquery	6.327	271	JavaScript	250
Firefox-ios	6.936	114	Swift	1.505

Através da análise dos resultados da nossa ferramenta para os projetos selecionados, construímos a Tabela 2, onde temos a porcentagem de *commits* que têm cada um dos esforços em relação à quantidade de *commits* de *merge* e a média de ações de cada esforço em relação à quantidade de *commits* de *merge*. Concluímos que o trabalho extra (*extra)* é o que tem menor ocorrência, acontecendo entre 0,47% a 12,42% dos casos. Vale notar que esses valores estão alinhados com o que é discutido na literatura [Menezes 2016], onde 13% dos *chunks* em conflitos foram resolvidos com código novo, ou seja, houve esforço adicional. Em contrapartida, o retrabalho (*rework)* e trabalho desperdiçado (*wasted*) ocorrem com mais frequência, entre 8,7% a 30,4% e 2,06% a 31,37%, respectivamente. É importante ressaltar que o projeto Firefox-ios é um *outlier* quando comparado aos outros. Se o excluíssemos da análise, obteríamos 16,54% a 30,4% de retrabalho e 11,96% a 31,37% de trabalho desperdiçado.

Tabela 2 Resultados de retrabalho, trabalho desperdiçado e trabalho extra

Projeto	Commits de Merge			Média de Ações		
	Rework	Wasted	Extra	Rework	Wasted	Extra
Sapos	30,07%	31,37%	12,42%	72,80	78,01	6,49
Voldemort	24,04%	18,99%	7,68%	101,02	178,66	11,72
DBAAS	16,54%	11,96%	4,14%	19,72	21,31	1,82
Jquery	30,40%	21,20%	8,00%	22,89	23,21	0,42
Firefox-ios	8,70%	2,06%	0,47%	3,20	3,60	0,25

Em relação a análise de média de ações de esforço, é possível notar que a quantidade média de ações de trabalho extra é significativamente menor que as demais. Além disso, observamos um padrão entre a quantidade média de ações de retrabalho e de trabalho desperdiçado, já que os valores são próximos. Por fim, um outro padrão observado é que a quantidade média de ações de retrabalho é sempre menor do que a de trabalho desperdiçado. Contudo, uma análise mais profunda e abrangente deve ser feita para obter evidências mais fortes sobre tais padrões e identificar as suas razões e implicações.

5. Trabalhos Relacionados

Prudêncio et al. [2012], com o intuito de reduzir a quantidade de conflitos, criaram a ferramenta Orion para avaliar qual é a política de controle de concorrência mais adequada para cada elemento de software. Para esse fim, a ferramenta calcula duas métricas: *merge effort* e *concurrency*. Apesar de utilizar o cálculo de esforço de merge, o objetivo da ferramenta não é retornar as métricas de esforço para o usuário da forma que nossa ferramenta faz, mas sim propor uma política de controle de concorrência. Posteriormente, Santos et al. [2012] realizaram um estudo em que propõem dez métricas diferentes para a estimar o esforço de *merge*. Dentre as métricas a que obteve melhor resultado foi *Quantidade de Conflitos Físicos*. Além disso, Santos et al. [2012] apresentaram métricas que se baseiam na quantidade de linhas modificadas. Ambos os trabalhos fazem uso de conjuntos para o cálculo de esforço de merge, e não multisets como propomos nesse trabalho, levando a potenciais erros quando as mesmas ações acontecem em paralelo. Além disso, os estudos não abordam a métrica de retrabalho.

Em um outro trabalho, Mehdi et al. [2014] propuseram a análise de *merge* através da análise de duas métricas: *merge blocks* e *merge lines*. *Merge blocks* representa a quantidade de blocos modificados para resolver o conflito e *merge lines* representa a quantidade de linhas modificadas nos blocos. O objetivo da ferramenta criada por Mehdi et al. [2014] é utilizar as métricas para poder comparar o quão bom ou ruim pode ser um algoritmo de *merge*. Portanto, a ferramenta não tem o objetivo de retornar ao usuário as métricas de esforço, mas somente utilizá-las como um meio para avaliar os algoritmos de *merge*.

Em um estudo mais recente, Cavalcanti et al. [2017] analisou os esforços de *merge* semi-estruturado e *merge* não estruturado (convencional) através da quantidade de conflitos falsos positivos, o tempo necessário para solucionar o conflito de *merge* e o tempo necessário para pensar em como solucionar esse conflito. Na pesquisa, foi proposta uma ferramenta para análise de *merge* semi-estruturado com o intuito de reduzir o número de conflitos falso positivos e falso negativos. Novamente, a ferramenta sugerida não tem o propósito de retornar ao usuário as métricas de esforço de *merge*. Além disso, as métricas sugeridas não permitem a construção de uma ferramenta automática, já que não há como quantificar automaticamente com precisão o tempo que um desenvolvedor leva para pensar em como solucionar um conflito.

Além dos pontos já abordados, nenhum dos estudos tem a sua técnica de cálculo de esforço de *merge* detalhada o suficiente para permitir a sua reprodutibilidade.

6. Conclusão

Existem estudos que retrataram o esforço de merge, mas não foi encontrado um estudo cujo foco era a coleta de métricas de esforço e que relatasse com detalhes a sua técnica. Além disso, nenhum dos estudos relacionados propôs o uso de *multisets* para representar o conjunto de ações, o que é o mais indicado, tendo em vista que é possível a repetição de ações.

Esse estudo expôs, detalhadamente, uma técnica para extração de três métricas: retrabalho, trabalho desperdiçado e trabalho extra, além de apresentar uma implementação para essa técnica. Exibimos, também, uma análise feita com a nossa ferramenta onde pudemos concluir que o trabalho extra não acontece com muita frequência. Além disso, identificamos que o retrabalho e o trabalho desperdiçado acontecem entre 10% a 30% dos *commits* de merge. Até onde temos conhecimento, nenhum outro trabalho estudou a frequência de ocorrência de retrabalho ou trabalho desperdiçado. Também foram analisadas as quantidades médias de ações de cada tipo de esforço e concluiu-se que também há uma média maior de ações de trabalho desperdiçado e retrabalho.

Optamos por fazer a coleta das métricas no nível físico por ser o nível mais usado pelos controles de versão atuais, mas essa escolha nos leva a limitações. Dessa forma a ferramenta pode retornar falsos positivos, como por exemplo, o caso em que o usuário troca *tab* por espaços. Por isso, em um trabalho futuro, pretendemos adaptar a ferramenta para que possa fazer análise sintática, no nível da Árvore Sintática Abstrata.

Por fim, vemos como trabalho futuro um estudo mais profundo entre a possível correlação entre retrabalho e trabalho desperdiçado. Além disso, pretendemos estender a nossa análise para um universo maior de projetos. Por fim, estudaremos as possíveis razões para a ocorrência de retrabalho, trabalho desperdiçado e trabalho extra.

Referências

Accioly, P., Borba, P. and Cavalcanti, G. (2017). Understanding semi-structured merge conflict characteristics in open-source Java projects. *Empirical Software Engineering*, pages 1–35. Springer.

Brun, Y., Holmes, R., Ernst, M. D. and Notkin, D. (2011). Proactive detection of collaboration conflicts. In *European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 168-178. ACM.

Cavalcanti, G., Borba, P. and Accioly, P. (2017). Evaluating and improving semistructured merge. *Proceedings of the ACM on Programming Languages*, v. 1, n. OOPSLA, page 59. ACM.

Grinter, R. (1995). Using a configuration management tool to coordinate software development. In *Proceedings of conference on Organizational computing systems*, pages 168 - 177. ACM.

Kasi, B. K. and Sarma, A. (2013). Cassandra: Proactive Conflict Minimization Through Optimized Task Scheduling. In *International Conference on Software Engineering*, ICSE '13, pages 732-741. IEEE.

Knuth, D. E. (1998). *The art of computer programming, Seminumerical Algorithms*. v. 2

Mehdi, A.-N., Urso, P. and Charoy, F. (2014). Evaluating software merge quality. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. Article No 9. ACM.

Menezes, G. (2016). On the nature of software merge conflicts. Universidade Federal Fluminense - UFF.

Prudêncio, J. G., Murta, L., Werner, C. and Cepêda, R. (2012). To lock, or not to lock: That is the question. *Journal of Systems and Software*, v. 85, n. 2, pages 277–289.

Santos, R. and Murta, L. G. P. (2012). Evaluating the Branch Merging Effort in Version Control Systems. In *26th Brazilian Symposium on Software Engineering*, SBES '12, pages 151-160. IEEE.

Yuzuki, R., Hata, H. and Matsumoto, K. (2015). How we resolve conflict: an empirical study of method-level conflict resolution. In *International Workshop on Software Analytics, SWAN*, pages 21-24. IEEE.

Towards an automated approach for bug fix pattern detection

Fernanda Madeiral[1], **Thomas Durieux**[2], **Victor Sobreira**[1], **Marcelo Maia**[1]

[1] Federal University of Uberlândia, Brazil

[2]INRIA & University of Lille, France

fernanda.madeiral@ufu.br, thomas.durieux@inria.fr, {victor, marcelo.maia}@ufu.br

Abstract. *The characterization of bug datasets is essential to support the evaluation of automatic program repair tools. In a previous work, we manually studied almost 400 human-written patches (bug fixes) from the Defects4J dataset and annotated them with properties, such as repair patterns. However, manually finding these patterns in different datasets is tedious and time-consuming. To address this activity, we designed and implemented PPD, a detector of repair patterns in patches, which performs source code change analysis at abstract-syntax tree level. In this paper, we report on PPD and its evaluation on Defects4J, where we compare the results from the automated detection with the results from the previous manual analysis. We found that PPD has overall precision of 91% and overall recall of 92%, and we conclude that PPD has the potential to detect as many repair patterns as human manual analysis.*

1. Introduction

Automatic program repair is a recent research field where approaches have been proposed to fix software bugs automatically, without human intervention [Monperrus 2018]. In automatic program repair, empirical evaluation is conducted by running repair tools on *known bugs* to measure the repairability potential. These known bugs are available on bug datasets, e.g., Defects4J [Just et al. 2014].

Building datasets of bugs is a challenging task. Despite the effort made by authors of bug datasets, such datasets generally do not include detailed information on the bugs and their patches (bug fixes), so a fair and advanced evaluation of repair tools becomes harder. We highlight two tasks that make the evaluation of repair tools more robust:

- *Selection of bugs* is used to filter out bugs that do not belong to the bug class which the repair tool under evaluation targets. For instance, NPEFix [Durieux et al. 2017] is a tool specialized in null pointer exception fixes, and it will probably fail on bugs that were not fixed by a human with a null pointer checking. So, a coherent and fair analysis would include only bugs within the target bug class(es) of the respective tools.
- *Correlation analysis* is an advanced analysis of the results produced by a repair tool, making possible to derive conclusions such as "the repair tool performs well on bugs having the property X". This kind of analysis requires a characterization of all bugs available in the used dataset.

To support these two tasks, in a previous work [Sobreira et al. 2018], we *manually* analyzed the patches of the Defects4J dataset [Just et al. 2014], which is a widely used dataset in automatic program repair field. As a result, we delivered a taxonomy of properties and the annotation of Defects4J patches according to such taxonomy. Despite the value of manual work, analyzing patches to calculate or to find properties when characterizing patches from different datasets is tedious and time-consuming. Nevertheless, the already built taxonomy is a useful resource to guide the automation of patch analysis.

In this paper, we present PPD (Patch Pattern Detector), a detector of *repair patterns* in patches, one of the existing types of property in our previous taxonomy. Repair patterns are recurring abstract structures in patches. For instance, a patch that affects only one line in the buggy code is an instance of the pattern *Single Line*, while a patch that adds an entire conditional block is an instance of the pattern *Conditional Block Addition*.

PPD analyzes a given patch by first retrieving edit scripts at Abstract-Syntax Tree (AST) level from the *diff* between the buggy and patched code using GumTree [Falleri et al. 2014]. Then, PPD searches for instances of patterns by analyzing the AST nodes of the *diff* using Spoon [Pawlak et al. 2015], a peer-reviewed library to analyze Java source code. We evaluated PPD and found that it has the potential of detecting the nine repair pattern groups from Sobreira et al. (2018). PPD can support automatic repair researchers on selecting bugs from bug datasets and performing correlation analysis between repaired bugs and their properties. Moreover, PPD can be useful in comparisons between different datasets of bugs. By discovering the constitution of bug datasets, it is possible to study the balance between them and also the flaws.

To sum up, the main contributions of this paper are 1) a tool to detect repair patterns from patches written in Java, which is publicly available, and 2) the detection of new 90 instances of the patterns on Defects4J.

2. Taxonomy of Repair Patterns

Previously, we delivered a taxonomy of repair patterns containing nine groups and 25 patterns in total [Sobreira et al. 2018]. PPD implements the detection of all these patterns. In this section, we briefly define the nine pattern groups. Additionally, we refer in this paper to patches from Defects4J (using a simple notation with project name followed by bug id) to provide examples with external links for patch visualization, e.g., Chart-1[1].

Conditional Block involves the addition or removal of conditional blocks (e.g., Lang-45).

Expression Fix involves actions on logic (Chart-1) or arithmetic (Math-80) expressions.

