IV Workshop on Software Visualization, Evolution and Maintenance

CBSOFT.ORG

REALIZAÇÃO:

EXECUÇÃO:

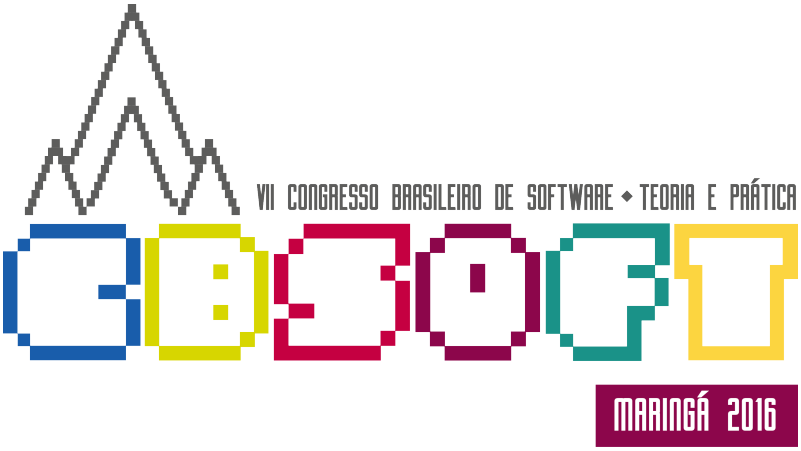
ORGANIZAÇÃO:

APOIO:

FOMENTO:

PATROCINADORES GIGA:

PATRCCINADORES MEGA:



4th WORKSHOP ON SOFTWARE VISUALIZATION, MAINTENANCE, AND EVOLUTION (VEM 2016)

September 21, 2016
Maringá, PR, Brazil

PROCEEDINGS

Sociedade Brasileira de Computação – SBC

PROGRAM COMMITTEE CHAIRS
Alexandre Bergel (Universidad de Chile)
Claudio Sant'Anna (UFBA)

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

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

REALIZATION
Sociedade Brasileira de Computação (SBC)

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

ISBN: 978-85-7669-334-5

Foreword

The continuing need to keep (i.e., identifying and correcting defects and improving the quality of the internal code) and develop (i.e., adding new features) software systems is a major challenge in modern software engineering. Several studies show that activities related to software evolution or maintenance have been responsible for a considerable cost on the development process, reaching in some cases, 90% of the total cost.

Complex software systems require advanced means for understanding the software artifacts to be modified during evolution or maintenance tasks. Software visualization relies on the use of visual resources to facilitate software comprehension. Due to this clear intersection of the research challenges involving the areas of Software Visualization, Software Evolution, and Software Maintenance, there is a need for a joint forum to discuss these challenges together.

The forth Workshop on Software Visualization, Maintenance, and Evolution (VEM 2016) aims at bringing together the research communities interested in topics related to these areas to share and discuss ideas, strengthen research group collaborations and identify new research opportunities. It builds upon the success of the first three editions of VEM, which in turn followed after two editions of the Brazilian Workshop on Software Visualization and ten editions of the Brazilian Workshop on Software Maintenance.

This year, we have received 27 submissions, from which 15 have been accepted. Eight submissions were written in English and 19 in Portuguese. Each submission was reviewed by at least three members of the program committee. The program committee included members from Brazil, Argentina, Chile, Germany and Switzerland. We thank all them for their effort to produce their reviews.

We also had three keynote speakers: Marco Aurelio Gerosa (University of São Paulo), Nabor Mendonça (University of Fortaleza) and Rohit Gheyi (Federal University of Campina Grande).

Alexandre Bergel (University of Chile, Chile)
Cláudio Sant'Anna (Federal University of Bahia, Brazil)
VEM 2016 Co-chairs

Technical committee

PC chairs

Alexandre Bergel (Universidad de Chile)
Claudio Sant'Anna (UFBA)

Program committee

Alexandre Bergel (Universidad de Chile)
Aline Vasconcelos (IFF)
Andre Hora (UFMS)
Andrei Chis (University of Bern)
Auri Marcelo Rizzo Vincenzi (UFG)
Bruno Silva (UNIFACS)
Christina Chavez (UFBA)
Claudio Sant'Anna (UFBA)
Dalton Serey (UFCG)
Eduardo Figueiredo (UFMG)
Eduardo Guerra (INPE)
Elder Cirilo (UFSJ)
Elisa Huzita (UEM)
Fabian Beck (University of Stuttgart)
Fernando Castor (UFPE)
Francisco Dantas (UERN)
Gustavo Rossi (Universidad Nacional de La Plata)
Heitor Costa (UFLA)
Igor Steinmacher (UTFPR)
Juan Pablo Sandoval (Universidad de Chile)
Leonardo Murta (UFF)
Lincoln Rocha (UFC)
Marcelo Maia (UFU)
Marcelo Pimenta (UFRGS)
Marcelo Schots (UERJ)
Marco Tulio Valente (UFMG)
Maria Istela Cagnin (UFMS)
Nabor Mendonca (UNIFOR)
Patrick Brito (UFAL)
Renato Novais (IFBA)
Ricardo Terra (UFLA)
Rodrigo Rocha (UFBA)
Rodrigo Spínola (UNIFACS)
Rosana Braga (ICMC-USP)

Santiago Vidal (Universidad Nacional del Centro de la Pcia. de Buenos Aires)
Tudor Gîrba (feenk gmhb)
Uirá Kulesza (UFRN)
Valter Camargo (UFSCar)

External reviewers

Hudson Borges (UFMG))
Paulo Meirelles (UnB))
André Luiz de Oliveira (ICMC-USP))
Leonardo Humberto Silva (UFMG)

Technical papers

Dolly or Shaun? A Survey to Verify Code Clones Detected using Similar Sequences of Method Calls

Alexandre Paiva, Johnatan Oliveira, Eduardo Figueiredo

Department of Computer Science – Federal University of Minas Gerais (UFMG)
Belo Horizonte, Brazil

ampaiva@gmail.com, {johnatan-si, figueiredo }@dcc.ufmg.br

Abstract. *Software developers usually copy and paste code from one part of the system to another. This practice, called code clone, spreads the same logic over the system, hindering maintenance and evolution tasks. Several methods have been proposed to detect code clones. This paper presents a survey to evaluate code clones automatically detected by analyzing similar sequences of method calls. The survey was conducted with 25 developers that, by means of code inspection, analyzed the code clone candidates. Results showed that more than 90% of participants confirmed the code clones. Therefore, we conclude that sequences of method calls are good indicators of code clone.*

1. Introduction

A common decision made by a software developer when coding is to reuse existing code as a reference [Rattan et al., 2013]. Reasons vary and are not mutual exclusive. They include getting an idea of how to solve a specific problem or using something already tested [Ducasse et al., 1999]. Copying existing code fragments and pasting them with or without modifications into other parts of the system is called code clone, an important area of software engineering research [Rattan et al., 2013; Ducasse et al., 1999; Roy et al., 2009]. This practice is considered a bad smell [Fowler, 1999] since it leads to problems during software development and maintenance tasks. The reason is that code clones duplicate logic and, therefore, increase points of refactoring and error fixing.

In order to detect code clones, several methods and tools have been proposed. The most common strategies are based on tree [Wahler et al., 2004], program dependency graph (PDG) [Krinke, 2001; Komondoor and Horwitz, 2001], text [Johnson, 1993], token [Hummel et al., 2010], metrics [Marinescu, 2004; Kontogiannis, 1997], and hybrid techniques [Göde and Koschke, 2011]. For instance, Baxter et al. (1998) propose a tree-based clone detection technique. In their approach, a program is first transformed on an abstract syntax tree (AST) and, for every sub-tree in the AST, similar sub-trees are compared pairwise.

In a previous work [Paiva, 2016], Paiva observed that code clones can also be detected by comparing similar sequences of method calls in different parts of the system. That is, when a code is copied from one part of the system and pasted into another, all instructions of the original code come together. Therefore, a code clone is a repetition of sequence of instructions in different points of the system. After the code clone operation, the parts can evolve independently, receiving different changes. These

changes make harder to detect the code clone. However, part of the original sequence of method calls is supposed to be preserved, since the original computation is one of the reasons for code cloning.

Although several different methods for code clone detection have been proposed, we lack empirical knowledge about the developers' subjective view of detected code clone. In other words, we need to know whether developers agree with code clone instances detected by these methods. To fill this gap, this paper presents and discusses the results of a survey with 25 software developers (Section 3). The survey contains 12 random samples of code clones automatically detected using similar sequences of method calls in 5 software systems. After filling a characterization form with their background profile, participants were asked to answer whether each sample (i.e., pair of code) is clone or not. In addition, the subjects should explain their reasons of choice.

The preliminary results (Section 4) show that, in general, more than 90% of subjects agree with the detected code clones. Subjects did not confirm only 2 out of 12 code clone candidates. Even these cases, the decision whether they are code clones or not was not a consensus. Indeed, in these two cases, subjects were divided half to half in their opinions. Therefore, results so far indicate that sequences of method calls is a prominent strategy for detecting code clones. This paper also discusses limitations of the study (Section 5) and related work (Section 6). Section 7 concludes this paper and points out directions for future work.

2. Background

This section provides background information to support the comprehension of our study. Section 2.1 briefly explains code clones, including definition and types of code clones. Section 2.2 discusses some common techniques to support code clone detection. Section 2.3 presents a technique for code clone detection based on similar sequences of method calls.

2.1. Code Clones

Code clones are fragments of source code that are similar, or exactly the same, with respect to structure or programming logic [Rattan, 2013]. Eventually, software developers clone code, i.e., copy and paste existing code from one part to another of a software system, to support the development of new system features [Rattan, 2013]. One reason for this practice is that code clones contains tested components, clear solutions, and useful programming logic that may be reused [Ducasse et al., 1999]. In addition, some development external factors, such as time constraints and productivity evaluation, may lead to code cloning practices [Ducasse et al., 1999].

Two code fragments may be considered code clones based on syntax similarity or also based on features provided by the fragments [Koschke et al., 2006]. Roy et al. [2009] discuss four main types of code clones as follows. Code clone *Type I* is related to exact copies of source code fragments, without modifications other than blank spaces and comments. *Type II* is defined as identical copies with respect to syntax. In this case, modifications cover variables, types, and function identifiers, for instance. Code clone *Type III* concerns copies with more significant modifications, such as addition, editing,

and removal of declarations. Finally, *Type IV* is related to two fragments of source code that have different structures, but provide the same computation. This type is considered the most complex because it requires an understanding of the code behavior to be detected [Bellon et al., 2007].

2.2. Common Techniques for Code Clone Detection

Many methods have been proposed in the literature to support the detection of code clones, using different detection strategies [Bellon et al., 2007; Hummel et al., 2010; Johnson, 1993]. Depending on the type of code clone we aim to detect, some of these strategies may provide better results than others. Some of the main strategies in literature are text-based, token-based, tree-based, and PDG-based [Rattan et al., 2013].

The text-based methods aim to detect code clones that differ at most in terms of code format layout and comments [Johnson, 1993]. Some transformation in the source code layout may be conducted to ease the textual analysis. In the token-based detection of code clones, tokens are extracted for the source code lines under analysis, in accordance with the analyzed programming language [Yuan and Guo, 2012]. Blank spaces, line breaks, and comments are removed for the comparison of tokens. Tree-based methods use Abstract Syntax Trees (AST) to support code clone detection [Baxter et al., 1998]. Identical or similar sub-trees are detected by a pairwise comparison. Finally, code clones can also be detected by means of the Program Dependency Graph (PDG) [Krinke, 2001]. Basically, a PDG is a representation of data and control dependencies in a software system. Subgraphs are analyzed for detection of code clones.

2.3. Code Clone Detection based on Sequence of Method Calls

In previous work [Paiva, 2016], Paiva propose a technique to detect code clones based on similar sequences of method calls. This technique analyses all method calls from a given software system. The occurrence of similar sequences of method calls in two different parts of the source code indicates a possible occurrence of code clone. The longer the sequence, the greater are the chances of these two parts being a code clone. The reason behind this technique is that, when source code fragments are copied from one part to another in the software system, these copies carry a set of instructions such as declarations, statements, operations, and method calls, for instance [Kim et al., 2004]. Therefore, code clones may be considered as replications of instruction sets in different parts of the system.

This code clone detection technique can be summarized in four consecutive steps. In Step 1, given an input software system, we extract all method calls from the system. Then, in Step 2, we gather the extracted method calls per method. In Step 3, we select only methods with at least 10[1] method calls. In Step 4, for each selected method, we pairwise the calls of this method with other methods (in the same file or in other files from the system) to assess whether they have similar sequences of calls. If the two compared methods have similar sequences of method calls, we consider both sequences as a code clone candidate.

[1] We used 10 as default in this study based on empirical observations with varying values [Paiva, 2016].

3. Study Settings

This paper aims to evaluate the technique proposed by Paiva [Paiva, 2016], to detect code clones based on similar sequences of method calls (Section 2.3). In this evaluation, we executed a survey with 25 participants. Each participant had to analyze 12 code clone candidates detected using the technique under evaluation.

Background of the Participants. The 25 participants of this survey are undergraduate and graduate students. Each participant filled in a characterization form[2] intended to collect profile data. The form questions asked the participants regarding their skills on the following areas: Object Oriented Programming, Java Programming, Bad Smells, Code Clone, and Work Experience. The characterization form asked the subjects to self-classify their skills on these areas using the following options: None, Few, Moderate, or Expert. The goal of collecting profile data is to further correlate with results of the main survey.

Survey Structure. The main survey was submitted to the participants via a specialized website[3]. Each subject was invited to analyze and decide for 12 pairs of code if they are clones. These 12 code clone candidates were randomly chosen from the total amount of 187 code clones automatically detected in 5 software systems using the technique presented in Section 2.3. Participants of this study rely on their subjective opinion to decide whether a pair of code fragment is clone. In addition, they were also asked to fill in a free text field explaining the reason of their choice. The same set of code clone candidates was presented to all subjects.

Training Session and Guidelines. All subjects received a 30-minute training session, divided in two parts. The first part was an explanation about refactoring code clones [Fowler, 1999]. For instance, Extracted Method was used to exemplify a recommended refactoring for code clone. In the second part of the training, we give general guidelines to answer the survey. The code clones of the training were not the ones of the applied survey. Both training and the survey took place in a controlled environment; i.e., a computer laboratory with 30 equally personal computers. After the training session, participants had one hour to finish the survey. This time was enough for all.

4. Results and Discussion

This section presents and discusses the results of our survey. Section 4.1 characterizes the participants of this survey with respect to five areas of expertise. Section 4.2 summarizes the main results on whether participants confirm the code clones detected by the evaluated technique. Section 4.3 discusses the results considering the different profiles of participants, with focus on their knowledge in code clone.

4.1. Characterization of Participants

Figure 1 shows the profiles of all 25 participants of survey according to five knowledge areas. For simplification purpose, we classify the knowledge of participants in two

[2] https://eSurv.org?u=characterizationform

[3] https://eSurv.org/?u=iscodeclone

groups: few/none and moderate/expert. The y-axis presents the absolute number of participants per knowledge level. Data in Figure 1 show that, in general, participants have balance experience in the asked areas. For instance, most participants consider themselves as moderate or expert in object-oriented programming (OOP), but with few or no experience in other areas. These results are somehow expected since participants are mostly undergraduate students.

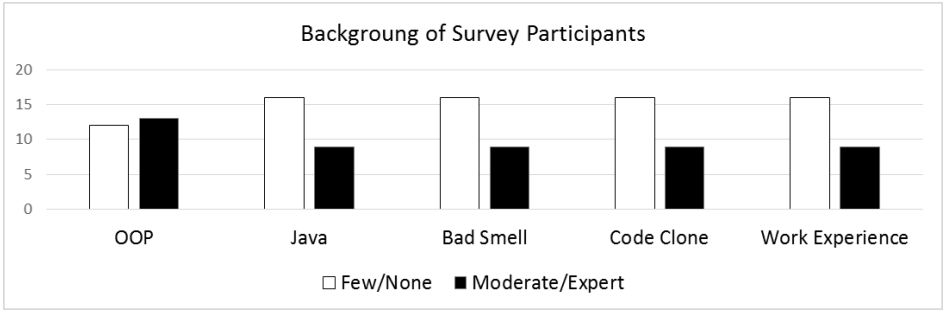


Figure 1. Background of the participants.

4.2. Overall Results

Figure 2 shows the overall agreement of participants in each code clone candidate. An agreement of 100% means that all participants indicated the pair of fragments as a code clone. This figure is ordered from lowest to highest agreement. That is, the first column in Figure 2 is not necessarily the first clone candidate in the survey. Figure 2 shows that, for seven candidates (i.e., columns 6 to 12), more than 90% of participants agree that they are actual code clones. In 3 out of 12 cases, between 50% and 60% of participants also agree with the codes as being clones. On the opposite, only two cases have agreement below 50%. Therefore, in general, results in Figure 2 indicate that a coincident sequence of method calls can be a good indicative of code clone.

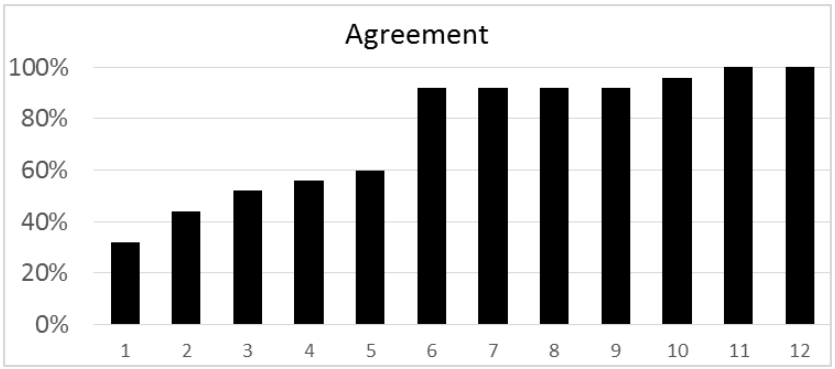


Figure 2. Agreement of the code clone candidates.

We observed in the qualitative answers that divergent judgment express how difficult is to have an agreement. Even for cases with low agreement, the open answers make it clear that the judgement of code clones is not easy. For instance, we present below two opposite points of view expressed by two participants with respect to the same code clone candidate. These participants disagree since the former considers both codes as doing same computation, while the latter does not.

"Yes, it is code clone. Although they are different from each other, codes do the same thing."

"No, it is not code clone. They are similar, but the methods do very different calculations."

4.2. Analysis of Participant Profiles

We investigate a possible relation between the background of participants and their tendency to indicate a code fragment clone or not. A general interesting observation is that, for most code fragments, participants with weak background – i.e., with the none or few knowledge in the asked expertise (see Section 3) – tend to agree with the code clone detection more often. As a representative of this observation, Figure 3 shows the results for all participants classified by their claimed expertise in code clone. Since the number of participants with each profile varies considerably, we present results as a percentage of participants with the specific profile. For instance, considering case 1 in Figure 3, almost 40% of participants with no or few experience in code clone indicated this pair of fragments as code clone.

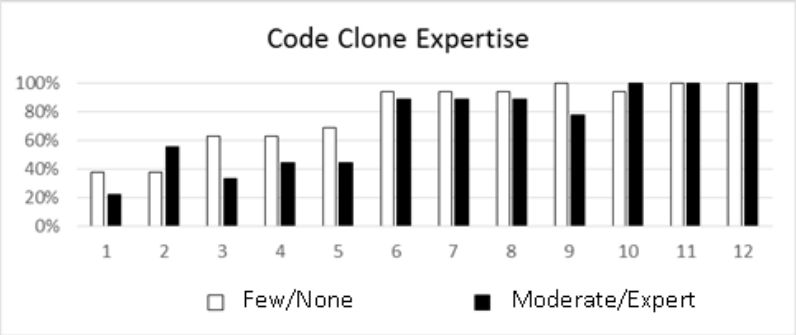


Figure 3. Background of the participants.

As presented in Figure 3, only in two cases, namely 2 and 10, a higher percentage of participants with moderate or high expertise in code clone confirmed the code clone candidates in comparison with participants in the few/none category. On the other hand, in eight cases, the percentage of participants with low profile are higher than the percentage of participants with better knowledge. Due to space constraints, we do not present the other analyzed profiles, but this general observation is similar to most cases of participants considering expertise in OOP, Java, and Bad Smell. Unlike the other profiles, participants with higher work experience confirmed more code clone candidates than participants with low work experience did.

5. Threats to Validity

Since the survey involves several steps, the various threats may have impacted on its validity [Wohlin et al., 2012]. The main question is whether the results can be generalized to a broader population. We cannot claim that our findings can directly apply to other software systems and to different clone detection techniques. However, we randomly selected code clone candidates for industry-strength projects and, so, we believe the most of our findings also hold to other similar contexts.

Another threat to generalization is that participants of our study are undergraduate and post-graduate students. However, a recent work have shown that the results of experiments with students are similar to the results with experienced professionals [Salman et al., 2015]. In addition, most of our results depend on the subjective opinion of the selected participants. Finally, the paper conclusions were

derived based on a discussion among the paper authors, and other researchers may come up with different a point of view. To easy replications and further analysis, we report the complete data in a supplementary website[4].

6. Related Work

Several techniques have been proposed to detect code clone [Bellon et al., 2007; Hummel et al., 2010; Johnson, 1993]. These detection techniques are also evaluated in published research work [Bellon et al., 2007; Roy et al., 2009]. For instance, Kim et al (2004) conducted an ethnographic study in order to understand programmers copy and paste practices. Based on their analysis, the authors constructed a taxonomy of code clone patterns. Bellon et al. (2007) performed an experiment to evaluate six techniques to detect code clone. The code clone candidates were evaluated by one of the authors as independent third party. Unlike Bellon et al., we rely on 25 undergraduate and graduate students to evaluate the code clone candidates.

7. Conclusions and Future Work

Many techniques for detecting code clones have been proposed in the past [Rattan et al, 2013]. However, it is hard to validate code clone candidates automatically detected by these techniques. This paper described the survey for evaluation of code clones detected using similar sequences of method calls [Paiva, 2016]. Twenty five participants with different developer profiles answered this survey. After a 30-minute training session, participants had one hour to confirm 12 code clone candidates randomly mined from 5 software systems.

The survey results indicate that more than 90% of participants confirmed the identified code clones. We also investigated the impact of participant profiles on their answers. In this sense, we observed that participants with weak background tend to confirm more code clones than participants with stronger background. However, the opposite happened with respect to work experience. That is, participants with more work experience confirmed more code clones than participants with low experience.

For future work, we plan to replicate this survey with more participants, focusing on different detection techniques and varying types of code clone candidates (e.g., types I, II, III, and IV, see Section 2.1). Based on the results of this study, we also plan to improve the technique to detect code clones based on similar sequences of method calls.

About the paper title: Dolly and Shaun are famous sheep. One of them is a clone.

References

Baxter, I. D., Yahin, A., Moura, L., SantAnna, M., and Bier, L. (1998). Clone Detection using Abstract Syntax Trees. In Proc. of Int'l Conference on Software Maintenance (ICSM).

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

Ducasse, S., Rieger, M., and Demeyer, S. (1999). A Language Independent Approach for Detecting Duplicated Code. International Conference on Software Maintenance (ICSM).

[4] http://ampaiva.github.io/mcsheep/

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

Gode, N. and Koschke, R. (2011). Frequency and Risks of Changes to Clones. In Proc. of International Conference Software Engineering (ICSE), pp. 311-320.

Hummel, B., Juergens, E., Heinemann, L., and Conradt, M. (2010). Index-based Code Clone Detection: Incremental, Distributed, Scalable. In Proc. of Int'l Conf. on Software Maintenance (ICSM), pp. 1-9.

Johnson, J. H. (1993). Identifying Redundancy in Source Code using Fingerprints. In Proc. of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON), pp. 171-183.

Kim, M., Bergman, L., Lau, T., Notkin, D. (2004) An Ethnographic Study of Copy and Paste Programming Practices in OOPL. In Proc. of Int'l Symp. on Empirical Soft. Eng. (ISESE), pp. 83-92.

Kontogiannis, K. (1997). Evaluation Experiments on the Detection of Programming Patterns using Software Metrics. In Proc. of Working Conference on Reverse Engineering (WCRE), pp. 44-54.

Komondoor, R. and Horwitz, S. (2001). Using Slicing to Identify Duplication in Source Code. In Proc. of International Static Analysis Symposium (SAS), pp. 40-56.

Koschke, R., Falke, R., and Frenzel, P. (2006) Clone Detection Using Abstract Syntax Suffix Trees. In Proc. of the 13th Working Conference on Reverse Engineering (WCRE), pp. 253-262.

Krinke, J. (2001). Identifying Similar Code with Program Dependence Graphs. In Proc. of the Working Conference on Reverse Engineering (WCRE), pp. 301-309.

Marinescu, R. (2004). Detection Strategies: Metrics-based Rules for Detecting Design Flaws. In Proc. of International Conference on Software Maintenance (ICSM), pp. 350-359.

Paiva, A. (2016). On the Detection of Code Clone with Sequence of Method Calls. Master Dissertation, Federal University of Minas Gerais (UFMG).

Rattan, D., Bhatia, R., and Singh, M. (2013). Software Clone Detection: a Systematic Review. Information and Software Technology (IST).

Roy, C., Cordy, J., Koschke, R. (2009) Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. Science of Computer Programming, 74(7), pp. 470-495.

Salman, I., Misirli, A., and Juristo, N. (2015). Are Students Representatives of Professionals in Software Engineering Experiments? In Proc. of Int'l Conference on Software Engineering (ICSE), pp. 666–676.

Wahler, V., Seipel, D., Wolff, J., and Fischer, G. (2004). Clone Detection in Source Code by Frequent Itemset Techniques. In Proc. of Int'l Workshop on Source Code Analysis and Manipulation (SCAM).

Wohlin, C. et al. (2012). Experimentation in software engineering. Springer.

Yuan, Y. and Guo, Y. (2012) Boreas: An Accurate and Scalable Token-based Approach to Code Clone Detection. In Proc. of Int'l Conf. on Automated Software Engineering (ASE).

Um Estudo em Larga Escala sobre o Uso de APIs Internas

Aline Brito, André Hora, Marco Tulio Valente

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

alinebrito@ufmg.br, {hora,mtov}@dcc.ufmg.br

Abstract. *Libraries and frameworks are increasingly used when building software systems, and hence their APIs. It is common to divide the APIs into two types, public and internal. While public APIs are stable, internal APIs handle implementation details, are unsupported and unstable. Therefore, their use is not recommended. However, some systems use internal APIs, which is a bad practice in software development. In this paper, we study a set of questions related to the use of internal APIs. Our database is composed of around 260K Java projects and 131M APIs. This paper also includes comments from clients and providers of internal APIs.*

Resumo. *Bibliotecas e frameworks são cada vez mais usados no desenvolvimento de sistemas, e consequentemente suas APIs. É comum a divisão dessas APIs em dois tipos, públicas e internas. Enquanto as APIs públicas são estáveis, as APIs internas envolvem detalhes de implementação, são instáveis e sem suporte. Consequentemente, os provedores não recomendam o uso. Entretanto, verifica-se que alguns sistemas clientes usam APIs internas, uma má prática no desenvolvimento de software. Nesse contexto, estuda-se nesse artigo um conjunto de questões relativas a frequência de utilização de APIs internas. Utiliza-se um dataset com cerca de 260 mil projetos Java e 131 milhões de APIs. O artigo também inclui uma consulta aos clientes e provedores de APIs internas.*

1. Introdução

O uso de bibliotecas e as suas APIs (*Application Programming Interfaces*) no desenvolvimento e manutenção de sistemas de *software* é cada vez mais popular, já que aumentam a produtividade [Mileva et al. 2010, Hora and Valente 2015]. É comum os provedores dessas bibliotecas dividirem suas APIs em dois tipos, internas e públicas [Hora et al. 2016, Mastrangelo et al. 2015]. As APIs públicas são estáveis e tendem a manter a compatibilidade com versões anteriores. Já as APIs internas referem-se a detalhes de implementação, e não são recomendadas para uso externo ao projeto. Além disso, a atualização de uma API interna não garante a compatibilidade com outras versões.

Entretanto, alguns clientes fazem referências a APIs internas nos seus sistemas, isto é, essa má prática de programação é adotada apesar das recomendações contrárias [Businge et al. 2015, Hora et al. 2016, Mastrangelo et al. 2015]. Essas referências a interfaces internas podem envolver um grande impacto no projeto. Pode-se ter comportamentos inesperados no *software* e até mesmo a interrupção do seu funcionamento.

Para distinguir APIs internas e públicas, alguns provedores incluem pacotes específicos. Por exemplo, na IDE Eclipse, APIs internas são implementadas em pacotes que

possuem o nome "internal" [Businge et al. 2015, Businge et al. 2013]. Já o JDK utiliza o prefixo "sun" [Mastrangelo et al. 2015]. O texto a seguir reproduz parte das diretivas do Eclipse e da Oracle relacionadas ao uso de interfaces internas.

Eclipse. "*Packages containing only implementation details have "internal" in the package name. Legitimate client code must never reference the names of internal elements. Client code that oversteps the above rules might fail on diferent versions and patch levels of the platform.*[1]"

JDK. "*The sun.* packages are not part of the supported, public interface. A Java program that directly calls into sun.* packages is not guaranteed to work on all Java-compatible platforms. In fact, such a program is not guaranteed to work even in future versions on the same platform.*[2]"

Para melhor quantificar o cenário que motiva esse trabalho, realizou-se um estudo preliminar onde minerou-se a quantidade de projetos que utilizam pelo menos uma API com as nomenclaturas citadas. Através da infraestrutura Boa analisou-se 263.425 projetos Java GitHub [Dyer et al. 2013]. *Foram encontrados 17.910 projetos (6.8%) usando APIs internas*. Assim, estuda-se neste artigo duas questões de pesquisa centrais relativas ao uso de APIs internas, descritas a seguir. Adicionalmente o estudo inclui o *feedback* de provedores e clientes de interfaces internas.

QP #1. Qual a frequência de utilização de interfaces internas?

QP #2. Qual a distribuição do uso das interfaces internas de uma biblioteca?

São analisadas 17 bibliotecas e suas APIs internas usadas por clientes, em um *dataset* com aproximadamente 260 mil projetos Java GitHub e 131 milhões de APIs. No melhor do nosso conhecimento, esse é o maior estudo sobre o uso de APIs internas Java. Estudos relacionados concentraram em bibliotecas específicas e analisaram menos clientes [Businge et al. 2015, Businge et al. 2013, Mastrangelo et al. 2015]. Um estudo relacionado ao uso de APIs internas apenas no contexto de *plug-ins* Eclipse, por exemplo, analisa 512 projetos [Businge et al. 2015]. Para melhor entendimento, o estudo proposto nesse artigo concentra-se em dois níveis de granularidade: APIs (por exemplo, quantos clientes estão usando org.junit.internal.AssumptionViolatedException?) e bibliotecas[3] (por exemplo, quantos clientes JUnit estão usando interfaces internas?).

O restante deste artigo está organizado conforme descrito a seguir. A Seção 2 apresenta a metodologia proposta; a Seção 3 os resultados obtidos, e a Seção 4 inclui os comentários de clientes e provedores de APIs internas. Na Seção 5 os riscos à validade são apresentados. A Seção 6 discute trabalhos relacionados e a Seção 7 conclui o estudo.

2. Metodologia

Neste estudo analisam-se 17 bibliotecas Java e o uso das suas interfaces internas em sistemas clientes. Para verificar o uso dessas interfaces, utiliza-se um *dataset* composto por 263.425 projetos e 16.386.193 arquivos Java, minerados através da infraestrutura Boa [Dyer et al. 2013]. Essa infraestrutura inclui diversos *datasets* do GitHub, assim como uma DSL (*Domain Specific Language*) para minerar repositórios de *software* e uma interface Web para executar os scripts de mineração.

Criou-se um *script* Boa para minerar projetos que fazem uso de pelo menos uma interface/biblioteca em um arquivo Java válido. Os metadados obtidos, foram então, in-

[1] www.eclipse.org/articles/article.php?file=Article-API-Use/index.html

[2] www.oracle.com/technetwork/java/faq-sun-packages-142232.html

[3] Neste trabalho, o termo "biblioteca" é utilizado para designar tanto *frameworks* quanto bibliotecas.

seridos em um banco de dados onde criou-se um *script* para agrupar as interfaces encontradas e contabilizar seus sistemas clientes. Em seguida minerou-se as interfaces internas. Para a biblioteca JDK, avalia-se interfaces que possuem o prefixo sun.* . Para as demais bibliotecas considera-se o respectivo prefixo e o pacote internal. A Tabela 1 apresenta a descrição das bibliotecas analisadas e o total de sistemas clientes encontrados.

3. Resultados

QP #1: Qual a frequência de utilização de interfaces internas?

Nessa primeira questão de pesquisa estuda-se a frequência com que clientes usam interfaces internas. A Tabela 1 apresenta o total de clientes de cada biblioteca que usam pelo menos uma interface interna. As bibliotecas Action Bar Sherlock e Eclipse apresentam cerca de 20% dos clientes usando interfaces internas. Entre 15% e 5%, encontram-se as bibliotecas Facebook, OpenQA, Twitter4j, Apache Maven e Google Inject. As demais bibliotecas possuem menos de 5% dos clientes usando interfaces internas.