Wraps/Unwraps consists of (un)wrapping existing code with/from high-level structures such as try-catch blocks (Closure-83) and low-level ones such as method calls (Chart-10).

Single Line is dedicated to patches affecting one single line or statement (Closure-55).

Wrong Reference occurs when the code references a wrong variable (e.g., Chart-11) or method call (e.g., Closure-10) instead of another one.

Missing Null-Check is related to the addition of a conditional expression or the expansion of an existing one with a null-check that was missing in the code (e.g., Chart-15).

Copy/Paste is the application of the same change to different points in the code (Chart-19).

Constant Change involves changes in literals or constant variables (e.g., Closure-65).

Code Moving involves moving code statements or statement blocks around, without extra changes to these statements (e.g., Closure-117).

[1] For paper printed version: all links on Defects4J patches can be built by inserting two parameters in `http://program-repair.org/defects4j-dissection/#!/bug/⟨project_name⟩/⟨bug_id⟩`. Example: Chart-1 contains the link `http://program-repair.org/defects4j-dissection/#!/bug/Chart/1`

3. PPD: a Detector of Bug Fix Patterns

The detection of repair patterns in patches falls in source code change analysis task. Analyzing source code changes can be performed at different levels of granularity such as file level, line level, and AST level. Our approach is at the AST level, and it consists of two main tasks to detect repair patterns in a given patch:

Retrieval of the AST diff: Given as input the buggy version of the program and the patch file (*diff* file), PPD retrieves the AST *diff* between the buggy and patched code (also known as *edit scripts*) using the GumTree algorithm [Falleri et al. 2014]. There are different implementations of the GumTree algorithm: we use GumTree Spoon[2] since such tool delivers the AST *diff* nodes in the representation of the Spoon library [Pawlak et al. 2015], based on a well-designed meta-model for representing Java programs. Therefore, we can analyze the edit scripts returned by GumTree with Spoon.

Analysis of the AST diff: With the AST *diff* retrieved, the AST nodes are analyzed to detect the repair patterns. PPD contains a set of detectors, one for each pattern group, because each pattern group has its own definition, which lead us to define a specific strategy for the detection of each of them[3]; the strategies are mainly based on searching and checking code elements/structures in the AST *diff*. However, all detectors follow the same general process: it analyzes edit scripts using Spoon based on the defined strategy to detect the pattern it was designed for. Thus, we choose one pattern, *Missing Null-Check*, to be used as an example to describe in details how PPD performs automatic detection. Given the edit scripts from a patch, the strategy of the *Missing Null-Check* detector is the following:

1. It searches for the addition of a binary operator where one of the two elements is `null`, i.e., a null-check;
2. It extracts from the null-check the variable being checked (`variable <operator> null`) or the variable being used to call a method where its return is being checked (`variable.methodCall() <operator> null`);
3. It verifies if the extracted variable is new, i.e., was added in the patch: a) if the variable is not new, a missing null-check was found; b) if the variable is new, it verifies if the new null-check wraps existing code: if it does, a missing null-check was found.

Consider the *diff* in Listing 1. In the buggy version of this code, in the old line 2166, a null pointer exception had been thrown when the variable `markers` was null and accessed for a method call. In the fixed version, a conditional block was added to check whether `markers` is null, and in such case, the method returns, so the program execution does not reach the point of the exception. The added null-check is an instance of the pattern *Missing Null-Check*. Note that the null-check was added in a new conditional block, so this patch also contains an instance of the pattern *Conditional Block Addition with Return Statement*. Additionally, this conditional block was added in four different locations on the code (see Chart-14), which consists in the *Copy/Paste* pattern.

```
2166       + if (markers == null) {
2167       +     return false;
2168       + }
2166  2169   boolean removed = markers.remove(marker);
```

Listing 1. Patch for bug Chart-14.

[2] `https://github.com/SpoonLabs/gumtree-spoon-ast-diff`

[3] PPD was designed in a modularized way that makes possible the addition of a new pattern detection by extending an existing class and implementing a strategy for the detection of the new pattern.

A missing null-check can appear in different variants beyond the addition of an entire conditional block. In Listing 2, for instance, the missing null-check was added in a new conditional by wrapping an existing block of code. This type of change consists of the pattern *Wraps-with if*. Different from *Conditional Block Addition*, the body of the conditional contains existing code in *Wraps-with if*.

```
1191 1191    ChartRenderingInfo owner = plotState.getOwner();
     1192 + if (owner != null) {
1192 1193       EntityCollection entities = owner.getEntityCollection();
1193 1194       if (entities != null) {
1194 1195          entities.add(new AxisLabelEntity(this, hotspot,
1195 1196                  this.labelToolTip, this.labelURL));
1196 1197       }
     1198 + }
```

Listing 2. Patch for bug Chart-26.

Since our detector searches for binary operators involving null-check, it also detects missing null-checks in other structures beyond `if` conditionals. In Listing 3, for instance, there is an example of a conditional using the ternary operator. When the ternary operator is used, and an existing expression is placed in the `then` or `else` expression, we have the pattern *Wraps-with if-else*. Note that this patch is also an instance of the pattern *Single Line* since only one line was affected by the patch.

```
29     - description.appendText(wanted.toString());
   29 + description.appendText(wanted == null ? "null" : wanted.toString());
```

Listing 3. Patch for bug Mockito-29.

For these three example patches, PPD was able to detect all the existing patterns in them, according to the taxonomy presented in Section 2.

4. Evaluation

Method. Our evaluation consists of running PPD on real patches to measure its ability at detecting the 25 repair patterns.

Subject Dataset. The patches used as input to PPD are from Defects4J [Just et al. 2014], which consists of 395 patches from six real-world projects (e.g. Apache Commons Lang and Mockito Testing Framework). We chose this dataset since it contains real bugs and all its patches have been annotated with repair patterns [Sobreira et al. 2018], allowing the direct comparison between results generated by PPD and the previous manual detection.

Result analysis. We analyzed the results in two steps. First, we calculated the precision and recall of the PPD for each pattern, using the available manual detection [Sobreira et al. 2018] as an oracle. We refer to such manual detection as *human detection*, while we refer to the detection produced by PPD as *automatic detection*. Second, we performed manual analysis on the disagreements between the automatic and human detection. For each pattern, two different authors of this paper analyzed all patches where there were disagreements and determined whether PPD actually missed or wrongly detected such pattern. We annotated the disagreements with one of the five diagnostics presented in the first column of Table 2. Then, we calculated the actual precision and recall for each pattern, using the following formulas: $TP = A + B + DC$, $precision = \frac{TP}{TP+DW}$, $recall = \frac{TP}{TP+HC}$, where A is the number of agreements between PPD and human detection, B is the disagreements when both PPD and human detection may be accepted, DC is the disagreements when PPD detection is correct and DW when PPD detection is wrong, and HC is the disagreements when the human detection is correct.

Table 1. PPD performance.

Pattern	Variant	Prior Precision (%)	Prior Recall (%)	Post Precision (%)	Post Recall (%)
Conditional Block	Addition	74.75	93.67	99.00	98.02
	″ with Return Statement	90.12	94.81	100.00	96.47
	″ with Exception Throwing	93.75	90.91	96.88	91.18
	Removal	60.71	77.27	86.67	89.66
Expression Fix	Logic Modification	82.22	75.51	91.11	83.67
	″ Expansion	90.20	95.83	92.16	97.92
	″ Reduction	76.92	83.33	76.92	100.00
	Arithmetic Fix	69.57	50.00	91.67	64.71
Wraps/Unwraps	Wraps-with if	74.19	95.83	83.87	96.30
	″ if-else	81.25	84.78	92.00	90.20
	″ else	16.67	100.00	33.33	100.00
	″ try-catch	100.00	100.00	100.00	100.00
	″ method	78.57	78.57	85.71	85.71
	″ loop	40.00	100.00	60.00	100.00
	Unwraps-from if-else	42.11	61.54	57.89	68.75
	″ try-catch	100.00	100.00	100.00	100.00
	″ method	45.45	83.33	54.55	85.71
Single Line	–	100.00	97.96	100.00	100.00
Wrong Reference	Variable	66.67	76.19	82.35	89.36
	Method	68.42	83.87	86.84	89.19
Missing Null-Check	Positive	95.45	84.00	100.00	100.00
	Negative	96.67	90.63	100.00	96.77
Copy/Paste	–	56.16	85.42	91.78	90.54
Constant Change	–	77.27	89.47	90.91	90.91
Code Moving	–	60.00	85.71	81.82	100.00
Overall		78.26	86.95	91.53	92.39

Results. The evaluation results are presented in Table 1: for each pattern, this table shows the precision and recall before (column "prior") and after (column "post") the disagreement analysis.

We observed that PPD has a high overall precision and recall, even when just comparing it directly with the human detection (see the last line in the table). For the most recurring pattern group, *Conditional Block*, both detections agreed on 194 instances of such pattern (prior). After the disagreement analysis, we found that PPD detected 39 new instances of such pattern, which increased the precision and recall of the PPD (post), for at least 86% and 89%, respectively.

For some less recurring patterns, *Single Line* and *Missing Null-Check*, PPD performed well by detecting 96 and 50 instances of these patterns in agreement with the human detection, respectively. In fact, for *Single Line*, the only two instances missing by PPD were not truly instances of such pattern.

However, we identified some particular patterns that PPD did not perform well. PPD found 95 instances of the patterns from the group *Wraps/Unwraps* in agreement with the human detection. On the disagreement analysis, we identified that PPD detected 7 new instances of this group, but that also generated 30 false positives. The major responsible for these false positives are the pattern variants involving `if`, `else` and `method`.

Table 2. Overall absolute results on the disagreement analysis and reasons for automatic detection differing from the manual detection.

Diagnostic	# Occurrences	Related Reason
DW (PPD false positive)	73	#1, #7
DC (PPD true positive)	90	#1, #4
HW (human detection false positive)	24	#5, #6
HC (human detection true positive)	65	#2, #7
B (both could be accepted)	33	#1, #3
A (agreements)	666	
TP (correct detection = A + B + DC)	789	

Discussion. During the disagreement analysis, we also investigated why PPD failed or differed from the human analysis. Table 2 relates the diagnostics with the reasons for the disagreements, which we discuss as follows.

Reason #1: Global human vision versus AST-based analysis. The GumTree algorithm identifies implicit structures that are not visible by humans. For instance, in Mockito-18, both automatic and manual detections found the pattern *Conditional Block Addition with Return Statement*. However, the automatic detection also found the pattern *Wraps-with if-else*. In this patch, the human sees the structure as in Listing 4, while the structure considered by PPD is like in Listing 5. In other words, the new conditional block wraps a part of the code, but with an implicit block. On these occurrences, we considered that both automatic and manual detection could be accepted.

```
+ } else if (type == Iterable.class) {
+     return new ArrayList<Object>(0);
  } else if (type == Collection.class)
      {
  [...]
```

Listing 4. Human vision.

```
+ } else {
+     if (type == Iterable.class) {
+         return new ArrayList<Object>(0);
      } else {
          if (type == Collection.class) {
  [...]
```

Listing 5. AST-based analysis.

Still on the global human vision versus AST-based analysis discussion, due to fine-grained changes, PPD takes into account small changes that do not make sense as the composition of a pattern in some cases. For instance, PPD detected the *Copy/Paste* pattern in Chart-3. Even though the two additions have a high similarity, these changes are not enough to be considered as an instance of the pattern *Copy/Paste*, so we determined this as a false positive generated by PPD.

In the same direction, PPD takes into account *relevant* small changes that humans may not identify in big patches. In these big patches, humans may intuitively consider only the global vision of the patch and miss smaller changes. For instance, Math-64 has several changes: one of them is the addition of a block with three lines of code in two different locations (i.e., *Copy/Paste*), which was missed by human detection.

Reason #2: The automatic detection relies on rules defined by humans (i.e., the authors of this paper), and it is difficult to identify all cases where an instance of a pattern may exist, thus PPD missed some pattern instances. The *Expression Fix* detector is the primary responsible for these missing detections. For instance, it missed the detection of an arithmetic expression fix in Math-77, where an arithmetic operation occurs with the assignment operator +=, which was replaced by a non-arithmetic assignment operator.

Reason #3: There are some borderline cases where a given pattern may fit or not. For instance, in line 1167 of Time-17, one could consider the removed method call as a part of the arithmetic expression used as argument for such method call, and another one could not. Only the manual detection detects such statement as an instance of the *Arithmetic Expression Fix* pattern, but we considered that detecting it or not can be both accepted.

Reason #4: The automatic detection applies the same rules for all the patches while it is a difficult task to be done by humans. Therefore, some pattern instances were missed by the manual detection due inconsistencies between patches.

Reason #5: In the manual analysis, humans may consider the semantic of the changes (even without noticing) and make assumptions on how the developer could write a patch that matches one of the patterns. For instance, Mockito-28 had been considered as an instance of the *Single Line* pattern. Semantically, it could be correct, but such pattern should be limited to changes affecting a single line or a single statement in a given patch.

Reason #6: A misconception of the patch can impact the human analysis. For instance, in Lang-50, the manual detection found an instance of the pattern *Logic Expression Modification* in line 285 (and also in the new line 463, which is the same case, i.e. *Copy/Paste*). However, the existing conditional block in line 285 was actually completely changed: the statement inside it was unwrapped in the patch, and the conditional was deleted. Then, an existing conditional block in the code took the place of the conditional considered as having its logic expression modified, i.e. a moving happened, not characterizing a genuine logic expression modification.