Tabela 1. Clientes Usando Interfaces Internas

Bibliotecas	**Descrição**	**Total Clientes**	**Interfaces Internas**	
			Total	**%**
Action Bar Sherlock	Criação de barra de ação em aplicativos Android	2.729	810	29.68
Eclipse	Integração e extensão com a plataforma Eclipse	9.774	2.304	23.57
Facebook	Integração de aplicativos com o Facebook	1.253	184	14.68
OpenQA	Desenvolvimento de testes	2.497	206	8.25
Twitter4j	Integração com serviços do Twitter	1.583	106	6.70
Apache Maven	Gerenciamento de projetos e compressão de software	2.304	145	6.29
Google Inject	Injeção de dependências	3.808	216	5.67
Mockito	Desenvolvimento de testes	7.393	331	4.48
Easy Mock	Desenvolvimento de testes	2.260	92	4.07
Jboss	Integração com servidores de aplicação JBoss	4.729	190	4.02
Hibernate	Persistência de dados	11.993	331	2.76
Android	Desenvolvimento de aplicativos Android	68.116	1.816	2.67
JDK	Desenvolvimento de aplicativos e componentes Java	237.898	5.838	2.45
Google Gson	Conversão de objetos JSON	7.906	147	1.86
Apache Cordova	Desenvolvimento de aplicativos móveis	3.006	39	1.30
JUnit	Desenvolvimento de testes	67.677	774	1.14
Hamcrest	Desenvolvimento de expressões em testes	8.219	50	0.61

Para algumas bibliotecas o baixo percentual de uso das interfaces internas não implica necessariamente em menos projetos. A Figura 1 apresenta as 5 bibliotecas com o maior número de clientes usando interfaces internas. A biblioteca JUnit, por exemplo, ocupa a penúltima posição considerando o percentual de clientes (1.14%), entretanto é a 5ª biblioteca (774 projetos) com o maior número de clientes usando interfaces internas.

QP #2: Qual a distribuição do uso das interfaces internas de uma biblioteca?

Nessa segunda questão de pesquisa analisa-se até a 50ª interface interna mais usada por biblioteca. A Figura 2 apresenta a distribuição das interfaces mais usadas, para as bibliotecas com o maior percentual de clientes usando interfaces internas. Já a Figura 3 apresenta a distribuição considerando todos os projetos. É mostrado um *box plot* e um *violin plot* (em amarelo), a fim de melhor apresentar a distribuição. O número em

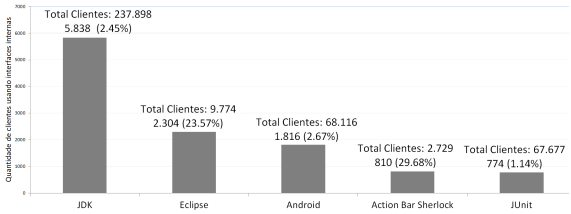
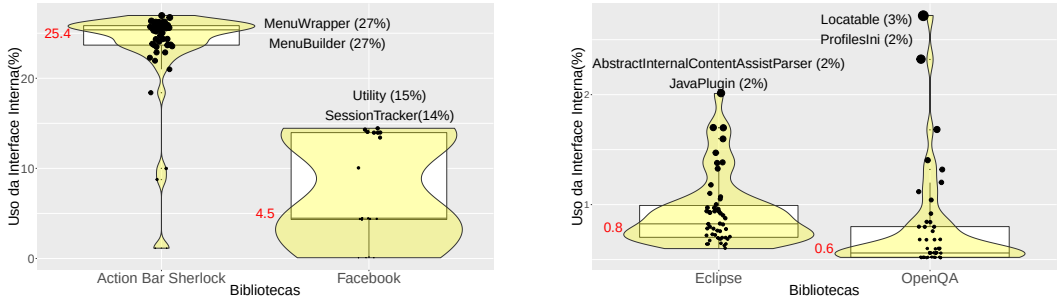


Figura 1. Bibliotecas com mais Clientes Usando Interfaces Internas

vermelho corresponde à mediana; cada ponto traçado corresponde a uma interface, e o seu tamanho é proporcional ao seu uso (quanto maior o ponto, mais usada é a interface).



(a) Action Bar Sherlock e Facebook

(b) Eclipse e OpenQA

Figura 2. Distribuição das Interfaces Internas por Biblioteca

A biblioteca Action Bar Sherlock possui 10 interfaces usadas por mais de 26% dos seus clientes, sendo que com.actionbarsherlock.internal.view.menu.MenuWrapper e com.-actionbarsherlock.internal.view.menu.MenuBuilder são usadas por 27% dos clientes. A biblioteca Facebook possui 18 interfaces internas usadas em sistemas clientes; acima de 14% encontram-se três interfaces, com.facebook.internal.Utility (15%, 181 projetos), com.facebook.internal.SessionTracker (14%, 179 projetos), com.facebook.internal.SessionAuthorizationType (14%, 176 projetos). A biblioteca Eclipse não possui interfaces internas com uso acima de 3%. Sua interface interna mais popular é org.eclipse.xtext.ui.editor.contentassist.-antlr.internal.AbstractInternalContentAssistParser (196 projetos, 2%). A biblioteca OpenQA também não possui interfaces internas com uso acima de 3%, suas interfaces internas mais usadas pertencem ao pacote selenium (org.openqa.selenium.internal.Locatable (3%) e org.-openqa.selenium.firefox.internal.ProfilesIni (2%)). As bibliotecas Google Gson, JDK, Mockito, Hibernate, e Hamcrest não possuem interfaces internas com uso acima de 1%. As bibliotecas Apache Cordova, Google Inject, Android, e JBoss não possuem interfaces internas usadas por mais de 2% dos seus clientes. Por fim, as bibliotecas Apache Maven, Twitter4j, e Easy Mock não tem interfaces internas usadas por mais de 5% dos clientes.

As 50 interfaces internas mais usadas (Figura 3) pertencem às bibliotecas JDK, Action Bar Sherlock e Android. As duas mais usadas pertencem ao JDK, sun.misc.-BASE64Encoder (1.502 projetos, 0.57%) e sun.misc.BASE64Decoder (798 projetos, 0.3%). De fato, o uso de interfaces internas do JDK é bem crítico e perigoso para clientes, e tem sido objeto de estudo da literatura recente [Mastrangelo et al. 2015]. A terceira interface é com.actionbarsherlock.internal.view.menu.MenuWrapper (736 projetos, 0.28%). As interfaces Android ocupam a 27ª posição, com.android.internal.telephony.Phone (694 projetos, 0.26%) e a 49ª posição, com.android.internal.telephony.TelephonyIntents (586 projetos, 0.22%).

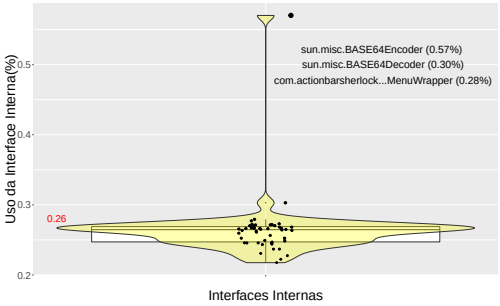


Figura 3. Distribuição das Interfaces Internas

O uso de interfaces internas é um indicativo para os provedores sobre quais elementos internos estão oferecendo recursos necessários para os seus clientes e que provavelmente não são disponibilizados por meio das interfaces públicas.

4. Comentários de Provedores e Clientes de Interfaces Internas

4.1. Consulta aos Clientes de Interfaces Internas

Nessa consulta, verifica-se o conhecimento dos clientes sobre as interfaces internas usadas em seus sistemas. Para tanto, foram consultados 10 desenvolvedores ativos de sistemas clientes cadastrados no GitHub. Para cada desenvolvedor criou-se uma *issue* com as perguntas descritas a seguir.

1. *You used an internal interface in "*name[1]*" class of your project. Do you know that it is an internal interface of the "*name[2]*" library?*
2. *Are you aware that internal interfaces are unstable and may change without backward-compatibility?*

Obteve-se 4 respostas (40%). O primeiro desenvolvedor usou a interface com.android.internal.telephony.Phone em seu projeto, e alegou que era um código-fonte de terceiros. Provavelmente esse desenvolvedor copiou o código-fonte da biblioteca para o seu projeto. O segundo desenvolvedor utilizou a interface interna org.hamcrest.internal.ArrayIterator, e justificou o seu uso devido a uma sugestão da IDE utilizada. O terceiro desenvolvedor utilizou a interface sun.misc.BASE64Encoder. Em sua resposta, ele informou que não sabia que tratava-se de uma interface interna do JDK. O texto a seguir apresenta um trecho da sua resposta.

"I was unaware of how unstable internal interfaces were. From looking into this I have noticed that Oracle have released a Base64 encoder/decoder in the java.util package, so I am now using that instead."

Por fim, o quarto desenvolvedor utilizou a interface com.google.inject.internal.Lists. Em sua resposta ele alegou que tinha conhecimento sobre interfaces internas mas não percebeu que estava utilizando. O texto a seguir apresenta um trecho da suas resposta.

"1. I had not noticed it is an internal. 2. Yes. Now that you have pointed it out, I will probably change it."

Assim, observa-se que o uso de interfaces internas está relacionado com o pouco conhecimento da biblioteca utilizada.

[1]Nome da classe onde a interface interna encontra-se.
[2]Nome da biblioteca. Exemplo: Google Gson.

4.2. Consulta aos Provedores de Interfaces Internas

Nessa consulta, tem-se por objetivo verificar o conhecimento dos provedores sobre os clientes que estão usando suas interfaces internas. Foram criadas *issues* em repositórios GitHub de 4 bibliotecas e obteve-se 3 respostas (75%). Para cada provedor foram realizadas as duas perguntas descritas a seguir. A *issue* também incluiu a lista de interfaces internas mineradas nesse estudo e a quantidade de usuários utilizando as mesmas.

1. *Did you know that internal interfaces are used by clients?*
2. *From the presented interfaces, there is some interface that is a candidate to be promoted to the public one?*

JUnit.[1] Quatro desenvolvedores da biblioteca JUnit responderam a *issue* que foi aberta, sendo um deles um membro *core* do projeto. Obteve-se seis respostas. Dentre os principais pontos discutidos, destaca-se a atualização da biblioteca JUnit a partir da quinta versão, onde o pacote internal será substituído pela notação @API(Internal) e possíveis intervenções para inibir o uso de interfaces internas. Um dos desenvolvedores informou que a equipe está ciente do uso de interfaces internas por clientes, e que essa é uma das razões para essa nova estratégia. O texto a seguir apresenta um trecho da sua resposta.

"Yes, we are aware that many of the originally internal interfaces are used by clients. It's one of the reasons why we're using a different approach of annotating APIs instead of using package names for JUnit 5."

Esse desenvolvedor alegou que cinco interfaces internas da lista apresentada já foram promovidas para o público, entretanto as interfaces foram depreciadas para manter a compatibilidade com versões anteriores. Essa promoção de interfaces internas é objeto de estudo na literatura recente [Hora et al. 2016].

Mockito.[2] Dois desenvolvedores da biblioteca Mockito responderam a *issue* que foi aberta. Obteve-se duas respostas. O primeiro desenvolvedor surpreendeu-se com o uso das interfaces internas por seus clientes, e comentou sobre prováveis ajustes realizados nesses projetos para usar essas interfaces. Adicionalmente, comentou sobre uma possível falta de recursos na biblioteca. O segundo desenvolvedor alegou que algumas funcionalidades deveriam ser adquiridas através de interfaces públicas. Por exemplo, a interface interna org.mockito.internal.verification.Times deve ser acessada via interface pública Mockito.times(). O mesmo informou que já ocorreram discussões sobre o uso de interfaces internas entre os membros da equipe e que realmente considera tal uso uma má pratica de programação. O texto a seguir apresenta um trecho da sua resposta.

"*It seems that quite some internal classes are used instead of relying on the implementation provided by the Mockito.* method. E.g. Times should be obtained via Mockito.times(). I am not sure if we can do something about that.*"

Google Gson.[3] Um dos desenvolvedores da biblioteca Google Gson informou que não tinha conhecimento da quantidade de clientes usando as interfaces internas, entretanto não surpreendeu-se com essa má prática adotada por alguns desenvolvedores clientes. Segundo ele, nenhuma das interfaces listadas por nós na *issue* são candidatas a promoção para o público. O texto a seguir apresenta um trecho da suas resposta.

[1] https://github.com/junit-team/junit5/issues/305
[2] https://github.com/mockito/mockito/issues/428
[3] https://github.com/google/gson/issues/874

"I didn't quite know this, but am not very surprised. None of these are candidates for promotion to public APIs."

Assim, após a consulta aos desenvolvedores de sistemas provedores de interfaces internas, observa-se que essa má prática é conhecida. As respostas dos desenvolvedores JUnit levantam também uma discussão em torno de outra atividade crítica, além do uso de interfaces internas, sistemas clientes estão usando interfaces internas depreciadas.

5. Riscos à validade

Validade externa. Os resultados desse estudo estão restritos a projetos Java GitHub, ou seja, eles não podem ser generalizados para outras linguagens e outros repositórios de código-fonte. Além disso, dentre os 260 mil sistemas analisados, alguns podem ser repositórios GitHub das bibliotecas analisadas nesse trabalho.

Validade interna. Projetos que importaram somente a biblioteca não foram considerados clientes das interfaces, já que a importação de uma biblioteca não assegura o uso de todas as suas interfaces no código-fonte.[1]

Validade de construção. O script Boa recuperou apenas interfaces/bibliotecas que foram importadas no código-fonte, e algumas dessas importações podem ser classes do próprio projeto. Além disso, pode-se ter interfaces que não foram utilizadas (*warning*), e projetos que copiaram o código-fonte da biblioteca.

6. Trabalhos Relacionados

Existem estudos que concentram-se na evolução e uso de interfaces e bibliotecas [Hora and Valente 2015, Mileva et al. 2010, McDonnell et al. 2013]. Adicionalmente alguns trabalhos relatam o uso indevido de interfaces por sistemas clientes, como por exemplo, a má prática de programação relacionada ao uso de interfaces internas [Businge et al. 2015, Mastrangelo et al. 2015]. Outros estudos concentram-se na promoção dessas interfaces internas para o público [Hora et al. 2016].

Um estudo relacionado ao uso de interfaces internas da biblioteca Eclipse serviu de inspiração para esse trabalho [Businge et al. 2015, Businge et al. 2013]. O mesmo analisa *releases* de 512 *plug-ins* Eclipse do repositório SourceForge, com o objetivo de investigar motivos que levam ao uso (ou desuso) de interfaces internas. Os autores também realizam entrevistas com alguns clientes. Outro estudo nessa área concentra-se no uso da interface interna sun.misc.Unsafe provida pelo JDK [Mastrangelo et al. 2015]. O artigo identifica 14 razões que levam clientes a usarem essa interface e analisa cerca de 86 mil arquivos Java. Ao contrário dos trabalhos mencionados, que concentram-se em bibliotecas específicas, este trabalho analisa o uso de interfaces internas na linguagem Java, e o conhecimento de clientes e provedores sobre o uso dessas interfaces. Além disso o trabalho faz uso de um *dataset* maior, com aproximadamente 260 mil projetos, 131 milhões de APIs e 16 milhões de arquivos.

7. Conclusões

Este trabalho apresentou um estudo empírico em larga escala sobre a utilização de interfaces internas em 260 mil projetos Java. Na primeira questão de pesquisa, investigou-se a frequência com que clientes usam interfaces internas. Na segunda questão de pesquisa

[1]Por exemplo, importar a biblioteca java.util.* não implica no uso das suas interfaces ArrayList e List.

analisou-se a distribuição do uso dessas interfaces por biblioteca. Além disso, realizou-se através de uma análise qualitativa, uma consulta aos clientes e provedores de interfaces internas. Apresenta-se a seguir os principais resultados desse estudo:

QP #1. Verificou-se que algumas bibliotecas tem mais de 20% dos seus clientes usando interfaces internas.

QP #2. Verificou-se que algumas interfaces internas das bibliotecas atraem mais clientes. Além disso, observou-se que as 50 interfaces internas mais usadas pertencem as bibliotecas JDK, Action Bar Sherlock e Android.

Entrevista. Na consulta com os provedores de algumas bibliotecas, observou-se que o uso de interfaces internas por sistemas clientes é conhecido, sendo considerada uma má prática de programação pelos desenvolvedores. Entre os clientes consultados, observou-se que o uso de interfaces internas é motivado pela falta de conhecimento da biblioteca.

Como trabalho futuro, pretende-se analisar o total de interfaces internas de cada biblioteca, e quantas dessas interfaces são usadas por clientes.

Agradecimentos: Esta pesquisa é financiada pela FAPEMIG e pelo CNPq. Agradecemos também aos desenvolvedores GitHub consultados.

Referências

Businge, J., Serebrenik, A., and van den Brand, M. G. J. (2013). Analyzing the eclipse API usage: Putting the developer in the loop. In *17th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 37–46.

Businge, J., Serebrenik, A., and van den Brand, M. G. J. (2015). Eclipse API usage: the good and the bad. In *Software Quality Journal*, pages 107–141.

Dyer, R., Nguyen, H. A., Rajan, H., and Nguyen, T. N. (2013). Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *35th International Conference on Software Engineering*, pages 422–431.

Hora, A., Valent, M. T., Robbes, R., and Anquetil, N. (2016). When should internal interfaces be promoted to public? In *24th International Symposium on the Foundations of Software Engineering (FSE)*, pages 1–12.

Hora, A. and Valente, M. T. (2015). apiwave: Keeping track of API popularity and migration. In *31st IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 321–323.

Mastrangelo, L., Ponzanelli, L., Mocci, A., Lanza, M., Hauswirth, M., and Nystrom, N. (2015). Use at your own risk: the Java unsafe API in the wild. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 695–710. ACM.

McDonnell, T., Ray, B., and Kim, M. (2013). An empirical study of API stability and adoption in the android ecosystem. In *International Conference on Software Maintenance*, pages 70–79.

Mileva, Y. M., Dallmeier, V., and Zeller, A. (2010). Mining API popularity. In *International Academic and Industrial Conference on Testing - Practice and Research Techniques*, pages 173–180.

Revelando Extensões de Conceitos no Código Fonte

Assis S. Vieira[1], **Bruno C. da Silva**[2], **Cláudio Sant'Anna**[1]

[1]Instituto de Matemática – Universidade Federal da Bahia (UFBA)
Cep 40170-110 – Salvador, BA – Brasil

[2]Universidade Salvador (UNIFACS)
Cep 41770-235 – Salvador, BA – Brasil

assis.sv@gmail.com, bruno.carreiro@pro.unifacs.br, santanna@dcc.ufba.br

Resumo. *Na modificação de software, a Localização de Conceitos é a etapa em que o desenvolvedor localiza no código-fonte os trechos de código que devem ser modificados. Nesta etapa, um método frequentemente usado pelos desenvolvedores é o CLDS – Concept Location by Dependence Search. As atuais técnicas que se baseiam neste método deixam a cargo do desenvolvedor a missão de identificar e delimitar os fluxos de controle e de dados presentes em um software. Tal missão, consome tempo e esforço do desenvolvedor. Para tratar este problema, desenvolvemos parcialmente um modelo heurístico que identifica os diferentes fluxos e seus elementos mais significativos. Com base neste modelo, propomos uma nova técnica para o método CLDS.*

1. Introdução

A localização de conceitos faz parte do processo de compreensão de software [Biggerstaff et al. 1993, Rajlich and Wilde 2002]. Esta tarefa é responsável em média por 62% do esforço e por 70% do tempo dos engenheiros durante uma atividade de modificação de software [Soh et al. 2013, Minelli et al. 2015].

Nas últimas décadas, pesquisadores têm desenvolvido modelos cognitivos para tentar explicar o processo de compreensão de software realizado por programadores [Hansen et al. 2012, Mayrhauser and Vans 1995]. Nesse contexto, Rajlich introduz na compreensão de software a noção de *conceito*, utilizado pela linguística e outras áreas do conhecimento [Rajlich and Wilde 2002, Rajlich 2009, Rajlich 2011]. Cada conceito é constituído por *nome*, *intenção* e *extensão*: o nome identifica o conceito; a intenção é uma sugestão de significado; e a extensão de um conceito é qualquer coisa no mundo real que reflete a intenção deste conceito. Para os autores, no contexto da engenharia de software as extensões no código-fonte podem ganhar diferentes formas. Em um código orientado a objeto, por exemplo, uma extensão pode ganhar a forma de um pacote, classe, método ou um conjunto destes. Assim, eles apontam que a localização de conceitos pode ser vista como um processo de localizar no código-fonte a extensão dos conceitos que se tem interesse.

Durante a leitura do código-fonte não há nada que indique ao programador que ele está passando de um conceito para outro, a não ser os nomes usados nos identificadores de classes e métodos. Entretanto, nem sempre os nomes usados são significativos o suficiente para tal tarefa [Eaddy et al. 2007]. Rajlich [2009] relata que um dos fatores para a dificuldade na recuperação do significado (intenção), considerando apenas um nome, é a

natural relação de muitos-para-muitos entre nome, intenção e extensão que constituem os conceitos. Exemplificando: dado isoladamente o identificador do método *dividir()*, não é possível distinguir se sua intenção refere-se a "dividir uma estrutura de dados para um algoritmo de busca" ou "inserir um divisor de ícones na barra de ferramentas de interface gráfica".

Portanto, para recuperar apropriadamente a intenção de um conceito é preciso antes identificar o contexto no qual ele está inserido. Considerando que o contexto de muitos conceitos que constituem um software encontra-se no próprio código-fonte, temos a seguinte questão de pesquisa: **como identificar os limites entre a extensão de um determinado conceito e as extensões dos demais conceitos que compreendem seu contexto?**

É fundamental conhecer os limites das extensões que consistem os conceitos que devem ser modificados, caso contrário, a modificação pode invadir inapropriadamente as extensões dos conceitos alheios à necessidade de modificação. Além disso, na localização de conceitos, conhecer tais limites é importante para concluir se uma extensão específica reflete de fato as propriedades contidas na intenção, ou se a localização de parte dessa extensão é apenas uma coincidência, tal como foi dado no exemplo do método *dividir()*. Considera-se que estes são alguns dos fatores que fazem os desenvolvedores se esforçarem tanto na recuperação do propósito dos elementos de código, elevando assim os custos de manutenção [Sillito et al. 2005, LaToza et al. 2006, Roehm et al. 2012].

Pesquisas sobre o processo cognitivo realizado por programadores durante atividades de compreensão de software apontam que, durante a leitura de um código-fonte desconhecido os programadores constantemente tentam compreendê-lo identificando os fluxos de controle e de dados [Mayrhauser and Vans 1995]. A nossa hipótese é de que os significados dos fluxos são mais facilmente alcançados pela leitura dos elementos de código que dão início a esses fluxos. Tais elementos tendem estar mais próximos do contexto do domínio da aplicação e dos termos conhecidos pelo desenvolvedor do que os demais elementos. Portanto, os demais elementos que compõe os fluxos podem ser inicialmente ignorados pelo desenvolvedor até que, em um momento posterior, a compreensão destes elementos periféricos se torne necessária.

Um método que apoia os desenvolvedores durante a leitura de um código-fonte é o CLDS (*Concept Location by Dependence Search*) [Chen and Rajlich 2000, Chen and Rajlich 2010]. As ferramentas que dão suporte a este método podem ser encontradas em ambientes de desenvolvimento, tal como o Eclipse[1]. Uma técnica específica para o CLDS também já foi proposta, o Ripples [Chen and Rajich 2001]. Contudo, não foi encontrada nenhuma abordagem que informa qual elemento origina os fluxos nos quais um específico elemento participa. Deste modo, as atuais técnicas deixam a cargo do desenvolvedor a missão de conhecer os limites dos diferentes fluxos de um código-fonte.

Neste cenário, propomos uma nova técnica que possibilita ao desenvolvedor conhecer os limites entre os fluxos de controle presentes no código-fonte, priorizando e destacando os elementos mais significativos de cada fluxo. Para verificar a viabilidade de tal proposta, apresentamos também como tal técnica pode ser aplicada em um sistema real.

[1] Ambiente de Desenvolvimento Integrado usado pela comunidade Java: http://www.eclipse.org.

Assim, para melhor compreensão, o presente artigo segue estruturado em quatro seções. A Seção 2 apresenta métodos e técnicas existentes de localização de conceitos. A Seção 3 descreve em alto nível a técnica proposta, destacando a diferença entre ela e as atuais técnicas. A Seção 4 demonstra a aplicação deste modelo em um software específico, o JHotDraw 7.6[2]. Por fim, na Seção 5, resume-se o que tem sido feito até o momento e os próximos passos para a continuidade da pesquisa.

2. Métodos e Técnicas de Localização de Conceitos

Um método de localização de conceitos amplamente conhecido é o *Concept Location by Dependence Search* (CLDS) [Chen and Rajlich 2000, Chen and Rajlich 2010]. Neste método os desenvolvedores leem o código-fonte e seguem as dependências estáticas que julgarem relevantes. Os ambientes de desenvolvimento têm fornecido recursos para auxiliar o desenvolvedor nesse sentido, tal como a funcionalidade de *Call Hierarchy* do Eclipse, além de *Plugins* como o Ripples [Chen and Rajich 2001]. Entretanto, essas iniciativas ainda não conseguem resolver o problema de compreensão das extensões dos conceitos no código-fonte.

Adicionalmente, outras técnicas baseadas em análise estática também foram propostas, porém não há evidências de que elas tenham mitigado o problema de compreensão de conceitos no código-fonte. No estudo realizado por De Alwis et al. [2007], os autores reportaram que durante as tarefas de modificação de software, não houve diferença significativa nos esforços dos desenvolvedores ao usar três ferramentas (JQuery, Ferret, Suade) que apoiam diferentes técnicas de localização de conceitos e as ferramentas convencionais disponibilizadas pelo ambiente de desenvolvimento Eclipse. Entre outras hipóteses, os autores explicaram que esse resultado pode ter ocorrido porque as técnicas testadas simplesmente re-empacotavam as informações, disponibilizadas pela IDE (*Integrated Development Environment*), em novas formas em vez de fornecer novos meios de resolver os problemas de compreensão. Esta hipótese converge com as observações de Sillito et al. [2005] e Ko et al. [2006] em outros estudos exploratórios em que os participantes utilizavam uma IDE amplamente conhecida.

No estudo realizado por Sillito et al. [2005], os autores observaram que, um modelo integrado que colocasse em evidência as relações entre as entidades identificadas como relevantes seria de grande valia. Os autores ainda destacaram que, mesmo quando lápis e papel foram usados, a integração ainda apresentou-se como um importante e difícil desafio. Os autores também puderam observar que as re-visitas de entidades e relações foram comuns, mas nem sempre simples. De fato, 57% das visitas observadas para os elementos de código foram re-visitas e esta proporção aumentava ao longo do tempo.

No estudo realizado por Ko et al. [2006], os pesquisadores relataram que as ferramentas do Eclipse utilizadas para a navegação do código causaram uma sobrecarga nos desenvolvedores. De acordo com a razão apresentada, sempre que uma dependência era analisada, o resultado da dependência anterior era perdido, forçando os desenvolvedores a repetir a busca. Os autores apontaram que estes problemas levaram os desenvolvedores a gastarem em média 35% do tempo em mecanismos de navegação.

[2]Framework para aplicações gráficas escritas em Java: https://sourceforge.net/projects/jhotdraw/.

3. Uma Nova Técnica para Localização de Conceitos

Neste cenário, propomos uma nova técnica baseada no método CLDS que apoia o desenvolvedor durante a localização de conceitos no código-fonte. Nesta técnica, um elemento de código inicial, chamado de *semente*, é selecionado pelo desenvolvedor. Uma ferramenta, por sua vez, apresenta um grafo de elementos relacionados à semente selecionada. Entretanto, diferente das atuais técnicas baseadas no CLDS, a referida proposta parte da perspectiva de que cada um dos elementos de código representa pequenos conceitos que se relacionam estaticamente para compor outros conceitos mais abstratos. A premissa é de que, dado um fluxo de controle ou de dados, o conceito mais significativo deste fluxo é representado pelo elemento de código que o inicia. Os demais elementos do fluxo são portanto coadjuvantes, pois possuem importância secundária na compreensão do fluxo. Os elementos apresentados pela ferramenta são então organizados em superconceitos e subconceitos da semente selecionada.

Os superconceitos da semente são os elementos que dão origem aos fluxos em que a semente participa. A semente tem um único superconceito quando ela é referenciada por um único elemento, ou quando ela é referenciada por dois ou mais elementos e estes forem referenciados, direta ou indiretamente, por um único elemento em comum. A semente pode ter dois ou mais superconceitos quando os elementos que a referenciam não possuem um elemento em comum que os referencie. Logo, os elementos que originam os distintos fluxos são os superconceitos da semente. Os subconceitos da semente são todos os elementos que, para serem alcançados, seja direta ou indiretamente, é necessário passar pela semente. Isso significa que a semente é o superconceito destes elementos.

A Figura 1 ilustra um possível grafo aplicando essa abordagem em um sistema real. Neste cenário, o desenvolvedor está tentando compreender o método *ApplicationModel.createActionMap(Application, View)* da biblioteca gráfica JHotDraw. Ao selecionar este método como semente, a técnica apresenta o grafo ilustrado pela referida figura.

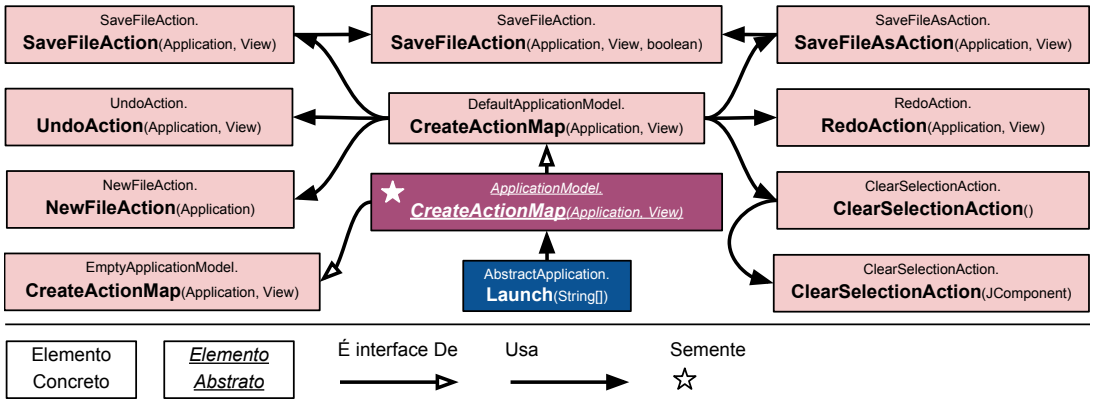


Figura 1. Grafo de conceitos apresentado ao desenvolvedor.

Neste grafo, todos os vértices são métodos. O elemento com uma estrela é a semente selecionada. O elemento logo abaixo da semente é o seu superconceito. Os demais são os subconceitos da semente. Este superconceito é o elemento que origina todo o fluxo no qual a semente participa. Coincidentemente, ele referencia a semente de modo indireto, através de outros 7 elementos que não são apresentados neste grafo (veja a Seção 4 para mais detalhes). Neste primeiro momento é possível que o desenvolvedor

conclua que a semente selecionada participa exclusivamente da inicialização da aplicação gráfica, uma vez que o identificador do superconceito - *AbstractApplication.launch(..)* - remete a isso. Porém, na utilização das demais técnicas da CLDS, no mínimo, vinte e uma dependências devem ser visitadas para chegar a esta mesma conclusão, interagindo com diversos mecanismos de interface da IDE. Estas dependências podem ser conferidas em detalhes na Seção 4.

Para implementar a técnica proposta, iniciamos o desenvolvimento de um modelo heurístico com um conjunto de regras que definem a delimitação entre diferentes extensões presentes no código-fonte. Considerando as restrições de espaço, apresenta-se a seguir algumas das regras desenvolvidas até o presente momento, omitindo a descrição detalhada do procedimento de aplicação. As regras omitidas referem-se principalmente as questões como dependências circulares. Contudo, as regras apresentadas são suficientes para compreender os resultados obtidos na aplicação deste modelo no *JHotDraw 7.6* (Seção 4).

1. **Regra da Composição:** a extensão de um conceito representado por um elemento de código *A* se estende unicamente aos elementos que são usados exclusivamente para defini-lo. Os demais elementos de código referenciados por *A*, porém de modo não exclusivo, não compõem a extensão do conceito composto por *A*. Por exemplo, se um método *m1()* compõe a extensão de um conceito C1, todos os elementos de código referenciados de modo exclusivo por *m1()* também farão parte da extensão do conceito C1. Os demais elementos referenciados por *m1()*, mas que também são referenciados por outros elementos de código não fazem parte da sua extensão.
2. **Regra da Referência Implícita:** quando um método concreto é invocado polimorficamente, o fluxo de controle passa do método chamador para o método abstrato, e este, por sua vez, decide então qual das suas especializações executar. Deste modo, em vez de considerar a relação direta do método chamador para o método concreto, considera-se primeiramente que, o método chamador faz referência ao método abstrato, e este, então, a uma referência implícita ao específico método concreto, como se o método abstrato possuísse uma implementação e nela tivesse referências de todas as suas especializações (métodos concretos). Assim, em tempo de execução, o método abstrato decide pelo método concreto que deve ser executado.

Algumas técnicas de localização de conceitos utilizam outras abordagens para organizar o código-fonte em superconceitos e subconceitos [Dit et al. 2013]. Tais técnicas se fundamentam na AFC - Análise Formal de Conceitos. A AFC é um modelo matemático que formaliza a noção de conceito através da teoria dos conjuntos [Wille 1992]. Diferente do modelo que está sendo proposto, o grafo de conceitos resultante da AFC tende a ser mais complexo do que o grafo de dependências estáticas fornecido como entrada, dificultando a compreensão dos conceitos presentes no código-fonte [Anquetil 2000].