Reason #7: GumTree is a sophisticated algorithm that may return imprecise results for some patches. For example, it can consider the change over some elements when it is not the case. As a consequence, PPD incorrectly detects or misses some pattern instances.

5. Threats to Validity

Internal validity. The ultimate precision and recall calculated when evaluating the performance of the PPD are based on a manual disagreement analysis. This analysis can be subject to small errors and misconception, typical of any manual work. To mitigate this, such analysis was performed to each pattern group by two authors of this paper, in live discussion sessions.

External validity. We have evaluated PPD on patches from Defects4J. However, since Defects4J may not be representative on all different cases on fixing bugs using one of the 25 patterns, it is possible that PPD still cannot generalize for systems including patches that differ a lot from those in Defects4J. Moreover, detected repair patterns are Java-based, therefore our detector is limited to systems written in this language.

6. Related Work

Martinez et al. [Martinez et al. 2013] also reported on the automatic detection of bug fix patterns at the AST level. The main differences between their work and our work are the following. First, they focused on 18 bug fix patterns from [Pan et al. 2009] while we focused on 25 patterns from [Sobreira et al. 2018]. Second, they used the ChangeDistiller AST differencing algorithm [Fluri et al. 2007] while we use GumTree [Falleri et al. 2014]. The latter outperforms the former by maximizing the number of AST node mappings, minimizing the edit script size, and detecting better move actions [Falleri et al. 2014]. Moreover, they pointed out that ChangeDistiller works at the statement level, preventing the detection of certain fine-grain patterns. Third, they formalized a representation for change patterns and used this representation to specify patterns. Then, to detect a pattern, a match of its specification must happen in a given edit script. However, such representation is based on change type (e.g. addition) over code elements (e.g. `if`), which does not support the specification of patterns such as *Single Line*.

7. Final Remarks

In this paper, we report on PPD, a detector of repair patterns in bug fixes. Through an evaluation on Defects4J, we found that PPD has a good performance in general, and for some patterns (e.g., *Missing Null-Check*) it can even perform better than human detection. Moreover, a fruit of the disagreement analysis, we found that human detection made fewer mistakes (24) than PPD (73), but also detected less exclusive occurrences (65) than PPD (90). As future work, we intend to conduct experiments over other bug datasets to evaluate the scalability of PPD and also to compare bug datasets, which may guide researchers on automatic program repair at choosing datasets when evaluating their tools. Finally, we intend to create a visualization for patches where the repair patterns are highlighted, to support the human patch comprehension task.

Tool Availability. PPD is part of the project ADD, which is publicly available at:

`https://github.com/lascam-UFU/automatic-diff-dissection`

One can find instructions in such repository on how to use PPD and also to reproduce the results on Defects4J presented in our evaluation (Section 4).

References

Durieux, T., Cornu, B., Seinturier, L., and Monperrus, M. (2017). Dynamic Patch Generation for Null Pointer Exceptions Using Metaprogramming. In *SANER '17*.

Falleri, J.-R., Morandat, F., Blanc, X., Martinez, M., and Monperrus, M. (2014). Fine-grained and Accurate Source Code Differencing. In *ASE '14*, pages 313–324.

Fluri, B., Wuersch, M., PInzger, M., and Gall, H. (2007). Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. *TSE*, 33(11):725–743.

Just, R., Jalali, D., and Ernst, M. D. (2014). Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *ISSTA '14*, pages 437–440.

Martinez, M., Duchien, L., and Monperrus, M. (2013). Automatically Extracting Instances of Code Change Patterns with AST Analysis. In *ICSM '13*, pages 388–391.

Monperrus, M. (2018). Automatic Software Repair: a Bibliography. *ACM Computing Surveys*, 51(1):17:1–17:24.

Pan, K., Kim, S., and Whitehead, Jr., E. J. (2009). Toward an understanding of bug fix patterns. *EmSE*, 14(3):286–315.

Pawlak, R., Monperrus, M., Petitprez, N., Noguera, C., and Seinturier, L. (2015). Spoon: A Library for Implementing Analyses and Transformations of Java Source Code. *Software: Practice and Experience*, 46:1155–1179.

Sobreira, V., Durieux, T., Madeiral, F., Monperrus, M., and Maia, M. A. (2018). Dissection of a Bug Dataset: Anatomy of 395 Patches from Defects4J. In *SANER '18*.

Explorando Como Bibliotecas Python Lançam Exceções ao Longo de Sua Evolução

Allan Gonçalves[1], **Cinthia Nascimento**[1], **Eiji Adachi**[1]

[1]Instituto Metrópole Digital – Universidade Federal do Rio Grande do Norte (UFRN)
Caixa Postal 1524 – 59.078.970 – Natal – RN – Brasil

{allangoncalves, cinthia.katiane}@ufrn.edu.br, eijiadachi@imd.ufrn.br

Resumo. *Para explorar como bibliotecas implementadas em Python estruturam o código responsável por lançar exceções, coletamos 26 bibliotecas-alvo do GitHub e analisamos como seus lançadores de exceção evoluíram ao longo do tempo. Analisamos os tipos de exceção mais comuns usados pelos lançadores e se estes tipos são definidos pelas próprias bibliotecas, pela biblioteca padrão de Python ou por código de terceiros. Os resultados do nosso estudo mostram que há uma preferência por lançar exceções com tipos pré-definidos pela biblioteca padrão de Python, que estes tipos são específicos e que as estratégias adotadas nas versões iniciais das bibliotecas costumam permanecer até as versões finais.*

1. Introdução

O reúso de software é uma meta almejada desde os primórdios da Engenharia de Software [McIlroy et al. 1968]. Por meio do reúso de software, almeja-se reduzir custos e tempo de desenvolvimento de soluções de software. Atualmente, com a consolidação e popularização de bibliotecas de software disponibilizadas em repositórios públicos na Internet, é impensável construir um sistema de software que não faça uso de funcionalidades providas por essas bibliotecas.

Um aspecto importante no projeto de uma biblioteca reutilizável é o seu tratamento de exceções. Tratamento de exceções refere-se à identificação de situações excepcionais que impedem a execução normal de um programa e à implementação de ações que respondem à ocorrência destas situações e que permitem um programa retornar à sua execução normal [Goodenough 1975]. No contexto de uma biblioteca reutilizável, o tratamento de exceções influencia tanto na sua robustez, uma vez que é necessário impedir que exceções causem falhas durante sua execução, e também em sua usabilidade, uma vez que é necessário lançar exceções indicando ao código cliente determinadas condições excepcionais que lhe impedem de prover seus serviços normalmente.

Apesar da importância do tratamento de exceções para o projeto e a implementação de bibliotecas, ainda há pouco conhecimento empírico sobre como bibliotecas implementam o tratamento de exceções. A literatura de tratamento de exceções tem focado principalmente em falhas causadas por defeitos em tratamento de exceções [Ebert et al. 2015, Barbosa et al. 2014], em como tratadores de exceção são implementados [Sena et al. 2016], ou como o código de tratamento de exceções de aplicações evolui ao longo do tempo [Osman et al. 2017]. Não há, no entanto, trabalhos focados em como bibliotecas lançam exceções e como tal estratégia evolui ao longo do tempo.

Este artigo apresenta um estudo exploratório cujo objetivo é caracterizar como o código de tratamento de exceções, em especial o código responsável por lançar as exceções, é estruturado ao longo da evolução de bibliotecas e de código aberto implementadas na linguagem de programação Python. Analisamos bibliotecas implementadas em Python devido a sua popularidade em ambientes de desenvolvimento, principalmente com o advento de bibliotecas para áreas "quentes", como análise de dados, aprendizado de máquina, dentre outros.[1] Além disso, ainda há poucos estudo empíricos realizados sobre código de tratamento de exceção em sistemas implementados nesta linguagem. Desta forma, este estudo preenche uma lacuna na literatura de tratamento de exceções e de bibliotecas de software. Os resultados do nosso estudo mostram que há uma preferência por lançar exceções com tipos pré-definidos pela biblioteca padrão de Python e que as estratégias adotadas nas versões iniciais das bibliotecas costumam permanecer até as versões finais.

2. Tratamento de Exceções em Python

Python possui um mecanismo de tratamento de exceções nativo baseado em cláusulas `try-except`. A estrutura genérica dessa cláusula segue o modelo:

```
1  try:
2          <try_block>
3  except ExceptionType1 as exc1:
4          <exception_block1>
5  except (ExceptionType2, ExceptionType3, ExceptionType4):
6          <exception_block2>
7  except:
8          <exception_block3>
9  else:
10         <else_block>
11 finally:
12         <finally_block>
```

A cláusula `try` delimita um conjunto de instruções que se deseja proteger da ocorrência de exceções. A uma cláusula `try` podem estar associados um ou mais tratadores, que em Python são definidos como blocos `except`. Cada bloco `except` declara como argumento um tipo de exceção que serve como filtro para capturar exceções: qualquer exceção que seja subtipo do argumento do bloco `except` será capturada por esse bloco. Em Python, também é possível declarar um bloco `except` com uma lista de exceções, como mostrado na linha cinco do exemplo anterior. A instrução `raise` é usada para indicar a ocorrência de um erro, lançando uma instância de exceção que for passada em seu argumento. Em alguns casos, quando não se quer manipular a exceção no bloco `except` onde ela foi capturada, a exceção pode ser capturada e relançada para ser tratada posteriormente por outro método.

Exceções em Python são representadas como objetos de classes que herdam de `BaseException`. Python também permite que desenvolvedores definam tipos de exceção específicos para suas aplicações. Para isto, há o tipo `Exception` e é considerada uma boa prática da linguagem usar tal tipo, ou um de seus subtipos, como ponto de extensão para os tipos de exceção definidos pelos usuários. Desta forma, os desenvolvedores podem aumentar o vocabulário de tipos de suas aplicações, representando erros específicos com tipos de exceção mais adequados do que os providos pela linguagem.

[1]Python aparece como a linguagem mais popular conforme o índice https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2017 e a quarta mais popular conforme o índice https://www.tiobe.com/tiobe-index

3. Configuração do Estudo Exploratório

Questões de Pesquisa e Coleta dos Dados. Para caracterizar como o código de tratamento de exceções, em especial o código responsável por lançar as exceções, é estruturado ao longo da evolução de bibliotecas de código aberto implementadas na linguagem de programação Python, nós investigamos as seguintes questões de pesquisa:

QP1: Quais categorias de tipos de exceção são mais usadas em bibliotecas implementadas em Python?
QP2: Quais os tipos de exceção são mais usados em bibliotecas implementadas em Python?

Na primeira questão de pesquisa, investigamos quais categorias de tipos de exceção são mais utilizadas nos lançadores de exceção das biblioteca analisadas. Definimos como *"categoria de tipo de exceção"* a origem onde o tipo de exceção é definida. Consideramos as seguintes origens: (i) *App* – abrange os tipos de exceção que são definidos explicitamente pela própria equipe de desenvolvimento da biblioteca analisada, (ii) *Python* – abrange os tipos de exceção que são definidos pela biblioteca padrão da linguagem Python; e (iii) *Outros* – abrange os tipos de exceção que não se encaixam nas duas categorias anteriores; são tipos de exceção definidos por código reutilizado de terceiros (outras bibliotecas) no contexto das bibliotecas analisadas. Ao investigarmos tal questão de pesquisa, nós pretendemos observar se existe uma preferência em definir tipos de exceção para serem lançados pelas bibliotecas analisadas, ou se estas tendem a lançar exceções com tipos definidos pela própria linguagem, ou por terceiros. Já na segunda questão de pesquisa, nós investigamos quais os tipos que são mais utilizados nos lançadores de exceção das biblioteca analisadas. Investigamos assim se existem tipos de exceção específicos que são comuns a bibliotecas de diferentes propósitos.

Para nos auxiliar a responder as questões de pesquisa propostas, nós coletamos métricas que quantificam diretamente quais os tipos que são utilizados nos lançadores de exceção, quantas ocorrências de cada tipo ocorre e a qual categoria cada tipo pertence. Para isto, implementamos uma ferramenta de análise estática baseada no módulo *Abstract Syntax Tree – ast* que é nativo da linguagem Python. Tal módulo provê uma interface para um analisador sintático de código fonte Python, provendo uma estrutura de árvores de sintaxe abstrata que implementam o padrão de projeto *Visitor* e que permite a extração de informações do código fonte através de análise estática. Desta forma, implementamos um analisador estático que visita as estruturas dos lançadores de exceção – instruções *raise* – e registra o tipo de exceção lançado.

Após registrados os tipos de exceção lançados no contexto das bibliotecas analisadas, passamos para o passo de categorizar os tipos. Para isto, realizamos o seguinte procedimento. Inicialmente, compilamos manualmente todos os tipos de exceção definidos pela biblioteca padrão da linguagem Python.[2] Em seguida, nossa ferramenta analisa o código das bibliotecas-alvo para identificar definições de classes que, por sua vez, são analisadas para identificar aquelas que estendem algum tipo de exceção. Caso exista uma classe estendendo um tipo de exceção, então o tipo definido por esta classe é classificado na categoria *App*. Os outros tipos identificados no código fonte e que não foram classificados nas categorias *Python* ou *App* são classificados na categoria *Outros*.