4. Aplicação da Técnica no JHotDraw

Para verificar a viabilidade do modelo foi aplicado manualmente as regras mencionadas em parte do código-fonte do *JHotDraw 7.6*, que é um projeto real de código-fonte aberto. No escopo dessa avaliação, foram considerados apenas os elementos

de código do tipo método. A semente selecionada foi o método *ApplicationModel.createActionMap(Application, View)*. A escolha desta semente teve como critério métodos em que fosse possível exercitar as regras apresentadas. Ao examinar a semente, aplicou-se a *Regra da Referência Implícita*, uma vez que este era um método abstrato com duas especializações (métodos concretos). Em seguida, foi descoberto que este método abstrato era a única referência para seus dois métodos concretos. Logo, aplicou-se a *Regra de Composição* nos três elementos, resultando na relação superconceito e subconceito, respectivamente entre o método abstrato e suas especializações. Deste modo, foi dada continuidade a exploração do código até que todos os elementos da extensão do conceito, representado pela semente, foram encontrados. O resultado desta investigação pode ser conferido na Figura 2. O elemento marcado com uma estrela é a semente selecionada, enquanto os demais são os subconceitos da semente.

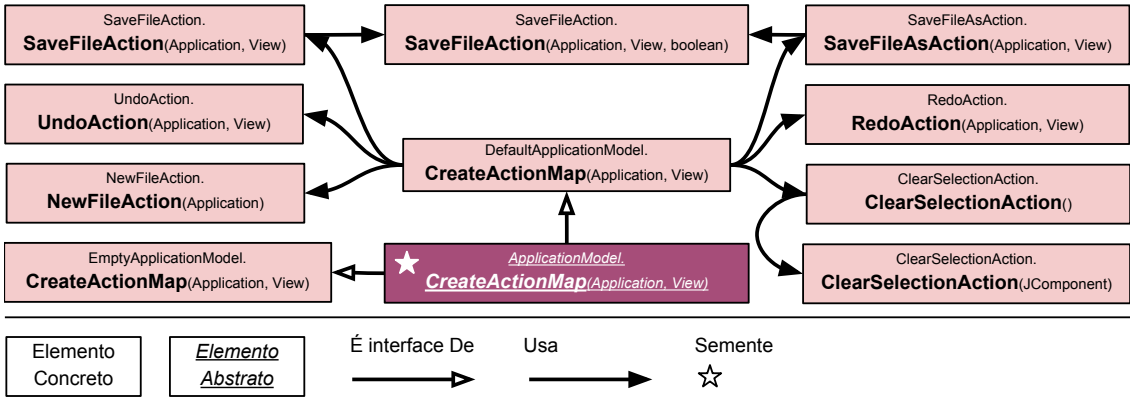


Figura 2. Todos os subconceitos da semente.

Por outro lado, ao navegar pelos métodos que referenciavam diretamente a semente, foram descobertos outros sete métodos, conforme ilustra a Figura 3. Com base na Regra de Composição, pôde-se concluir que a semente e o conjunto dos novos métodos descobertos delimitavam conceitos distintos. Em outras palavras, apesar de ter sido referenciada, a semente não era subconceito de nenhum destes sete elementos. Neste sentido, deu-se prosseguimento a exploração do código até que o método *AbstractApplication.Launch(String[])* foi descoberto sendo o superconceito da semente. Este método originava todos os fluxos em que a semente participava. O resultado desta inspeção pode ser conferido no grafo da Figura 3. O elemento com uma estrela representa a semente, enquanto que o elemento no canto inferior direito do grafo representa o superconceito da semente. Os demais elementos interligam a semente ao seu superconceito. Esses elementos, incluindo a semente, são alguns dos subconceitos do referido superconceito.

5. Conclusão

Embora diversas técnicas de localização de conceitos tenham sido propostas nas últimas décadas, o processo de localizar a extensão de conceitos no código-fonte ainda parece consumir mais da metade do esforço e tempo dos desenvolvedores em tarefas de modificação de código-fonte. Trabalhos sobre modelos cognitivos apontam que desenvolvedores compreendem software desconhecido através da identificação dos fluxos de controle e de dados no código-fonte. Baseado no ponto de vista da programação como um processo de tradução, este trabalho assume a premissa de que os elementos que iniciam um fluxo de

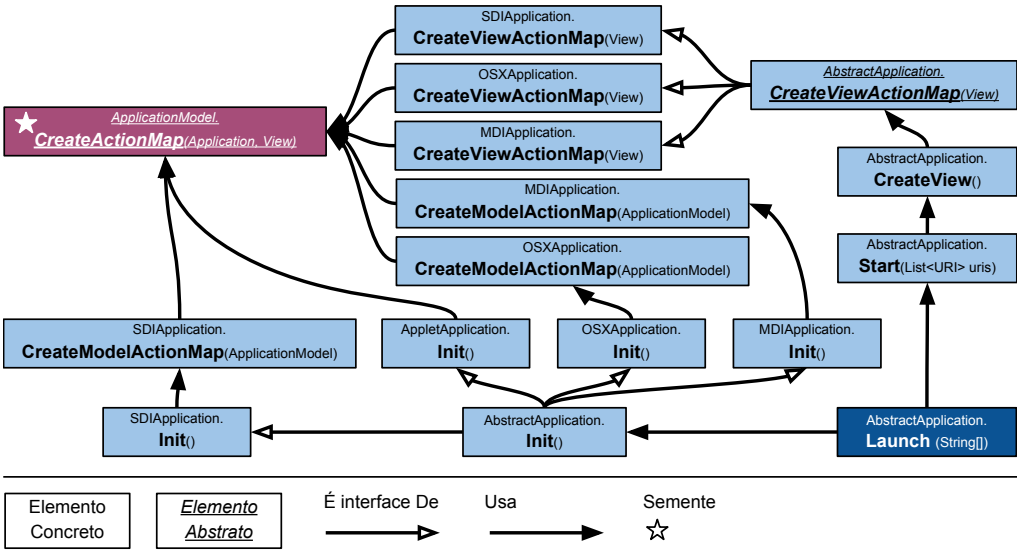


Figura 3. Os elementos que conectam a semente ao seu superconceito.

controle são mais significativos para o desenvolvedor do que os demais elementos deste fluxo. Desta forma, propõe-se uma nova técnica de localização de conceitos baseada no método CLDS. Nesta nova abordagem, o papel do desenvolvedor é selecionar uma semente. Enquanto que, diferente das demais técnicas, nesta proposta o papel da técnica é apresentar os subconceitos e superconceitos da semente. Nesta apresentação, a técnica destaca os limites entre a extensão do conceito representado pela semente e as extensões dos conceitos que compõe o contexto da semente. Para delimitar dos conceitos presentes no código-fonte, iniciamos o desenvolvimento de um modelo heurístico. Apesar de estar em fase inicial, o modelo demonstrou-se viável após sua aplicação em partes do JHotDraw.

Assim, as próximas atividades para o desenvolvimento dessa pesquisa consistem em: (i) concluir o modelo heurístico que identifica os fluxos de controle no código-fonte; (ii) concluir o desenvolvimento da ferramenta de apoio à técnica proposta; e (iii) avaliar a aplicabilidade da técnica em um experimento sobre localização de conceitos.

Agradecimentos. Esse trabalho foi apoiado pela FAPESB e pelo CNPq por meio do Instituto Nacional de Ciência e Tecnologia para Engenharia de Software (processo 573964/2008–4) e Projeto Universal (processo 486662/2013-6).

Referências

Anquetil, N. (2000). A comparison of graphs of concept for reverse engineering. In *Program Comprehension, 2000. Proceedings. IWPC 2000. 8th International Workshop on*, pages 231–240.

Biggerstaff, T. J., Mitbander, B. G., and Webster, D. (1993). The concept assignment problem in program understanding. In *Proceedings of the 15th International Conference on Software Engineering*, ICSE '93, pages 482–498, Los Alamitos, CA, USA. IEEE Computer Society Press.

Chen, K. and Rajich, V. (2001). Ripples: tool for change in legacy software. In *Software Maintenance, 2001. Proceedings. IEEE International Conference on*, pages 230–239.

Chen, K. and Rajlich, V. (2000). Case study of feature location using dependence graph. In *Program Comprehension, 2000. Proceedings. IWPC 2000. 8th International Workshop on*, pages 241–247.

Chen, K. and Rajlich, V. (2010). Case study of feature location using dependence graph, after 10 years. In *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*, pages 1–3.

Dit, B., Revelle, M., Gethers, M., and Poshyvanyk, D. (2013). Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95.

Eaddy, M., Aho, A., and Murphy, G. C. (2007). Identifying, assigning, and quantifying crosscutting concerns. In *Assessment of Contemporary Modularization Techniques, 2007. ICSE Workshops ACoM '07. First International Workshop on*, page 2.

Hansen, M. E., Lumsdaine, A., and Goldstone, R. L. (2012). Cognitive architectures: A way forward for the psychology of programming. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2012, pages 27–38, New York, NY, USA. ACM.

LaToza, T. D., Venolia, G., and DeLine, R. (2006). Maintaining mental models: A study of developer work habits. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 492–501, New York, NY, USA. ACM.

Mayrhauser, A. V. and Vans, A. M. (1995). Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55.

Minelli, R., Mocci, A., and Lanza, M. (2015). I know what you did last summer - an investigation of how developers spend their time. In *2015 IEEE 23rd International Conference on Program Comprehension*, pages 25–35.

Rajlich, V. (2009). Intensions are a key to program comprehension. In *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on*, pages 1–9.

Rajlich, V. (2011). *Software Engineering: The Current Practice*. Chapman & Hall/CRC, 1st edition.

Rajlich, V. and Wilde, N. (2002). The role of concepts in program comprehension. In *Program Comprehension, 2002. Proceedings. 10th International Workshop on*, pages 271–278.

Roehm, T., Tiarks, R., Koschke, R., and Maalej, W. (2012). How do professional developers comprehend software? In *2012 34th International Conference on Software Engineering (ICSE)*, pages 255–265.

Sillito, J., Voider, K. D., Fisher, B., and Murphy, G. (2005). Managing software change tasks: an exploratory study. In *2005 International Symposium on Empirical Software Engineering, 2005.*, pages 10 pp.–.

Soh, Z., Khomh, F., Guéhéneuc, Y. G., and Antoniol, G. (2013). Towards understanding how developers spend their effort during maintenance activities. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 152–161.

Wille, R. (1992). Concept lattices and conceptual knowledge systems. *Computers Mathematics with Applications*, 23(6):493 – 515.

Uma análise da associação de co-ocorrência de anomalias de código com métricas estruturais

Carlos E. C. Dantas, Marcelo de A. Maia

Faculdade de Computação - FACOM
Universidade Federal de Uberlândia (UFU)

carloseduardodantas@iftm.edu.br,marcelo.maia@ufu.br

Abstract. *Source code anomalies (code bad smells) were characterized as symptoms of poor programming practices that should be either avoided or eliminated by refactoring. Several recent studies have recognized that the impact caused by these source code anomalies does not have the same severity. Although these studies have been conducted in order to understand the dynamics of the life cycle of the source code anomalies, there is little knowledge about how the source code anomalies evolve in the affected entities, especially regarding the question of how a code anomaly in an entity may induce the co-occurrence of other code anomaly during its evolution. This work studied the evolution of 5 systems, analyzing 12 types of anomalies that were introduced during the life cycle of their respective developments. We observed that 6 inter-relations anomalies between the anomalies analyzed were more prominent, and Long Method has great influence in all founded anomalies. With this anomaly, classes have highter coupling and lower cohesion, besides being more likely to introduce new types of anomalies.*

Resumo. *Anomalias de código (code bad smells) foram caracterizadas como sintomas de más práticas de programação que deveriam ser evitadas ou então eliminadas por meio de refatoração. Vários estudos recentes já reconhecem que os danos causados pelas anomalias de código não têm a mesma severidade. Apesar de esses estudos terem sido conduzidos com o intuito de entender a dinâmica do ciclo de vida das anomalias de código, ainda existe pouco conhecimento a respeito de como tais anomalias evoluem nas entidades afetadas, especialmente em relação à questão de como uma anomalia de código em uma entidade pode induzir co-ocorrência de outras anomalias durante sua evolução. Este trabalho estudou a evolução de código de 5 sistemas, analisando 12 tipos de anomalias que foram introduzidas no decorrer das suas respectivas evoluções. Observou-se que 6 inter-relações entre as anomalias analisadas foram mais proeminentes, com a anomalia Long Method exercendo influência em todas as inter-relações encontradas. Com a presença desta anomalia, foi constatado que classes possuem maior acoplamento e menor coesão, além de estarem mais propensos a introduzir novos tipos de anomalias.*

1. Introdução

Para atender os requisitos solicitados no menor tempo e custo possíveis, sistemas que adotam orientação a objetos (OO) deveriam seguir princípios básicos de *design*, como

polimorfismo, encapsulamento, abstração de dados e modularidade. Contudo, os desenvolvedores podem violar alguns desses princípios por vários motivos, seja pela urgência na entrega do sistema [Lavallée and Robillard 2015], pela adoção de uma solução deficiente no projeto das classes do sistema, ou simplesmente pelo desconhecimento quanto às boas práticas em desenvolvimento de sistemas OO [Palomba et al. 2014]. Estas violações resultam no surgimento de algumas anomalias nas classes desses sistemas. Várias dessas anomalias foram catalogadas por [Fowler et al. 1999], onde foram apelidadas de *code bad smells* (ou *code smells*). Em linhas gerais, a presença de anomalias nas classes de um sistema pode apresentar danos como o decremento da compreensibilidade [Abbes et al. 2011] e manutenibilidade [Yamashita and Moonen 2012]. Além disso, tais sistemas podem se tornar mais propensos a falhas, e sofrer mais mudanças no decorrer do tempo [Khomh et al. 2012].

Enquanto várias pesquisas têm analisado o comportamento das anomalias de maneira uniforme, alguns estudos têm buscado o discernimento sobre a severidade que cada tipo de anomalia exerce sobre as classes. Por exemplo, [Sjøberg et al. 2013] mostraram que nem toda anomalia é igualmente maléfica, pois algumas não são propensas a afetar diretamente a manutebilidade das classes de um sistema. Já [Lozano et al. 2007] apresentaram que certas anomalias podem não apenas dificultar a manutenibilidade, mas também influenciar no surgimento de novas anomalias, além de permanecerem por um longo período de tempo nas classes. Por fim, [Yamashita and Moonen 2013] mostraram que grupos específicos de anomalias possuem a tendência de estarem presentes nas mesmas classes, e que juntas podem multiplicar os danos sobre as mesmas. Com isso, cada grupo representa um conjunto de danos para as classes de um sistema.

Diante dos trabalhos citados, necessita-se de estudos sobre como os diferentes tipos de anomalias podem evoluir no decorrer dos *commits* de uma classe, e com isso, influenciar no surgimento de outras anomalias. Além disso, esta inter-relação entre as anomalias podem multiplicar os efeitos danosos sobre as classes. Assim, alguns tipos de anomalias poderiam ser priorizados para remoção em detrimento de outros, visando evitar a multiplicação dos efeitos danosos sobre as classes de um sistema, como mencionado por [Yamashita and Moonen 2012]. O cenário ideal seria que, ao serem detectadas, todas as anomalias fossem removidas. Contudo, aplicar operações de refatoração possuem um certo custo, além dos riscos de novos defeitos serem introduzidos no sistema. Além disso, outros trabalhos já mostraram que nem sempre o desenvolvedor está interessado em remover anomalias, o que eleva a importância da priorização mencionada [Lavallée and Robillard 2015].

O restante deste trabalho está organizado como segue. A metodologia do estudo, com os objetivos, a descrição sobre a coleta e análise dos dados são descritas na Seção 2. Os resultados do estudo são apresentados na Seção 3. Os trabalhos relacionados são citados na Seção 4. Na Seção 5 são apresentadas as limitações deste trabalho, e as conclusões e os trabalhos futuros são expostos na Seção 6.

2. Metodologia do Estudo

Os objetivos deste trabalho são: 1) avaliar a evolução dos tipos de anomalias sobre as classes de sistemas com código-fonte aberto, descobrindo possíveis inter-relações entre as mesmas sobre um conjunto de classes, e 2) analisar os resultados de medições envolvendo

indicadores de coesão e acoplamento para as classes afetadas por tais inter-relações, e compará-los com classes não afetadas por tais inter-relações. Com isso, será possível avaliar se determinadas inter-relações precisam de priorização para remoção, evitando a multiplicação dos seus efeitos danosos.

2.1. Coleta de Dados

Os sistemas utilizados neste estudo foram: *Apache Ant* [1], *JHotDraw* [2], *Apache Log4j* [3], *Lucene-Solr* [4] e *Apache Tomcat* [5]. Todos estes foram extraídos do *Github* [6]. O *download* dos sistemas e a extração dos *commits* de cada classe foram realizados através de comandos próprios do controle de versão *git*, como *git clone*, *git log* e *git show* . A **Tabela 1** apresenta as características de cada sistema utilizado neste trabalho, onde são mostrados a quantidade de classes analisadas de cada sistema (foram desconsideradas classes que são destinadas a exemplos e testes unitários, pois estas não possuem funcionalidades aderentes ao propósito de cada sistema). Além disso, também é apresentado a quantidade de classes onde algum um tipo de anomalia tenha sido detectado em pelo menos um *commit*. Por fim, são apresentados a quantidade de *commits* que cada sistema possui, e o intervalo de tempo entre o primeiro e o último *commit* de cada sistema. O critério para escolha dos sistemas ocorreu em função da relevância de cada um para a comunidade de desenvolvedores Java, e pela variação quanto ao propósito de cada sistema (arquiteturas distintas), quantidade de classes, anomalias e *commits*.

Tabela 1. Sistemas utilizados para análise das anomalias de código

Sistema	Classes	Classes c/ Anomalias	*Commits*	Intervalo dos *Commits*
Ant	2.778	1.315	13.342	Jan/2000 a Jun/2016
JHotDraw	1.691	882	747	Out/2000 a Dez/2015
Log4j	1.032	595	3.275	Nov/2000 a Jun/2015
Lucene-Solr	7.163	4.102	24.958	Set/2001 a Abr/2016
Tomcat	3.320	1.933	17.246	Mar/2006 a Jun/2016
Total	15.984	8.827	59.568	-

Para detectar as anomalias sobre cada *commit* de cada classe, foi utilizada a ferramenta *DECOR* [Moha et al. 2010], com suas métricas já estabelecidas por padrão. A **Tabela 2** apresenta a quantidade de classes em que pelo menos um *commit* apresentou algum dos diferentes tipos de anomalias detectados pelo *DECOR*. Também é mostrado o percentual de ocorrências em que cada tipo de anomalia incide sobre o total de classes de cada sistema. Observa-se que as anomalias mais frequentes sobre as classes foram *Long Method, Lazy Class, Long Parameter List* e *Complex Class*. Além disso, em virtude das diferenças arquiteturais entre os sistemas, algumas anomalias são mais frequentes em alguns sistemas em detrimento de outros. Por exemplo, a anomalia *Anti Singleton* ocorre em 11,3% das classes do sistema *Log4j*, enquanto que em apenas 0,8% no sistema *Ant*.

[1] https://github.com/apache/ant.git
[2] https://github.com/wumpz/jhotdraw.git
[3] https://github.com/apache/log4j.git
[4] https://github.com/apache/lucene-solr.git
[5] https://github.com/apache/tomcat.git
[6] http://www.github.com

Em contrapartida, a anomalia *Long Method* ocorre com proporções semelhantes em todos os sistemas, indicando que, independente do propósito do sistema, é comum ocorrer de métodos possuírem uma quantidade elevada de linhas de código.

Tabela 2. Quantidade de classes que apresentaram cada tipo de anomalia detectada pelo DECOR [Moha et al. 2010]

Anomalia	*Ant*	*JHotDraw*	*Log4j*	*Lucene-Solr*	*Tomcat*
Anti Singleton(AS)	23(0,8%)	35(2,0%)	117(11,3%)	311(4,3%)	279(8,40%)
Base Class S. Be Abstract(BC)	13(0,4%)	5(0,2%)	0(0%)	23(0,3%)	8(0,2%)
Class Data S. Be Private(CD)	31(1,1%)	33(1,9%)	24(2,3%)	418(5,8%)	103(3,1%)
Complex Class(CC)	298(10,7%)	299(17,6%)	143(13,8%)	1.267(17,6%)	500(15,0%)
Large Class(LGC)	4(0,1%)	0(0%)	4(0,3%)	24(0,33%)	39(1,1%)
Lazy Class(LZC)	377(13,5%)	168(9,9%)	162(15,6%)	1.046(14,6%)	968(29,1%)
Long Method(LM)	916(32,9%)	564(33,3%)	387(37,5%)	2.788(38,7%)	1.108(33,3%)
Long Parameter List(LPL)	341(12,2%)	251(14,8%)	137(13,2%)	997(13,9%)	318(9,5%)
Many Field Attributes(MFA)	0(0%)	0(0%)	0(0%)	14(0,1%)	4(0,1%)
Refused Parent Bequest(RPB)	15(0,5%)	0(0%)	9(0,8%)	51(0,7%)	11(0,3%)
Spaghetti Code(SC)	2(0,0%)	10(0,5%)	15(1,4%)	46(0,6%)	37(1,1%)
Speculative Generality(SG)	0(0%)	1(0,0%)	0(0%)	20(0,2%)	0(0%)

2.2. Análise de Dados

Para descobrir quais relações são mais frequentes entre os diferentes tipos de anomalias, foram utilizadas regras de associação aplicando o algoritmo *APRIORI* [Agrawal and Srikant 1994]. Os limiares empregados foram 3% de suporte sobre as classes que apresentaram algum tipo de anomalia, e 80% de confiança sobre cada regras encontrada. O critério para a escolha do percentual de suporte ocorreu em virtude da busca pelo equilíbrio entre obter a maior quantidade possível de tipos de anomalias distribuídos entre as regras encontradas, desde que cada regra possua frequência de pelo menos algumas dezenas de classes, em função de maximizar a quantidade de casos para cada regra. Por fim, o critério para o percentual de confiança se deve pelo interesse em descobrir inter-relações entre tipos de anomalias que ocorrem constantemente a partir das regras encontradas.

Após obter o conjunto de regras, foram aplicadas quatro métricas estruturais sobre as classes envolvidas em cada relação. Segue uma breve descrição de cada métrica.

- *CBO (Coupling Between Object Classes)* - quantidade de classes que estão acopladas com a classe analisada.
- *RFC (Response for Class)* - quantidade de métodos que são executados pela classe internamente quando alguma mensagem é recebida de outra classe.
- *LCOM (Lack of Cohesion)* - efetua um cálculo sobre a quantidade de métodos que não utilizam determinado atributo da classe.
- *Ca (Afferent couplings)* - quantidade de classes que utilizam alguma funcionalidade da classe analisada.

Para calcular tais métricas, foi utilizada a ferramenta *COPE* [Kakarontzas et al. 2013]. Para todas as métricas, quanto maior for o valor calculado, mais danosos são os efeitos que determinado conjunto de anomalias empregam sobre a classe analisada.

3. Resultados

A **Tabela 3** apresenta os resultados da aplicação do algoritmo *APRIORI* sobre as classes dos sistemas empregados neste trabalho. São mostradas as regras extraídas para cada sistema, com a quantidade de classes envolvidas. Além disso, também é apresentado percentual de confiança para cada regra. No total, foram encontradas 14 regras, sendo 6 regras distintas, envolvendo de 6 tipos de anomalias. Observa-se ainda que todas as regras encontradas implicam no tipo de anomalia *Long Method*. Isso indica que este tipo de anomalia exerce um papel centralizador sobre as demais anomalias com o qual esta se relaciona. A seguir, serão apresentados exemplos sobre 4 das 6 regras encontradas.

Tabela 3. Resultado do algoritmo APRIORI sobre as classes afetadas por diferentes tipos de anomalias

Sistema	Regra	Confiança
Ant	(CC,LPL)(124) → LM(113)	91%
Ant	CC(298) → LM(259)	87%
JHotDraw	(CC,LPL)(71) → LM(57)	80%
Log4j	(AS,LPL)(21) → LM(19)	90%
Log4j	(AS,CC)(25) → LM(22)	88%
Log4j	(CC,LPL)(44) → LM(36)	82%
Lucene-Solr	(CC,LPL)(438) → LM(386)	88%
Lucene-Solr	CC(1.267) → LM(1061)	84%
Lucene-Solr	(CD,CC)(160) → LM(130)	81%
Tomcat	(AS,CC)(122) → LM(111)	91%
Tomcat	(CC,LC)(77) → LM(67)	87%
Tomcat	(CC,LPL)(157) → LM(135)	86%
Tomcat	(AS,LPL)(72) → LM(61)	85%
Tomcat	CC(500) → LM(402)	80%

Para exemplificar a regra *(Complex Class → Long Method)* encontrada no sistema *Ant*, a classe `org.apache.ant.core.execution.ExecutionFrame.java` possui a anomalia *Complex Class* desde a sua criação, em virtude dos diversos fluxos anternativos existentes nos seus métodos, como por exemplo o método `parsePropertyString()`. Em virtude da complexidade da classe, alguns métodos estão propensos a possuírem várias linhas, apresentando a anomalia *Long Method*, como o método `fillinDependencyOrder()`, que possui 60 linhas de código no *commit 362903a*.

Com relação à regra *(Complex Class,Lazy Class → Long Method)* encontrada no sistema *Tomcat*, a classe `javax.servlet.http.HttpUtils.java` foi concebida para possuir métodos estáticos sem atributos, por isso foi detectada a presença da anomalia *Lazy Class*. Contudo, o método estático `parseName()` apresenta vários caminhos de execução distintos, apresentando a anomalia *Complex Class*, como consta no *commit 50a1d0b*.

Por fim, a última regra a ser exemplificada é a *(Class Data Should Be Private,Complex Class → Long Method)* encontrada no sistema *Lucene-Solr*. Esta apresenta uma situação diferente das demais regras. Como exemplo, a classe

`org.apache.lucene.queryParser.ParseException.java` apresenta atributos públicos desde a sua criação, possibilitando edições por parte das classes que estão acopladas a esta. Contudo, esta classe possui métodos complexos que fazem uso desses atributos públicos. O método `getMessage()`, apresenta a anomalia *Complex Class*, quebrando o encapsulamento da classe. E como esta classe foi crescendo a cada commit, surgiu a anomalia *Long Method*.

Para avaliar o quão danoso é a presença do tipo de anomalia *Long Method* sobre as regras, foram calculadas as métricas estruturais *CBO, LCOM, RFC e Ca* sobre as classes envolvidas em cada regra, separando pelas classes que possuem e as que não possuem o tipo de anomalia *Long Method*. Por exemplo, como foi mostrado na **Tabela 3**, a regra *(Complex Class → Long Method)* encontrada no sistema *Ant* apresentou 259 casos onde a anomalia *Long Method* surgiu, e 39 casos onde tal anomalia não foi detectada. A **Tabela 4** apresenta os resultados sobre os itens citados, e também mostra o percentual de classes que apresentam outros tipos de anomalias além dos tipos que cada regra já apresenta.

Como pode ser observado, as classes que apresentaram o tipo de anomalia *Long Method* apresentaram valores consideravelmente maiores para as métricas estruturais, ou seja, tal anomalia contribui muito mais para a baixa coesão e o alto acoplamento entre as classes. Por exemplo, a classe *org.apache.catalina.core.StandardContext* do sistema *Tomcat* apresentou os valores 50724 para *LCOM*, 75 para *CBO*, 15 para *Ca* e 663 para *RFC*. Esta classe possui dezenas de atributos para finalidades distintas, com cada método utilizando um subconjunto desses atributos. Um detalhe importante é que além das anomalias mostradas na regra, esta classe também apresentou as anomalias *Spagheti Code* e *Large Class*. O mesmo aconteceu para diversas outras classes, como mostra o percentual na **Tabela 4**. Este percentual mostra que, as classes que apresentaram o tipo de anomalia *Long Method* estão mais propensas a serem infectadas com outras anomalias, se comparadas às classes que não apresentaram tal tipo de anomalia.

Tabela 4. Métricas de acoplamento e coesão sobre as regras encontradas

Regra	Classes com *Long Method*					Classes sem *Long Method*				
	CBO	RFC	LCOM	Ca	%	CBO	RFC	LCOM	Ca	%
(CC,LPL) → LM	15,17	71,10	344,38	9,60	30,49	7,40	32,34	64,13	2,01	23,47
CC → LM	12,23	60,33	256,23	8,23	53,40	5,56	31,02	57,13	4,55	44,32
(AS,LPL) → LM	19,24	127,89	973,24	8,16	30,49	3,00	27,00	18,66	3,33	23,47
(AS,CC) → LM	17,16	119,41	1.171,64	17,52	79,78	2,00	38,00	155,00	6,00	64,28
(CD,CC) → LM	14,44	92,60	709,41	11,30	80,95	4,75	18,61	30,18	4,31	46,29
(CC,LC) → LM	15,74	90,80	1.247,90	10,11	73,07	1,34	25,23	48,30	11,17	57,89

4. Trabalhos Relacionados

Diversos estudos empíricos foram realizados para explicar os comportamentos de anomalias que surgem nas classes dos sistemas. [Yamashita and Moonen 2012] fizeram um estudo sobre o impacto das anomalias sobre a manutenibilidade dos sistemas, onde concluíram que aspectos importantes em manutenibilidade, como simplicidade, encapsulamento e duplicidade de código-fonte são degradados em função das anomalias que surgem nas classes. Este se difere desse trabalho, que analisa a relação entre as anomalias, sem analisar os efeitos negativos que estas transmitem às classes. Além disso, o trabalho citado não destaca características individuais de cada tipo de anomalia.

[Lozano et al. 2007] fez um estudo empírico sobre a evolução da anomalia *Code Clones* em cima do histórico de versões do código-fonte. O estudo observou que uma anomalia possui tendência de promover o aparecimento de outras anomalias. O trabalho proposto busca mostrar uma relação semelhante, mas destacando quais anomalias se relacionam, e estendendo a várias anomalias e não apenas ao *Code Clone*.

[Sjøberg et al. 2013] conduziram uma pesquisa que buscava verificar se as anomalias exercem influência sobre o esforço de manutenção nas classes afetadas. Concluíram que a grande maioria dessas anomalias oferecem maior esforço, exceto a anomalia *Refused Bequest*. Isto indica que anomalias não possuem a mesma gravidade sobre as classes. Neste trabalho, métricas estruturais envolvendo acoplamento e coesão são avaliadas para verificar o impacto das anomalias em tais classes.

O trabalho mais relacionado com este se trata de [Yamashita and Moonen 2013], que pesquisou sobre a inter-relação entre as anomalias, enfatizando nos efeitos que determinadas anomalias juntas podem afetar a manutenibilidade das classes. Para tal finalidade, foram analisadas 12 tipos de anomalias, com detecção de 3 problemas de manutenibilidade e 5 fatores que relacionam as anomalias. Com isso, obtém-se tendências de comportamento sobre as classes. Embora este trabalho também encontra inter-relação entre as anomalias, o objetivo consiste em explicar o comportamento de tais inter-relações sobre as classes, enquanto que este trabalho verifica o nível de deterioração das classes com tais inter-relações.

Por fim, [Tufano et al. 2015] fizeram uma análise em cerca de 500.000 *commits*, onde 9.164 desses possuíam classes afetadas com anomalias. Neste estudo, uma das conclusões foi que as classes geralmente são afetadas por anomalias logo na sua criação, ao invés das manutenções que a classe sofre no decorrer dos *commits*. Para coletar tais resultados, foi construída uma ferramenta chamada *History Miner* que utiliza a ferramenta *DECOR* [Moha et al. 2010] para detectar as anomalias. Esta metodologia possui algumas interseções com este trabalho, contudo são coletados outros indicadores como acoplamento e coesão das classes, além da utilização de regras de associação para detectar a inter-relação entre as anomalias.

5. Conclusões e Trabalhos Futuros

Em geral, as principais conclusões obtidas neste trabalho foram:

1. Existem tipos de anomalias que possuem fortes inter-relações, tendo em vista que o percentual de confiança empregado foi consideravelmente alto. Além disso, foi constatado que alguns tipos de anomalias ocorrem com maior frequência nas classes, dificultando a obtenção de regras dos demais tipos de anomalias, por não possuírem amostras suficientes para obter tais regras.
2. O tipo de anomalia *Long Method* surgiu em todas as regras encontradas, onde foi constatado que a presença deste tipo de anomalia nas classes é extremamente danoso, além de facilitar no surgimento de novos tipos de anomalias nas classes. Caso tal anomalia são seja detectada, mesmo que a classe apresente outros tipos de anomalias, os danos às classes são consideravelmente menores, reforçando que nem todas as anomalias são igualmente danosas.

Para trabalhos futuros, recomenda-se efetuar uma avaliação sobre todos os tipos de anomalias, obtendo quais destas são mais danosas. Além disso, obter um conjunto

maior de sistemas para coletar uma quantidade maior de amostras para anomalias que surgem menos frequentemente, buscando regras para estas.

Referências

Abbes, M., Khomh, F., Guéhéneuc, Y.-G., and Antoniol, G. (2011). An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *15th Conf. on Software Maintenance and Reengineering (CSMR)*, pages 181–190.

Agrawal, R. and Srikant, R. (1994). Fast algorithms for mining association rules in large databases. In *20th Int. Conference on Very Large Data Bases (VLDB)*, pages 487–499.

Fowler, M., Beck, K., Brant, T., Opdyke, W., and Roberts, D. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA.

Kakarontzas, G., Constantinou, E., Ampatzoglou, A., and Stamelos, I. (2013). Layer assessment of object-oriented software: A metric facilitating white-box reuse. *Journal of Systems and Software*, 86(2):349 – 366.

Khomh, F., Penta, M. D., Guéhéneuc, Y.-G., and Antoniol, G. (2012). An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering*, 17(3):243–275.

Lavallée, M. and Robillard, P. N. (2015). Why good developers write bad code: An observational case study of the impacts of organizational factors on software quality. In *37th Int. Conference on Software Engineering (ICSE)*, pages 677–687.

Lozano, A., Wermelinger, M., and Nuseibeh, B. (2007). Assessing the impact of bad smells using historical information. In *9th Int. Workshop on Principles of Software Evolution: In Conjunction with the 6th ESEC/FSE Joint Meeting*, pages 31–34.