[2]Esta listagem pode ser encontrada em CTPS://docs.python.org/2/library/exceptions.html

Por fim, para compreender como o código que lança exceções nas evolui ao longo do tempo, coletamos as métricas na primeira e na última versão disponível no repositório de cada biblioteca. Embora esta seja uma análise simplificada da evolução do código dos lançadores de exceção, ela é suficiente para uma caracterização inicial de como as decisões tomadas nas versões iniciais comportam-se ao longo do tempo.

População e Seleção da Amostra. A população alvo do nosso estudo é formada por bibliotecas de código aberto implementadas em Python. Para a realização deste estudo, selecionamos uma amostra dessas bibliotecas que estão disponíveis na plataforma de hospedagem de código fonte GitHub. Atualmente, o GitHub é a maior plataforma de hospedagem de código aberto além de prover uma API de acesso aos projetos de código aberto hospedados na plataforma.

Com base nas recomendações dadas por Kalliamvakou et al. [Kalliamvakou et al. 2014], definimos os seguintes critérios para selecionarmos nossa amostra dentre os projetos disponíveis no GitHub: (i) o projeto deve ser uma biblioteca reutilizável implementada em Python; (ii) o projeto deve ser popular, evitando assim projetos de pouca relevância prática; (iii) o projeto deve ter um histórico de evolução relevante, de modo a observarmos bibliotecas que já atingiram um estágio de maturidade mínimo; e (iv) o projeto deve ser ativo, de modo a observarmos bibliotecas que ainda estão sendo mantidas e evoluídas.

A partir da API do GitHub, nós listamos os 1000 projetos implementados em Python com maior número de *seguidores* (do inglês *watchers*). O número de *seguidores* foi utilizado neste estudo como um indicador da popularidade dos projetos disponíveis no GitHub. Em seguida, aplicamos um filtro para selecionar apenas projetos com um histórico de evolução relevante. Para isto, filtramos apenas os projetos com pelo menos 30 *releases* publicamente disponíveis. De acordo com a documentação oficial do GitHub, toda *release* cadastrada deve estar associada a uma *tag*. Como a funcionalidade de cadastrar *releases* foi implementada apenas em 2013, consideramos *releases* todas as *tags* cadastradas pelos desenvolvedores em seus respectivos repositórios.

Após esta filtragem, restaram 263 projetos. Destes projetos restantes, analisamos manualmente a descrição presente no *read-me* ou na *wiki* de cada um e selecionamos apenas aqueles que explicitamente se denominavam como uma biblioteca (procuramos pelo termo *library*). Dos 263 projetos, restaram 38 projetos. Destes projetos restantes, novamente, analisamos manualmente cada um para descartarmos *forks* de projetos-base. Foram descartados por este critério outros 12 projetos, restando uma amostra contendo 26 projetos de bibliotecas de código aberto implementadas em Python. Esta amostra compreende bibliotecas de diferentes domínios, como bibliotecas para computação simbólica (SymPy), aprendizado de máquina (Scikit-Learn), construção de gráficos (Matplotlib), computação paralela (Dask), dentre outros domínios. A listagem das bibliotecas analisadas é apresentada na Tabela 1.

4. Resultado e Discussão

A Tabela 1 apresenta os dados coletados que nos auxilia a responder nossa primeira questão de pesquisa. Cada linha da tabela apresenta uma das bibliotecas analisadas, agrupando do lado esquerdo os dados coletados para a sua versão inicial (primeira *release* disponível publicamente no GitHub) e do lado direito os dados para sua versão

Tabela 1. Evolução dos Lançadores e suas Categorias de Tipos

Nome	Versão Inicial				Versão Atual			
	App	Python	Outros	Total	App	Python	Outros	Total
Sympy	107	275	40	422	644	2252	15	2911
Scipy	0	212	303	515	42	2210	78	2330
Pandas	5	139	1	145	162	1664	43	1869
Theano	76	797	23	896	148	1554	23	1725
Scikit-Learn	52	77	1	130	35	1072	69	1176
Matplotlib	96	612	43	751	66	928	12	1006
Moto	1	115	0	116	592	209	33	834
Dask	0	3	1	4	15	528	6	549
Scapy	0	0	0	0	128	134	245	507
Cryptography	27	32	0	59	117	384	0	501
Keras	0	16	0	16	0	484	0	484
Bokeh	9	35	0	44	52	317	64	433
Scikit-image	0	15	0	15	3	384	9	396
Gensim	0	34	0	34	2	266	18	286
Docker-py	0	9	2	11	101	52	60	213
Hypothesis	2	4	0	6	135	51	2	188
TensorLayer	0	40	0	40	0	179	0	179
GitPython	4	3	1	8	29	129	19	177
PyEthereum	0	34	1	35	52	107	2	161
Pika	19	5	4	28	37	84	2	123
Psutil	0	1	0	1	53	64	3	120
Requests	16	13	2	31	30	23	0	53
Snips NLU	0	55	0	55	8	39	6	53
spaCy	0	3	0	3	0	49	1	50
Dedupe	0	34	0	34	8	28	0	36
ChatterBot	0	0	0	0	0	0	27	27
Total	414	2526	422	3399	2459	13191	737	16387

final (última *release* disponível publicamente no GitHub). As colunas representam o número de lançadores por categoria de tipo: *App*, *Python* e *Outros*, além da soma total de lançadores em cada versão.

Ao analisar os dados apresentados na Tabela 1, é perceptível que entre as versões inicial e atual de todas as bibliotecas há aumento no número total de lançadores de exceção. Isto é esperado, pois a medida que as bibliotecas evoluem, novas funcionalidades são implementadas e novas condições excepcionais são identificadas, justificando o aumento no número de lançadores.

Analisando a estratégia de lançamento de exceções na versão inicial das bibliotecas, percebem-se os seguintes fatos: 20 das 26 bibliotecas preferem lançar exceções fornecidas pelo Python, 3 das 26 preferem lançar suas próprias exceções, 1 das 26 prefere lançar exceções de terceiros e 2 das 26 bibliotecas não lançam qualquer exceção. Em uma visão geral sobre os lançadores implementados nas primeiras versões das bibliotecas analisadas, tem-se que 12% de todos os lançadores usam exceções definidas pela própria aplicação, 75% contém exceções definidas pela linguagem e 13% contém exceções definidas por aplicações de terceiros. Ou seja, nas primeiras versões das bibliotecas, 88% dos lançadores usam exceções prontas, sejam estas providas pela linguagem de programação ou por código de terceiros.

Analisando a estratégia de lançamento de exceções na versão atual das bibliotecas, observa-se fatos similares aos observados nas versões iniciais. A predileção da origem das exceções está estruturada da seguinte forma: 4 das 26 bibliotecas favorecem suas próprias exceções, 20 das 26 optam por exceções fornecidas pela linguagem, e 2 das

26 usam exceções de outras fontes. Observa-se ainda que na versão atual todas bibliotecas lançam alguma exceção. Ao estudarmos os lançadores, percebemos que 15% contém exceções definidas por suas respectivas bibliotecas, 80% favorecem exceções entregues pelo Python e 5% dos lançadores usam exceções definidas por terceiros. Ou seja, nas versões atuais das bibliotecas, 85% dos lançadores usam exceções prontas que são providas pela linguagem ou por terceiros. Comparando com as versões iniciais, há um pequeno aumento na porcentagem de lançadores que usam exceções definidas pela própria aplicação (aumento de 12% para 15%).

Ao analisarmos cada linha da tabela, podemos analisar se as bibliotecas analisadas alteram suas estratégias de lançamento de exceções no período de evolução analisado. Nesta análise, percebe-se que 6 bibliotecas alteraram sua estratégia de lançamento no que se refere à preferência de categoria de tipos das exceções lançadas. Os projetos Moto, Docker-Py e Hypothesis inicialmente privilegiavam o uso de exceções definidas pela linguagem, mas passaram a priorizar suas próprias exceções. Em um comportamento inverso, durante a versão inicial das bibliotecas GitPython e Pika observava-se um uso maior de suas próprias exceções, mas nas versões atuais passaram a usar mais exceções cedidas por Python. Cabe salientar que nestas duas observações houve aumento no número de exceções em todas categorias de tipos usadas nessas bibliotecas; a mudança ocorreu apenas na preferência (i.e., na categoria que representa a maior porcentagem do total). Por fim, a biblioteca Scipy, que inicialmente utilizava em maior número exceções de terceiros, acabou alterando para exceções da biblioteca padrão de Python. Neste caso específico, houve uma redução no número de lançadores com exceções de terceiros acompanhado do aumento de lançadores nas outras categorias. Houve, portanto, a mudança do tipo da exceção usada em determinados lançadores.

Através destas observações, nota-se que, em grande parte, as bibliotecas escritas em Python priorizam exceções definidas pela própria linguagem e que ao longo da evolução tendem a não alterar a preferência adotada na versão inicial, visto que das 20 bibliotecas que iniciaram seus desenvolvimentos utilizando em maior número exceções definidas pelo Python, apenas três realizaram alterações (Moto, Docker-Py e Hypothesis).

As Tabelas 2 e 3 apresentam os dados que nos auxiliam a responder a segunda questão de pesquisa, apresentando os tipos de exceções mais recorrentes na fase inicial e atual das bibliotecas, respectivamente. As tabelas apresentam o número total de lançadores em todas as bibliotecas analisadas (coluna *#Lan.*), bem como o número de bibliotecas em que um determinado tipo foi usado (coluna *#Bib.*) e o nível que um determinado tipo de exceção está na árvore de tipos de exceções (coluna *Nível*). Neste caso, um valor maior do nível define um tipo mais específico e um valor menor define um tipo mais genérico. Quanto mais específico um tipo de exceção é, mais específica é a preocupação em detalhar qual tipo de erro foi encontrado.

Todos os tipos de exceção mais comumente usados nas versões iniciais das bibliotecas, aqueles apresentados na Tabela 2, são definidos pela biblioteca padrão de Python. Nós analisamos o quão específicos são esses tipos, isto é, o quão distante esses tipos estão do tipo de exceção base de Python, o tipo `BaseException`. Dentre os tipos mais comuns, o tipo `Exception`, que é o quarto mais comum nas versões iniciais, é o único que herda diretamente de `BaseException`. Durante a fase inicial das bibliotecas, as exceções mais recorrentes foram encontradas em 2514 lançadores, e, dentre esses, cerca

Tabela 2. Exceções Iniciais

Nome	#Lan.	#Bib.	Nível
ValueError	919	19	4
NotImplementedError	563	14	5
TypeError	509	11	4
Exception	254	12	2
RuntimeError	134	11	4
KeyError	47	9	5
ImportError	30	10	4
IndexError	26	6	5
IOError	13	6	5
AttributeError	19	4	4

Tabela 3. Exceções Atuais

Nome	#Lan.	#Bib.	Nível
ValueError	7567	25	4
TypeError	2108	23	4
NotImplementedError	1582	22	5
RuntimeError	477	20	4
Exception	409	17	2
AttributeError	173	19	4
ImportError	173	17	4
IndexError	161	17	5
KeyError	136	19	5
AssertionError	188	12	4

de 10% (254 lançadores) utilizam o tipo `Exception` que é um tipo bastante genérico. Já os tipos `ValueError`, `TypeError`, `ImportError`, `AttributeError` e `RuntimeError` estão três níveis abaixo do tipo `BaseException` enquanto os tipos `KeyError`, `NotImplementedError`, `IOError` e `IndexError` estão quatro níveis abaixo do tipo `BaseException`. Além disso, aproximadamente 90% das ocorrências estão divididas em quatro diferentes tipos: `ValueError`, `NotImplementedError`, `TypeError` e `Exception`. Entretanto, apenas dois desses tipos (`ValueError` e `NotImplementedError`) encontram-se em mais da metade das bibliotecas analisadas.

Fato similar foi observado na versão atual das bibliotecas: apenas um tipo comum nas versões iniciais não apareceu nos mais comuns das versões atuais. A diferença entre os tipos mais recorrentes durante as diferentes versões das bibliotecas foi a troca de `IOError` para `AssertionError`. Outrossim, observa-se que o número de lançadores que se relacionavam com as exceções contidas nas tabelas cresceu em aproximadamente 416% durante as duas fases. Posto isso, pode-se dizer que os tipos preferidos no início estenderam-se até a fase atual das bibliotecas, quando apenas `AssertionError` não se propagou em mais da metade das bibliotecas. Embora tenhamos observado algumas mudanças nas estratégias em termos de categorias de tipos, como discutido anteriormente, em relação aos tipos mais comuns não observamos mudanças nas estratégias de lançamento. Por fim, o fato de 97% das ocorrências conterem exceções que pertencem a níveis mais fundos na árvore de tipos (81% em três níveis abaixo e 16% quatro níveis abaixo do tipo `BaseException`) mostra que as bibliotecas preocuparam-se minimamente em lançar exceções com tipos mais específicos, provavelmente buscando tipos que indiquem mais claramente aos seus usuários quais erros ocorreram.

5. Trabalhos Relacionados

Os estudos de Ebert et al. [Ebert et al. 2015] e de Barbosa et al. [Barbosa et al. 2014] analisaram sistemas de software *open-source* e categorizaram falhas ocasionadas por defeitos no código de tratamento de exceções. Algumas destas categorias estão relacionadas ao lançamento de exceções, em particular ao lançamento de exceções com tipos muito genéricos. No nosso estudo, observamos que algumas das bibliotecas analisadas adotam a prática de lançar exceções muito genéricas desde as versões iniciais, mantendo esta prática até suas versões atuais. Similarmente ao nosso estudo, Sena et al. [Sena et al. 2016] realizaram um estudo empírico do tratamento de exceção em bibliotecas. Todavia, estes focaram em analisar os fluxos excepcionais, categorizar as estratégias de tratamento de exceção e antipadrões de tratamento de exceções existentes bibliotecas Java, enquanto o presente estudo analisa as estratégias adotadas no lançamento de exceções em bibliotecas com código fonte Python. Já Osman et al. [Osman et al. 2017] apresentaram uma comparação entre a evolução do tratamento de exceções em bibliotecas e aplicações. Os autores também categorizaram as exceções e efetuaram uma análise sobre os lançadores em bibliotecas Java. Assim como em nosso estudo, estes autores também observaram que bibliotecas Java parecem dar preferência ao uso de tipos de exceção definidos pela própria linguagem ao invés de tipos próprios definidos pela biblioteca.

6. Conclusão

Nosso estudo observou que há uma preferência por tipos de exceção definidos pela biblioteca padrão de Python e que os tipos mais comumente usados estão em níveis mais profundos da árvore de tipos de exceção. Isto pode sugerir que o projeto da árvore de tipos da linguagem Python apresenta um bom vocabulário de erros para as necessidades das bibliotecas. Por outro lado, observamos que algumas bibliotecas ainda lançam exceções com tipos muito genéricos e pouco informativos sobre o erro encontrado, prática já reportada na literatura como sendo causa comum de falhas em sistemas de software. Isto pode indicar que o tratamento de exceções, especialmente o lançamento de exceções, ainda não é apropriado no projeto de algumas bibliotecas. Por fim, observamos que as preferências quanto ao lançamento de exceções adotadas na versão inicial não costumam mudar ao longo da evolução do projeto. Isto pode indicar que as decisões tomadas nas versões iniciais são difíceis de serem modificadas ao longo da evolução e, portanto, o tratamento de exceção deve ser bem projetado e implementado desde as versões iniciais das bibliotecas.

Referências

Barbosa, E. A., Garcia, A., and Barbosa, S. D. J. (2014). Categorizing Faults in Exception Handling: A Study of Open Source Projects. In *Proceedings of the XXVIII Brazilian Symposium on Software Engineering (SBES'14).*

Ebert, F., Castor, F., and Serebrenik, A. (2015). An exploratory study on exception handling bugs in java programs. *Journal of Systems and Software*, 106:82–101.

Goodenough, J. B. (1975). Exception handling: issues and a proposed notation. *Communications of the ACM*, 18(12):683.

Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M., and Damian, D. (2014). The promises and perils of mining github. In *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR'14)*, pages 92–101.

McIlroy, M. D., Buxton, J., Naur, P., and Randell, B. (1968). Mass-produced software components. In *Proceedings of the 1st International Conference on Software Engineering (ICSE'68)*, pages 88–98.

Osman, H., Chiş, A., Corrodi, C., Ghafari, M., and Nierstrasz, O. (2017). Exception evolution in long-lived java systems. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR'17)*, pages 302–311.

Sena, D., Coelho, R., Kulesza, U., and Bonifácio, R. (2016). Understanding the exception handling strategies of java libraries: An empirical study. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR'16)*, pages 212–222.

Identifying Confusing Code in Swift Programs

Fernando Castor[1]

[1]Informatics Center – Federal University of Pernambuco (UFPE)
Recife – PE – Brazil

castor@cin.ufpe.br

Abstract. *Atoms of confusion are small code patterns that are likely to cause confusion in developers and can be replaced by functionally equivalent alternatives that are less likely to confuse. The only existing catalog of atoms of confusion targets the C language. Our long term goal is to devise a similar catalog for the Swift programming language. However, this is not straightforward because it requires specialist knowledge about the programming language under study. Also, there is currently no direct definition of atom of confusion that makes it possible to unequivocally differentiate atoms from related concepts and there are no principles or guidelines for identifying new atoms. This paper presents preliminary work towards our long term goal and makes three contributions: (i) provides a direct, structured definition of atoms of confusion; (ii) examines factors that cause existing atoms to be confusing; and (iii) presents a preliminary catalog of atoms of confusion for Swift.*

1. Introduction

Understanding code is an important activity in the development of software systems. Code that is difficult to understand can negatively impact various development-related activities, such as testing, code review [2, 6], and editing and navigation [16]. For example, it is one of the factors that negatively affects the performance of reviewers during code reviews, causing confusion [6] and slowing down the process [2]. One of the obstacles faced by developers performing the activity of understanding the code of software systems is their complexity. Complexity can create a conflict between what a developer trying to understand the functioning of a system understands and the actual behavior of that system, i.e., how a computer would understand it. Fred Brooks [3] stated that there are two types of complexity in software systems: essential and accidental. In the first case, the problem that the program solves is inherently difficult and its solution is therefore complex. In the second one, the complexity stems from decisions made by developers and is not inherent. Accidental complexity can manifest itself in a variety of ways: code smells [7], anti-patterns [4], identifiers that either are too short or do not clearly express the intent of developers [1, 11], among others. Recent work has identified another form of accidental complexity, the **atoms of confusion** [8]. According to Gopstein et al. [8], atoms of confusion are

> *"[...] the smallest pieces of code that can routinely cause programmers to misunderstand code. [...] These atoms can serve as an empirical and quantitative foundation for understanding what makes code confusing."*

Two experiments, one with 73 participants and the other with 43, found that small C programs including atoms of confusion are more difficult to understand than functionally equivalent programs that do not include these atoms [8]. In addition, a recent study [9] involving 14 large-scale open-source projects written in the C language has revealed that there are plenty of atoms in real and successful projects (such as Git and the Linux kernel). The presence of these atoms in a project has a strong correlation with bug fixing commits and long code comments.

In this position paper, we present preliminary work on the study of atoms of confusion in the Swift programming language.[1] According to Apple[2], Swift is a *"powerful language that is also easy to learn"*, *"safe by design"*, and *"intuitive"*. In fact, Swift avoids by construction some of the atoms of confusion presented by Gopstein et al. [8], such as omitted curly braces and assignment expressions. This motivates us to ask the question: **What are the atoms of confusion of the Swift programming language?** Identifying new atoms for a programming language is non-trivial, however. Firstly, because it requires specialist knowledge about the programming language under study. Secondly, because there is currently no direct definition of atom of confusion that makes it possible to unequivocally differentiate atoms from related concepts, such as code smells. Thirdly, because there are no principles or guidelines for identifying new atoms. In this paper, we address the latter two issues and present a preliminary list atom candidates for Swift. Furthermore, we discuss a number of future avenues for research in this area.

2. Atoms of Confusion

This section starts by providing a structured definition of atom of confusion (Section 2.1). It then proceeds by showing that this definition makes it possible to clearly distinguish between atoms of confusion and two related concepts, code smells and antipatterns (Section 2.2).

2.1. A definition

In their original work, Gopstein et al. [8] present a set of 15 atoms they extracted from winning programs of the International Obfuscated C Code Contest[3]. As the authors themselves acknowledge, they target *"[...]specific situations where a programmer might tend to misunderstand the behavior of a piece of code."*. Furthermore, they state that *" We restricted our definition of an atom to only minimal portions of code so that our findings would be generalizable and occur frequently in real projects."*. Table 1 presents some examples of atoms they have identified, together with functionally equivalent alternative code patterns that do not include them. A complete list is available elsewhere [8, 9].

The creators of the concept of atom of confusion do a good job of providing an intuition about what it means for a code pattern to be an atom. Nonetheless, the concept of atom is not defined in a direct manner, clearly specifying the necessary conditions for a code pattern to be considered an atom. To assist researchers interested in the study of atoms of confusion, we believe it is useful to have a direct, structured definition. Based on the information available in the aforementioned two papers, we define an atom of confusion as a code pattern that is

- precisely identifiable,
- likely to cause confusion,

[1]https://docs.swift.org/swift-book/GuidedTour/GuidedTour.html
[2]https://developer.apple.com/swift/
[3]https://www.ioccc.org

Atom Name	Atom Example	Transformed
Literal Encoding	`printf("%d", 013)`	`printf("%d", 11)`
Assignment as Value	`V1 = V2 = 3;`	`V2=3;` `V1 = V2;`
Post- and Pre-Increment	`V1 = V2++;`	`V1 = V2;` `V2 += 1;`
Conditional Operator	`V2 = (V1==3)?2:V2`	`if (V1==3) { V2 = 2; }`
Omitted Curly Brace	`if (V) F(); G();`	`if (V) { F(); } G();`
Repurposed Variable	`argc = 7;`	`int V1=7;`
Implicit Predicate	`if(4%2)`	`if(4%2 != 0)`
Logic as Control Flow	`V1 && F2();`	`if (V1) F2();`
Comma Operator	`V3 = (V1 += 1, V1);`	`V1 += 1; V3 = V1;`

Table 1. Some atoms of confusion for the C language and functionally equivalent, less confusing transformed versions. Adapted from [9]

.

- replaceable by a functionally equivalent code pattern that is less likely to cause confusion, and
- indivisible.

In the definition above, "code pattern" may be a specific language construct, e.g., the Conditional Operator or Assignment as Value, or a usage pattern, e.g., Repurposed Variable or Implicit Predicate. By "precisely identifiable", we mean that the atom can be identified independently of subjective opinions. For example, it may be the case that the `F2()` function call is in fact part of the condition in the atom example of Logic as Control Flow in Table 1. In this scenario, the transformed code pattern would not be appropriate. Nonetheless, either the call is part of the condition or it is not. This is different, for example, from a long method code smell [7], where the definition of what constitutes "long" is dependent on personal opinion [13]. Furthermore, "precisely identifiable" is not the same as "mechanically identifiable". For example, because of cases such as the aforementioned example, it is not possible to build a program that detects instances of Logic as Control Flow with precision.

The "likely to cause confusion" part of the definition is implied by a basic premise: that an empirical evaluation has been conducted showing that the atom candidate is in fact confusing. The paper introducing the concept of atoms [8] describes an experiment where the performance of developers in understanding small programs including atom candidates was measured in terms of how often they correctly identified the output of these programs and how long it took them to do so. Moreover, this part of the definition is strongly connected to the third one ("can be replaced..."). Gopstein et al. [8] state that *"We define atoms relative to functionally equivalent code that does not confuse programmers."* For example, one of the atom candidates of the aforementioned paper is Pointer Arithmetic. Although this candidate is verifiably confusing, it is not considered an atom of confusion because the developers participating in the experiment were as confused by the evaluated alternative code patterns as they were by the atom candidate. The existence of a less confusing, functionally equivalent alternative code pattern where the atom is not present is a necessary condition for a code pattern to be considered an atom.

Finally, an atom is "indivisible" in the sense that it cannot be simplified and remain an atom nor be broken down into smaller atoms. This part of the definition emphasizes that an atom represents the simplest possible example of a potential cause for confusion.

2.2. Atoms, code smells, antipatterns

Atoms of confusion are similar to the well-known code smells [7] which have been studied extensively (For example, [12, 14, 17]). According to Fowler [7], code smells are symptoms of deeper problems that may occur in software systems. Atoms differ from code smells at the conceptual and practical level. At the conceptual level, atoms of confusion are hints of readability problems. On the other hand, code smells may indicate problems related to a wide range of quality attributes, e.g., maintainability [7], performance [10], and energy-efficiency [18]. Furthermore, atoms of confusion exist at a fine level of granularity, in the context of a definition, statement, or expression. Code smells can involve several classes and high level design issues. At the practical level, atoms of confusion can be precisely identified. Code smells, on the other hand, often depend on subjective judgement, which may lead to conflicting understandings about their presence or absence [13]. Furthermore, the requirements of empirical validation and indivisibility are not part of the definition of code smell. The existence of a less confusing, functionally equivalent alternative solution, however, is implied because code smells were introduced as a motivation for refactoring [7].

Atoms and smells are sometimes directly related. For example, one of the code smells from Fowler's book [7] is Switch Statements. Since `switch` statements are trivially identifiable, this smell could be easily explained as an atom of confusion, as long as it can be empirically shown that (i) it hinders readability and (ii) there is a functionally equivalent alternative that is less confusing. As an additional example, the presence of Comments is another smell from the book. As pointed out by Gopstein et al. [9], code that includes atoms usually exhibits longer comments than code that does not.

Antipatterns [4] are bad solutions to recurring design problems. They have a negative impact on the quality attributes of a system. According to Brown et al. [4], antipatterns manifest mainly at the development, architecture, and process levels. This definition emphasizes the broad scope of antipatterns when compared to atoms of confusion. For example, the is no correspondence between process level antipatterns and atoms. As shown by Moha and colleagues [14], development and architecture level antipatterns and code smells have a direct relationship; code smells are concretizations of these antipatterns. Since the latter are very abstract, even when referring directly to code elements, none of the elements of the definition presented in Section 2.1 apply to antipatterns in general. Antipatterns sometimes negatively impact readability, e.g., Spaghetti Code may include long methods and pervasive use of global variables. Notwithstanding, they may impact other quality attributes of software development processes and products.