Moha, N., Guéhéneuc, Y.-G., Duchien, L., and Le Meur, A.-F. (2010). Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36.

Palomba, F., Bavota, G., Penta, M. D., Oliveto, R., and Lucia, A. D. (2014). Do they really smell bad? A study on developers' perception of bad code smells. In *30th Int. Conference on Software Maintenance and Evolution (ICSME)*, pages 101–110.

Sjøberg, D. I. K., Yamashita, A. F., Anda, B. C. D., Mockus, A., and Dybå, T. (2013). Quantifying the effect of code smells on maintenance effort. *Trans. Soft. Eng.*, 39(8).

Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., and Poshyvanyk, D. (2015). When and why your code starts to smell bad. In *37th Int. Conference on Software Engineering (ICSE)*, pages 403–414.

Yamashita, A. F. and Moonen, L. (2012). Do code smells reflect important maintainability aspects? In *28th Int. Conference on Software Maintenance (ICSM)*, pages 306–315.

Yamashita, A. F. and Moonen, L. (2013). Exploring the impact of inter-smell relations on software maintainability: an empirical study. In *35th Int. Conference on Software Engineering (ICSE)*, pages 682–691.

Uma Ferramenta para Conversão de Código JavaScript Orientado a Objetos em ECMA 5 para ECMA 6

Daniel V. S. Cruz[1], **Marco Tulio Valente**[1]

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

{danielvsc,mtov}@dcc.ufmg.br

Abstract. *This paper presents a tool to convert source code in JavaScript that follows the object oriented Programming paradigm on ECMAScript 5 to the newest version: ECMAScript 6. The conversion is focused on the new syntax used to define classes instead of using the simulation. We also describe the use of the proposed tool in two popular JavaScript systems, when we were successfully able to convert them to the new ECMAScript 6 syntax.*

Resumo. *Este artigo apresenta uma ferramenta para converter códigos-fonte em JavaScript que adotam o paradigma de Programação Orientada a Objetos em ECMAScript 5 para a nova versão: ECMAScript 6. A conversão é focada na nova sintaxe utilizada para definir classes ao invés de utilizar a simulação. Descrevemos também o uso da ferramenta proposta em dois sistemas JavaScript populares, quando estávamos com sucesso capaz de convertê-los para a nova sintaxe ECMAScript 6.*

1. Introdução

JavaScript é uma linguagem muito conhecida e cada vez mais popular. Ela representa, por exemplo, 15% dos repositórios do GitHub[1]. Inicialmente concebida para ser uma linguagem de script voltada apenas para dar dinamismo à páginas web, a proporção da sua utilização se estendeu e atingiu outros patamares. Além de não possuir concorrentes diretos nesse segmento, através de aplicações como o Node.js[2] , frameworks como Angular.js[3] e conceitos como JavaScript Isomórfico, a linguagem passou a ser utilizada no desenvolvimento de uma ampla gama de sistemas, extrapolando o ecossistema web.

Desde sua criação, programas em JavaScript têm sido implementados seguindo diversos paradigmas, como por exemplo os paradigmas de programação imperativa e funcional. Além destes, o paradigma da programação orientada a objetos é bastante utilizado [L.Silva et al. 2015b], usando princípios de prototipagem [Arnström et al. 2016]. Este conceito de prototipagem define uma abordagem diferente da orientação a objetos tradicionalmente utilizada em linguagens como Java ou C#. Adicionalmente, existe um problema sintático, dado que não existem em JavaScript palavras-chave comumente utilizadas, como por exemplo, **class**. Essas diferenças podem acabar causando dificuldade na implementação do código orientado a objetos.

[1]http://githut.info/
[2]https://nodejs.org/
[3]https://angularjs.org/

Essa divergência sintática foi alvo das atualizações da ECMAScript, que em sua nova versão, ES 6 (ECMAScript 6) [ecm 2015] introduz uma nova sintaxe para classes. A proposta é manter o funcionamento baseado em protótipos [Blaschek 2012] , mas com uma forma de implementação mais conveniente e sintaticamente próxima dos conceitos tradicionais de orientação a objetos. Para que os desenvolvedores possam usufruir dos benefícios desta nova versão, este trabalho apresenta uma ferramenta, intitulada *ECMA Class Parser*, para migração de classes de código JavaScript ES 5 (ECMAScript 5) [ecm 2011] para nova sintaxe proposta de ES 6. Com isso, espera-se ajudar os milhares de desenvolvedores que possuem código que emula classes de acordo com a sintaxe antiga a se beneficiarem de forma automática da sintaxe nativa de classes proposta pela ES 6.

O restante deste artigo está organizado como descrito a seguir. A Seção 2 introduz as diferenças entre a sintaxe de ES 5 e ES 6, focando nos conceitos de orientação a objetos. A Seção 3 descreve as regras propostas no trabalho para conversão de código JavaScript ES 5 para ES 6. A Seção 4 apresenta os resultados das conversões, quando aplicados em 2 sistemas JavaScript populares. A Seção 5 discute trabalhos relacionados, enquanto a Seção 6 conclui o artigo e apresenta sugestões de trabalhos futuros.

2. Background

Esta seção apresenta os conceitos de orientação a objetos suportados por código JavaScript ES 5 e ES 6.

2.1. Definição de Classes

Em ES 6, as palavras-reservadas *class* e *constructor* foram adicionadas permitindo a declaração de classes de uma maneira similar a outras linguagens, como Java. A palavra-reservada *class* substitui o uso da palavra-reservada *function* e os parâmetros desta, que passam a pertencer ao construtor da classe. Por exemplo, o Código 1 apresenta a declaração de uma classe em ES 5, de nome *Foo*, atributo *attr* e método *action*. O código 2 apresenta a mesma classe, porém implementada na versão ES 6. A ferramenta proposta neste trabalho visa converter a classe apresentada no Código 1, para a nova sintaxe de classes, conforme ilustrado no Código 2.

```
1 //Classe Foo declarada como function
2 function Foo(attr){
3     //Atributos
4     this.attr = attr;
5     //Metodos
6     function Action(){}
7 }
```

Código 1. Exemplo de Classe ES 5

```
1 //Classe Foo com o uso da nova palavra-chave
2 class Foo {
3     //Construtor da Classe
4     constructor(attr) {
5         //Atributos
6         this.attr = attr;
7         this.Action();
8     }
```

```
9       //Metodos
10      Action(){}
11  }
```

Código 2. Nova sintaxe para declaração de Classes - ES 6

Adicionalmente, em ES 5, existem variantes de declaração dos métodos, os quais podem ser declarados via protótipo, ao invés de implementados internamente em uma função. O código 3 apresenta um exemplo:

```
1  Foo.prototype.Action = function(){};
```

Código 3. Método adicionado via protótipo

2.2. Herança

Em JavaScript, para herdar de uma classe, é necessário realizar uma cópia do protótipo de um objeto da classe base (conforme mostrado na Linha 9 do Código 4) e realizar sua associação ao protótipo da classe filha (Linha 10).

```
1  //Classe base
2  function Foo(attr){
3      this.attr = attr;
4  }
5  //Classe filha
6  function Bar(){}
7
8  //Copia e Associacao dos prototipos
9  Bar.prototype = Object.create(Foo.prototype);
10 Bar.prototype.constructor = Foo;
```

Código 4. Herança na sintaxe de ES 5

Por outro lado, a nova sintaxe de ES 6 inclui a palavra-reservada *extends* que realiza o mesmo processo de maneira mais intuitiva, conforme mostrado no Código 5. Neste código, a classe *Bar* é declarada como uma subclasse de *Foo* (Linha 8).

```
1  //Classe  base alvo
2  class Foo{
3      constructor(attr) {
4          this.attr = attr;
5      }
6  }
7  //Bar herda de Foo
8  class Bar extends Foo{
9  }
```

Código 5. Herança em ES 6

2.3. Acesso a Classes Base

Em ES 5, o acesso a classe base ocorre por meio de uma chamada ao construtor da classe base em um objeto filho (conforme mostrado na Linha 5 do Código 6). Esse acesso é realizado através de uma invocação direta de função, usando o método *call*.

```
1
2  //Classe Bar
```

```
3 function Bar(attr){
4     //Chamada ao construtor de Foo com o argumento attr
5     Foo.call(this, attr);
6 }
```

Código 6. Acesso a classe Base em ES 5

Em ES 6, a adição da palavra-reservada *super* permite realizar o mesmo acesso de maneira mais simples, pois encapsula a chamada à função e requer que apenas sejam passados parâmetros, conforme mostrado no Código 7.

```
1     //Acesso a classe base com o argumento attr
2     super(attr);
```

Código 7. Acesso à classe Base em ES 6

3. Regras de Conversão

A ferramenta, para o processo de conversão de ES 5 para ES 6, inicialmente realiza uma classificação de todas as funções de um programa ES 5 em três categorias: Classe, Método ou Independente. Essas categorias são definidas nas subseções seguintes.

3.1. Classe

Uma função é identificada como classe se não for anônima, isto é, se possui um identificador, e se não for implementada no escopo de outra função não anônima. Além disso, é necessário que uma função usada para definir classes seja usada como parâmetro do operador *new*, como ilustra o Código 8. Alternativamente, a função deve ter um método adicionado em seu protótipo, como no Código 9.

```
1     var foo = new Foo();
```

Código 8. Criação de um objeto da classe Foo

```
1 Foo.prototype.Action = function(){};
```

Código 9. Adição do método Action à classe Foo

3.2. Método

Uma função é identificada como sendo um método se for associada a uma classe. Essa associação pode ocorrer via protótipo de uma função já identificada como possível classe, como mostrado no Código 10 (Linha 3). Nesse exemplo, *Bar* é classificada como um método da classe *Foo*

```
1     function Bar(){};
2     //Metodo Bar da Classe Foo
3     Foo.prototype.Bar = Bar;
```

Código 10. Método Bar adicionado à classe Foo

Adicionalmente, essa associação pode ocorrer implementando a função a ser classificada como um método internamente em uma função já classificada como possível classe, como mostrado no Código 11.

```
1     function Foo(){
2        function Bar(){};
3     }
```

Código 11. Método bar declarado internamente à classe Foo

3.3. Geração do Código ES 6

Para iniciar a conversão do código para ES 6, inicialmente, obtém-se a AST (*Abstract Syntax Tree*) através do *parser* esprima.js[4] . Em seguida, a AST é percorrida buscando por três tipos de estruturas sintáticas:

1. Declaração de Funções: Criação de funções com nome definido e sem associação, como por exemplo: *function Foo(){}*
2. Declaração de Expressões: Chamadas de funções, criação de objetos, estruturas de repetição e todo o restante de expressões. Um exemplo fundamental é a estrutura da implementação de protótipos: *Foo.prototype.Do = function(action){}*
3. Declaração de Variáveis: Declaração de variavéis que podem estar sendo atríbuidas a uma função, para criação de uma classe, como por exemplo: *var Foo = function(){}*

A seguir, descrevem-se as transformações realizadas sobre cada um dos elementos mencionados.

3.3.1. Declaração de Funções

As estruturas desse tipo são as primeiras a serem verificadas, com o objetivo de criar uma estrutura de dados chamada MIC (Matriz de Identificação de Classes). Essa matriz armazena as possíveis classes que poderão ser alvo do processo de conversão. Cada declaração de função é testada para as três categorias: Classe, Método e Independente, em ordem de verificação, respectivamente. Os testes de classificação de uma função são baseados em duas propriedades: Identificador e Escopo.

Toda função é inicialmente validada quanto a seu identificador. Em sua ausência, a função em questão já é descartada como Classe, conforme a primeira regra de conversão da ferramenta. Na presença de um identificador, o escopo da função é verificado conforme a segunda regra de conversão: se a função for interna a uma função que possui identificador, ela é um método. Esta verificação exclui a opção de classificação dessa função como classe, pois em ES 6 não há suporte para aninhamento de classes. Uma função que satisfaz as duas condições é marcada como Classe em potencial. A confirmação final ocorre após a ferramenta percorrer toda a AST. As funções que não forem confirmadas como Classe são automaticamente classificadas como independentes e descartadas.

3.3.2. Delaração de Variáveis

Para complementar a construção da MIC, as declarações de variáveis são verificadas, pois podem implicar na criação de uma classe através da palavra-reservada *var*. Uma declaração de variável consiste em um identificador e um inicializador (como mostra a Linha 1 do Código 12), sendo este último, facultativo:

```
1    var foo = 'init';
```

Código 12. Declaração de Variável

[4]http://esprima.org/

Variáveis sem inicializador são descartadas, enquanto as demais tem o tipo do seu inicializador verificado. Caso o inicializador possua a estrutura de uma declaração de função, ele é enviado para o processo anterior (descrito na seção 3.3.1), para que a função seja avaliada e classificada como Classe, Método ou independente.

3.3.3. Declaração de Expressões

Após a construção da MIC, realiza-se uma busca por expressões que validem os candidatos a classe. Para isso, consideram-se dois tipos fundamentais de estrutura: *Assignment Expression* e *Call Expression*. A primeira é verificada quando possui composição pela palavra-reservada *prototype*. O termo à esquerda desta palavra-reservada representa o identificador da classe, enquanto o termo à direita, designa um método. A segunda expressão, *Call Expression*, é usada para detectar o uso de herança, pois, quando utilizada na associação do protótipo de uma função (como mostrado na Linha 1 do Código 13), pode estar definindo a cópia do protótipo da Classe Base. Por exemplo, no código 14, ambas as classes, *Bar* e *Foo* são atualizadas na MIC, pois *Foo* é confirmada como Classe Base, enquanto *Bar* é confirmada como classe que herda de *Foo*.

```
1    Bar.prototype = Object.Create(Foo.prototype);
2    Bar.prototype.constructor = Foo;
```

Código 13. Confirmação de Classe - Herança por Prototipagem

```
1    function Bar(){
2        Foo.call(this);
3    }
```

Código 14. Confirmação de Classe - Acesso à classe Base

Uma *Call Expression* também é verificada de maneira independente. Se uma função identificada como Classe em potencial possuir uma *Call Expresion*, onde o termo à direita é o método *call*, ela é confirmada na MIC pois representa um acesso a uma classe Base. O termo a esquerda é o identificador de sua classe base. Por exemplo, no código 14, *Bar* e *Foo* são confirmados como sendo classes, sendo *Bar* uma subclasse de *Foo*.

Ao final do processo, é realizada a formatação do código-fonte convertido. Esta formatação é realizada utilizando a ferramenta js-beautify[5].

4. Resultados

Após testes com sistemas triviais, foram realizadas conversões em dois sistemas reais do GitHub. O primeiro, 2048.js, é um jogo implementado em JavaScript. Com o uso da ferramenta proposta, foram convertidas 10 classes e 56 métodos. Foram realizados testes para verificar se houve alteração no comportamento do jogo, não sendo detectada nenhuma falha. O código convertido está disponível em: `https://github.com/DVSCross/2048`.

No segundo sistema, Isomery.js , uma biblioteca gráfica foram convertidas 7 classes e 47 métodos. Para verificação do comportamento do sistema, foi recriada uma área de

[5]https://www.npmjs.com/package/js-beautify

utilização da biblioteca[6] com o código convertido, que foi testada exaustivamente. Não houve alteração em nenhuma circunstância reproduzida. O código convertido está disponível em: `https://github.com/DVSCross/isomer/`. Uma das classes convertidas, *Canvas*, é mostrada no código 15. Como pode ser veritificado, essa classe possui um construtor (Linhas 2-7) e dois métodos: *clear* (Linhas 8-10) e *path* (Linhas 11-17).

```
1   class Canvas {
2     constructor(elem) {
3       this.elem = elem;
4       this.ctx = this.elem.getContext('2d');
5       this.width = elem.width;
6       this.height = elem.height;
7     }
8     clear() {
9       this.ctx.clearRect(0, 0, this.width, this.height);
10    }
11    path(points, color) {
12      this.ctx.beginPath();
13      this.ctx.moveTo(points[0].x, points[0].y);
14      for (var i = 1; i < points.length; i++) {
15        this.ctx.lineTo(points[i].x, points[i].y);
16
17      }
18      this.ctx.closePath();
19      this.ctx.save();
20      this.ctx.globalAlpha = color.a;
21      this.ctx.fillStyle = this.ctx.strokeStyle = color.toHex();
22      this.ctx.stroke();
23      this.ctx.fill();
24      this.ctx.restore();
25    }
26  }
```

Código 15. Código ES 6 gerado para a classe Canvas do sistema Isomery.js

5. Trabalhos Relacionados

A análise de código JavaScript é alvo de estudos que buscam identificar desde más práticas de codificação [Fard and Mesbah 2013] até a utilização do paradigma de orientação a objetos [L.Silva et al. 2015a]. Existem muitas ferramentas de refatoração para linguagens estaticamente tipadas, porém poucas para linguagens dinâmicas como JavaScript [Feldthaus et al. 2011]. O principal motivo é a complexidade inerente à execução das linguagens dinâmicas [G.Richard et al. 2010]. As refatorações incluem também refatorações sintáticas, visando principalmente a melhoria da legibilidade do código-fonte [Feldthaus and Møller 2013]. A própria ECMA, detentora da sintaxe da ECMAScript, fundamentou grande parte das alterações da nova ES 6 em alterações de caráter sintático. Com o propósito de permitir a utilização da ES 6, mesmo com a falta de compatibilidade encontrada em diversos ambientes, uma ferramenta vem sendo amplamente utilizada: Babel.js[7]. Trata-se de um compilador que realiza a tradução de códigos ES 6 para ES 5, permitindo a utilização da nova sintaxe. Ou seja, essa ferramenta faz o processo inverso da

[6]http://jdan.github.io/isomer/playground/
[7]https://babeljs.io/

ferramenta apresentada neste artigo. O objetivo é exatamente possibilitar a interpretação de código ES 6 em máquinas virtuais que ainda não suportam esse novo padrão.

6. Conclusão e Trabalhos Futuros

Este trabalho propõe uma ferramenta de conversão de códigos JavaScript ES 5 para ES 6. Tal ferramenta se mostra útil, visto que os navegadores vem rapidamente provendo suporte aos novos recursos de ES 6 [Kangax 2016]. E, mesmo sem esse suporte completo, grandes companhias[8] já adotam a sintaxe de ES 6 junto a ferramentas de retrocompatibilidade. A migração para a ES 6 torna-se necessária então, para aqueles que não desejam converter manualmente o código em ES 5 mas desejam aproveitar os novos recursos da linguagem.

Como trabalho futuro, pretende-se aplicar a ferramenta proposta a um número maior de sistemas. Pretende-se também validar as conversões realizadas pela ferramenta proposta com os desenvolvedores de tais sistemas.

Referências

(2011). Ecma-262: Ecmascript language specification, edition 5.1.

(2015). Ecma-262: Ecmascript language specification, 6th edition.

Arnström, M., Christiansen, M., and Sehlberg, D. (2003 (acessado em 18 de Junho, 2016)). Prototype-based programming. `http://www.idt.mdh.se/kurser/cd5130/msl/2003lp4/reports/prototypebased.pdf`.

Blaschek, G. (2012). *Object-Oriented Programming: with Prototypes*. 1th edition.

Fard, A. M. and Mesbah, A. (2013). JSNose: Detecting JavaScript code smells. pages 116 – 125.

Feldthaus, A., Millstein, T., Møller, A., Schäfer, M., and Tip, F. (2011). Tool-supported refactoring for JavaScript. *SIGPLAN Notices*, 46(10):119–138.

Feldthaus, A. and Møller, A. (2013). Semi-automatic rename refactoring for JavaScript. *SIGPLAN Not.*, 48(10):323–338.

G.Richard, Lebresne, S., B.Burg, and J.Vitek (2010). An analysis of the dynamic behavior of JavaScript programs. *SIGPLAN Notices*, pages 1–12.

Kangax (2016). `https://kangax.github.io/compat-table/es6/`.

L.Silva, D.Hovadick, M.T.Valente, A.Bergel, N.Anquetil, and A.Etien (2015a). JSClassFinder: A Tool to Detect Class-like structures in JavaScript, pages 1–8.

L.Silva, M.Ramos, M.T.Valente, A.Bergel, and N.Anquetil (2015b). Does Javascript Software Embrace Classes? *SANER*, pages 73–82.

[8] https://babeljs.io/users/

Avaliação Heurística de um Ambiente Virtual para Análise de Rotas de Execução de Software

Filipe Fernandes, Claudia Rodrigues e Cláudia Werner

COPPE/UFRJ - Universidade Federal do Rio de Janeiro (UFRJ)
Caixa Postal 68.511 – CEP 21.945-970 – Rio de Janeiro – RJ – Brasil

{ffernandes,susie,werner}@cos.ufrj.br

Abstract. *Dynamic analysis aims to examine the implementation of a software system. However, dealing with scalability due to the large amount of data has been a challenge in understanding and visualizing software. This paper presents a heuristic usability evaluation of a new alternative for dynamic analysis software through Virtual Reality. The study revealed that 63% of participants judged as positive the usability of the virtual environment that is under development. In addition, it was observed that participants, despite the difficulties of interaction, have increased their interest of exploring more information UML sequence diagram due to its 3D representation.*

Resumo.*Análise dinâmica tem como princípio examinar a execução de um sistema de software. Contudo, lidar com questões de escalabilidade devido ao grande volume de dados tem sido um desafio na compreensão e visualização de software. Este trabalho apresenta a avaliação heurística de usabilidade de uma nova alternativa na análise dinâmica de software por meio de Realidade Virtual. O estudo revelou que 63% dos participantes julgaram como positivo a usabilidade do ambiente virtual que está em desenvolvimento. Além disso, observou-se que os participantes, apesar das dificuldades de interação, aumentaram seu interesse de explorar mais informações do diagrama de sequência UML devido a sua representação em 3D.*

1. Introdução

A análise dinâmica é uma técnica tipicamente adotada para compreensão de software durante o processo de manutenção e evolução, pois fornece uma "imagem precisa" devido à captura de informações do comportamento corrente do sistema, conhecidas como rotas de execução. Estas rotas registram informações sobre a ordem temporal da troca de mensagens entre objetos, de acordo com um cenário ao qual se deseja investigar [Dit *et al.* 2013]. No entanto, visualizar informação dinâmica emerge como um problema de escalabilidade, devido ao grande volume de dados, podendo gerar sobrecarga cognitiva durante o processo de manutenção [Cornelissen *et al.* 2011].

Para lidar com esta questão de escalabilidade foram propostas abordagens para a redução de rotas de execução e metáforas visuais não tradicionais. No entanto, técnicas de redução podem omitir informações importantes durante o processo de compreensão, enquanto que visualizações não tradicionais podem adicionar mais sobrecarga cognitiva devido à dificuldade de relacionar as formas visuais com os dados da execução do software [Cornelissen *et al.* 2009]. Segundo Maletic *et al.* (2002), novos dispositivos de

visualização e interação, bem como novas representações, podem apoiar a compreensão de um grande volume de dados, tal como a utilização de Realidade Virtual (RV).

Este trabalho tem como principal objetivo apresentar a avaliação heurística de usabilidade de um ambiente virtual para apoiar a análise e compreensão de rotas de execução de software, denominada VisAr3D-Dynamic. Por se tratar de uma nova alternativa de compreensão de software, usuários podem não estar familiarizados em manipular diagramas de sequência UML em 3D, bem como outros recursos do ambiente. Portanto, decidiu-se primeiramente identificar falhas de usabilidade e no futuro realizar um experimento controlado comparando a abordagem proposta com uma tradicional, sem problemas de usabilidade.

O restante do trabalho está organizado como segue: a Seção 2 apresenta o conceito de RV, bem como sua aplicação na área de visualização. A Seção 3 apresenta o ambiente virtual VisAr3D-Dynamic. A Seção 4 apresenta a avaliação heurística realizada e, por fim, a Seção 5 conclui o artigo.

2. Visualização de Dados em Realidade Virtual

A RV é uma interface avançada para aplicações computacionais, onde o usuário pode navegar e interagir, em tempo real, em um ambiente tridimensional gerado por computador, usando dispositivos multissensoriais [Kirner e Tori 2004]. O usuário tem a impressão de estar atuando dentro destes ambientes virtuais em tempo real. Em virtude destas características, a RV tem sido utilizada por diversas áreas com aplicabilidades distintas, inclusive na visualização de dados complexos [Van Dam 2000].

Em visualização científica, usuários podem facilmente explorar e compreender estruturas naturalmente tridimensionais complexas por meio de uma experiência imersiva através de dispositivos multissensoriais, que possibilitam ao usuário, além de visualizar, ouvir, tocar e, até mesmo, sentir o odor e o paladar [Van Dam 2000]. Ao contrário da científica, a visualização de informação cria representações gráficas deum grande volume de dados abstratos gerados por simulações computacionais. Visualizações em RV proporcionam uma experiência imersiva na manipulação destes dados sob vários ângulos e posições, permitindo uma ampla exploração dos mesmos, inclusive de propriedades matemáticas intrínsecas [Kirner e Tori 2004].

3. VisAr3D-Dynamic

Neste contexto de visualização de informação por meio de RV, está em desenvolvimento um ambiente virtual para análise de rotas de execução de sistemas de software de larga escala. A VisAr3D-Dynamic [Fernandes *et al.* 2015]estende a visão de comportamento da abordagem VisAr3D (Visualização da Arquitetura de Software em 3D) [Rodrigues e Werner 2016]. Esta abordagem propõe um ambiente de ensino-aprendizagem de modelos UML de sistemas complexos, no entanto, a visão de comportamento também pode ser aplicável à compreensão de programas, no contexto de manutenção e evolução de software.

O ambiente virtual proposto renderiza diagramas de sequência UML em 3D a partir de dados capturados durante a execução do software, persistidos no formato XMI. A fim de reduzir a sobrecarga cognitiva das rotas de execução foram implementadas funcionalidades, as quais serão brevemente descritas.

O diagrama de sequência 3D complexo (Figura 1) é exibido aos poucos através de uma animação onde o usuário pode acompanhar o seu funcionamento através de uma *Timeline*. A *Timeline* transmite a percepção temporal ao aplicar efeitos *fade-in* e *fade-out*, tanto em *lifelines* quanto em mensagens, conforme o acionamento do botão *Iniciar* ou pela interação através do *slider* (Figura 1(a)), possibilitando o avanço ou recuo da animação. Cores amarela e cinza também são utilizadas para indicar visualmente mensagens que uma determinada *lifeline* envia ou recebe, respectivamente.

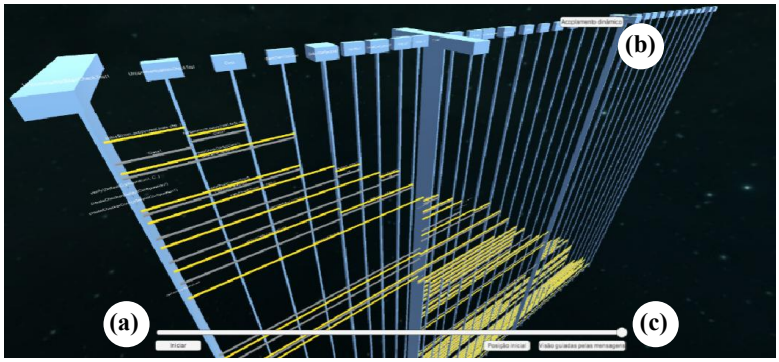


Figura 1. Diagrama de sequência UML 3D em perspectiva no VisAr3D-Dynamic

Quando a função de acoplamento dinâmico é acionada (Figura 1(b)), a forma geométrica em *z* de cada *lifeline* é alterada, ou seja, um valor é calculado e aplicado ao seu comprimento, utilizando a métrica EOC (*Export Object Coupling*) [Yacoub*et al.*2000]. E como forma de interagir com o ambiente, os seis graus de liberdade[1] foram disponibilizados através do teclado (A, S, W e D e pelas setas direcionais) e *mouse*, que também disponibiliza o recurso de *zoom*.

4. Avaliação Heurística da VisAr3D-Dynamic

O propósito da pesquisa é desenvolver um ambiente virtual para apoiar a compreensão de rotas de execução de sistemas complexos. A fim de alcançar este objetivo, decidiu-se avaliar a usabilidade do ambiente com as funcionalidades descritas na Seção 3.

4.1 Objetivo do Estudo

Seguindo a abordagem GQM–Goal/Question/Metric [Basili *et al.* 1994], o objetivo do estudo pode ser descrito conforme mostra a Tabela 1.

Tabela 1. Objetivo do estudo segundo a abordagem GQM

Analisar	a interação com um diagrama de sequência em 3D por meio de um ambiente virtual
Com o propósito de	caracterizar
Com respeito às	heurísticas de usabilidade
Do ponto de vista dos	pesquisadores
No contexto de	tarefas de compreensão de rotas de execução por alunos de graduação e de pós-graduação

[1] Graus de liberdade é o número de eixos de coordenadas que podem ser manipulados simultaneamente durante o processo interativo.

Na seção a seguir são apresentadas as heurísticas de usabilidade para ambientes virtuais adotadas neste trabalho.

4.2. Avaliação Heurística para Ambientes Virtuais

A avaliação de projeto de interfaces tem por objetivo identificar, classificar e contar o número de problemas de usabilidade de um sistema computacional. Sutcliffe e Gault (2004) propuseram um conjunto de doze heurísticas, baseadas nas heurísticas de Nielsen (1994), a serem utilizadas com intuito de verificar os componentes disponíveis no ambiente virtual (Tabela 2).

Tabela 2. Heurísticas de usabilidade [Sutcliffe e Gault 2004]

Identificação	Descrição
H1	*Engajamento Natural*: a interação deve atender a expectativa do usuário em relação ao mundo real ou um conhecimento prévio. Idealmente, o usuário deve ter consciência de que a realidade é virtual. Essa heurística dependerá da exigência de naturalidade e sensação de presença e engajamento do usuário.
H2	*Compatível com as tarefas do usuário e do domínio:* o ambiente virtual e o comportamento dos objetos devem corresponder à expectativa do usuário em relação aos objetos do mundo real, seu comportamento e *affordances*[2].
H3	*Expressão natural da ação:* a representação de presença no ambiente virtual deve permitir ao usuário agir e explorar de uma maneira mais natural e implementar leis básicas da física. Esta heurística pode ser limitada pelos dispositivos disponíveis.
H4	*Representação e coordenação da ação*: a representação de presença e comportamento no ambiente virtual deve ser fiel às ações do usuário. O tempo de resposta entre o movimento do usuário e atualização no ambiente de exibição deve ser inferior a 200 milissegundos para evitar problemas de enjoo.
H5	*Feedback realista:* os efeitos das ações do usuário em objetos do mundo virtual devem ser imediatamente visíveis e em conformidade com as leis da física e expectativas de percepção do usuário.
H6	*Fidelidade dos pontos de vista:* a representação visual do ambiente virtual deve mapear a percepção normal do usuário, e a mudança do ponto de vista pelo movimento da cabeça deve ser processada sem demora.
H7	*Suporte à navegação e orientação:* os usuários devem ser capazes de encontrar onde eles estão no ambiente virtual e voltar para posições definidas ou conhecidas.
H8	*Entrada e saída:* os meios de entrar e sair do mundo virtual devem ser claramente comunicados.
H9	*Ações consistentes:* ações no ambiente virtual devem ser claramente identificadas, como por exemplo, ações de substituição de energia (comuns em jogos) ou alternância entre formas de navegação.
H10	*Suporte à aprendizagem:* objetos ativos devem ser identificáveis ou, se necessário, explicá-los para promover aprendizagem do ambiente virtual.
H11	*Turn-taking:* aplica-se à conversação em que os "avatares"[3] podem se comunicar com o usuário ou quando o sistema toma a iniciativa. A alternância na comunicação deve ser clara para o usuário.
H12	*Senso de presença*: a percepção e engajamento do usuário de estar em um mundo "real" devem ser o mais natural possível.

As heurísticas são aplicadas em relação ao ambiente virtual que será objeto de estudo, podendo ser utilizadas todas ou algumas delas. Cada questão relaciona-se exclusivamente com uma heurística, porém cada heurística pode estar vinculada a várias

[2] *Affordance* é a qualidade de um objeto que permite ao usuário identificar sua funcionalidade sem prévia explicação.

[3] Avatar é a representação humana virtual do usuário.

questões, conforme Tabela 3.Em situações que o avaliador julgar uma questão como problemática de usabilidade, ou seja, escolher a opção "Não", este indicará um nível de gravidade, conforme mostra a Tabela 4.

4.3.Projeto Experimental

A avaliação foi conduzida individualmente por seis participantes no total, 83% da área de Ciências de Computação e 17% de Engenharia Elétrica, que se apresentaram voluntariamente aos convites. Com relação ao perfil dos participantes:(i) 33% estão cursando a graduação, 33% são alunos de mestrado, 17% possuem pós-graduação *lato-sensu* e 17% são doutores; (ii) 83% utilizaram diagramas UML em sala de aula;(iii) 50% possuem experiência com orientação a objetos adquirida em projetos em sala de aula; e por fim, (iv) quanto a experiência com a linguagem Java, 17% foram obtidas em projetos na indústria, 33% em projetos pessoais, 33% com prática em projetos em sala de aula e 17% em livros ou em salas de aulas.

Neste trabalho, a avaliação apresentou os seguintes procedimentos: (i) aceitação do termo de consentimento para a realização da avaliação;(ii) preenchimento do formulário de caracterização de perfil;(iii) breve apresentação do objetivo do ambiente virtual, informando as possibilidades de interação por meio do teclado e *mouse*, não informando a função de cada tecla ou botão; (iv) livre navegação e exploração no ambiente virtual por 5 minutos, aproximadamente; (v) realização das tarefas no ambiente; e (vi) preenchimento do questionário contendo as heurísticas de usabilidade.

Para a execução da avaliação foi utilizado um notebook com monitor de 15,6 polegadas, teclado integrado e um *mouse* com 3 botões, utilizado tanto para o preenchimento dos formulários digitais, quanto para a interação com o ambiente virtual.

4.4. Ameaças à Validade

Alguns fatores podem ser considerados como ameaças à validade neste estudo. Não foram realizados testes estatísticos para identificar *outliers*. Tomou-se essa decisão devido ao pequeno número de participantes que não é representativo. A quantidade de participantes, portanto, é considerada uma outra ameaça à validade do estudo. Por fim, outra possível ameaça é em relação aos perfis dos participantes, onde grande parte não são de engenharia de software.

4.5. Resultados e Discussões

Como o intuito da avaliação heurística é identificar problemas de usabilidade, neste estudo foram identificadas três questões críticas: Q4, Q7 e Q9, as quais podem ser consultadas na Figura 2. Cada uma das três é descrita, iniciando pela mais crítica.

A finalidade da Q7 é de identificar se a navegação e exploração proporciona desorientação espacial. Todos os participantes julgaram como um problema de usabilidade, pois encontraram dificuldades de navegação no ambiente, ou seja, obteve 100% de rejeição. Este resultado induz que, possivelmente, os participantes tenham julgado com altos níveis de gravidade (entre 3 e 4). Contudo, 50% dos participantes avaliaram como nível de gravidade 2 e 33% como nível 3. Acredita-se que os participantes já tivessem a expectativa de desorientar-se, por se tratar de um ambiente virtual e tridimensional, característica comum em jogos.

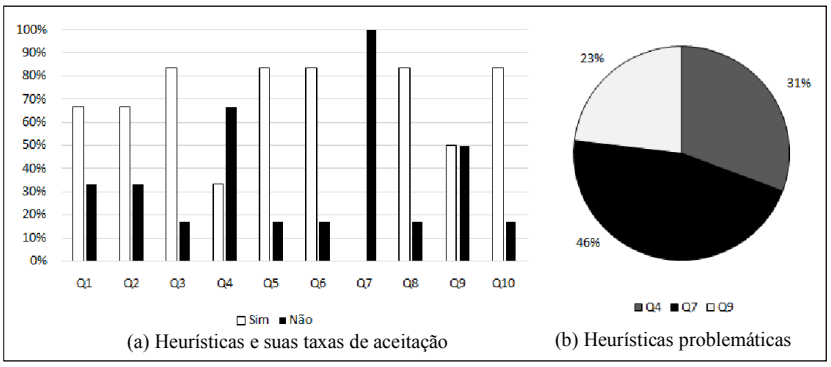
Tabela 3. Questões utilizadas na avaliação relacionadas com as heurísticas

Identificação	Heurística	Descrição
Q1	H1	A manipulação dos objetos disponíveis no ambiente virtual foi intuitiva?
Q2	H2	O diagrama de sequência 3D no ambiente virtual transmitiu o mesmo significado de um diagrama de sequência tradicional?
Q3	H2	Após interagir com os objetos no ambiente virtual, estes se comportaram como o esperado?
Q4	H3	A exploração e navegação no ambiente virtual por meio de suas interfaces (*mouse* e teclado) foram satisfatórias?
Q5	H4	Mediante a interação com o ambiente virtual, o tempo de resposta de alguma ação foi imediato?
Q6	H5	Os efeitos aplicados aos objetos (*lifelines*) e mensagens transmitem a ideia de eventos ocorridos em tempo de execução?
Q7	H7	No ambiente virtual NÃO é possível se "perder" durante a navegação e exploração?
Q8	H7	O ambiente virtual dispõe de mecanismos para ajudar na orientação espacial?
Q9	H10	O ambiente virtual é composto por componentes virtuais autoexplicativos?
Q10	H12	O ambiente virtual transmitiu algum grau de sensação de "estar dentro" do ambiente?

Tabela 4. Níveis de Gravidade

Níveis	Descrição da Gravidade
0	não é um problema de usabilidade
1	parcialmente um problema de usabilidade
2	problema normal de usabilidade
3	problema relevante de usabilidade
4	problema crítico de usabilidade

Em relação à orientação, um dos participantes comentou: "*Queria que ele 'me levasse' junto com a execução do diagrama. Ou que eu pudesse 'me libertar' dessa execução e tivesse liberdade de acompanhar do ângulo que eu achasse melhor"*. Este comentário também está relacionado com Q9 (uso do botão "visão guiada pelas mensagens"). Apesar de julgar os elementos no ambiente parcialmente autoexplicativos, sentiu falta de um "*on/off*" no botão "visão guiada pelas mensagens" ((c)) ou algo que indicasse melhor se o botão estaria ativo ou não.



(a) Heurísticas e suas taxas de aceitação (b) Heurísticas problemáticas

Figura 2. Resultados das heurísticas

O segundo mais crítico é Q4 com 67% de rejeição, onde procurou-se investigar o quanto as interfaces tradicionais interferem no ambiente virtual. Em relação aos graus de gravidade, 1/3 dos participantes avaliaram, respectivamente, com níveis 0, 1 e 3. Entre os

problemas, destacam-se a forma como interagiram com o botão *scroll* do *mouse*. Na etapa de exploração livre do ambiente, notou-se que todos os participantes tentaram efetuar o *zoom* rolando o *scroll*. Apenas 17% conseguiram utilizar esta função corretamente, mantendo o *scroll* pressionado e movimentando o *mouse* para frente e para trás.

Analisando os comentários, 33% dos participantes explicitaram o desconforto na utilização deste botão. Dentre eles, destaca-se: "*Achei a resposta do mouse lenta, os movimentos de zoom não são intuitivos, de início tentei usar a roda para dar zoom. Tive dificuldade na forma de retornar à posição original depois de mover as lifelines no ambiente 3d. O movimento pelas setas poderia ser um pouco mais rápido ou configurável. O que menos me agradou foi o tempo de resposta do zoom"*.

Por fim, a Q9 obteve um resultado parcial com 50% de rejeição. Metade dos participantes julgaram como item problemático. Nesta questão, procurou-se investigar se os elementos de interface de usuário eram autoexplicativos, ou seja, se o formato visual e nome indicavam intuitivamente suas funcionalidades. Alguns participantes comentaram que o entendimento do botão "visão guiada pelas mensagens" não foi alcançada, bem como a utilização de dicas explicativas para cada interface poderiam ser adicionadas ao ambiente.

Embora não seja comum a manipulação de diagramas de sequência 3D e ser um ambiente virtual em desenvolvimento, 63% dos participantes julgaram como positivo a usabilidade e interação com o ambiente virtual. As cores nas mensagens, bem como a alteração da forma de cada *lifeline* em *z*, ajudou identificar questões de qualidade de software, tal como o acoplamento em tempo de execução. Quanto aos aspectos negativos, a exploração por meio dos dispositivos de teclado e *mouse*, desorientação espacial e componentes autoexplicativos foram os mais criticados na avaliação.

No entanto, observou-se que, enquanto os participantes interagiam com o ambiente virtual, estes despertavam a curiosidade em explorar a tridimensionalidade do ambiente, principalmente, com intuito de obter algum tipo de informação atrás do diagrama de sequência. Acredita-se, que este tipo de reação é devido ao fato de que o mundo real, intrinsecamente, seja tridimensional e, naturalmente, se comportaram da mesma forma explorando e interagindo com o diagrama no ambiente virtual.

A RV torna a interação com a visualização 3D o mais natural possível, transmitindo uma experiência imersiva na exploração de dados complexos. Apesar de limitar-se aos dispositivos convencionais, como o teclado, *mouse* e monitor, atualmente, tem-se popularizado tecnologias e dispositivos imersivos que estão viabilizando o uso de RV em diversos domínios, inclusive em visualização de software.

5. Considerações Finais

A finalidade deste trabalho foi apresentar uma abordagem para análise dinâmica de software por meio de Realidade Virtual, denominadaVisAr3D-Dynamic, e a avaliação de sua usabilidade para análise de rotas de execução de sistema de software de larga escala. Para isso, adotou-se heurísticas de ambientes virtuais por tratar de questões específicas de aplicações em Realidade Virtual.

A partir deste estudo, observou-se que a interação por meio de dispositivos não tradicionais pode ser uma barreira durante a navegação e exploração da informação dentro do ambiente virtual, apesar das vantagens da visualização em 3D. Como trabalhos futuros,

os itens críticos apontados neste estudo serão corrigidos, outros requisitos serão implementados, tais como múltiplas visões, a utilização de HMD ou Joystick, e, posteriormente, a realização de outra avaliação, visando obter evidências do potencial da experiência imersiva na exploração de dados complexos de software.

Referências

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

Cornelissen, B., Zaidman, A., & van Deursen, A. (2011). "A controlled experiment for program comprehension through trace visualization". IEEE Transactions on Software Engineering, 37(3), 341-355.

Cornelissen, B., Zaidman, A., Van Deursen, A., Moonen, L., & Koschke, R. (2009). "A systematic survey of program comprehension through dynamic analysis". IEEE Transactions on Software Engineering, 35(5), 684-702.

Dit, B., Revelle, M., Gethers, M., & Poshyvanyk, D. (2013). "Feature location in source code: a taxonomy and survey". Journal of Software: Evolution and Process, 25(1), 53-95.

Fernandes, F. A.; Rodrigues, C. S.; Werner, C. M. (2015). "Um Ambiente de Realidade Virtual para Apoiar a Compreensão de Aspectos Dinâmicos do Software". Anais do II Fórum de Educação em Engenharia de Computação, V Simpósio Brasileiro sobre Engenharia de Sistemas de Computação, pp. 1-4.

Kirner, C., & Tori, R. (2004). "Introdução à realidade virtual, realidade misturada e hiper-realidade". Realidade Virtual: Conceitos, Tecnologia e Tendências. 1ed. São Paulo, 1, 3-20.

Maletic, J. I., Marcus, A., & Collard, M. L. (2002). "A task oriented view of software visualization". In Visualizing Software for Understanding and Analysis, 2002. Proceedings. First International Workshop on (pp. 32-40). IEEE.

Nielsen, J. (1994). "Usability engineering". Elsevier.

Rodrigues, C. S. C., Werner, C. M., & Landau, L. (2016). "VisAr3D: an innovative 3D visualization of UML models". In Proceedings of the 38th International Conference on Software Engineering Companion (pp. 451-460).Software Engineering Education and Training Track (SEET). ACM.

Sutcliffe, A., & Gault, B. (2004). "Heuristic evaluation of virtual reality applications". Interacting with computers, 16(4), 831-849.

Van Dam, A., Forsberg, A. S., Laidlaw, D. H., LaViola, J. J., & Simpson, R. M. (2000). "Immersive VR for scientific visualization: A progress report". IEEE Computer Graphics and Applications, 20(6), 26-52.

Yacoub, S. M., Ammar, H. H., & Robinson, T. (2000). "A methodology for architectural-level risk assessment using dynamic metrics". In Software Reliability Engineering, 2000. ISSRE 2000. Proceedings. 11th International Symposium on (pp. 210-221). IEEE.

SPPV: Visualizing Software Process Provenance Data

Gabriella C. B. Costa[1], Marcelo Schots[1], Weiner E. B. Oliveira[2], Humberto L. O. Dalpra[2], Cláudia M. L. Werner[1], Regina Braga[2], José Maria N. David[2], Marcos A. Miguel[2], Victor Ströele[2], Fernanda Campos[2]

[1]COPPE - Systems Engineering and Computer Science Department
UFRJ - Federal University of Rio de Janeiro
21945-970 - Rio de Janeiro - RJ - Brazil

[2]Computer Science Department
UFJF - Federal University of Juiz de Fora
36036-900 - Juiz de Fora - MG - Brazil

{gabriellacbc, schots, werner}@cos.ufrj.br, {woliveira82, humbertodalpra}@gmail.com, {regina.braga, jose.david, fernanda.campos}@ufjf.edu.br, {marcos.miguel, victor.stroele}@ice.ufjf.br

Abstract. *Provenance data can provide implicit and strategic information for process improvement, helping to establish potential causes for process success, failure, delays, among others. However, when it comes to software processes, the large amount of execution data generated makes their analysis complex and arduous, requiring proper ways to store and represent provenance information. In this sense, this paper presents the application of a goal-oriented process to build a visualization tool geared to provenance data. The suitability of the visual attributes mapped was preliminarily assessed through an exploratory analysis with stakeholders from two software development companies, using their own process provenance data.*

1. Introduction

A software process can be defined as a set of activities, methods, practices and transformations that people use to develop and maintain software and associated products. A standard software process encompasses the essential process assets: activities, artifacts, resources, and procedures [Falbo and Bertollo, 2005].

Provenance "is a record that describes the people, institutions, entities, and activities involved in producing, influencing, or delivering a piece of data or a thing" [Belhajjame *et al*. 2013]. Data provenance can be used in the context of software process development to provide additional information about it. For example, during process modeling and execution, data provenance can be captured to establish process success, failure, delays and errors [Wendel *et al*. 2010].

Assuming that software development organizations in their intra-organizational context perform similar processes, the knowledge acquired in previous executions of these processes can be reused to establish better policies to adopt in future projects, thus supporting continuous process improvement. This knowledge can also be acquired by investigating the data generated during process executions. However, the increase of

process data generated during this execution makes the data analysis more complex. Thus, it requires techniques that allow a proper examination of these data, extracting records and facts that will actually contribute to process improvement and choose a proper representation for these data to enable their exploration and understanding.

Most software process managers are focused on process monitoring and improvement, and they do not have advanced knowledge about models, methods, and techniques for software process data analysis. For analyzing software process data, the use of provenance techniques and models and appropriate data visualizations can be of great help. Our main contribution is focused in proposing an appropriate data provenance visualization in a specific domain (software development process), in order to facilitate the understanding about these data and to support software process improvement. We did not find any related contribution targeted to software process provenance data and a way to visualize them. Generic provenance visualization tools do not use a familiar notation to process managers and often provided data that do not offer any meaning for them. The approach presented in this paper aims to assist process managers in (i) understanding software process provenance data, and (ii) using them to support decision-making on the analyzed process.

The remainder of this paper is structured as follows. The next section provides an overview of the PROV-Process approach [Dalpra *et al*. 2015], used as the visualization backend for software process provenance data handling and storage. Section 3 discusses other works related to provenance visualization. Section 4 describes how the visualization attributes were defined, using a goal-oriented process [Schots and Werner 2015]. The SPPV (Software Process Provenance Visualization) tool, developed to concretize the proposed approach, is presented in Section 5 with an exploratory analysis, using software process provenance data from two Brazilian software development companies. Finally, the conclusions are presented in Section 6.

2. The PROV-Process Approach

The PROV-Process approach was used for structuring and storing the software process provenance data to be visualized. It is part of the iSPuP (improving Software Process using Provenance) approach [Costa *et al*. 2016] and consists of a specified architecture for capturing, storing and analyzing processes provenance data, using the PROV model [Moreau *et al*. 2015]. PROV-Process' database was modeled and implemented based on PROV-DM [Belhajjame *et al*. 2013], which suggests three vertices to represent *entities*, *activities* and *agents*, including causal relationships between them, such as *wasGeneratedBy*, *wasStartedBy*, *wasEndedBy*, *wasInvalidatedBy*, *wasDerivedFrom*, *alternateOf*, *specializationOf*, *used* and *hadMember*. In this approach, the *entities* represent the artifacts of software process, *activities* are used with the same concept of software process activities, and *agents* represent software process resources.

In addition to allowing the storage of provenance data, PROV-Process offers an interface to build an OWL (Ontology Web Language) file with the captured provenance data of a software process using an extension of PROV-O ontology [Lebo *et al*. 2013], named PROV-Process Ontology. Figure 1 shows an example of an activity instance (called *Solution_Implementation_14*) represented in PROV-Process Ontology. As it can be seen, this activity used four entities and was associated with the agent *VB6_2* (as

illustrated by the used and *wasAssociatedWith* relationships). It can be noticed that, as the amount of data increases, the analysis of them in an ontology tool (such as Protégé) becomes a non-trivial task. It is known that software became bigger over the years, generating large amounts of daily information. Thus, the purpose of using software visualization applied to provenance data is to help understanding the processed information in order to improve the efficiency of software development processes.

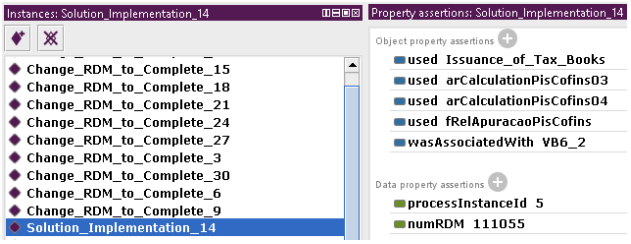


Figure 1. Activity example represented in the ontology tool.

3. Related Work

NoWorkflow [Murta *et al*. 2014] captures provenance of experiment scripts and includes a graph-based visualization mechanism. This approach only captures provenance of Python scripts and is not focused on capturing, storing, and analyzing software process provenance data. Besides, it does not use mechanisms to emphasize a given item (for example, an agent), as SPPV does.

Chen and Plale (2015) use visualization techniques to support real time analysis of large provenance graphs, applying it to the provenance captured from the network layers of large-scale E-Science distributed applications. Unlike this proposal, our work combines the use of provenance data, ontology and visualization attributes, in order to facilitate project managers in understanding the analyzed software process.

Some tools such as GraphViz [Ellson *et al*. 2001] and PROV-O-Viz [Hoekstra and Groth 2014] deal with ontology visualizations as graphs. Differently from GraphViz, SPPV focuses on ontology individuals, and is targeted to experts in software process management, who does not necessarily have much knowledge in ontology. Besides, SPPV uses BPMN, a more familiar notation to process managers. PROV-O-Viz, in turn, is a web-based visualization tool for PROV-based provenance traces collected from various sources, including ontologies, which leverages diagrams to reflect the flow of information through activities. Since our focus is not at analyzing the flow of information, PROV-O-Viz becomes unsuitable for our specific goals.

Although Wendel *et al*. (2010) present an approach that handles provenance and software development processes, we did not find contributions targeted to software process provenance data and a way to visualize them. SPPV uses an explicit rationale to identify appropriate visualization attributes [Schots and Werner 2015], and was subjected to an exploratory analysis with real data from two Brazilian companies.

4. Mapping Process Provenance Data and Visualization

When creating a visualization tool, there are specific goals to be achieved. The challenge lies on transforming software process provenance data into a corresponding visualization. To do this, we used the staged process proposed by Schots and Werner

(2015) for mapping managerial needs to visualization attributes. To develop the proposed provenance visualization system, the meet-in-the-middle strategy was chosen.

4.1. Mapping Goals and Questions

In this study, two goals were previously defined: (**G1**) *Analyze whether a process agent, who is a company employee engaged in the process tasks, is overloaded in a given process*; and (**G2**) *Analyze how often process requests are started by clients*. The first goal (**G1**) aims to assist the process manager in making decisions regarding employees who are overloaded or idle in the process, being able to perform a different distribution of tasks in the process. The second goal (**G2**) aims to assist in analyzing client demands, considering that a process generates a software product that is consumed by a client.

Considering these goals, the following questions were raised: (**Qa**) *How many entities/artifacts were manipulated by the agent?* (related to **G1**); (**Qb**) *How often are these entities manipulated by the agent over time?* (related to **G1**); (**Qc**) *How many activities/tasks were done by the agent?* (related to **G1**); (**Qd**) *How often are these activities done by the agent over time?* (related to **G1**); (**Qe**): *How many activities/tasks, related to new requests, were started by the client?* (related to **G2**). Although tabular or list-based views can help answering these questions, the context in which the entities, agents and activities were involved and, most importantly, the other relations between them, cannot be interpreted as easily as by using a visualization.

4.2. Mapping Questions and Tasks

The tasks associated with these questions are: (**Ta**) *Analyze the amount of activities/ tasks performed by each agent* (related to **Qc** and **Qd**); (**Tb**) *Analyze the amount of entities/artifacts manipulated by each agent* (related to **Qa** and **Qb**); (**Tc**) *Analyze the amount of activities/tasks related to new requests started by each client* (related to **Qe**).

4.3. Mapping Tasks and Data

The data required to support the tasks are: (i) **agent information**: agent name (**Ta**, **Tb**, **Tc**) and type: Organization, Person or Software Agent (whose roles can be a client, a developer, or a software to automate a certain task); (ii) **activity information**: activity name (**Ta**, **Tc**) and its respective data properties; (iii) **entity information**: entity name (**Tb**); (iv) **relation information**: relation name, source, target, type (asserted or inferred[1]), and amount of relations between the same source and target (**Ta**, **Tb**, **Tc**).

4.4. Mapping Data and Visualization

One of PROV requirements [Moreau *et al*. 2015] is to provide a single layout convention used throughout PROV specifications. This convention uses blue rectangles, yellow ellipses, and orange pentagons for activities, entities, and agents, respectively. This layout convention was adopted in our mapping, but since process experts are not familiar with the conventional layout used by PROV, the developed tool also enables the visualization of processes provenance data using the BPMN 2.0 notation [OMG,

[1] An asserted relation comes from the original process data within a knowledge base, while an inferred relation represents additional process data that some ontology reasoned provided/inferred from the asserted data.

2011]. Table 1 depicts the mapping of software process provenance data to visual attributes. In addition, SPPV also highlights elements and relationships associated to an element when hovering the mouse on it. It also provides filters to allow the selection of specific elements of the ontology (i.e., an activity, an entity, or an agent).

Table 1. Data-to-Visualization Mapping.

Visual Attribute	Data	Value	Description
shape / icon	Agent	pentagon OR person icon*	Represents an agent in PROV or a role in BPMN.
	Activity	rectangle	Represents an activity in PROV or a task in BPMN.
	Entity	ellipse OR paper icon	Represents an entity in PROV notation or an artifact in BPMN.
	Used	arrow	Represents the use of an entity by an activity.
	wasAssociatedWith		Represents an association between an activity and an agent.
	wasAttributedTo		Represents the assignment of an entity to an agent.
	Influenced		Represents an abstract relation of influence.
	wasInfluencedBy		
color	Agent	orange	Used to identify the type of vertex (as the agent symbol does not exist in BPMN, we created an icon to represent the respective nodes.)
	Activity	blue	
	Entity	yellow	
	relation type	- black, for asserted relations - red, for inferred relations - olive, for inferred and asserted relations	The use of ontology allows inferences to be made using provenance data. Then, the arrows in black display provenance data and the arrows in red display inferred data.
transparency	amount of relations in which the vertex (agent, activity, or entity) is part	- high: for 1 relation - medium: for 2 or 3 relations -low: for 4 or more relations	According to the vertex brightness, it will be easier to identify the one that has more relations than others.
width	amount of relations between the same vertex source and target	- thick: for 4 or more relations - medium: for 2 or 3 relations - slim: for 1 relation	When there is more than one relation between the same source and target, the arrow width will facilitate to identify this amount of relations.

*As the agent symbol does not exist in BPMN, we created an icon to represent the respective nodes.

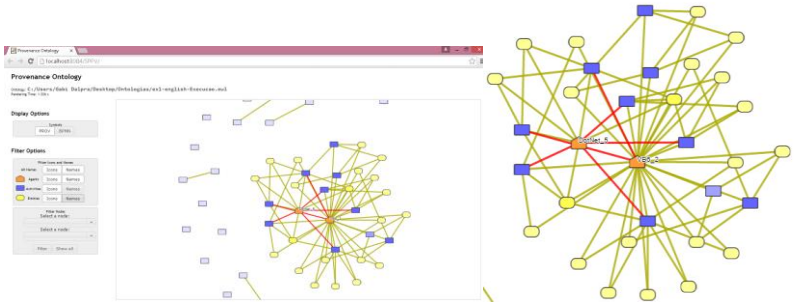
5. The Software Process Provenance Visualization Tool (SPPV) in Action

The SPPV tool loads the provenance data from ontology as can be seen in Figures 2 to 5. An exploratory study was performed using SPPV to analyze the data from two real industry processes. One process is used to manage change requests in a 19-years-old Brazilian software company (referred to as company α) that deals with business management software. The other process involves the implementation of new features and error handling in an Enterprise Resource Planning project from a small Brazilian software development company (referred to as company β). Both companies provided a log from the executed process instances through spreadsheets, transformed in a CSV file to be used as input by PROV-Process. The data were then anonymized and stored in the relational database. Ten process execution instances from each company were used. The goal of the exploratory study was to evaluate if the goals established in Subsection 4.1 could be achieved with the visualization attributes implemented in SPPV.

The visualization of the provenance data from the first process is shown in Figures 2 and 3. By executing the proposed tasks using SPPV, the goals G1 and G2 were met as follows: (**G1)** Process agent *VB6_2* is overloaded, while agent *DotNet_5* performed few tasks (Figure 2 (b)); (**G2)** Client *Client_1* opened a much higher number of requests when compared with *Quality_3* and *Support_4* (Figure 3(b)).

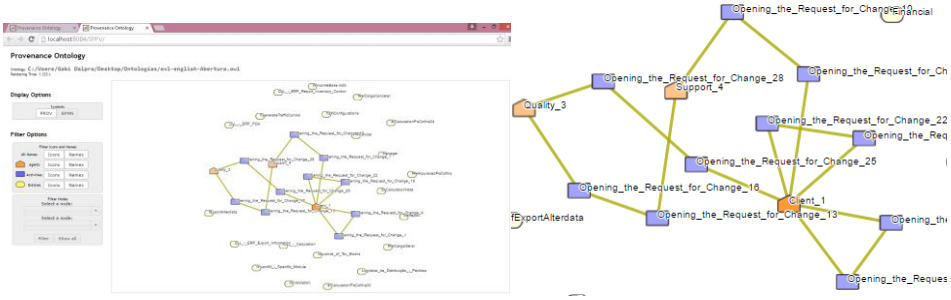
The visualization of the provenance data from the second process is shown in Figures 4 and 5. The goals set in **G1** and **G2** were achieved with SPPV as follows: (**G1)**: Process agent *Developer_4* is overloaded, while agents *Developer_6*, *Developer_7*, and

Developer_8 performed only one task (Figure 4(b)). (**G2**) Client *Client_E* did a much higher number of requests when compared to all the other clients (Figure 5(b)).



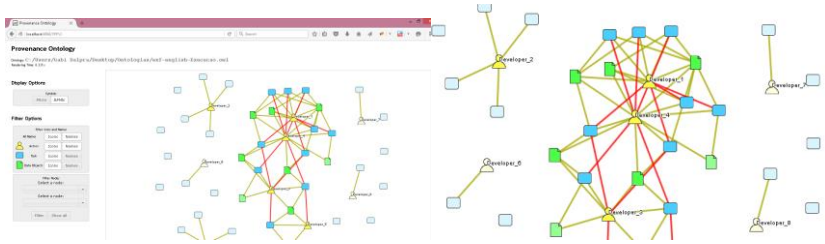
(a) (b)

Figure 2. Process 1 - visualization for **G1**: process agent overloaded.



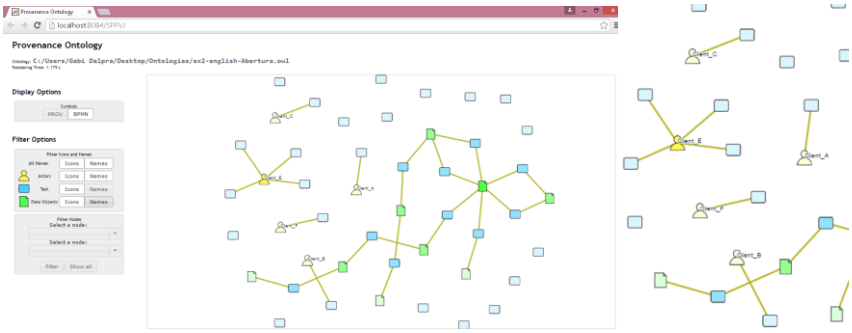
(a) (b)

Figure 3. Process 1 - visualization for **G2**: process requests started.



(a) (b)

Figure 4. Process 2 - visualization for **G1:** process agent overloaded.



(a) (b)

Figure 5. Process 2 - visualization for **G2**: process requests started.

A preliminary evaluation about the proposed tool was done through interviews composed by some specific questions with people who participate in the processes. From company α, the interviewee was involved in the process for five years as a

software developer. From company β, the interviewee was the process manager, who was involved in the process since its inception (for more than ten years).

When participants were asked about which person performs more tasks in their company processes in the last ten instances, **Rα** stated that "a way to do this is by executing a single query in our database," while **Rβ** affirmed, "the company uses Mantis, and I can find the information on it." Participants were also asked to identify the agents who were most overloaded, based on SPPV. Both answered correctly. They were then asked if they could see something in this visualization, besides the overload of tasks. **Rα** mentioned "the relation between the agents and system artifacts" and **Rβ** said, "tasks that involved more than one person for their resolution caught my attention. This may indicate that the task was complex or required several revisions until the process was completed." The last question about **G1** was: "By knowing that an agent is overloaded with tasks, what actions would you take?" **Rα** answered, "one possible action is to analyze who has done few tasks in the process and try to delegate the tasks that have been made by the person who is overloaded," while **Rβ** said that "this type of information may indicate that (...) either the person who is apparently overloaded needs his/her responsibilities to be reviewed, or maybe this person is actually more efficient than the rest of the team and, thus, has done more tasks than the others." These answers allow concluding that the definition of what to do with this information depends on the prior knowledge of the person who is performing the analysis.

Related to **G2**, participants were asked about which client performs more requests, without using SPPV. **Rα** said that he could obtain this information through the sales sector contact, and **Rβ** said that he cannot obtain this type of information. Both participants correctly identified clients who performs more requests. By seeing a client who performs more requests than others, **Rα** said that he would "analyze why this is happening," and **Rβ** answered that "this information is important because we can analyze if the client who has opened more requests in last executions has a system version that presents many problems, or if his/her requirements are related to new implementations to this system; if the latter is true, the company's commercial sector needs to be contacted for a possible re-examination of this client's service pack."

When participants were asked about what they liked in SPPV and what they would change, **Rα** answered that "the relations between nodes are not clear (...); besides, the tool does not need all types of symbols (BPMN and PROV). One is enough." **Rβ** said that "the SPPV visualization using the proposed graphs and symbols is interesting. The information discovered using the visualization might be useful to increase the company's revenues"; he added, "the number of relations of each edge should be displayed. One visualization – BPMN or PROV – is enough, as both show the same things. A filter to select actors from a specific team would be interesting, too."

6. Conclusion

This paper described the usage of a goal-oriented mapping process for planning SPPV, a tool for visualizing software process provenance data. An exploratory analysis using two real-world industry processes was carried out, with a positive feedback from the process experts. Although the exploratory analysis was performed in a real world context, it cannot be generalized to the context of other industry software processes.

As future work, additional goals and attributes (e.g., time spent on activities) based on provenance data can be defined with their respective visualization(s). We also intend to: (1) create filters to select actors from a specific team (as suggested by a study participant), (2) include a timeline in the proposed tool (in order to understanding how agents and activities behave over time), and (3) analyze SPPV's visual scalability.

Acknowledgments

We would like to thank CEOsoftware and Projetus TI, for kindly sharing their data and providing feedback to our research, and CNPq and FAPERJ, for their financial support.

References

Belhajjame, K., BFar, R., Cheney, J., Coppens, S., Cresswell, S., Gil, Y., Groth, P., Klyne, G., Lebo, T., McCusker, J., Miles, S., Myers, J., Sahoo, S., Tilmes, C., Moreau, L. and Missier P. (2013) "PROV-DM: The PROV Data Model", W3C.

Chen, P., Plale, B. (2015) "Big Data Provenance Analysis and Visualization", International Symposium on Cluster, Cloud and Grid Computing, pp. 797-800.

Costa, G. C. B. (2016) "Using Data Provenance to Improve Software Process Enactment, Monitoring and Analysis", International Conference on Software Engineering (ICSE 2016), Austin, USA, pp. 875-878.

Dalpra, H. L. O., Costa, G., Sirqueira, T. F. M., Braga, R., Werner, C. M., Campos, F., David, J. M. N. (2015) "Using Ontology and Data Provenance to Improve Software Processes", Proceedings of the Brazilian Seminar on Ontologies, pp. 10-21.

Ellson J., Gansner, E., Koutsofios, L., North, S. C., Woodhull, G. (2001) "Graphviz - open source graph drawing tools", Springer Berlin Heidelberg, pp. 483-484.

Falbo, R., Bertollo, G. (2005) "Establishing a Common Vocabulary for Helping Organizations to Understand Software Processes", Proceedings of the 1st VORTE, Enschede, The Netherlands.

Hoekstra, R., Groth, P. (2014) "PROV-O-Viz-understanding the role of activities in provenance," Provenance and Annotation of Data and Processes", Springer International Publishing, pp. 215-220.

Moreau, L., Groth, P., Cheney, J., Lebo, T., Miles, S. (2015) "The rationale of PROV", Web Semantics: Science, Services and Agents on the World Wide Web.

Murta, L., Braganholo, V., Chirigati, F., Koop, D., Freire, J. (2014) "noWorkflow: Capturing and analyzing provenance of scripts", Proceedings of the Provenance and Annotation of Data and Processes. Springer International Publishing, pp. 71-83.

OMG (2011) "Business Process Model and Notation - Version 2.0", Technical Report.

Schots, M., Werner, C. (2015) "On Mapping Goals and Visualizations: Towards Identifying and Addressing Information Needs", III Workshop de Visualização, Evolução e Manutenção de Software (VEM 2015), Belo Horizonte, Brasil.

Wendel, H., Kunde, M., Schreiber, A. (2010) "Provenance of software development processes", Provenance and Annotation of Data and Processes, Lecture Notes in Computer Science, v. 6378, Springer Berlin Heidelberg, pp. 59-63.