3. Sources of atoms of confusion

To devise an atom catalog for C programs, Gopstein et al. [8] relied on purposely complex code examples that appeared in the International Obfuscated C Code Contest. To win this competition, developers need to build programs that are difficult to understand, using potentially esoteric C-language features. As the name implies, IOCCC is unique to C programs and there are no similar competitions for other languages.[4] Thus, this methodology cannot be used to devise new atom catalogs targeting different programming languages.

[4]There used to be an analog competition for Perl programs, but the last edition occurred in 2000.

The development of a catalog of atoms of confusion requires expert knowledge about the language and also about the typical **sources of confusion**, the underlying reasons that make a code pattern difficult to understand. Although it is not possible to leverage IOCCC programs to devise atoms for other languages in a direct manner, we hypothesize that the sources of confusion for the identified set of atoms can be leveraged to devise new atom catalogs. Taking that hypothesis as a starting point, we performed reverse engineering of the atom catalog of Gopstein and colleagues [8]. We classified the atoms based on the sources of confusion that each one exhibits, as discussed by the authors themselves (Table 1 of [8]). The resulting classification can then serve as a basis for the identification of atoms in other languages. Since this process relies on subjective judgement, we employed open card sorting [19], assigning one atom description to each card. We then proceeded to grouping the cards, sometimes combining simple groups, sometimes splitting complex ones. Since our goal in eliciting sources of confusion is to provide guidance to designers of new atom catalogs and not define a rigid categorization, we admitted the possibility of a card belonging to more than one group. The card sorting process yielded 5 sources of confusion, presented in Table 2. We use these sources of confusion as a starting point to devise a catalog of atom candidates for Swift (Section 4).

4. Atom of Confusion Candidates for Swift

We devised a preliminary set of atom candidates for Swift. To identify the atom candidates, we conducted the following steps, guided by the atom catalog of Gopstein and colleagues [8] and the sources of confusion of Table 2:

Existing catalog. We started out by analyzing each atom in the original catalog and excluded the ones that cannot happen in Swift due to its differences from C. This step excluded 9 of the original 15 atoms. For example, Assignment as Value, Omitted Curly Braces, and Pre- and Post-Increments do not exist in Swift. The remaining 6 atoms are all atom candidates for Swift.

Uncommon Constructs. We then proceeded by adapting the Swift-AST[5] Swift parser to collect information about the frequency of use of the syntactic constructs from 5 large Swift projects that were analyzed in a recent study [5]. For each project, we selected the 5 least popular ones and checked the feasibility of implementing functionally equivalent code patterns that do not include these constructs. This process produced one additional atom candidate, Defer Statement, and showed that usage of Conditional Expression is uncommon in these Swift projects.

Usage beyond Lexical Meaning and Syntactically Similar, Semantically Different Constructs. Based on expert knowledge[6] and language documentation, we identified cases where there are different ways of using a construct and also cases where the same symbol or identifier is employed with different semantics, depending on the context of use. There are many instances of these cases in Swift. This step yielded 7 atom candidates.

Obscuring programming practices. This is the most pragmatics-dependent source of

[5]https://github.com/yanagiba/swift-ast

[6]The authors have been working with, teaching, and researching on Swift programming since January 2015, 7 months after the first release of the language.

Source of Confusion	Description	Affected Atoms
Uncommon constructs	The use of little-known language constructs. This is a source of confusion for many of the atoms of confusion.	Literal Encoding Comma Operator Conditional Operator
Construct usage beyond lexical meaning	Some constructs are typically used in a certain way. However, additional, less common usage patterns are also possible and tend to obscure program meaning.	Logic as Control Flow Assignment as Value
Syntactically similar, semantically different constructs	The semantics of a symbol or identifier changes depending on the context of use.	Post- and Pre-Increment
Complex rules	Ambiguity and lack of clarity because language rules are non-obvious and not explicit.	Operator precedence
Obscuring programming practices	Usage patterns of common programming language constructs that obscure the meaning of the program.	Repurposed Variables Omitted Curly Braces Implicit Predicate

Table 2. Sources of confusion. Each one represents a potential reason for a code pattern to be confusing.

confusion. In this work, we have identified one example of programming practice that is commonplace but obscures readability, based on previous work on visualizing Swift code [15].

We also identified one additional atom candidate that is a rarely-used Swift-specific derivation of Literal Encoding. Furthermore, we could not find more atom candidates besides Operator Precedence when considering the Complex Rules source of confusion. Table 3 presents a subset of the atom candidates we identified, together with the functionally equivalent alternatives. We omit candidates that have a direct correspondence to previously identified atoms [8], except for Literal Encoding because Swift has additional options and slightly different conventions for expressing number encoding. In Table 3, Hexadecimals with Exponents is a special case of number encoding where a floating point hexadecimal is written as a product of an hexadecimal integer and a power of 2. The `defer` statement is rarely used in the Swift projects we analyzed. The code block associated with a `defer` statement is executed right before jumping out of the enclosing block. If multiple `defer` statements are executed within the same scope, they create a nested structure where the code block of the last `defer` is the first one to be executed.

Swift supports higher-order functions and two atom candidates are related to that language feature. Shorthand Arguments refers to the use of implicit parameter names when employing closures. In the example, `$0` is the name of an implicitly defined parameter of the closure `{$0 > 11}(V1)`, which takes a number as argument and checks whether it is greater than 11. In the transformed version. the parameter `V` is explicitly named. Trailing Closures are special syntax for cases where the last parameter of a function is a closure. In the example, the use of the `reduce` function will sum all the elements of array `V1`, since it takes an initial value and a two parameter function and combines that initial value with all the elements of the array using the given function.

Swift programs make frequent use of optional types. A variable of an optional type is one that potentially holds `nil`. For example, a variable of type optional integer (`Int?`) either stores a integer number (`Int`) or stores `nil`, but a variable of type `Int` is guaranteed to store an integer. If a variable of an optional type actually stores an integer, obtaining this integer requires a process known as unwrapping. Swift makes a number of unwrapping approaches available for developers. One of the atom candidates

<table>
<tr><th>Atom Candidate Name</th><th>Example</th><th>Transformed</th></tr>
<tr><td>Literal Encoding</td><td><code>V1 = 0x12 | 0x6</code></td><td><code>V1 = 0b10010 | 0b00110</code></td></tr>
<tr><td>Hexadecimals with Exponents</td><td><code>V1 = 0x50p-2</code></td><td><code>V1 = 0x50 / 4</code></td></tr>
<tr><td>Defer Statement</td><td><code>defer { V1 += 1 }</code>
<code>V1 = 3</code></td><td><code>V1 = 3</code>
<code>V1 += 1</code></td></tr>
<tr><td>Shorthand Arguments</td><td><code>V1 = 9</code>
<code>{$0 > 11}(V1)</code></td><td><code>V1 = 9</code>
<code>{V in return V > 11}(V1)</code></td></tr>
<tr><td>Trailing Closures</td><td><code>V1=[5,7,8,10,3]</code>
<code>V3=V1.reduce(0) {V2,V4 in V2+V4}</code></td><td><code>V1=[5,7,8,10,3]</code>
<code>V3=V1.reduce(0, {V2,V4 in V2+V4})</code></td></tr>
<tr><td>Overloaded “!”</td><td><code>V1 = readLine()</code>
<code>if V1! != "OK" {</code>
<code> print("Not ok")</code>
<code>}</code></td><td><code>V1 = readLine()!</code>
<code>if V1 != "OK" {</code>
<code> print("Not ok")</code>
<code>}</code></td></tr>
</table>

Table 3. Swift atom candidates.

of Table 3 refers to unwrapping operators. In the example of Overloaded “!”, function `readLine()` returns a value of type `String?`. The following line unwraps the values stored in `V1` using the forced unwrapping operator (`!`) and then checks whether the wrapped string is different (`!=`) from `"OK"`. The problem is that the two semantically different, textually close uses of the “!” symbol in this example may be surprising for developers and cause them to assume that the program has a bug.

5. Concluding Remarks

In this paper, we proposed a definition for atoms of confusion, identified some sources of confusion that exist in the original atoms catalog, and presented a preliminary atom catalog for Swift. Atoms of confusion are an exciting topic of research for which many avenues remain open. The first and most obvious one is to identify new atom candidates, including atoms for other languages, and empirically evaluate their proneness to cause confusion. In fact, this is the next step we intend to take with the preliminary set of candidates presented here. Furthermore, deriving a theory to explain why certain code patterns are harder to read than others is a worthy goal. In this paper, we have given a first step in that direction by introducing the notion of sources of confusion.

We also think that attempting to understand how atoms impact real world development activities is important. Recent work [9] has shown that atoms are often related to bugs. We believe that attempting to understand how atoms impact code review and testing activities may help developers become more productive and the code produced by them more readable. In addition, although atoms of confusion are simple code patterns whose impact should be measurable in isolation, studying the ways in which these atoms are typically combined can pave the way for the development of new analysis and reengineering solutions.

Acknowledgements. We’d like to thank Felipe Ebert and Dan Gopstein, who read preliminary versions of this paper. This research was partially funded by INES 2.0, (FACEPE PRONEX APQ 0388-1.03/14 and APQ-0399-1.03/17, and CNPq 465614/2014-0), FACEPE APQ-0839-1.03/14, and CNPq 304220/2017-5.

References

[1] Eran Avidan and Dror G. Feitelson. Effects of variable names on comprehension: an empirical study. In *Proceedings of the 25th ICPC*, pages 55–65, Buenos Aires, Argentina, May 2017.

[2] Amiangshu Bosu, Jeffrey C. Carver, Christian Bird, Jonathan D. Orbeck, and Christopher Chockley. Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at microsoft. *IEEE Trans. Software Eng.*, 43(1):56–75, 2017.

[3] Frederick P. Brooks Jr. No silver bullet - essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, 1987.

[4] William J. Brown, Raphael C. Malveau, Hays W. McCormick, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley, 1st edition, 1998.

[5] Nathan Cassee, Gustavo Pinto, Fernando Castor, and Alexander Serebrenik. How swift developers handle errors. In *Proceedings of the 15th MSR*, Gothenburg, Sweden, May 2018.

[6] Felipe Ebert, Fernando Castor, Nicole Novielli, and Alexander Serebrenik. Confusion detection in code reviews. In *Proceedings of the 33rd ICSME*, pages 549–553, Shanghai, China, September 2017.

[7] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[8] Dan Gopstein, Jake Iannacone, Yu Yan, Lois DeLong, Yanyan Zhuang, Martin K.-C. Yeh, and Justin Cappos. Understanding misunderstandings in source code. In *Proceedings of the 11th ESEC/FSE*, pages 129–139, Paderborn, Germany, September 2017.

[9] Dan Gopstein, Hongwei Henry Zhou, Phyllis Frankl, and Justin Cappos. Prevalence of confusing code in software projects. In *Proceedings of the 15th MSR*, Gothenburg, Sweden, May 2018.

[10] G. Hecht, N. Moha, and R. Rouvoy. An empirical study of the performance impacts of android code smells. In *Proceedings of the 3rd MOBILESoft*, pages 59–69, Austin, USA, May 2016.

[11] Dawn J. Lawrie, Christopher Morrell, Henry Feild, and David W. Binkley. What's in a name? A study of identifiers. In *Proceedings of the 14th ICPC*, pages 3–12, Athens, Greece, June 2006.

[12] Isela Macia, Joshua Garcia, Daniel Popescu, Alessandro Garcia, Nenad Medvidovic, and Arndt von Staa. Are automatically-detected code anomalies relevant to architectural modularity?: An exploratory analysis of evolving systems. In *Proceedings of the 11th AOSD*, pages 167–178, Potsdam, Germany, 2012.

[13] Mika Mäntylä and Casper Lassenius. Subjective evaluation of software evolvability using code smells: An empirical study. *Empirical Software Engineering*, 11(3):395–431, 2006.

[14] Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Françoise Le Meur. DECOR: A method for the specification and detection of code and design smells. *IEEE Trans. Software Eng.*, 36(1):20–36, 2010.

[15] Rafael Nunes, Marcel Rebouças, Francisco Soares-Neto, and Fernando Castor. Visualizing swift projects as cities. In *Companion to the Proceedings of the 39th ICSE*, pages 368–370, Buenos Aires, Argentina, May 2017.

[16] Akond Rahman. Comprehension effort and programming activities: Related? or not related? In *Proceedings of the 15th MSR*, Gothenburg, Sweden, May 2018.

[17] Jan Schumacher, Nico Zazworka, Forrest Shull, Carolyn Seaman, and Michele Shaw. Building empirical support for automated code smell detection. In *Proceedings of ESEM 2010*, pages 8:1–8:10, Bolzano-Bozen, Italy, 2010.

[18] Roberto Verdecchia, René Aparicio Saez, Giuseppe Procaccianti, and Patricia Lago. Empirical evaluation of the energy impact of refactoring code smells. In *5th ICT4S*, pages 365–383, Toronto, Canada, May 2018.

[19] T. Zimmermann. Card-sorting: From text to themes. In Tim Menzies, Laurie Williams, and Thomas Zimmermann, editors, *Perspectives on Data Science for Software Engineering*, pages 137–141. Morgan Kaufmann, 2016.

DiffMutAnalyze: Uma abordagem para auxiliar a identificação de mutantes equivalentes

Juliana Botelho[1], Carlos Henrique Pereira[1], Vinicius H. S. Durelli[2], Rafael S. Durelli [1]

[1]Universidade Federal de Lavras (UFLA)
Lavras – MG – Brasil

[2]Universidade Federal de São João del-Rei (UFSJ)
São João del-Rei – MG – Brasil

juliana.botelho@posgrad.ufla.br, carloshpereira27@gmail.com
durelli@ufsj.edu.br, rafael.durelli@dcc.ufla.br

Abstract. *The mutation test is considered an effective technique for fault localization, but it has the disadvantage of the high cost of its application. Thus, it determines one of the obstacles to its adoption, since it is necessary to manually identify which mutants behave (i.e., produce the same output) as the original code for all possible inputs. These codes are considered equivalent and need to be analyzed manually to confirm this equivalence. In order to assist test analysts in the identification of these mutants, this paper present an approach that supports the generation of mutants, inspects the codes and informs the analysis data. Thus, it is possible for the analyst to perform the mutation test and to analyze the mutants that remained alive in a single tool.*

Resumo. *O teste de mutação é considerado uma técnica eficaz para localização de falhas, porém tem como desvantagem o alto custo de sua aplicação. Com isso, determina um dos obstáculos à sua adoção, pois é necessário identificar manualmente quais mutantes se comportam (i.e., produzem a mesma saída) como o código original para todas as entradas possíveis. Esses códigos são considerados equivalentes e precisam ser analisados manualmente para confirmação dessa equivalência. A fim de auxiliar os analistas de teste na identificação desses mutantes, o artigo propõe uma abordagem que apoia a geração dos mutantes, inspeção dos códigos e informa os dados da análise. Assim, é possível o analista de teste realizar o teste de mutação e analisar os mutantes que permaneceram vivos em uma única ferramenta.*

1. Introdução

No âmbito da engenharia de software, um dos propósitos é garantir a qualidade dos sistemas de software desenvolvidos. Durante o processo de implementação, o sistema de software deve ser avaliado garantindo que o código implementado esteja em conformidade com o que foi especificado. Uma maneira de avaliar o código é utilizando testes de unidade, porém esses testes também precisam atingir um bom nível de qualidade. Assim, para garantir a qualidade dos casos de teste, o critério de teste denominado teste de mutação tem a finalidade de adequar os casos de testes para alcançar uma boa cobertura do sistema de software [Delamaro et al. 2016].

Há uma medida para avaliar a qualidade da atividade de teste de mutação, denominada *"escore de mutação"* (*mutation score*). O escore de mutação é dado pela razão do número de mutantes mortos pelo total de mutantes gerados subtraído pelo número de mutantes equivalentes [DeMillo 1980]. O escore de mutação varia entre 0 e 1 (100%). Como o cálculo do escore de mutação depende da quantidade de mutantes equivalentes, é importante realizar a análise dos mesmos. Uma vez que os mutantes equivalentes atuam como falsos positivos, pois nenhum teste pode detectá-los [Grün et al. 2009]. Isso significa que a eliminação desses mutantes não afeta na geração dos casos de teste, mas no cálculo do escore de mutação. Assim, a precisão da métrica se torna questionável.

A análise de mutantes equivalentes é uma etapa fundamental para a condução do teste de mutação, pois somente por meio da identificação desses mutantes é que os testadores podem calcular devidamente o escore de mutação. Em outras palavras, tal passo é essencial para possibilitar que os testadores avaliem a qualidade do conjunto de casos de teste existente. Como a distribuição de mutantes tende a ser imprevisível e a identificação da equivalência dos mutantes é um problema indecidível [Budd and Angluin 1982], é difícil julgar a qualidade do conjunto de testes com base no escore de mutação sem que os mutantes equivalentes sejam identificados.

Segundo Grün et al., leva-se aproximadamente 15 minutos para verificar manualmente se um mutante é equivalente ao sistema de software original [Grün et al. 2009]. Considerando esse tempo e acrescentando a quantidade de mutantes gerados pelos operadores de mutação, que geralmente são gerados em grande quantidade, é necessário averiguar maneiras de minimizar o custo dessa análise, uma vez que são gastas horas de um analista de teste para realizar a inspeção dos códigos, o que contribui para o custo humano da análise.

Dentro desse contexto e para auxiliar a inspeção manual da equivalência entre os códigos (i.e., original e mutantes), a abordagem proposta visa auxiliar os analistas de teste na identificação desses mutantes, objetivando minimizar o tempo gasto com a análise desses mutantes. Assim, a ferramenta DiffMutAnalyze foi desenvolvida com o intuito de gerar os mutantes no projeto sendo testado e exibir as duas versões do código (i.e., original e mutante) lado a lado, para verificação guiada e manual da equivalência. Essa visualização do código facilita sua inspeção, permitindo ao analista de teste comparar diretamente o código original e seu mutante, além de evidenciar a alteração realizada.

Durante a realização da análise da equivalência, DiffMutAnalyze contabiliza o tempo gasto com a inspeção dos códigos. Dessa forma, é contabilizado o tempo efetivamente gasto com a análise. Além disso, a ferramenta permite que o analista de teste indique o grau de dificuldade em analisar o dado mutante. Assim, é possível identificar os mutantes mais difíceis de serem analisados e, consequentemente, verificar qual operador de mutação que o gerou, permitindo ao analista conhecer os operadores de mutação que geram a maior quantidade de mutantes equivalentes e quais possuem maior dificuldade em sua análise.

Esse artigo apresenta as seguintes contribuições: *(i)* auxiliar a identificação de mutantes equivalentes; *(ii)* centralizar a geração dos mutantes e a análise dos equivalentes em um só ambiente; e *(iii)* fornecer relatórios sobre: o tempo total gasto com a análise, o tempo gasto em cada mutante e o grau de dificuldade em analisar tais mutantes.

O restante do trabalho está organizado da seguinte maneira: Seção 2 introduz informações sobre teste de mutação e mutantes equivalentes. A ferramenta proposta é descrita na Seção 3. Os trabalhos relacionados estão referenciados na Seção 4. Por fim, na Seção 5, estão as observações finais e trabalhos futuros.

2. Background

Esta seção fornece informações sobre o teste de mutação e elucida o problema dos mutantes equivalentes. Além disso, fornece informações relacionadas aos operadores de mutação empregados na ferramenta Major.

2.1. Teste de Mutação

O teste de mutação, proposto por DeMillo et al. [DeMillo et al. 1978], é uma proposta eficaz para realizar testes em sistemas de softwares em desenvolvimento. Esse critério de teste introduz mudanças sintáticas no código original e requer que essas alterações sejam descobertas pelos casos de teste construídos. Caso o conjunto de testes definidos inicialmente não identifiquem essas mudanças, novos testes são necessários, ou ainda, os mutantes gerados podem ser equivalentes ao software original. Dessa forma, o teste de mutação contribui com dois fatores [Kintis and Malevris 2014]: *(i)* melhoria da qualidade dos casos de teste; e *(ii)* descoberta de falhas durante o desenvolvimento.

Para avaliar a qualidade de um dado conjunto de testes, os casos de testes existentes são executados nos mutantes gerados, a fim de determinar se a alteração introduzida pode ser detectada, ou seja, se o mutante pode ser morto. Diz-se que um mutante é morto se sua saída for diferente da saída do sistema de software original, quando ambos são executados com o mesmo caso de teste; caso contrário, o mutante permanece vivo. Dessa forma, o ideal é que todos os mutantes sejam mortos pelos casos de teste. Porém, nem todos os mutantes podem ser mortos, esses mutantes são funcionalmente equivalentes ao programa original e, consequentemente, nenhum caso de teste é capaz de detectá-los [Kintins and Malevris 2013]. Estes mutantes são denominados mutantes equivalentes e, como descrito a seguir, eles constituem um custo no teste de mutação.

2.2. Mutantes Equivalentes

A realização de testes de mutação implica em grande quantidade de esforços humanos, contribuindo para o aumento do custo. Um esforço humano bastante significativo está relacionado com a identificação dos mutantes equivalentes. Um mutante é equivalente quando ele possui a mesma semântica do sistema de software original, porém os códigos são sintaticamente diferentes, o que leva a uma possível mudança não observável no comportamento [Jia and Harman 2011]. Uma vez que tem como resultado a mesma saída que o software original. Dessa forma, não existe um caso de teste capaz de distinguir esse mutante do código original [Papadakis et al. 2014].

Com a geração desse tipo de mutante, ele se tornou um dos principais obstáculos da adoção do teste de mutação. O problema do mutante equivalente foi mostrado como sendo indecidível em sua forma geral [Budd and Angluin 1982]. Isso implica que uma solução totalmente automatizada não pode ser atingida [Kintis and Malevris 2014]. Portanto, esses mutantes precisam ser analisados manualmente, para verificar se há, de fato, a equivalência entres os códigos original e mutante. Assim, esses mutantes constituem um custo importante no teste de mutação.

2.3. Ferramenta de Mutação Major

O teste de mutação faz alterações no programa a ser testado e essas alterações são baseadas em um conjunto de regras. Essas regras são implementadas nos operadores de mutação, que por sua vez são projetados para realizar a transformação do código e gerar os mutantes [Ammann and Offutt 2008]. Geralmente esses operadores estão empregados em ferramentas de mutação e, para a abordagem proposta neste artigo, a ferramenta Major[1] foi definida.

Major é uma ferramenta de mutação integrada ao compilador Java e não requer uma estrutura de análise de mutação específica. Por isso, ela pode ser usada em qualquer ambiente baseado em Java. Major manipula a árvore sintática abstrata (*AST - Abstract Syntax Tree*) analisando o código-fonte do programa em teste. Como a Major realiza as alterações utilizando a AST, que representa o código desenvolvido, evita gerar mutantes que não podem ser mapeados para uma localização específica no código original [Just 2014].

3. DiffMutAnalyze

A ferramenta de apoio computacional proposta, denominada DiffMutAnalyze [2], objetiva auxiliar os analistas de teste na identificação dos mutantes equivalentes. A principal motivação para o desenvolvimento dessa ferramenta é tentar minimizar o alto custo da análise desses mutantes. Além de fornecer evidências do grau de dificuldade em analisar tais mutantes e o tempo efetivamente gasto com a análise. A ferramenta possui o propósito de localizar a alteração realizada e apresentar ao analista de teste a comparação, lado a lado, da diferença entre o código original e o mutante. Desse modo, os objetivos da ferramenta incluem: *(i)* acessar o projeto diretamente pelo GitHub ou importando uma pasta *.zip* do projeto; *(ii)* gerar os mutantes através da ferramenta Major; *(iii)* disponibilizar os mutantes vivos para análise da equivalência; *(iv)* associar o grau de dificuldade na análise com o mutante gerado; *(v)* contabilizar o tempo da análise; e *(vi)* disponibilizar relatórios com os dados da análise.

3.1. Proposta da Abordagem

DiffMutAnalyze é uma ferramenta que gerencia e controla os mutantes analisados. Um gráfico (ou quadro) é gerado com todos os mutantes que foram destinados para análise. Assim, o analista de teste consegue gerenciar e controlar a análise desses mutantes. Após a inspeção dos códigos – original e mutante – o analista de teste determina a equivalência de ambos e avalia a análise de acordo com o grau de dificuldade. Finalizada a avaliação, o analista de teste procede com a análise do próximo mutante.

Caso haja dúvida da equivalência de um mutante com seu código original correspondente, DiffMutAnalyze permite que o analista de teste prossiga para a próxima análise e volte posteriormente para esse mutante. Dessa forma, o gráfico dos mutantes possibilita ao analista de teste identificar os mutantes que foram analisados e determinada sua equivalência, assim como, identificar os mutantes que foram analisados, porém sua equivalência não foi determinada. Além disso, os mutantes que não foram analisados são marcados, para que o analista de teste obtenha a visualização desses mutantes e possa escolher qual será o próximo a ser inspecionado.

O tempo gasto pelo analista de teste em analisar os mutantes é contabilizado em cada análise e, ao final, é possível obter o tempo total gasto. Para que o tempo seja contabilizado de maneira real, um *timer* fica disponível na tela para que o analista possa pausar a contagem do tempo, caso seja necessário. Um relatório é exibido ao analista

[1] Major. Disponível em: http://mutation-testing.org/

[2] Disponível em: https://github.com/PqES/DiffMutAnalyze

de teste contendo todos os dados da análise, como quantidade de mutantes equivalentes, avaliação do grau de dificuldade e tempo gasto com a análise.

3.2. Visão geral da DiffMutAnalyze

Conforme ilustrado na Figura 1, a ferramenta DiffMutAnalyze permite que o analista de teste inclua projetos diretamente pelo GitHub, inserindo a *URL* do projeto, ou submeta uma pasta (*.zip*) do projeto que será analisado (Figura 1 (a)), os projetos inseridos precisam conter casos de teste criados previamente. Após inserir o projeto, a ferramenta gera os mutantes utilizando os operadores da ferramenta de mutação Major (Figura 1 (b)). Os mutantes gerados são armazenados em um diretório do projeto, contendo todas as classes Java alteradas. Com os mutantes gerados, os casos de teste são executados nos mutantes e são verificados quais mutantes foram mortos e quais sobreviveram (Figura 1 (c, d)).