Um Estudo em Larga Escala sobre Estabilidade de APIs

Laerte Xavier, Aline Brito, André Hora, Marco Tulio Valente

[1]ASERG, Departamento de Ciência da Computação (DCC)
Universidade Federal de Minas Gerais (UFMG), Brasil

{laertexavier,hora,mtov}@dcc.ufmg.br, alinebrito@ufmg.br

Abstract. *APIs are constantly being evolved. As a consequence, their clients are compelled to update and, thus, benefit from the available improvements. However, some of these changes may break contracts previously established, resulting in compilation errors or behavioral changes. In this paper, questions related to API evolution and stability are studied, with the purpose of characterizing (i) the frequency of changes inserted, (ii) the behavior of these changes among time and (iii) the impact in client applications. Therefore, the top-100 GitHub most popular Java libraries are analyzed, and so their possible clients. As a result, insights are provided for the development of tools to support both library developers and clients in evolution and maintenance activities.*

Resumo. *APIs estão em constante evolução. Como consequência, seus clientes são compelidos a atualizarem-se e, assim, aproveitarem as melhorias disponibilizadas. Entretanto, algumas dessas mudanças podem quebrar contratos previamente estabelecidos, resultando em erros de compilação ou mudanças comportamentais. Neste artigo, estudam-se questões relativas a evolução e estabilidade de APIs, visando caracterizar (i) a frequência de mudanças inseridas, (ii) o comportamento dessas mudanças ao longo do tempo e (iii) o impacto em aplicações clientes. Dessa forma, foram analisadas as 100 bibliotecas Java mais populares do GitHub, bem como seus possíveis clientes. Como resultado, são apresentados insights de ferramentas para auxiliar ambos os clientes e os desenvolvedores de bibliotecas em suas atividades de evolução e manutenção.*

1. Introdução

Em desenvolvimento de *software*, mudança é uma constante. Estima-se que 80% do custo desse processo está relacionado às fases de manutenção e evolução. Nesse contexto, APIs (*Application Programming Interfaces*) também estão susceptíveis a atualizações e, como consequência, aplicações clientes são naturalmente compelidas a migrarem para novas versões. Compatibilidade torna-se, então, um desafio relevante na evolução de APIs, uma vez que mudanças introduzidas em uma nova versão podem quebrar contratos previamente assumidos com milhares de clientes.

Dessa forma, as mudanças em APIs podem ser classificadas em ***breaking changes:*** quebram compatibilidade com versões anteriores, alterando ou removendo elementos previamente disponíveis; e ***non-breaking changes:*** não quebram compatibilidade e, em geral, incluem adição de novos componentes ou extensão de funcionalidades [Dig and Johnson 2006]. Assim, diversos trabalhos utilizam essa classificação com o objetivo de analisar questões sobre a evolução e a estabilidade de

APIs [Raemaekers et al. 2012, McDonnell et al. 2013]. No entanto, nota-se que esses estudos são realizados apenas em pequena escala, ou seja, num contexto onde poucos sistemas são avaliados. Além disso, eles possuem uma importante restrição em suas validações, uma vez que não avaliam o impacto das *breaking changes* nos reais interessados, i.e., as aplicações clientes.

Neste trabalho, essas questões são analisadas em larga escala, relacionando os resultados obtidos ao possível impacto produzido em clientes reais. Tem-se como objetivo medir *(i)* a frequência de mudanças entre versões de bibliotecas, *(ii)* o comportamento dessas mudanças ao longo do tempo, e *(iii)* o impacto em aplicações clientes. Especificamente, são propostas três questões de pesquisa centrais:

QP #1. Qual a frequência de mudanças de bibliotecas?
QP #2. Como a frequência de *breaking changes* se comporta ao longo do tempo?
QP #3. Qual o impacto real das *breaking changes* em aplicações clientes?

Para responder a essas questões, são analisadas as 100 bibliotecas Java mais populares do GitHub, bem como potenciais clientes afetados por alterações em seus componentes. Assim, as principais contribuições deste trabalho são: *(i)* prover dados e análises para auxiliar clientes a escolherem bibliotecas baseados no critério de estabilidade, *(ii)* prover bases para o desenvolvimento de uma ferramenta que mitigue os riscos de migração entre versões de bibliotecas, e *(iii)* fornecer *insight* de ferramenta que alerte os desenvolvedores de bibliotecas acerca do impacto de possíveis mudanças pretendidas.

O restante deste artigo está organizado da seguinte forma: a Seção 2 aprofunda as definições de *breaking change* e *non-breaking change*, apresentando o catálogo de mudanças utilizado. Na Seção 3 é descrita a metodologia dos estudos realizados e na Seção 4 são apresentados os resultados obtidos. A Seção 5 discute os resultados, apresentando aplicações práticas. Na Seção 6 são apresentados os riscos à validade deste estudo e, por fim, as Seções 7 e 8 discutem trabalhos relacionados e as conclusões obtidas.

2. Catálogo de Mudanças em APIs

APIs são definidas como componentes de um sistema de *software* que podem ser reusados por aplicações clientes. Utilizam-se, portanto, de modificadores de visibilidade para expor interfaces ditas estáveis (e.g., em Java utiliza-se `public` ou `protected`). Durante seu ciclo de vida, entretanto, estão sujeitas a mudanças evolutivas, tais como adição, remoção, modificação ou depreciação de seus elementos. Essas mudanças foram catalogadas em estudos anteriores no contexto de *refactoring* [Dig and Johnson 2006] e são classificadas em *breaking change* e *non-breaking change*. Neste trabalho, utiliza-se tal classificação, aplicando o catálogo proposto sem as referências a refatoração.

Dessa forma, são consideradas ***breaking changes*** alterações abruptas que quebram contratos estabelecidos, sem notificação prévia por meio de mensagens de depreciação. São elas: remoção de elementos (tipos, atributos ou métodos); modificação da assinatura de métodos (nome, visibilidade e tipos de retorno, de exceção ou de parâmetro); e alterações em atributos (tipos e valores de inicialização). Por outro lado, consideram-se ***non-breaking changes*** aquelas mudanças inseridas em elementos depreciados, ou aquelas que adicionam novas funcionalidades, mas mantém compatibilidade com clientes. Em geral, representam adição de elementos ou funcionalidades às bibliotecas.

3. Metodologia

Seleção de Sistemas. A fim de selecionar os sistemas cujas APIs serão analisadas, utilizou-se uma base de dados desenvolvida em trabalhos anteriores [Brito et al. 2016]. Nessa base, foram classificados em *Library* e *Non-Library* 623 repositórios Java hospedados no GitHub, escolhidos e ordenados pelo número de estrelas. Desses, foram selecionados para análise no presente estudo os 100 sistemas mais populares com classificação de *Library*. Dessa forma, os sistemas analisados neste trabalho possuem, na mediana, 1.190,5 estrelas e 23,5 *releases*. Entre os melhores classificados, destacam-se: NOSTRA13/ANDROID-UNIVERSAL-IMAGE-LOADER com 9.793 estrelas e 28 *releases*; SQUARE/PICASSO com 6.948 estrelas e 20 *releases*; e LIBGDX/LIBGDX, com 6.839 estrelas e 29 *releases*.[1]

Extração de Métricas. Com o propósito de responder às questões de pesquisa propostas, foram coletadas as frequências de ocorrência das *breaking changes* e *non-breaking changes* descritas na Seção 2. Para tanto, foi implementado um *parser* baseado na biblioteca `JDT` do Eclipse que compara tais mudanças entre as versões analisadas. Sendo N a última *release* lançada do sistema, e 1 a primeira, na *QP #1* essas frequências foram comparadas entre as versões N e $N-1$ (finais). Já para a *QP #2*, estendeu-se a coleta para as versões 1 e 2 (iniciais) e $N/2$ e $N/2-1$ (intermediárias). Assim, observa-se que tais *releases* constituem momentos representativos do ciclo de desenvolvimento das bibliotecas (iniciais, intermediários e finais), mitigando o alto custo de análise de todo histórico de versões.

Por outro lado, para responder a *QP #3*, utilizou-se o `JAVALI`[2], uma ferramenta para comparar e categorizar APIs Java de acordo com o uso por aplicações clientes. Ela utiliza um *dataset* com 263.425 projetos e 16.386.193 arquivos Java hospedados no GitHub e cujas informações foram extraídas utilizando a linguagem e infraestrutura `Boa` [Dyer et al. 2013]. Os clientes das bibliotecas em estudo foram, então, identificados através da análise das declarações de *import* nos projetos disponíveis no `Boa`, buscando por referências aos tipos modificados. Dessa forma, foram analisadas as quantidades de projetos e arquivos possivelmente impactados pelos tipos que sofreram *breaking change* em pelo menos um de seus elementos nas versões finais das bibliotecas analisadas.

4. Resultados

QP #1: Qual a frequência de mudanças de bibliotecas?

A frequência de mudanças foi calculada para tipos, atributos e métodos entre as duas últimas versões das 100 bibliotecas analisadas. Desse total, 81 sistemas apresentaram pelo menos uma *breaking change*, somando 70.040 mudanças (23,3%). Por outro lado, 230.064 *non-breaking changes* foram observadas em 82 sistemas (76,7%). Com o objetivo de estudar a estabilidade dessas bibliotecas nas versões mais recentes, foram analisados os valores absolutos para ambos os tipos de mudanças e, em seguida, avaliados os valores percentuais daquelas mudanças que afetam aplicações clientes.

A distribuição dos valores absolutos registrados para cada elemento de API é apresentada na Figura 1. O primeiro quartil, a mediana e o terceiro quartil de *breaking changes* em tipos são iguais a 0, 1 e 25,5. Para mudanças em atributos, o primeiro quartil é 0, a

[1] A lista completa dos sistemas avaliados está disponível em: `https://goo.gl/VaQ8jI`

[2] `http://java.labsoft.dcc.ufmg.br/javali`

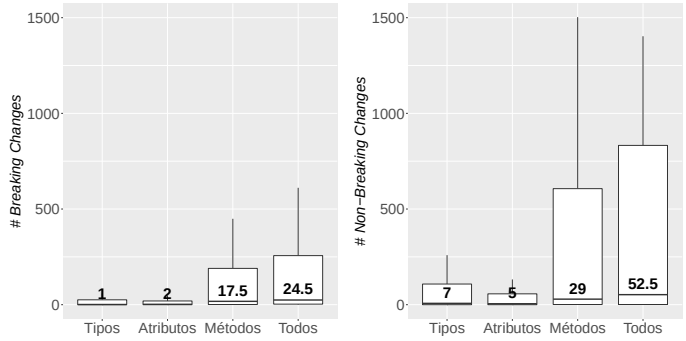


Figura 1. Total de mudanças na versão final dos sistemas analisados.

mediana, 2, e o terceiro quartil, 19,5. *Breaking changes* em métodos apresentam ainda primeiro quartil igual a 1, mediana igual a 17,5, e terceiro quartil 189,7. Por outro lado, a frequência de *non-breaking changes* em tipos tem primeiro quartil 0, mediana 7 e terceiro quartil 108. Ainda nesse contexto, atributos apresentam primeiro quartil, mediana e terceiro quartil iguais a 0, 5 e 56,7. Para métodos, observa-se que o primeiro e terceiro quartis são, respectivamente, 1 e 606,5, com mediana 29. Por fim, considerando todos os elementos de API, os valores obtidos são: 3, 24,5 e 256,5, para *breaking changes*; e 1, 52,5 e 832,5, para *non-breaking changes*.

Em termos percentuais, a Figura 2 apresenta a distribuição das taxas de ocorrência de *breaking changes* nos elementos de API das bibliotecas em estudo, em relação ao seu total de mudanças. Dessa forma, considerando todos os elementos analisados, observa-se uma elevada taxa de 24,6% de mudanças desse tipo, com destaque para atributos e métodos, com valores medianos correspondentes a 22,1% e 21,3%, respectivamente. Por fim, verifica-se que apenas 5,7% das mudanças em tipos são *breaking changes*.

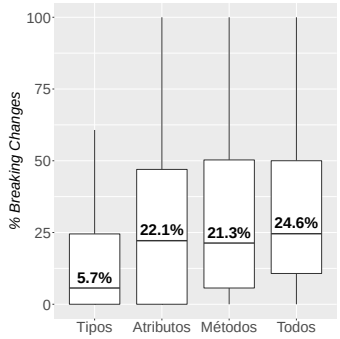


Figura 2. Taxa de *breaking changes* na versão final dos sistemas analisados.

Resumo: Em valores medianos, 24,6% das mudanças que ocorrem em APIs afetam aplicações clientes, isto é, são *breaking changes*. Ou seja, um quarto das interfaces consideradas estáveis podem afetar seus clientes, causando problemas de compatibilidade.

QP #2: Como a frequência de breaking changes se comporta ao longo do tempo?

A fim de analisar a estabilidade durante o ciclo de vida das 100 bibliotecas em estudo, foram calculadas as frequências de *breaking changes* entre as versões iniciais e inter-

mediárias, comparando-as com os valores finais obtidos na *QP #1*. Entre as versões iniciais, foram encontradas 76.047 mudanças desse tipo. Por outro lado, 73.053 e 70.040 foram registradas nas versões intermediárias e finais, respectivamente.

A Figura 3 apresenta a distribuição do percentual de *breaking changes* em cada elemento de API, em relação ao seu total de mudanças, nas versões iniciais, intermediárias e finais. Para mudanças em tipos, observa-se que os valores obtidos nas medianas são iguais a 13,6%, 18,6% e 5,7%. Já para atributos, a mediana entre versões da fase inicial é 29,2%, 33,3% da intermediária e 22,1% da final. Por fim, nota-se que as medianas para métodos em cada uma das fases analisadas são, respectivamente: 26,6%, 36,7% e 21,3%.

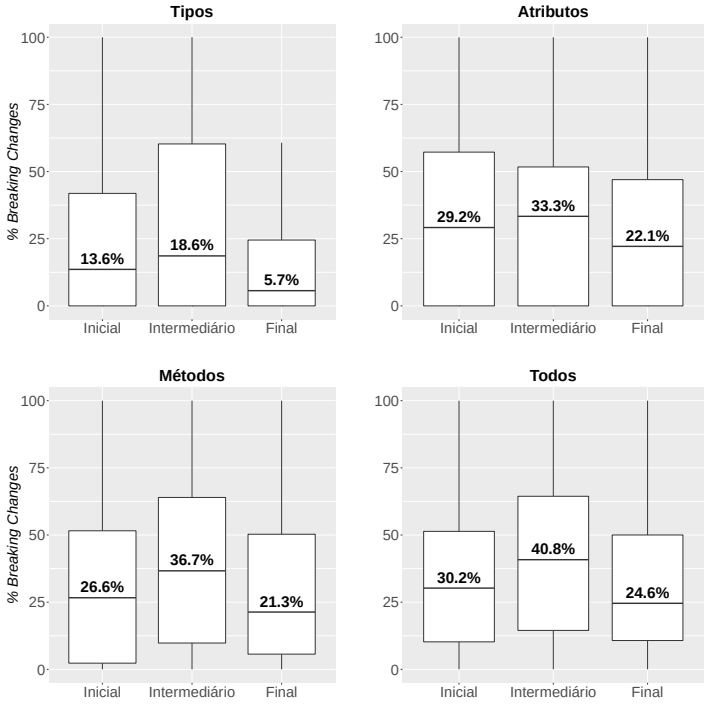


Figura 3. Taxa de *breaking changes* nas versões inicial, intermediária e final.

Considerando todos os elementos de API, observa-se que a taxa de *breaking changes* tende a aumentar entre as fases inicial e intermediária, com medianas iguais a 30,2% e 40,8%, respectivamente. Entretanto, observa-se também que, na segunda metade do seus tempos de vida, esses valores diminuem de maneira que a taxa de mudanças entre as versões intermediária e final possuem medianas iguais a 40,8% e 24,6%. Em ambos os casos, foram obtidos *p-values* $< 0{,}05$ no teste de Mann-Whitney. Isto é, existe uma diferença estatística na taxa de *breaking changes* entre as versões analisadas.

Resumo: Na primeira metade do tempo de vida das APIs, o percentual de *breaking changes* aumenta; na metade final, observa-se uma redução relativa desse tipo de mudança. Isso mostra que interfaces fornecidas por bibliotecas tendem a ficar mais estáveis.

QP #3: Qual o impacto real das breaking changes em aplicações clientes?

Com o propósito de mensurar o possível impacto das *breaking changes* em aplicações clientes, foram sumarizados os tipos que sofreram essas alterações nas versões finais das

bibliotecas estudadas. Em seguida, minerou-se, a partir dos 263 mil projetos e 16 milhões de arquivos do `JAVALI`, o número de clientes de tais tipos em termos de projetos e arquivos. Do total de 70.040 *breaking changes* registradas em todos os elementos de API, observa-se que tais mudanças estão relacionadas a 10.206 tipos. Desses, 4.024 possuem pelo menos uma aplicação cliente que foi, possivelmente, afetada.

Na Figura 4, apresenta-se a distribuição da quantidade de projetos e arquivos impactados por tipos que sofreram alguma *breaking change* e possuem pelo menos uma aplicação cliente. No primeiro quartil, verifica-se que 1 projeto e 3 arquivos são possivelmente afetados. Na mediana, verifica-se que esses valores são iguais a 4 e 11, respectivamente. Por fim, no terceiro quartil, observam-se os totais de 11 projetos e 37 arquivos.

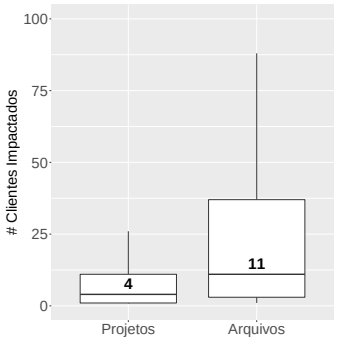


Figura 4. Projetos e arquivos impactados por *breaking changes* na versão final.

Apesar dos tipos que sofreram *breaking changes* nas APIs das bibliotecas estudadas não possuírem, no geral, quantidades elevadas de clientes, alguns casos merecem destaque. Por exemplo, o tipo `org.hibernate.Query` da biblioteca HIBERNATE/HIBERNATE-ORM, que é utilizado em 3.594 projetos e 20.897 arquivos clientes. Nesse sistema, entre as *releases* 4.2.23 e 5.2.0 foram inseridas um total de 26 *breaking changes*, algumas delas reportadas pelos usuários. Essas mudanças geraram a criação de uma *issue* de alta prioridade que acarretou na restauração do código legado menos de 30 dias após o lançamento da nova versão.[3]

Resumo: Em valores medianos, 4 projetos e 11 arquivos clientes são possivelmente impactados pelas *breaking changes* nas bibliotecas analisadas. O baixo impacto nos clientes fornece indícios de que os tipos alterados são internos, ou que os desenvolvedores são cautelosos quando introduzem *breaking changes* em certos elementos.

5. Aplicações Práticas

A partir dos resultados obtidos, observa-se que clientes devem tomar precauções antes de escolherem uma biblioteca, bem como antes de decidirem atualizar a versão daquelas que já utilizam. Por outro lado, seus desenvolvedores devem ser cautelosos antes de produzirem *breaking changes* em elementos muito utilizados. Dessa forma, evidenciam-se as seguintes aplicações práticas:

[3]Maiores detalhes sobre a *issue* podem ser encontrados em: `https://hibernate.atlassian.net/browse/HHH-10839`

Aplicação #1: A fim de auxiliar clientes a decidirem a respeito da utilização de uma determinada biblioteca, as métricas utilizadas neste trabalho podem ser estendidas para o histórico completo de versões, construindo-se uma curva que exponha o comportamento evolutivo de uma biblioteca em termos de *breaking changes*, e forneça uma visão geral da estabilidade do sistema.

Aplicação #2: Uma ferramenta de análise de impacto com o propósito de auxiliar o cliente a mensurar o custo de migração entre versões de bibliotecas. Para tanto, deve-se observar quais elementos sofreram *breaking changes* entre a versão atual do cliente e a que se pretende atualizar; em seguida, analisar o código do cliente a fim de observar quais dessas mudanças o impactam diretamente.

Aplicação #3: Uma ferramenta de análise de mudanças cujo objetivo seja alertar os desenvolvedores de biblioteca a respeito do impacto de possíveis *breaking changes*. Dessa forma, antes de fazer *commit* de alguma modificação que possivelmente afete pelo menos um cliente externo, um alerta seria produzido informando o impacto daquela mudança e sugerindo a utilização de mensagens de depreciação.

6. Ameaças à Validade

Validade Externa. Este estudo limitou-se à analise de 100 bibliotecas *open source* Java. Portanto, seus resultados não podem ser generalizados para outras linguagens ou para sistemas comerciais. Entretanto, foram selecionados sistemas relevantes, com alta popularidade, e classificados como *Library*, aumentando, assim, a relevância dos resultados.

Validade Interna. Dentre os aspectos que podem afetar os resultados apresentados, destaca-se o *parser* implementado para contabilizar as *breaking changes*. A fim de minimizar essa ameaça, utilizou-se a biblioteca `JDT` do Eclipse. Além disso, o teste de Mann-Whitney foi utilizado com o objetivo de assegurar a validade das análises.

Validade de Construção. Para cada um dos sistemas em estudo, foram analisadas seis versões, em três fases distintas: inicial, intermediária e final. Entretanto, observa-se que essa análise não caracteriza todo o seu desenvolvimento. Dessa forma, não se pode afirmar que os resultados apresentados refletem a evolução completa dos sistemas analisados.

7. Trabalhos Relacionados

Diversos trabalhos abordam questões sobre evolução de APIs, propondo ferramentas de apoio aos desenvolvedores [Hora and Valente 2015, Hora et al. 2014, Henkel and Diwan 2005], e fornecendo melhor entendimento acerca das características de mudanças [Dig and Johnson 2006]. Por outro lado, existe um esforço no sentido de caracterizar e mensurar a estabilidade de APIs [Raemaekers et al. 2012], apresentando-se evidências do impacto de *breaking changes* em aplicações clientes [Brito et al. 2016, McDonnell et al. 2013, Robbes et al. 2012].

No entanto, nota-se que esses estudos são realizados apenas em pequena escala, num contexto onde poucos sistemas são avaliados. Além disso, eles possuem uma importante restrição em suas validações, uma vez que não avaliam o impacto das *breaking changes* nas aplicações clientes. Este trabalho avança o entendimento dessas questões, a partir de um estudo em larga escala, e da análise do impacto nos clientes, i.e., 100 bibliotecas, 263 mil sistemas clientes e 16 milhões de arquivos clientes.

8. Conclusões

Neste estudo, caracterizou-se a estabilidade de APIs através da análise de ocorrência de *breaking changes* em 100 bibliotecas *open source* Java. Verificou-se que uma quantidade relevante de alterações correspondem a esse tipo de mudança (24,6%). Observou-se, ainda, que esses valores são maiores em versões anteriores (30,2% na inicial e 40,8% na intermediária), sugerindo uma possível estabilização com o passar do tempo. Por fim, constatou-se que o impacto dessas mudanças é relativamente baixo (4 projetos e 11 arquivos), fornecendo indícios de que os tipos usualmente alterados são internos, ou que os desenvolvedores são cautelosos quando introduzem *breaking changes* em elementos muito utilizados. A partir desses resultados, três aplicações práticas foram brevemente apresentadas com o objetivo de auxiliar tanto clientes como desenvolvedores de bibliotecas em suas atividades de evolução e manutenção.

Agradecimentos

Esta pesquisa é financiada pela FAPEMIG, e pelo CNPq.

Referências

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 *23rd International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 360–369.

Dig, D. and Johnson, R. (2006). How do APIs evolve? A story of refactoring. In *22nd International Conference on Software Maintenance (ICSM)*, pages 83–107.

Dyer, R., Nguyen, H. A., Rajan, H., and Nguyen, T. N. (2013). Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *35th International Conference on Software Engineering (ICSE)*, pages 422–431.

Henkel, J. and Diwan, A. (2005). Catchup!: Capturing and replaying refactorings to support API evolution. In *27th International Conference on Software Engineering (ICSE)*, pages 274–283.

Hora, A., Etien, A., Anquetil, N., Ducasse, S., and Valente, M. T. (2014). APIEvolutionMiner: Keeping API evolution under control. In *IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), Tool Demonstration Track*, pages 420–424.

Hora, A. and Valente, M. T. (2015). apiwave: Keeping track of API popularity and migration. In *31st IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 321–323.

McDonnell, T., Ray, B., and Kim, M. (2013). An empirical study of API stability and adoption in the Android ecosystem. In *29th International Conference on Software Maintenance (ICSM)*, pages 70–79.

Raemaekers, S., van Arie Deursen, and Visser, J. (2012). Measuring software library stability through historical version analysis. In *28th International Conference on Software Maintenance (ICSM)*, pages 378–387.

Robbes, R., Lungu, M., and Rothlisberger, D. (2012). How do developers react to API deprecation? The case of a smalltalk ecosystem. In *20th International Symposium on the Foundations of Software Engineering (FSE)*, pages 56:1–56:11.

Uma Análise Preliminar de Projetos de Software Livre que Migraram para o GitHub

Luiz Felipe Dias[1], Igor Steinmacher[1], Igor Wiese[1], Gustavo Pinto[2], Daniel Alencar da Costa[3], Marco Gerosa[4]

[1]Universidade Tecnológica Federal do Paraná (UTFPR)
Campo Mourão – PR – Brasil

[2]Instituto Federal do Pará (IFPA)
Santarém – PA - Brasil

[3]Universidade Federal do Rio Grande do Norte (UFRN)
Natal – RN – Brasil

[4]Universidade de São Paulo (USP)
São Paulo – SP – Brasil

luizdias@alunos.utfpr.edu.br,{igorfs, igor}@utfpr.edu.br

gustavo.pinto@ifpa.edu.br,danielcosta@ppgsc.ufrn.br,gerosa@ime.usp.br

Abstract. *Social coding environments such as GitHub and Bitbucket are changing the way software is built. Not surprisingly, several mature, active, non-trivial open-source software projects are switching their decades of software history to these environments. There is a belief that these environments have the potential of attracting new contributors to open-source projects. However, there is little empirical evidence to support these claims. In this paper, we quantitatively studied a curated set of open-source projects that migrated to GitHub, aiming at understanding whether this migration fostered collaboration. Our results suggest that although interaction in some projects increased after migrating to GitHub, the rise of newcomers is not straightforward. In this preliminary analysis, we could not assess causality, and there may be factors unrelated to the migration influencing the rise of contributors and contributions.*

Resumo. *Ambientes sociais de codificação tais como GitHub e Bitbucket estão mudando a maneira de construir software. Sem grandes surpresas, vários projetos ativos, não triviais e de código aberto, estão migrando suas décadas de história de software para estes ambientes. Há uma crença de que estes ambientes possuem o potencial de atrair novos contribuidores para projetos de código aberto. No entanto, há pouca evidência empírica para apoiar estas alegações. Neste artigo, nós estudamos um conjunto específico de projetos de software livre que migraram para o GitHub, com o objetivo de entender se esta migração promove a colaboração. Nossos resultados sugerem que, embora a interação em alguns projetos aumente após a migração para o GitHub, o aumento do número de novatos não foi tão relevante. Nesta análise preliminar não foi possível estabelecer relações de causa e efeito dos fenômenos, podendo existir fatores não relacionados à migração que influenciam o aumento na quantidade de contribuidores e contribuições.*

1. Introdução

Projetos de software livre introduziram ao desenvolvimento de software uma nova perspectiva quanto ao conceito de contribuição. Voluntariamente, desenvolvedores ao redor do mundo contribuem a projetos de software livre da maneira que podem, de forma a somar seus esforços para a evolução de uma comunidade, de modo geral, ou de um software, em particular. Com o crescimento desta forma de manifestação, novas plataformas também começaram a surgir para melhor apoiar a atividade de desenvolvimento de software livre. Dentre suas principais mudanças, tais ambientes passam a mudar o modo ao qual desenvolvedores de software se comunicam, colaboram e contribuem com projetos de código aberto [Pham et al. 2013a]. Esses ambientes são chamados de ambientes sociais de codificação [Tsay et al. 2014, Pham et al. 2013a, Thung et al. 2013] oferecem aos desenvolvedores algumas funcionalidades de redes sociais, como menções e perfil e a possibilidade de compartilhar as atividades realizadas, seguir atividades de outros desenvolvedores e/ou projetos em um único ambiente web [Thung et al. 2013].

Um dos exemplos mais conhecidos desse tipo de ambiente é o GitHub, que é um serviço de hospedagem para projetos de software livre (gratuito) e proprietários (pago). Este serviço é disponível para projetos que utilizam o sistema de controle de versão *Git*. Contando com mais de 38 milhões de repositórios hospedados e 15 milhões de usuários cadastrados[1], GitHub é considerado não só o maior *website* para hospedagem de códigos do mundo, mas também um dos mais ricos, quando se refere as suas funcionalidades sociais. Além disso, o GitHub é frequentemente usado em estudos recentes de engenharia de software (por exemplo, [Pinto et al. 2016], [Moura et al. 2015], [Tsay et al. 2014]).

Apesar de sua notoriedade, o GitHub não é o único ambiente de codificação utilizado por desenvolvedores de software. Existem diversas opções, como o *CodePlex*, específico para tecnologias Microsoft, e o *BitBucket*, que permite trabalhar com diferentes sistemas de versionamento de código. No entanto, nos últimos meses, tem sido comum a migração de vários sistemas de versionamento para o GitHub, em particular, devido a sua notória popularidade, bem como sua funcionalidades sociais que facilitam a criação e gerenciamento times colaborativos.

O objetivo desse estudo é investigar, quantitativamente, a influência da migração de projetos de software livre para o GitHub, em particular, com relação ao número de contribuintes e de contribuições realizados nestes projetos. Para isso, selecionamos projetos representativos que antes eram hospedados em repositórios tradicionais de codificação, como o *Sourceforge*, e que migraram para o GitHub no decorrer do seu ciclo de vida. Para guiar esta pesquisa, a principal pergunta de pesquisa é:

> ***Q:*** *Quanto o processo de migração para ambientes sociais de codificação impacta na entrada de novos contribuintes e no número de contribuições?*

Para responder à questão de pesquisa, utilizamos dados e meta-dados adquiridos dos repositórios, e buscamos entender se esse processo de migração beneficiou os projetos. Com isso, analisamos número de novos contribuidores, número de contribuidores ativos e contribuições realizadas.

[1] `https://GitHub.com/about/press`

2. Trabalhos Relacionados

Existem trabalhos na literatura que analisam ambientes sociais de codificação. Os trabalhos em questão discutem, sob diferentes perspectivas, os possíveis fatores que possam impactar na migração de projetos. Quanto a este contexto, gostaríamos de ressaltar trabalhos relacionados à: análise de fatores sociais na retenção de novatos, características sociais presentes no GitHub, e o fenômeno dos contribuidores casuais.

Influência de Fatores Sociais sobre a retenção de novatos: Alguns estudos na literatura estão focados em analisar a influência de fatores sociais sobre a retenção de novatos em projetos de software livre ([Steinmacher et al. 2015, Zhou and Mockus 2015, Ducheneaut 2005, Bird 2011]). A fim de entender, por meio das redes sociais (e.g., extraídas de listas de emails), com quem os novatos colaboram, e como estas redes evoluíram ao longo dos anos. Jensen et al. (2011) analisaram quatro projetos para entender se novatos costumam ser respondidos rapidamente, se a idade ou nacionalidade dos mesmos impacta no tipo de resposta que recebem e se o tratamento recebido é similar aos dos demais membros do projeto. Apesar dos estudos se concentrarem na relação de aspectos sociais quanto a retenção de novatos, eles não analisam os ambientes sociais de codificação como um meio que possibilite novas contribuições.

Características Sociais e o GitHub: Diversos são os estudos voltados a aspectos sociais no GitHub. Marlow et al. (2013) encontraram evidência de que desenvolvedores utilizam de sinais presentes nos perfis do GitHub, tais como habilidades e relacionamentos, para formar primeiras impressões de usuários e projetos. Dabbish et al. (2012) investigaram a influência existente no comportamento de usuários do GitHub, e relataram que o número de observadores em um projeto é um fator que pode atrair novos desenvolvedores. Na sequência, Tsay et al. (2014) evidenciaram que desenvolvedores usam tanto de informações técnicas como sociais para influenciar avaliações em projetos de software livre. O tamanho da comunidade também foi evidenciado como possível indicador de sucesso em projetos de software livre. McDonald e Goggins (2013), ao entrevistar mantenedores de projetos no GitHub, encontraram que as funcionalidades oferecidas pelo GitHub são uma das principais razões do crescimento de contribuições em projetos de software livre. Esses estudos estão focados nos ambientes sociais de codificação como responsáveis pela atração de novos desenvolvedores e pela geração de sinais e impressões entre projetos e desenvolvedores. Entretanto, nenhum destes estudos investiga como a migração para ambientes sociais de codificação influencia a entrada de novatos, e o número de contribuições recebidas.

Contribuidores Casuais: Certos trabalhos exploram o fenômeno dos contribuidores casuais no contexto dos ambientes sociais de codificação. Vários autores tem reconhecido a existência e o crescimento deste comportamento [Pham et al. 2013b, Pham et al. 2013a, Gousios et al. 2014, Vasilescu et al. 2015, Pinto et al. 2016]. No entanto, estes trabalhos não analisam o impacto da migração de projetos de software livre para os ambientes sociais de codificação.

3. Método de Pesquisa

Nesta seção, são descritos os projetos selecionados (Seção 3.1), e o processo de coleta e análise dos repositórios (Seção 3.2).

3.1. Projetos

Entende-se por software livre aquele que respeita a liberdade e senso de comunidade dos usuários [Fogel 2013]. De maneira geral, os usuários devem possuir a liberdade de executar, copiar, distribuir, estudar, mudar e melhorar o software [Stallman 1999]. Para representar esta forma de manifestação, foram selecionados três projetos popularmente conhecidos, de domínios diferentes, inicialmente hospedados em um ambiente de desenvolvimento não-colaborativo, mas que migraram para o GitHub em algum momento no seu ciclo de vidas. Ademais, tais projetos são não-triviais, em termos do número de contribuição, contribuintes e tempo de vida. São eles:

- **`Ruby`**, linguagem de programação dinâmica e orientada à objetos. Lançada em 1998, migrou ao GitHub em fevereiro de 2010. Escrita principalmente nas linguagens C e Ruby.
- **`MongoDB`**, banco de dados orientado a documentos. Lançado em outubro de 2007, migrou ao GitHub em janeiro de 2009. Escrito principalmente em C++.
- **`Jenkins`**, serviço de integração contínua. Lançado em novembro de 2006, teve sua data de migração quatro anos após, em novembro de 2010. Escrito principalmente em Java.

Estes projetos foram escolhidos pois: são bem estabelecidos em suas respectivas comunidades, todos com mais de nove anos de existência. Contabilizam um total de contribuições por projeto satisfatório, os três superiores a dez mil. Estão abertos à *pull-requests* de terceiros. Contam com uma média de mais de trezentos contribuidores por projeto, aos quais uma média de 27 contribuidores permanecem ativos mensalmente, e somam em média 280 contribuições mensais. A Tabela 1 apresenta detalhes adicionais sobre os projetos selecionados.

Tabela 1. A diversidade de nossas aplicações alvo. LoC significa Linhas de Código. PR significa Pull Requests. A idade é apresentada em anos.

Projetos	Lançado em	Migrou em	LoC	Contribuidores	Contribuições	PR	Idade
`jenkins`	Nov. 2006	Nov. 2010	191K	556	21K	2K	10
`ruby`	Jan. 1998	Feb. 2010	1,001K	95	40K	1K	18
`mongodb`	Oct. 2007	Jan. 2009	2,104K	324	31K	1K	9

3.2. Coleta e Análise dos Repositórios

Para cada um dos três projetos, foram extraídos dados dos repositórios de código utilizando técnicas de mineração de logs. Entre os dados coletados estão: o número de novatos que ingressaram em cada projeto, o número de contribuições, e o número de contribuidores ativos. Definimos um contribuidor ativo como aquele que teve sucesso ao realizar ao menos uma contribuição ao repositório, sem distinções de tipo de arquivo alterado, seja ela uma contribuição em termos de código, documentação ou tradução. As contribuições são definidas como qualquer alteração de arquivos do projeto, descrita pela

documentação do GitHub como um *commit* ou *pull-request*. Já os novatos são definidos de forma similar aos contribuidores ativos, identificados a partir da data em que fizeram sua primeira contribuição. Por exemplo, um contribuidor é considerado novato na exata data em que o mesmo realizou sua primeira contribuição ao projeto, contribuições posteriores serão descartadas para essa métrica.

Após realizarmos a coleta dos dados, comparamos as distribuições de cada métrica coletada antes e após a migração de cada um dos projetos. Por exemplo, nós comparamos o número de novos contribuidores de um dado projeto antes e após a migração para o ambiente social de codificação. Sendo que, deste modo, nos tornamos capazes de verificar se a migração pode impactar na colaboração de projetos. Para uma melhor visualização do que foi coletado, criamos gráficos apresentando em três informações distintas: número de novatos, de contribuidores e contribuições.

Para avaliar nossas comparações, utilizamos os testes estatísticos não-paramétricos Mann-Whitney-Wilcoxon (MWW) e o delta de Cliff (tamanho do efeito). Os testes foram escolhidos porque as métricas coletadas antes e após a migração não seguiam uma distribuição normal. O teste MWW foi usado para verificar se duas distribuições de métricas são diferentes para um $\alpha = 0.05$. O delta de Cliff foi utilizado para dimensionar o tamanho da diferença das medidas coletadas antes e depois da migração. Quanto maior o valor do delta de Cliff obtido, maior é a diferença entre as distribuições. Para interpretar os resultados do tamanho da diferença foi utilizado a escala provida por [Romano et al. 2006]: *delta* < 0.147 (diferença insignificante), *delta* < 0.33 (diferença baixa), *delta* < 0.474 (diferença média), *delta* $>= 0.474$ (diferença alta).

4. Resultados

A Figura 1 apresenta uma visão geral dos dados coletados. Estes gráficos trazem uma perspectiva temporal de diferentes características pesquisadas. Por exemplo, a linha pontilhada em verde representa o número de contribuidores novatos que tiveram sucesso ao realizar no mínimo uma contribuição. A linha tracejada em azul representa o número de contribuições realizadas em todo o período de execução do projeto. A linha em vermelho representa o número de contribuidores ativos. E finalmente na vertical, a linha tracejada em preto indica a exata data em que cada projeto migrou para o GitHub. Além disto, trazemos através da Tabela 2, todos os resultados estatísticos obtidos nesta pesquisa (p-valor e de *effect size*).

Tabela 2. Resultados estatísticos. Células em verde indicam um alto tamanho de efeito, enquanto células em amarelo indicam um tamanho de efeito médio.

Projetos	Novatos		Contribuidores		Contribuições	
	p-value	$delta$	p-value	$delta$	p-value	$delta$
jenkins	0.001	0.131	2.66^{-06}	0.147	0.138	-0.031
ruby	0.489	-0.015	5.16^{-07}	0.106	2.20^{-16}	0.478
mongodb	0.178	0.143	1.41^{-07}	0.403	2.20^{-16}	0.710

Percebeu-se um aumento no número de contribuições. Em dois dos três projetos escolhidos foi possível notar um aumento significativo no número de contribuições. O que pode indicar que a migração de fato está relacionada a este crescimento. Estes resultados

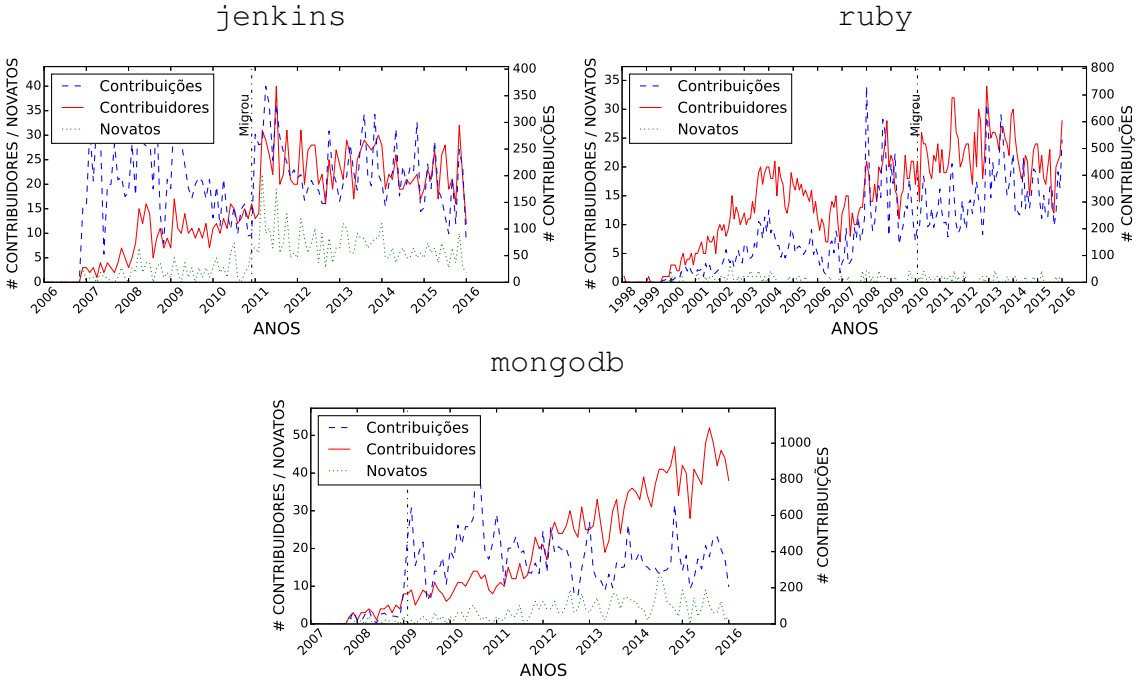


Figura 1. Gráfico de contribuições (linha azul tracejada), contribuidores (linha vermelha) e novatos (linha verde pontilhada), divididos em períodos antes e após a migração (linha tracejada em preto).

foram obtidos pelos projetos `mongodb` (p-valor = 2.20^{-16}, $delta$ = 0.710) e `ruby` (p-valor = 2.20^{-16}, $delta$ = 0.478). Estes resultados indicam que, ao comparar o cenário anterior e posterior a migração, existe uma relevante diferença em número de contribuições após a migração. Uma das hipoteses é que este crescimento esteja relacionado a aspectos sociais presentes no GitHub, que motivem de certa forma os desenvolvedores a contribuir. Entrentanto, não é possível generalizar essa conclusão, uma vez que o projeto `jenkins` apresentou um *effect size* insignificante ($delta$ = -0.031).

Existe uma maior frequência de contribuidores. Ademais, é observado que os gráficos do `jenkins` e do `mongodb` mostram um aparente crescimento em número de contribuidores ativos por período. Quanto mais contribuidores ativos em um projeto, possivelmente mais contribuições existirão. Entretanto, o efeito em termos estatísticos só foi notório no `mongodb` (*p-value* $= 1.41^{-07}$, $delta = 0.403$), que obteve um efeito considerado médio.

5. Ameaças à validade

Pode-se argumentar que são poucos os projetos de código aberto por nós analisados, o que portanto, limita a generalização de nossos resultados. Entretanto, os projetos de código aberto selecionados são de diferentes domínios, tamanhos e idades. Além disso, a intenção deste trabalho é explorar de forma preliminar o fenômeno.' Para trabalhos futuros pretende-se aumentar o tamanho da amostra analisada.

Outra ameaça para validade de nossa pesquisa está na forma como selecionamos os autores de *commits*. Visto que alguns dos projetos selecionados migraram de ambientes com outros sistemas de controle de versão, tais como `SVN`, que não distinguem autores de

committers. Neste caso, nós usamos o endereço de *email* para diferencia-los. No entanto, em repositórios `SVN` não existe a necessidade de se informar um *email* ao realizar uma contribuição, bem como um contribuidor pode usar diferentes *emails* ao contribuir. Estes fatores tem o potencial de criar falsos-positivos, onde por exemplo, um contribuidor pode ser contado mais de uma vez. Para suavizar esta ameaça, utilizamos técnicas de remoção de ambiguidade, além de testes estatísticos visando mitigar ameaças de generalização, de acordo com nossas hipóteses.

6. Conclusão

Ambientes sociais de codificação estão mudando o modo como *software* é construído. Estes ambientes possuem uma diversidade de aspectos que fazem com que as contribuições se tornem muito mais visíveis. Com toda sua popularidade e melhorias, estes ambientes aparentam ser responsáveis pela solução de diversas barreiras enfrentadas por projetos de código aberto. Neste artigo, nós estudamos se e como estas melhorias realmente podem ser creditadas a tais ambientes.

Em resposta à nossa questão de pesquisa (*Quanto o processo de migração para ambientes sociais de codificação impacta na colaboração em projetos de software livre em termos de novos contribuintes e número de contribuições?*), podemos dizer que foi possível observar que o crescimento da quantidade de contribuições e contribuidores após a migração para o GitHub não é algo verdadeiro para todos os casos. Não pudemos evidenciar aumento nas contribuições ou nos contribuintes em um dos projetos avaliados (`ruby`). Portanto, existem indícios de que a crença de que o próprio GitHub será eficaz na captação de novos contribuintes para projetos de OSS não é de todo verdadeiro. Encontramos, entretanto, em dois casos foi possível evidenciar aumento na quantidade de contribuição após a migração para o GitHub, mas o crescimento em número de novatos não é algo garantido. Acreditamos que existam fatores não relacionados à migração que influenciam o aumento na quantidade de contribuidores e contribuições (por exemplo, receptividade da comunidade, complexidade do projeto, interesse da comunidade).

Para o futuro, nós planejamos expandir o escopo deste estudo conduzindo um estudo em mais larga escala e conduzir análise qualitativas que possam indicar as razões para o crescimento e quais fatores podem influenciar o crescimento das contribuições e contribuidores.

References

[Bird 2011] Bird, C. (2011). Sociotechnical coordination and collaboration in open source software. In *ICSM*, pages 568–573, Washington, DC, USA. IEEE Computer Society.

[Dabbish et al. 2012] Dabbish, L., Stuart, C., Tsay, J., and Herbsleb, J. (2012). Social coding in github: Transparency and collaboration in an open software repository. In *CSCW*, pages 1277–1286, New York, NY, USA. ACM.

[Ducheneaut 2005] Ducheneaut, N. (2005). Socialization in an open source software community: A socio-technical analysis. *CSCW*, 14(4):323–368.

[Fogel 2013] Fogel, K. (2013). *Producing Open Source Software: How to Run a Successful Free Software Project*. O'Reilly Media, first edition.

[Gousios et al. 2014] Gousios, G., Pinzger, M., and Deursen, A. v. (2014). An exploratory study of the pull-based software development model. In *ICSE*, pages 345–355.

[Jensen et al. 2011] Jensen, C., King, S., and Kuechler, V. (2011). Joining free/open source software communities: An analysis of newbies' first interactions on project mailing lists. In *Proceedings of the 44th Hawaii International Conference on System Sciences*, HICSS '10, pages 1–10. IEEE.

[Marlow et al. 2013] Marlow, J., Dabbish, L., and Herbsleb, J. (2013). Impression formation in online peer production: Activity traces and personal profiles in github. In *CSCW*, pages 117–128.

[McDonald and Goggins 2013] McDonald, N. and Goggins, S. (2013). Performance and participation in open source software on github. In *CHI*, pages 139–144.

[Moura et al. 2015] Moura, I., Pinto, G., Ebert, F., and Castor, F. (2015). Mining energy-aware commits. In *MSR*, pages 56–67.

[Pham et al. 2013a] Pham, R., Singer, L., Liskin, O., Figueira Filho, F., and Schneider, K. (2013a). Creating a shared understanding of testing culture on a social coding site. In *ICSE*, pages 112–121.

[Pham et al. 2013b] Pham, R., Singer, L., and Schneider, K. (2013b). Building test suites in social coding sites by leveraging drive-by commits. In *ICSE*, pages 1209–1212.

[Pinto et al. 2016] Pinto, G., Steinmacher, I., and Gerosa, M. (2016). More common than you think: An in-depth study of casual contributors. In *SANER*, pages 112–123.

[Romano et al. 2006] Romano, J., Kromrey, J. D., Coraggio, J., and Skowronek, J. (2006). Should we really be using t-test and cohen's d for evaluating group differences on the nsse and other surveys? In *Annual meeting of the Florida Association of Institutional Research*.

[Stallman 1999] Stallman, R. (1999). The gnu operating system and the free software movement.

[Steinmacher et al. 2015] Steinmacher, I., Conte, T., Gerosa, M. A., and Redmiles, D. F. (2015). Social barriers faced by newcomers placing their first contribution in open source software projects. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*, CSCW '15, pages 1–13, New York, NY, USA. ACM.

[Thung et al. 2013] Thung, F., Bissyande, T. F., Lo, D., and Jiang, L. (2013). Network structure of social coding in github. In *Software maintenance and reengineering (csmr), 2013 17th european conference on*, pages 323–326. IEEE.

[Tsay et al. 2014] Tsay, J., Dabbish, L., and Herbsleb, J. (2014). Influence of social and technical factors for evaluating contribution in github. In *ICSE*, pages 356–366.

[Vasilescu et al. 2015] Vasilescu, B., Filkov, V., and Serebrenik, A. (2015). Perceptions of diversity on github: A user survey. In *CHASE*.

[Zhou and Mockus 2015] Zhou, M. and Mockus, A. (2015). Who will stay in the floss community? modelling participant's initial behaviour. *IEEE Transactions on Software Engineering*, 41(1):82–99.

Towards an Approach to Prevent Long Methods Based on Architecture-Sensitive Recommendations

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

[1] Department of Computer Science
Federal University of Bahia – Salvador, BA – Brasil

[2] Department of Information Systems
Federal University of Sergipe – Itabaiana, SE – Brasil

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

Abstract. *Long methods can be a software design erosion symptom. Existing approaches to identify this code smell are mainly based on the use of the number of lines of code metric together with a generic threshold value. However, using generic threshold values generates many false positives and false negatives because the class architectural concern is disregarded. This work presents a new approach that considers architectural concerns to identify threshold values and recommend long methods to developers. We evaluate the proposal with nine versions of MobileMedia system and the initial results show higher accuracy when compared with existing approaches.*

1. Introduction

The gradual increase of long methods in source code accelerates software design erosion. Design erosion is inevitable because of the way software is developed. However, good development methods help to increment system longevity [van Gurp & Bosch 2002]. According to Fontana et al. (2013), long methods are among the most common code fault in different application domains.

Code review is a common practice to maintain the quality of source code. Recently, automatic metrics-based approaches have been proposed to help this process [Marinescu 2004; Arcoverde et al. 2012; Palomba et al. 2013]. They define generic threshold values before code analysis for each used metric. A code smell is identified when established threshold values are exceeded. For instance, a long method is identified when the number of lines of code per method (SLOC/Method) exceeds a predetermined threshold value.

However, generic threshold values disregard system design decisions and the architecture concern of each class. The class architectural concern defines the main responsibility of the class within the evaluated code design. For example, in a system that follows the layer architectural style, are there differences in the average of SLOC/Method of classes in the persistence layer and classes in the business layer? If this occurs, generic thresholds may hide design problems (false negatives) or detect unimportant problems (false positives). Zhang et al. (2013) show evidences that information about application context, such as application domain, programming language and age should be considered in the metrics utilization. When this context is not considered, a large number of false positive and false negative can be sent to software developers. The excessive amount of warnings may lead to mistrust and lack of motivation to use automatic review approaches.

Furthermore, common wisdom suggests that software aging [Parnas 1994] and novice developers are the main cause of design erosion. However, recent studies show that code smells affect code artifacts since their creation [Tufano et al. 2015]. Expert developers are also responsible for introducing code anomalies in source code produced under the pressure of organizational factors [Lavallee & Robillard 2015]. However, the major approaches to check code anomalies are executed after the source code development. Postponing the repair of anomalies to final of development phase can lead to lack of motivation and time to fix software problems.

In this context, this paper presents a new approach for detecting and recommending long methods for software developers. Our approach allows to identify and to define specific thresholds for each architectural concern in a system. The threshold values of these architectural concerns are extracted from a design reference system that follows the same reference architecture [Angelov et al. 2009]. The reference system may be an earlier version of the same system or another system which follows the same reference architecture.

This approach differs from the existing proposals which suggest threshold values extracted from a set of systems [Arcelli Fontana et al. 2015; Vale & Figueiredo 2015]. However, the use of a set of different systems disregards design decisions of the system under evaluation, often requiring manual calibration of the values found. Furthermore, the quality of the systems could not be properly evaluated. Our approach uses only one system as a sample which can easily be reviewed by a quality team.

The remainder of this paper is organized as follows. Section 2 presents related work. Section 3 describes the proposed method to identify long method using class architectural context. Section 4 presents the experiment setup. Section 5 discusses threats to validity of the evaluation. Section 6 shows the results and discusses the research questions. Finally, Section 7 presents conclusions and insights for future work.

2. Related Works

A variety of approaches have been proposed to avoid design erosion through metric-based code smells detection strategies. Some studies discuss the importance of considering context in the assessment of source code. Marinescu (2006) shows how detection accuracy of Data Class and Feature Envy smells can be improved by taking into account particularities of enterprise applications. The study revealed that considering architecture roles has a big impact on eliminating false positives from classical detection rules.

Guo et al. (2010) claim that code smells detection rules should also take domain-specific characteristics into consideration. Setting thresholds is critical to an effective approach for detecting domain-specific code smells. Our work distinguishes from theirs in the way that we aim to propose a generic method to identify architectural concerns. Our approach does not require from the developer prior knowledge of the design decisions. The design decisions are automatically extracted from a reference system.

Macia et al. (2013) have explored the binding of the architecture structure with code smells. They propose a suite of detection strategies that use architecture-sensitive metrics. Their evaluation indicated on average 50% more architecturally-relevant code anomalies than those gathered with the use of conventional strategies. However, their

approach uses threshold values extracted from distinct systems. Our approach differs because we intended to use threshold values extracted from a system that follows the same reference architecture of the evaluated system.

3. The Proposed Approach

The proposed approach aims to identify long methods according to the architectural concern of each evaluated class. The class architectural concern identifies the main architectural responsibility of each class. On systems that adopt reference architectures, class responsibility is commonly assigned by class inheritance, class annotations or interfaces implementation of the reference architecture.

Our approach uses this knowledge to find out threshold values for each architectural concern. These values are extracted from a reference system that must follow the same reference architecture and design rules. The goal is to extract knowledge from a system that has the same design decisions and can be easily reviewed by a quality team to become a benchmark of code quality. We strongly recommend that this review is always carried out in the version used as code reference design. If such review is not performed, the approach can recommend code anomalies based on a code with inadequate design decisions.

The approach is divided into two phases. The first phase calculates thresholds for identified architectural concerns in the reference system (Figure 1). In the second phase (Figure 2), each class to be evaluated is classified into an architectural concern. This classification will select the threshold values, identified in the first phase, which will be used to evaluate the methods of this class.

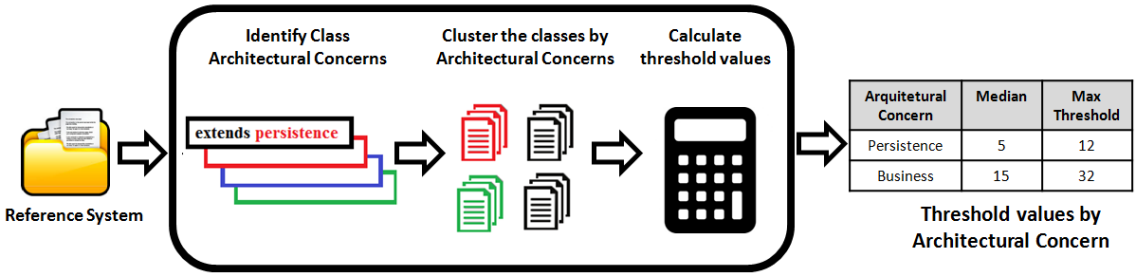


Figure 1. Defining threshold values from a reference system.

Figure 1 illustrates the first process phase. The approach is divided into three steps: (i) identifying the main architectural concern of each class in the sample system, (ii) grouping these classes according to the architectural concern and (iii) calculating threshold values for each identified architectural concern. The output of this process is a table holding all the identified architectural concerns and their respective threshold values.

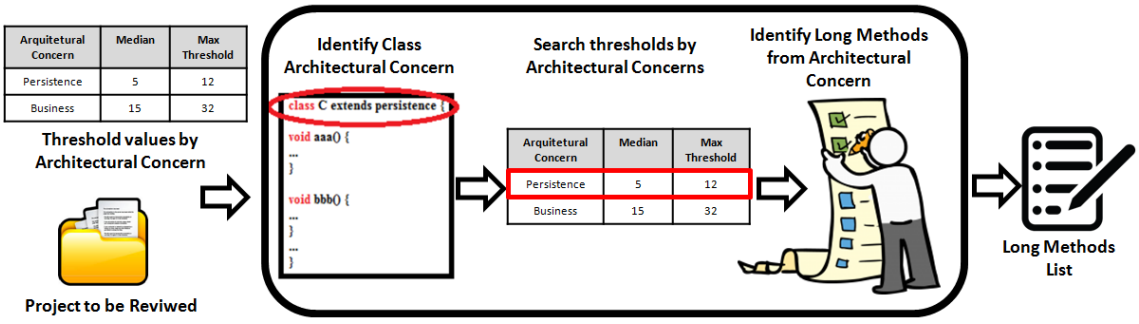


Figure 2. Identifying long methods based on architectural concerns.

Figure 2 shows the identification process of long methods considering the architectural concern. The input of the process is the table which contains threshold values for each architectural concern calculated in the first phase. The process is divided into three steps: (i) identifying the evaluated class architectural concern, (ii) obtaining threshold values in the table corresponding to architectural concern and (iii) identifying long methods according to the values found in the previous phase. The output of this process is a list of all the long methods in the evaluated system.

A key step in the two phases is the process of grouping classes in architectural concerns in both sample system and evaluated system. The method uses the hierarchy of classes to group them and then calculating the threshold values of each group. The first criterion, groups together the classes that extend the same class. After we group the classes that implement a system-defined interface itself. Finally, we group the classes that implement a non-system-defined interface. When a class is not categorized according to the specified criteria, a new group called *util* is created to link these classes. Additionally, the groups with only one element are merged with *util* group. The algorithm below describes how architectural concerns are identified:

Algorithm: Let $C = \{c_1, c_2, ..., c_n\}$ be a set of classes of a system, $1 < i \leq n$. The architectural concern of a class ci is represented by the function $AC(c_i) = (Di, Ii, Ei)$, where:

i) Di is the set of classes extended by the class ci.
ii) Ii is the set of interfaces defined by the system (internal interfaces) and implemented by the class ci.
iii) Ei is the set of interfaces not defined by the system (external interfaces) and implemented by the class ci.

Let $AC = \{AC_1, AC_2, ..., AC_m)$ a set of architectural concerns of the sample system,$1 < j \leq m$. Each architectural concern in AC is represented by the tuple $ACj = (Dj, Ij. Ej)$. A class ci is associated to an architectural concern ACj according to the following criteria:

1. c_i is associated to the AC_j when $D_i \neq \{\}$ and $D_i \subset Dj$. If $D_i \neq \{\}$ and it is not associated to any element of AC, a new architectural concern is added to AC represented as follows $AC_{m+1} = (Di, \{\}, \{\})$
2. c_i is associated to the AC_j when $I_i \neq \{\}$ and $I_i \subset I_j$. If $I_i \neq \{\}$ and it is not associated to any element of AC, a new architectural concern is added to AC represented as follows $AC_{m+1} = (\{\}, I_i, \{\})$
3. c_i is associated to the AC_j when $E_i \neq \{\}$ and $E_i \subset E_j$. If $E_i \neq \{\}$ and it is not associated to any element of AC, a new architectural concern is added to AC represented as follows $AC_{m+1} = (\{\}, \{\}, E_i)$
4. c_i is associated to the AC_{util} when $D_i = \{\}$, $I_i = \{\}$ and $E_i = \{\}$. AC_{util} is represented as follows $AC_{util} = (\{\}, \{\}, \{\})$.

At the end of processing of all classes, the algorithm observes if there are some element in the AC set that have only one associated class. These elements are excluded from the set and joined with AC_{util} elements. Finally AC_{util} is added to the set AC.

We defined the standard threshold values for long methods identification in each AC_J as the median and the 75th percentile (third quartile) of the ordered set of values of the NLOC/Method metric. The percentile value can be adjusted to another value.

4. Experimental Setup

The main objective of this study is to *analyze* our approach to detect long methods *for the purpose of* evaluating *with the respect to their* accuracy *from the point of view of* software developers *in the context of* available approaches to detect long methods.

Target Systems: To achieve this goal we used nine versions of the MobileMedia system, a software product line (SPL) for applications that manipulate photo, music, and video on mobile devices [Figueiredo et al. 2008]. We selected this system because it have already been used in a study conducted by [Paiva et al. 2015] that analyzed the accuracy of InFusion, JDeodorant and PMD tools using the default settings in relation to a reference list of three code smells*: feature envy, god class and god method*. According to Paiva et al. (2015), experts identified the code smells occurrences that form this reference list. There is a reference list for each of the nine versions. We only used the data about the god method smell, considering that a god method is also a long method. The difference is that the existing approaches use other metrics, besides NLOC/Method, to identify god methods.

Research Questions: We made two comparisons with existing approaches. First, we compared a single generic threshold extracted from a sample system. Second, we compared reference values for each identified architectural concern also extracted from a sample system. As existing approaches we used results obtained by Paiva et al. (2015). We formulated the following research questions, one for each situation:

RQ1. *Does using a threshold value extracted from a sample system that follows the same reference architecture improve the accuracy of long method detection?*

RQ2. *Does using threshold values for each architectural concern, extracted from a sample system that follows the same reference architecture, improve the accuracy of long method detection?*

Data Collection: We chose the first version as the sample system to calculate the thresholds and evaluated the other versions in order to detect long methods. All the nine versions follow the same architecture reference. We also collect data using two percentiles (75 and 90) for setting the maximum threshold values. These percentiles were selected by convenience for an initial assessment of their influence on the results. To apply of the proposed approach and allow the reproducibility of study, we developed an open source plug-in for Eclipse, called ContextLongMethod[1]. Also, the collected data were compiled and made available on a website[2].

Data Analysis: We calculate recall, precision and F-score to compare the obtained results with the proposed approach and available tools. The F-Score or F-measure is a measure of accuracy. It considers both precision and recall measures to compute the score. The best F-score has value 1 and the worst score has value 0.

5. Threats to Validity

Below we present potential threats to the validity of the experiment and the actions taken to minimize them.

Internal validity: The application of the approach is automatic. The only threats to internal validity are related to the selection of the sample system and the percentile to

[1] https://github.com/marcosdosea/ContextSmellDetector/
[2] https://sites.google.com/site/cbsoft2016/

be considered to calculate the thresholds. As our study is exploratory, we selected two different percentiles in order to verify the variation on the results. Choosing initial versions to evaluate later ones seems to make sense when the versions follow the same architecture. But this is a decision we intend to further evaluate in future studies.

External validity: The results obtained are valid only for MobileMedia system and the long method smell. We do not suggest generalizing the results to other systems and other code smells.

Construct validity. The use of reference lists produced by experts to calculate metrics such as precision and recall usually represent threats to validity. In our study, we used a reference list produced by other researchers, so there is no bias in favor of our approach. In addition, the results of other approaches we compared our results with were calculated based on the same reference list.

6. Results and Discussion

Table 1 summarizes the findings obtained by Paiva et al. (2015) compared with the our findings. The results of precision, recall and F-Score of each approach are shown. Each value represents the average of the values obtained for all nine versions (versions 1 to 9). Detailed results, including the number of false positives and false negatives used to calculate precision and recall of each method, are available on our website.

The first three lines show the results obtained with the use of InFusion, Jdeodorant and PMD. The following lines present the four configurations executed in our study. The third and forth lines show the results of the use of a generic threshold value extracted from a sample system (the first version of MobileMedia). The last two lines are about the use of a different threshold for each architectural concern.

Table 1. Results of analyzes performed in MobileMedia

		% Precision	% Recall	% F-Score
Paiva et al. (2015)	inFusion	100	26	41,27
	Jdeodorant	35	50	41,18
	PMD	100	26	41,27
Proposed Approach	Percentile 75	27	100	42,52
	Percentile 90	56	95	70,46
	Percentile 75 + architectural concern	32	100	48,48
	Percentile 90 + architectural concern	60	89	71,68

Given these results, we discuss the two research questions as follows:

RQ1. *Using a threshold value extracted from a sample system that follows the same reference architecture improves the accuracy of long method detection?*

The results obtained by using a generic threshold extracted from the first version of MobileMedia showed improvements in F-score. Despite the precision of 100% obtained with inFusion and PMD, their recall values show that these tools have not found a high number methods from the reference list. JDedorant found 50% of the reference list of methods, but its low precision (35%) generates the lowest F-Score (41,18%). Our approach using the 75th percentile, despite the low precision (27%), found all the methods of the reference list (100%). With the 90th percentile we found the best F-score (70.46%) when compared with the tools. A precision of 56% and a

recall of 95% generate this high value of F-Score. Thus, we found that only using a previous version for extracting the threshold value was already enough to improve the accuracy of long method detection.

We notice that the F-Score with the 90th percentile was higher than with the 75th percentile. However, it is noteworthy that by using the 90th percentile we did not reach 100% of recall, as with the 75th percentile. We believe that achieving greater recall in this case is more important than a high precision. It is easier for developers to analyze false positives (low precision) than analyzing false negatives (low recall). In the evaluation performed by Paiva et al. (2015), no approach was able to detect all the long methods identified by the experts, i.e. none approach obtained 100% recall. This can lead to a false sense that the system has few design flaws.

***RQ2**. Using threshold values for each architectural concern, extracted from a sample system that follows the same reference architecture, improves the accuracy of long method detection?*

Comparing the two configurations of our approach, we noticed small differences on the F-Score when architectural concern was considered. With the 75th percentile, we observed a small improvement in the precision (from 27% to 32%) when we considered architectural concerns. With the 90th percentile, there was a decrease in recall (from 95% to 89%), but there was also improvement in precision (from 56% to 60%) when we considered architectural concerns. This last configuration (90th percentile + architectural concerns) produced the best F-score (71.68%) among all approaches.

In summary, because the differences were too small, we are not able to claim that the use of architectural concerns in our approaches improves the accuracy of detecting long methods. The only claim that we can make is that both configurations of our approaches (with and without architectural concerns) improved the accuracy in comparison with existing tools. We intend to carry out further studies with larger systems to verify whether there are advantages on the use of architectural concerns.

7. Conclusion and Future Works

In this work we present a new approach to indicate long methods for developers. We also show an initial assessment of this approach. The approach proposes to extract threshold values of a sample system considering the class architectural concern. The initial evaluation of this approach showed better results compared to existing tools. One major benefit of the method is the increase of recall rate. This means reduction in the number of false negatives which are usually more difficult to be detected by developers.

As future work, we are going to extend the architectural concern identification method to consider more levels of class hierarchy and improve the precision of the approach. We are also extending the approach to consider other code smells. We also intend to perform a detailed analysis of classes classified in each group to understand how the proposed algorithm works in different system architectures. Finally, we plan to evaluate our approach in the context of real systems for more reliable results.

Acknowledgements. This work was supported by CNPq: grant 573964/2008-4 (National Institute of Science and Technology for Software Engineering) and grant 486662/2013-6.

References

Angelov, S., Grefen, P. & Greefhorst, D., 2009. A classification of software reference architectures: Analyzing their success and effectiveness. *2009 Joint Working*

IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture, pp.141–150.

Arcelli Fontana, F. et al., 2015. Automatic Metric Thresholds Derivation for Code Smell Detection. In *2015 IEEE/ACM 6th International Workshop on Emerging Trends in Software Metrics*. IEEE, pp. 44–53.