Os mutantes que não foram detectados pelos casos de teste (ou permaneceram vivos) são agrupados e destinados para análise entre eles e o código original (Figura 1 (e)). DiffMutAnalyze exibe para o analista de teste os códigos original e mutante que permaneceu vivo lado a lado para serem analisados manualmente (Figura 1 (f)). A cada mutante analisado, o analista de teste indica se há ou não a equivalência entre os códigos e a ferramenta contabiliza o tempo gasto da análise do mutante em questão (Figura 1 (g)). Além disso, o analista de teste deve indicar o grau de dificuldade da análise (Figura 1 (h)). Assim, a análise é realizada para todos os mutantes que sobreviveram e relatórios (Figura 1 (i)) são gerados com os dados da análise.

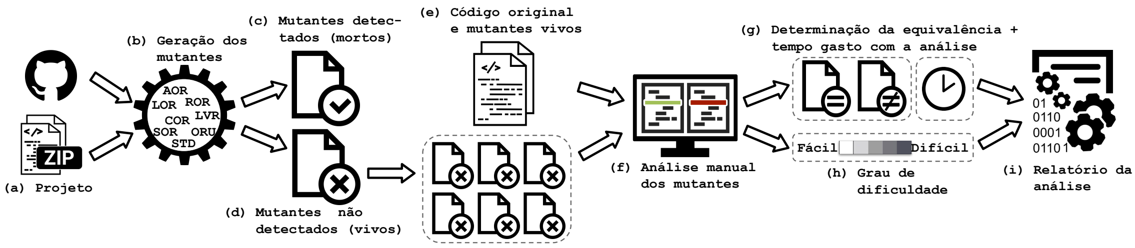


Figura 1. Passo a passo da ferramenta DiffMutAnalyze.

A ferramenta DiffMutAnalyze foi desenvolvida em Java, utilizando tecnologias como o *framework* Spring Boot[3], JavaScript e HTML e os dados são armazenados no banco de dados MySQL. DiffMutAnalyze possui um *design web*, porém seu código será disponibilizado para que seja utilizada no servidor local. O principal motivo de utilizar a ferramenta localmente, é devido à segurança do código do projeto. Uma vez que o mesmo será submetido à ferramenta e seu código inserido no banco de dados, para que todo o processo da mutação seja realizado.

Como pode ser observado na Figura 2, a ferramenta DiffMutAnalyze disponibiliza ao analista de teste a visualização dos códigos lado a lado e permite que o usuário informe se os códigos são equivalentes e o grau de dificuldade em analisá-los. Mais detalhes sobre os procedimentos para realização da análise estão descrito na seção a seguir.

3.3. Procedimentos para realização da análise

1) Seleção do projeto a ser analisado: Para facilitar a geração e análise dos mutantes sobreviventes, DiffMutAnalyze permite que o analista de teste insira o projeto e, no mesmo

[3]https://projects.spring.io/spring-boot/

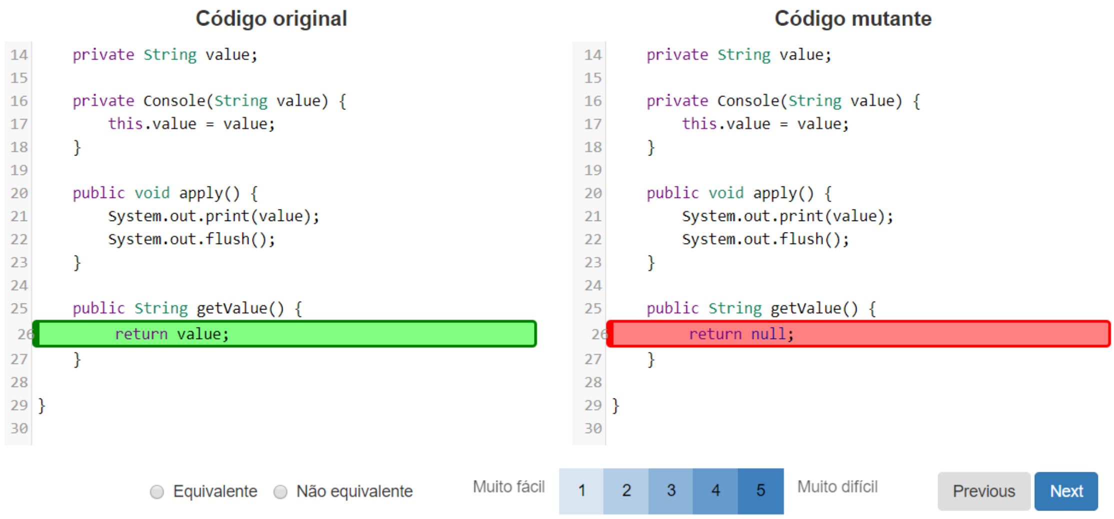


Figura 2. Analisador DiffMutAnalyze.

ambiente, a ferramenta gera os mutantes, executa os casos de teste e mantém os mutantes que serão analisados. Uma vez que os mutantes vivos são selecionados, DiffMutAnalyze busca pelo primeiro mutante e o apresenta para análise.

2) Projeto inserido: DiffMutAnalyze gera uma árvore contendo a estrutura do projeto inserido, essa funcionalidade permite ao analista de teste navegar pelo projeto e escolher a classe que deseja analisar.

3) Quadro dos mutantes: Os mutantes que sobreviveram são inseridos em um quadro. Assim, é possível identificar quais mutantes serão analisados em cada classe do sistema de software. Cada mutante é representado por um retângulo e possui três cores distintas: *(i)* azul; *(ii)* verde; e *(iii)* vermelha, no qual cada cor representa um *status* do mutante. Os mutantes representados pela cor azul são os mutantes que não foram analisados. A cor verde referencia mutantes que foram analisados e avaliados, determinada sua equivalência e o grau de dificuldade da análise. A cor vermelha representa os mutantes que foram inspecionados, porém, sua equivalência não foi definida. O quadro contendo todos os mutantes e seus status possibilita ao analista de teste obter uma visão geral da situação da análise, além de permitir selecionar um mutante para análise clicando sobre o retângulo correspondente.

4) Buscar mutante não analisado: Durante a análise dos mutantes podem surgir dúvidas quanto a sua real equivalência. Dessa forma, o analista de teste pode seguir adiante e inspecionar outro mutante, retornando à esse mutante posteriormente. Assim, a ferramenta busca pelo primeiro mutante não analisado ou "pulado" por meio do botão *"Buscar mutante não analisado"*, colocando-o em evidência no campo de análise dos códigos.

5) *Diff* entre os códigos original e mutante: DiffMutAnalyze exibe os códigos original e mutante lado a lado e coloca em evidência a mutação realizada. Isso é um benefício para tornar a análise mais rápida. Suponha uma classe com muitas linhas de código, assim, o analista de teste não necessita procurar onde foi efetuada a mutação.

6) Determinação da equivalência entre os códigos: Após a inspeção dos códigos, o analista de teste indica se há equivalência ou não do mutante com o código original.

7) Avaliação do grau de dificuldade: Além de determinar se o mutante é equivalente a seu original correspondente, o analista de teste indica o grau de dificuldade da análise. Essa funcionalidade foi determinada para auxiliar a identificar, ao final da análise, os mutantes mais difíceis de serem analisados e, consequentemente, determinar o operador de mutação que o gerou, considerando assim a complexidade da análise.

8) Geração do relatório da análise: O relatório da análise é emitido através do botão *"Gerar relatório"*. Os dados são exibidos em três estágios: *(i)* resultados da análise total dos mutantes; *(ii)* resultados da análise por classe do sistema de software sendo testado; e *(iii)* resultados da análise por mutante. Além disso, o cálculo do escore mutação é informado. O primeiro estágio informa: o tempo total gasto com a análise de todos os mutantes, o tempo médio por mutante, o grau de dificuldade médio da análise, a quantidade total de mutantes que foram determinados como equivalente e a quantidade de mutantes não equivalentes. O segundo estágio informa os resultados por meio das mesmas variáveis do estágio anterior, a diferença é que os resultados são exibidos por cada classe do projeto. Por fim, o terceiro estágio do relatório, exibe os resultados por mutante, informando o tempo gasto com a análise de determinado mutante, o grau de dificuldade atribuído a ele e a equivalência definida.

4. Trabalho relacionado

Uma abordagem de gamificação para identificar os mutantes equivalentes de maneira interativa foi proposta por Houshmand e Paydar, denominada EMVille [Houshmand and Paydar 2017]. Essa abordagem propõe uma estrutura que procura reduzir as limitações das técnicas automatizadas existentes. EMVille é na verdade um sistema baseado na *web* onde os analistas de teste analisam os mutantes sobreviventes através de um jogo.

Os resultados do experimento mostram que o envolvimento dos participantes é visivelmente aumentado pela abordagem de gamificação. Além disso, EMVille possui o *Diff Visualizer*, um visualizador que mostra as diferenças do código original e do mutante de uma forma que o jogador pode compará-los, assim, os participantes do experimento informaram que os componentes que contemplam essa abordagem auxiliam na análise das instâncias dos mutantes. Dessa forma, é possível perceber que a existência de uma ferramenta para auxiliar a análise dos mutantes sobreviventes é útil durante o processo do teste de mutação. Em relação à ferramenta proposta neste artigo, além de possuir a visualização dos códigos lado a lado, há as informações do grau de dificuldade em analisar os mutantes e a contabilização real do tempo gasto com a análise.

5. Conclusão

No contexto do teste de mutação, os mutantes formam os objetivos do processo de teste. Assim, casos de teste que são capazes de distinguir os comportamentos dos mutantes e do código original são indicados a identificar possíveis falhas no sistema de software [Papadakis et al. 2017]. Na prática, alguns desses mutantes são equivalentes, ou seja, eles formam versões funcionalmente equivalentes ao sistema de software original, dessa forma, eles não contribuem para melhorar o conjunto de casos de teste e precisam ser analisados para verificar se são realmente equivalentes ao código original ou se mais casos de teste são necessários para identificar os mutantes sobreviventes.

Com base nesse contexto, este artigo propôs uma abordagem que auxilia na identificação dos mutantes equivalentes. Como resultados, espera-se que a ferramenta Diff-

MutAnalyze contribua de forma eficiente com a realização da análise dos mutantes, sendo possível inspecionar os mutantes sobreviventes comparando-os com sua versão original e obter informações para verificar o custo real da análise dos mutantes. Uma vez que a ferramenta permite que o analista de teste determine a equivalência dos mutantes analisados e indique o grau de dificuldade de cada análise, além de contabilizar o tempo gasto total e de cada análise dos mutantes.

Em suma, as informações extraídas podem auxiliar a outros pesquisadores a utilizar os operadores de mutação pontualmente. Uma vez que serão analisados quais os operadores que mais contribuem com o custo, gerando maior quantidade de mutantes equivalentes e quais os operadores mais complexos de serem analisados, consequentemente, o analista de teste pode decidir quais operadores serão utilizados para outros sistemas de software. Além disso, a ferramenta será útil para estudos empíricos em projetos de pesquisa. Como trabalho futuro, pretende-se realizar um experimento para avaliar a funcionalidade da ferramenta e obter informações reais do custo em analisar os mutantes sobreviventes.

AGRADECIMENTOS

Este projeto é apoiado pela FAPEMIG.

Referências

Ammann, P. and Offutt, J. (2008). *Introduction to Software Testing*. Cambridge University Press.

Budd, T. A. and Angluin, D. (1982). Two notions of correctness and their relation to testing. *Acta Informatica*, 18(1):31–45.

Delamaro, M., Maldonado, J., and Jino, M. (2016). *Introdução ao teste de software*. Elsevier, 2nd edition.

DeMillo, R. A. (1980). Mutation analysis as a tool for software quality assurance. Technical report, Georgia Int of Tech Atlanta School of Information and Computer Science.

DeMillo, R. A., Lipton, R. J., and Sayward, F. G. (1978). Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41.

Grün, B. J., Schuler, D., and Zeller, A. (2009). The impact of equivalent mutants. In *2nd International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 192–199. IEEE.

Houshmand, M. and Paydar, S. (2017). Emville: A gamification-based approach to address the equivalent mutant problem. In *7th International Conference on Computer and Knowledge Engineering (ICCKE)*, pages 326–332.

Jia, Y. and Harman, M. (2011). An analysis and survey of the development of mutation testing. *IEEE transactions on Software Engineering*, 37(5):649–678.

Just, R. (2014). The major mutation framework: Efficient and scalable mutation analysis for java. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 433–436. ACM.

Kintins, M. and Malevris, N. (2013). Identifying more equivalent mutants via code similarity. In *20th Asia-Pacific Software Engineering Conference (APSEC)*, pages 180–188.

Kintis, M. and Malevris, N. (2014). Using data flow patterns for equivalent mutant detection. In *7th International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 196–205.

Papadakis, M., Delamaro, M., and Le Traon, Y. (2014). Mitigating the effects of equivalent mutants with mutant classification strategies. *Science of Computer Programming*, 95:298–319.

Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Le Traon, Y., and Harman, M. (2017). Mutation testing advances: An analysis and survey. *Advances in Computers*.

Lista de Autores | *Author Index*