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

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

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

Guo, Y. et al., 2010. Domain-specific tailoring of code smells: an empirical study. *2010 ACM/IEEE 32nd International Conference on Software Engineering*.

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

Lavallee, M. & Robillard, P.N., 2015. Why Good Developers Write Bad Code: An Observational Case Study of the Impacts of Organizational Factors on Software Quality. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. IEEE, pp. 677–687.

Macia, I. et al., 2013. Enhancing the detection of code anomalies with architecture-sensitive strategies. *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, pp.177–186.

Marinescu, C., 2006. Identification of Design Roles for the Assessment of Design Quality in Enterprise Applications. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*. IEEE, pp. 169–180.

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

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

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

Parnas, D.L., 1994. Software aging. In *Proceedings of 16th International Conference on Software Engineering*. IEEE Comput. Soc. Press, pp. 279–287.

Tufano, M. et al., 2015. When and Why Your Code Starts to Smell Bad. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*.

Vale, G.A. Do & Figueiredo, E.M.L., 2015. A Method to Derive Metric Thresholds for Software Product Lines. *2015 29th Brazilian Symposium on Software Engineering*.

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

SARA^{MR}: Uma Arquitetura de Referência para Facilitar Manutenções em Sistemas Robóticos Autoadaptativos

Marcos H. de Paula[1], Marcel A. Serikawa[1], André de S. Landi[1], Bruno M. Santos[1]
Renato S. Costa[1], Valter V. de Camargo[1]

[1]Departamento de Computação – Universidade Federal de São Carlos (UFSCar)
Caixa Postal 676 – 13.565-905 – São Carlos – SP – Brasil

{marcos.paula,marcel.serikawa,andre.landi}@dc.ufscar.br

{bruno.santos,renato.costa,valter}@dc.ufscar.br

Abstract. *The Self-Adaptive Systems (SAS) architecture has two semantically distinct parts: (i) the managed system, which contains the domain specific features; and (ii) the system manager that monitors and changes the managed system behavior when necessary. In many systems these parts are not modularized difficulting maintenances. This article presents the reference architecture SARA^{MR}, which aims to separate the SAS parts in order to facilitated the management system maintenance. The architecture has been implemented in Java and a case study was developed using a Lego robot. A preliminary assessment was conducted to quantify the maintenance impact on the system. We realized that the reference architecture tends to minimize the maintenance activities produced by evolution impacts of self-adaptatives robotic applications.*

Resumo. *A arquitetura de Sistemas AutoAdaptativos (SAA) possui duas partes semanticamente distintas: (i) o sistema gerenciado, que representa as funcionalidades específicas do domínio; e (ii) o sistema gerenciador, que monitora e altera o comportamento do sistema gerenciado quando necessário. Em muitos sistemas essas partes não estão modularizadas, dificultando manutenções. Este artigo apresenta a arquitetura de referência SARA^{MR}, cujo objetivo é modularizar essas partes de um SAA de forma que as manutenções no sistema gerenciador sejam facilitadas. A arquitetura foi implementada em Java e foi desenvolvido um estudo de caso utilizando-se um robô Lego. Uma avaliação preliminar foi conduzida para quantificar o impacto de uma manutenção no sistema. Percebeu-se que a arquitetura de referência tende a minimizar as atividades de manutenção produzidas por impactos de evolução em aplicações robóticas autoadaptativas.*

1. Introdução

Robôs Móveis Autônomos (RMAs) são capazes de realizar atividades com pouca ou nenhuma intervenção externa. Para que um robô seja considerado autônomo, ele precisa de um controlador que seja um Sistema AutoAdaptativo (SAA) [Baker et al. 2011]. Muitos estudos demonstram que SAAs são baseados na teoria do controle do campo de Engenharia de controle (*Control Theory*). Sendo assim, intrinsecamente usam *loops* de controle (*Control Loops*) em sua arquitetura para realizar as adaptações [Cheng et al. 2005, Brun et al. 2009]. Um *loop* de controle tem as seguintes responsabilidades [Weyns et al.

2013, Kokar et al. 1999]: (i) monitorar e capturar dados de um determinado processo/-contexto; (ii) analisar os dados capturados desse processo/contexto perante um valor de referência; (iii) planejar uma adaptação; e (iv) realizar adaptações no processo/contexto de forma que se aproxime os próximos dados capturados da referência. Geralmente esses *loops* de controle são compostos por quatro elementos: Monitores, Analisadores, Planejadores e Executores. Atualmente, um dos modelos conceituais mais difundidos para esses *loops* de controle é o MAPE-K idealizado pela IBM [IBM 1994].

Em consequência dessas características, a arquitetura de um SAA pode ser denominada como duas partes distintas, sendo: (i) um subsistema controlado e um subsistema controlador. O subsistema controlado representa o sistema propriamente dito, já o subsistema controlador representa o *loop* de controle que realiza adaptações no subsistema controlado. Apesar dessa divisão conceitual de um SAA, esses subsistemas usualmente não se encontram modularizadas e separados no código-fonte. Vários autores apontam que uma das formas de melhorar os níveis de manutenibilidade de SAAs pode ser por meio da modularização e separação desses subsistemas, bem como dos *loops* de controle em entidades de primeira classe, já que diversas atividades de manutenção e evolução em SAAs são voltadas a esses quatro componentes [Weyns et al. 2013, Cheng et al. 2005, Garlan et al. 2004]. Isso significa que esses *loops* de controle devem ser implementados de forma modular e evidente no código-fonte ao invés de ficar espalhado e entrelaçado com outras classes do sistema. Embora alguns autores apresentem propostas de arquitetura para sistemas robóticos autônomos, apenas Albus [Albus 2000] e Affonso e Nakagawa [Affonso et al. 2013] apresentam soluções claras e concretas.

A principal motivação para este trabalho é que, assim como sistemas tradicionais, sistemas robóticos autônomos também necessitam de manutenções, muitas vezes no sentido de evoluir o software que controla o robô. Entretanto, apesar de vários autores reconhecerem que manutenções, nesse contexto, muitas vezes são mais desafiadoras do que em sistemas tradicionais, porém, pouco deles concentram-se em facilitar essas manutenções [Brugali 2007, Georgas and Taylor 2008, Edwards et al. 2009]. Neste artigo é proposta uma arquitetura de referência, chamada SARA^{MR}, para estruturar softwares robóticos autônomos de forma que sua manutenção seja facilitada. Isto é, vislumbra-se que o emprego da SARA^{MR} faz com que atividades de inclusão, alteração e remoção de elementos relacionados aos *loops* de controle sejam realizados de forma mais concentrada e controlada. A arquitetura SARA^{MR} está implementada como um *framework* Java, com classes abstratas e concretas, sendo assim para usá-la cria-se classes e métodos concretos que estendem os elementos abstratos. Os estudos de caso apresentados foram implementados utilizando-se a SARA^{MR} e testados com um robô Lego do *kit Mindstorm NXT*.

2. Arquiteturas de Referência

Arquitetura de referência é uma estrutura que fornece a caracterização das funcionalidades de um sistema a partir de um domínio específico [Eickelmann et al. 1996]. Dessa forma, uma arquitetura de referência serve como um guia para aumentar as chances de um desenvolvimento com sucesso de um sistema e pode ser considerado o primeiro passo essencial para o desenvolvimento de frameworks de aplicação.

De acordo com Eickelmann e Richardson [Eickelmann et al. 1996], a proposta de uma arquitetura de referência para sistemas de um domínio específico não é uma tarefa

trivial, pois, requer um profundo conhecimento sobre o domínio que a arquitetura de referência está sendo criada. Além disso, uma arquitetura de referência permite o reúso do projeto arquitetural de uma área específica [Affonso et al. 2013].

3. A Arquitetura SARA^{MR}

A proposta da arquitetura SARA^{MR} é fazer com que todas as funcionalidades de autoadaptação (monitorar, analisar, planejar, agir) sejam modularizadas e externalizadas para fora da aplicação base do robô. Além disso, também é a criação de uma camada de integração que forneça propriedades de baixo acoplamento entre o sistema controlado e o sistema controlador. Dessa forma, os módulos de *loop* de controle podem ser implementados, incluídos ou excluídos com menor impacto de manutenção. Além disso, a arquitetura propõe uma abstração para modularizar quatro funções básicas de um robô: (i) comportamento; (ii) representação do ambiente; (iii) implementação de sensores e (iv) atuadores. Com isso, essas funcionalidades também podem ser incluídas ou excluídas com menor impacto de manutenção. A Figura 1 apresenta a proposta da arquitetura mencionada.

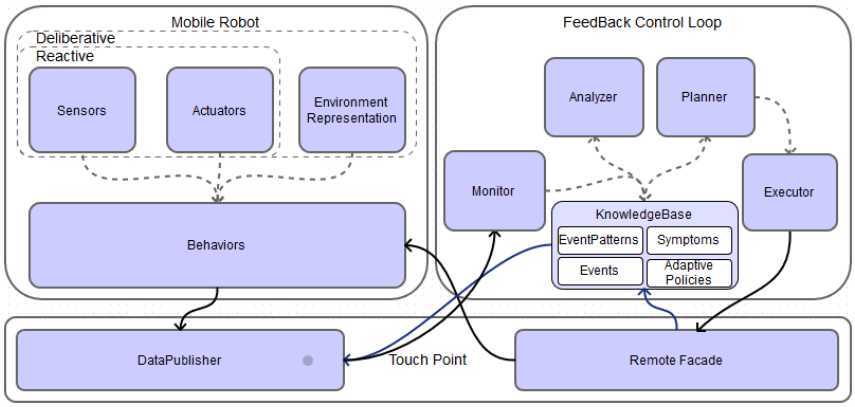


Figura 1. Diagrama de Blocks da arquitetura SARA^{MR}

A arquitetura de referência SARA^{MR} foi implementada em Java na forma de um *framework* caixa branca, possuindo classes e métodos abstratos. É apresentado na Figura 2 a arquitetura de referência criada que é composta por três módulos principais: i) *Mobile Robot Module* (`mobileRobot`) - que representa o sistema gerenciado, isto é, a aplicação base do robô; ii) *Feedback Control Loop Module* (`feedbackControlLoop`) - que representa o sistema gerenciador, o qual agrupa os componentes e as atividades de autoadaptação; e iii) *Touch Point Module* (`touchPoint`) - que representa a interface de comunicação entre os dois módulos citados anteriormente. Na parte superior da Figura 2 encontram-se os principais componentes da arquitetura de referência representados pelo pacote `feedbackControlLoop` e pelo `touchPoint`. Já na parte inferior da figura são representadas as classes concretas da aplicação desenvolvida como estudo de caso (`inDoorMonitoringBiding`, `inDoorMonitoring`). Sendo assim, para desenvolver uma nova aplicação seguindo essa arquitetura é necessário a criação de classes e métodos concretos, como é representado na figura, mais especificamente no pacote `inDoorMonitoring`. Nessa figura algumas partes estão representadas apenas por pacotes como `mobileRobot` e `inDoorMonitoringBinding`. Esses pacotes não serão detalhados pois o enfoque deste artigo e apenas no módulo de autoadaptação. A estrutura de classes da aplicação é apresentada com maiores detalhes na Seção 4.

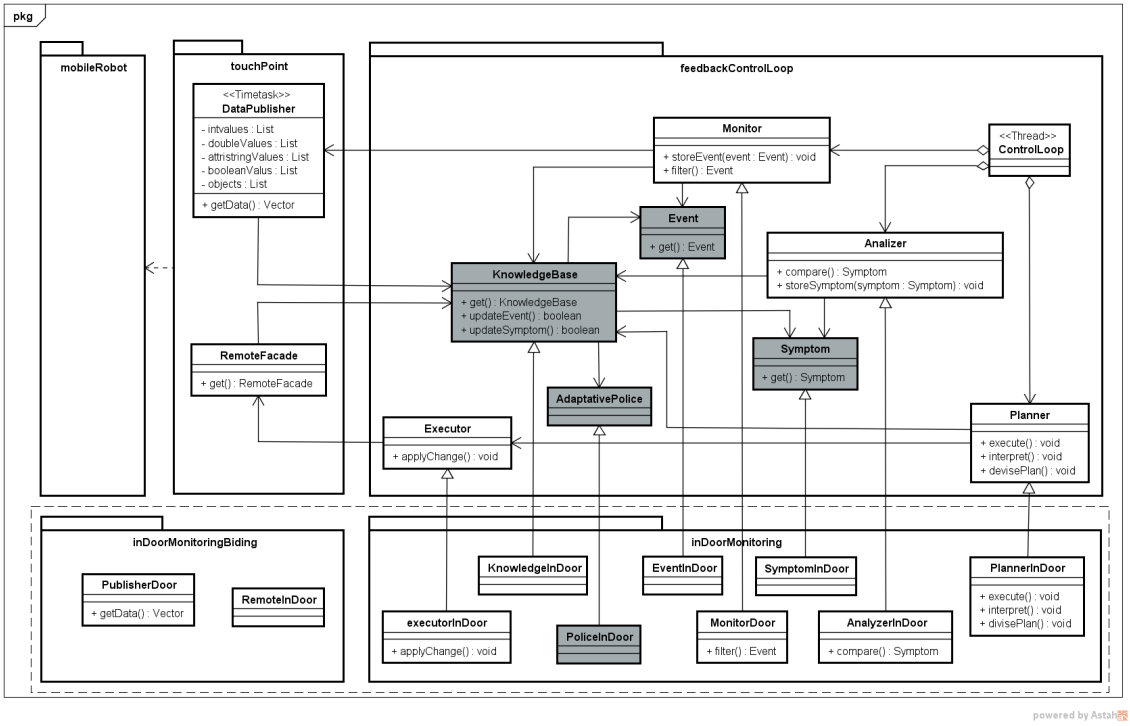


Figura 2. Arquitetura Implementada e Instanciação de uma Aplicação

O pacote `feedbackControlLoop` representa o sistema gerenciador de acordo com o modelo apresentado por Kephart e Chess [Kephart and Chess 2003]. Um ponto importante desse módulo é a presença de classes para modularizar os principais elementos de um *loop* de controle. Dessa maneira, manutenções que envolvem esses elementos podem ser realizadas de forma mais concentrada e controlada que em sistemas nos quais o código-fonte desses elementos encontra-se espalhado pelas classes do sistema. A seguir, seguem simples explicações sobre as classes desse pacote:

- `Monitor`: Detecta o processo gerenciado, filtra os dados coletados e armazena eventos relevantes na base de conhecimento para referencia futura.
- `Analizer`: compara dados de eventos para diagnosticar sintomas e armazena os sintomas na base de conhecimento para referencia futura.
- `Planner`: interpreta os dados analisados e elabora um plano de ação para mudar o processo do subsistema gerenciado.
- `Executor`: aplica esse plano por meio dos atuadores.
- `KnowledgeBase`: a base de conhecimento armazena os dados compartilhados entre os elementos do *loop* de controle.
- `Event`: pode ser um valor ou conjunto de valores recolhidos a partir dos sensores e atuadores que poderiam indicar um estado particular do sistema.
- `Symptom`: é a interpretação resultante da comparação entre um evento do sistema contra um evento padrão. Para cada problema há um conjunto de políticas de adaptação estabelecidas em conformidade com os objetivos do sistema.
- `AdaptivePolicies`: algoritmos reutilizáveis ou blocos de comandos que podem ser executados pelos atuadores, alterando o comportamento do robô.
- `ControlLoop`: é uma *Thread* que coordena a execução do fluxo de dados entre as classes do *loop* de controle.

O pacote `touchPoint` representa uma camada para facilitar a integração e fornece a propriedade de baixo acoplamento entre os módulos que estão sendo integrados. Esse pacote minimiza os efeitos de alterações efetuadas causando pouco ou nenhum impacto sobre outro modulo que seja integrado. A estrutura interna deste pacote é composta por duas classes. A classe `DataPublisher` é o elemento que tem acesso a todos os dados utilizados no comportamento do robô. Ou seja, os dados coletados dos sensores e atuadores são armazenados em sua estrutura de dados interna. E a classe `RemoteFacade` é quem possui acesso aos comandos disponíveis nos comportamentos do robô e da base de conhecimento de um *loop* de controle. O *RemoteFacade* deve ser implementado como o padrão *Facade*, simplificando a utilização dos comandos de comportamentos do robô.

A exemplificação do funcionamento de um SAA aderente a SARA^{MR} ocorre da seguinte forma. De acordo com uma taxa pré-determinada de tempo, os valores obtidos dos sensores e atuadores são atualizados nos `DataPublishers`. Os monitores registram esses valores na base de dados traduzindo-os em eventos, por meio dos métodos `storeEvent()` e `filter()`. Os analisadores inferem sintomas e os registra na base de dados por meio dos métodos `storeSymptom()` e `compare()`. Os planejadores interpretam os sintomas com o método `interpret()`, e preparam um plano de correção. Para isso o método `devisePlan()` realiza uma busca na base de dados pela melhor política de adaptação. Os executores recebem o plano e aplicam a correção nos comportamentos do robô, por meio do padrão *Facade*.

4. Estudo de Caso e Avaliação Preliminar

Como estudo de caso, foi desenvolvida uma aplicação com apoio da arquitetura de referência SARA^{MR} com o objetivo de demonstrar as propriedades de facilidade de evolução e manutenção. A aplicação é um robô de monitoramento de ambientes fechados cuja estratégia é um comportamento que o faz seguir em frente mantendo uma distancia de 20cm da parede. O mecanismo de ajuste de distância é implementado na forma de um *loop* de controle e utiliza um algoritmo de PID como politica de adaptação. O sistema autoadaptativo submete os valores coletados dos sensores ao processo de monitoramento e análise, em seguira um plano de correção é efetuado com base no valor de retorno do algoritmo, finalmente o plano é traduzido em ação por meio de comandos aplicados nos atuadores. Assim, o *loop* de controle atua como um piloto automático sobre o comportamento do robô fazendo os ajustes necessários para que ele continue mantendo a distância desejada.

Entretanto, esse *loop* de controle não ofereceu um bom desempenho, pois os parâmetros de ajuste do algoritmo de PID (Kp, Ki e Kd), são fixos e o robô o acaba fazendo "zig zags", comprometendo o tempo de deslocamento. Dessa forma, optou-se pela estratégia de inclusão de um segundo *loop* de controle atuando com ajustes nos parâmetros do algoritmo do primeiro *loop*. O segundo *loop* também possui o ciclo completo de autoadaptação, com as funções de monitoramento, análise, planejamento e ação.

A abordagem escolhida para a avaliação da arquitetura se baseia na demonstração de atividades de manutenção no código necessárias para evolução da aplicação robótica. Simulou-se uma atividade de manutenção, que foi a inclusão de um novo *loop* de controle, conforme Tabela 1. A motivação foi a percepção de que o primeiro loop não estava sendo suficiente para fazer com que o robô obtivesse um bom desempenho. Assim, percebeu-se a necessidade de um outro loop de controle que atuasse como um sistema gerenciador

sobre o primeiro *loop*, o qual passou a fazer o papel de sistema gerenciado.

Tabela 1. Atividades de Manutenção na Aplicação 1

Evolução	Descrição da Necessidade de Evolução	Atividade	Qtd
Inclusão de um novo *loop* de controle	Em consequência do mau desempenho do robô, opto-se por incluir um novo *loop* de controle para melhorar o desempenho.	Criação de Classes	10
		Implementação de Métodos	10
		Declaração de Atributos	7

As atividades de manutenção demonstraram que a arquitetura SARA^{MR} cumpre a função de guia e orienta o mantenedor do código a identificar facilmente os pontos de alteração. Tanto nas tarefas de inclusão ou exclusão de *loop* de controle como na implementação do ciclo de autoadaptação. Todos esses pontos estão referenciados na arquitetura. A estratégia de utilização de classes espelho e políticas de adaptação facilitou no cenário de exclusão de sensores. No caso de aplicações que não utilizam a arquitetura SARA^{MR} ou/e que não levam em conta a estratégias de classes espelho, bem como separação do módulo de *loop* de controle, as atividades de manutenção podem ser bem mais complexas e custosas.

Com o objetivo de avaliar o esforço gasto com as atividades de manutenção evolutiva, realizou-se uma avaliação do impacto que as alterações de códigos efetuadas em um determinado componente podem causar aos outros componentes da arquitetura. O termo impacto aqui se refere ao fato de que uma determinada alteração em um componente pode disparar necessidades de alterações em outros componentes.

Dessa forma, foi elaborada uma matriz de relacionamentos, conforme é mostrada na Tabela 2, contrastando as atividades de manutenção com o impacto de alterações que eventualmente possam se propagar pela arquitetura. Na primeira linha estão os módulos e as atividades de manutenção (Inclusão, Alteração, Exclusão). Na primeira coluna são apresentados os módulos que recebem algum impacto. Nas intersecções são mostradas as letras N (Não) para assinalar que não há impacto e a letra S (Sim) para assinalar que há impacto entre atividade e módulo. Conforme é mostrado na Tabela 2, as atividades de inclusão e alteração de sensores não causam impacto de manutenção em outros módulos da aplicação. Já a atividade de exclusão de sensores pode ocasionar impactos nos módulos de Comportamentos e *Loops* de Controle. As atividades de inclusão, alteração ou exclusão de *Loops* de Controle não causam impactos nos outros módulos da aplicação.

Nas duas colunas da direita são indicadas as quantidades que apontam uma relação entre o número total de atividades de manutenção e o total de impactos causados. Observa-se que do total de 60 atividades de manutenção, apenas 10 causam impactos de manutenção, ou seja, um total de 17%.

5. Trabalhos Relacionados

A arquitetura 4D/RCS [Albus 2000] fornece um modelo de referência para veículos militares não tripulados. É uma arquitetura híbrida com capacidade de planejamento e ação em tempo real para responder e reagir aos estímulos do ambiente. Cada camada possui vários *loops* de controle que são chamados de nós computacionais. Apenas os níveis mais baixos foram totalmente implementados. Os elementos são organizados na estrutura de um *feedback loop*. Onde cada nó funciona como um *loop* de controle, lendo dados dos sensores e enviando comandos aos atuadores.

Tabela 2. Matriz de relacionamentos (Manutenção x Impacto)

		Manutenção (Inclusão, Alteração, Exclusão)															Total de Impactos	
		Sensores			Atuadores			Comportamentos			Ambiente			*Loops* de Controle				
		I	A	E	I	A	E	I	A	E	I	A	E	I	A	E	Sim	Não
Impacto	Sensores				N	N	N	N	N	N	N	N	N	N	N	N	0	12
	Atuadores	N	N	N				N	N	N	N	N	N	N	N	N	0	12
	Comportamentos	N	N	S	N	N	S				N	S	S	N	N	N	4	8
	Ambiente	N	N	N	N	N	N	N	N	N				N	N	N	0	12
	Loops de Controle	N	N	S	N	N	S	N	S	S	N	S	S				6	6
Totais =																	**10**	**50**
																	17%	**83%**

A RA4SaS [Affonso et al. 2013] e uma arquitetura de referência baseada no recurso de reflexão para inspeção e modificação de entidades de software em tempo de execução. Apesar de não ser uma arquitetura para o domínio de RMAs, essa proposta é interessante para o contexto deste trabalho, pois essa arquitetura utiliza *loops* de controle para realizar a autoadaptação. Outro ponto importante é que o contexto de adaptação da RA4SaS ocorre diretamente na estrutura das entidades de software diferenciando do método de adaptação apresentado na proposta deste trabalho, criando uma perspectiva interessante de comparação.

6. Conclusão

Neste artigo foi apresentada uma arquitetura de referência para apoiar construção e manutenção de software autoadaptativo para robôs móveis autônomos. Com o apoio da arquitetura SARA^{MR}, conforme é demonstrado no decorrer do trabalho as atividades de manutenção evolutiva são facilitadas de acordo com os aspectos a seguir:

- Inclusão ou alteração de sensores e atuadores não geram impactos de manutenção nos outros módulos da aplicação.
- Os comportamentos são blocos individuais de código e podem ser adicionados sem gerar impacto. No caso de alteração e remoção os impactos são mínimos, podendo ocorrer apenas no módulo de *loop*.
- A representação de elementos no ambiente é efetuada por meio de informações métricas, facilitando as atividades de manutenção.
- Inclusão de novos elementos não geram impactos de manutenção. Alteração ou exclusão de elementos podem gerar impactos porem apenas nos módulos de comportamentos e *loops* de controle.
- Inclusão, alteração ou exclusão de *loops* de controle não geram impactos na aplicação, as funcionalidades de autoadaptação são separadas da aplicação base do RMA.

Embora a atual versão da arquitetura SARA^{MR} esteja implementada em um *framework* Java e somente tenha sido testada em um robô Lego. Acredita-se que as abstrações oferecidas na estrutura da arquitetura possam servir de guia para a implementação em qualquer outra linguagem orientada a objetos. Sua implementação em C++, por exemplo, poderia propiciar seu uso em outros tipos de robôs. Os resultados obtidos demonstram que a arquitetura de referência minimiza as atividades de manutenção produzidas por atividades de evolução das aplicações robóticas autoadaptativas. A arquitetura de referencia fornece uma padronização de módulos para as principais funcionalidades do domínio de RMAs. Fornece também uma estrutura com diretrizes, técnicas e padrões para melhorar as atividades ligadas a construção e manutenção evolutiva no software de RMAs.

7. Agradecimentos

O presente trabalho possui relação direta com o processo Fapesp 2012/00494-0

Referências

Affonso, F. J. et al. (2013). A reference architecture based on reflection for self-adaptive software. In *Software Components, Architectures and Reuse (SBCARS), 2013 VII Brazilian Symposium on*, pages 129–138.

Albus, J. S. (2000). 4-d/rcs reference model architecture for unmanned ground vehicles. In *Robotics and Automation, 2000. Proceedings. ICRA '00. IEEE International Conference on*, volume 4, pages 3260–3265 vol.4.

Baker, C. R. et al. (2011). Toward adaptation and reuse of advanced robotic software. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 6071–6077.

Brugali, D. (2007). Software abstractions for modeling robot mechanisms. In *2007 IEEE/ASME international conference on advanced intelligent mechatronics*, pages 1–6.

Brun, Y. et al. (2009). *Software Engineering for Self-Adaptive Systems*, chapter Engineering Self-Adaptive Systems Through Feedback Loops, pages 48–70. Springer-Verlag, Berlin, Heidelberg.

Cheng, S.-W., Garlan, D., and Schmerl, B. (2005). *Self-star Properties in Complex Information Systems*, chapter Making Self-adaptation an Engineering Reality, pages 158–173. Springer-Verlag, Berlin, Heidelberg.

Edwards, G. et al. (2009). Architecture-driven self-adaptation and self-management in robotics systems. In *2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 142–151.

Eickelmann, N. S. et al. (1996). An evaluation of software test environment architectures. In *Software Engineering, 1996., Proceedings of the 18th International Conference on*, pages 353–364.

Garlan, D. et al. (2004). Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54.

Georgas, J. C. and Taylor, R. N. (2008). Policy-based self-adaptive architectures: A feasibility study in the robotics domain. In *Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-managing Systems*, SEAMS '08, pages 105–112, New York, NY, USA. ACM.

IBM (1994). *Autonomic computing white paper: An architectural blueprint for autonomic computing*. IBM White Paper.

Kephart, J. O. and Chess, D. M. (2003). The vision of autonomic computing. *Computer*, 36(1):41–50.

Kokar, M. M., Baclawski, K., and Eracar, Y. A. (1999). Control theory-based foundations of self-controlling software. *IEEE Intelligent Systems*, 14(3):37–45.

Weyns, D. et al. (2013). *Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*, pages 76–107. Springer Berlin Heidelberg, Berlin, Heidelberg.

Distribuição de Conhecimento de Código em Times de Desenvolvimento - uma Análise Arquitetural

Mívian M. Ferreira[1], **Kecia Aline M. Ferreira**[2], **Marco Tulio Valente**[1]

[1] Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte, MG – Brasil

[2]Centro Federal de Educação Tecnológica de Minas Gerais (CEFET-MG)
Belo Horizonte, MG – Brasil

{mivian.ferreira,mtov}@dcc.ufmg.br, kecia@decom.cefetmg.br

Abstract. *Understanding the distribution of knowledge about the code among the members of a development team is an important task for project management. Truck factor is a metric that indicates the number of developers whose departure impairs or complicates the survival of the project. Currently the algorithms proposed for computing this metric take into account only the amount of files authored by a developer, but not the importance of these files to the system. This study investigates whether the set developers indicated by such algorithms are those who actually have knowledge of the most complex components of the system. The results indicate that the analysis of the truck factor associated with the importance of the classes in a system brings relevant information about the influence of developers in the system.*

Resumo. *Saber como se dá a distribuição de conhecimento do código entre os membros de um time de desenvolvimento é uma questão importante para o gerenciamento de projetos. Truck factor é uma métrica que indica o número de desenvolvedores cuja saída dificulta ou inviabiliza a continuidade do projeto. Os algoritmos propostos para o cálculo dessa métrica levam em consideração apenas a quantidade de arquivos nos quais um desenvolvedor possui autoria, e não a importância desses arquivos para o sistema. O presente trabalho investiga se o conjunto de desenvolvedores indicados por tais algoritmos são aqueles que de fato possuem conhecimento dos componentes mais complexos do sistema. Os resultados sugerem que a análise do truck factor considerando a importância das classes de um sistema traz detalhamentos relevantes acerca da influência dos desenvolvedores no sistema.*

1. Introdução

Truck factor é uma métrica que tem por objetivo detectar como o conhecimento sobre o código-fonte está distribuído entre os membros de uma equipe de um projeto de software. Segundo Martin Bowler[1], essa métrica indica o número de desenvolvedores que, ao serem desligados da equipe de desenvolvimento, fazem com que o projeto entre em sérios problemas. *Truck factor* pode ser utilizada para identificar e prevenir possíveis riscos decorrentes da dependência do projeto em relação a certos desenvolvedores [Ricca et al. 2011]. Essa aplicação se faz importante principalmente em projetos

[1]http://www.agileadvice.com/2005/05/15/agilemanagement/truck-factor/

de desenvolvimento de software nos quais a taxa de saída e entrada (*i.e.*, *turnover*) de desenvolvedores tende a ser alta.

Existem na literatura alguns algoritmos para o cálculo de *truck factor* [Zazworka et al. 2010, Avelino et al. 2016]. Esses algoritmos baseiam-se na premissa de que todas as classes possuem a mesma importância para o sistema. Sendo assim, o *truck factor* de um sistema será dado pelo menor conjunto de desenvolvedores cuja remoção da equipe de desenvolvimento fará com que a maior parte dos arquivos do sistema percam todos os seus autores (*i.e.,* desenvolvedores que possuem conhecimento sobre o arquivo). Entretanto, não existem na literatura algoritmos que levem em consideração a importância das classes para o sistema, por exemplo, a quantidade de falhas associadas às classes ou às complexidades das mesmas. O presente trabalho visa contribuir nesse contexto.

Especificamente, este trabalho tem por objetivo analisar como o conhecimento dos desenvolvedores que compõem o *truck factor* de um sistema está distribuído em sua arquitetura. Buscaremos identificar uma associação mais realista entre o conhecimento dos desenvolvedores e as classes presentes nos componentes mais críticos do sistema. Para tanto, é apresentada uma avaliação empírica com quatro sistemas Java. A arquitetura dos sistemas será representada pelo modelo *Little House* [Ferreira 2011], que fornece a visualização macroscópica e genérica da arquitetura de um software.

O restante deste trabalho está organizado da seguinte forma: Seção 2 apresenta os principais conceitos e trabalhos relacionados a este trabalho; Seção 3 relata a metodologia aplicada para desenvolvimento do trabalho; Seção 4 apresenta os resultados obtidos; Seção 5 apresenta uma breve discussão sobre os achados do trabalho; Seção 6 discute as ameaças à validade do trabalho; e Seção 7 traz as conclusões e indicações de trabalhos futuros.

2. Trabalhos Relacionados

2.1. Truck Factor

Existem na literatura poucos trabalhos relacionados ao tema *truck factor*. Alguns trabalhos têm por objetivo propor abordagens para o cálculo de *truck factor* [Zazworka et al. 2010, Cosentino et al. 2015, Avelino et al. 2016]. A abordagem proposta por Zazworka et al. [2010] calcula, para cada combinação possível de desenvolvedores, o número de arquivos que continuariam com autores caso aquele conjunto de desenvolvedores fosse retirado do sistema. Porém, essa abordagem não escala para projetos com mais de 30 desenvolvedores [Ricca et al. 2011]. Já na abordagem proposta por Consentino et al. [2015], os desenvolvedores utilizados para o cálculo de *truck factor* são aqueles que possuem um determinado nível de conhecimento sobre o sistema. A utilização dessa abordagem é dificultada pela necessidade de inserção de parâmetros e métricas de conhecimento sobre código-fonte. A proposta de Avelino et al. [2016] consiste em estabelecer os autores de cada arquivo do sistema através da fórmula de *Degree of Authorship* [Fritz et al. 2014]. Após estabelecer os autores do sistema, retiram-se da lista de avaliação os autores com maior número de arquivos. As retiradas são realizadas até que mais de 50% dos arquivos estejam sem autores. Os dados referentes ao *truck factor* usados na avaliação empírica deste artigo foram obtidos pela aplicação da abordagem desenvolvida por Avelino et al. [2016].

Aplicando a abordagem desenvolvida por Zazworka et al. [2010], alguns trabalhos fazem uso do conceito de *truck factor* para identificar: *thresholds* para o uso da métricas [Torchiano et al. 2011]; dificuldades inerentes ao cálculo da métrica [Ricca et al. 2011] e a complexidade computacional do algoritmo [Hannebauer and Gruhn 2014]. Entretanto, no melhor no nosso conhecimento, não existem trabalhos que, como este, busquem correlacionar os dados obtidos através do *truck factor* de um sistema com sua arquitetura.

2.2. O Modelo *Little House*

Little House é um modelo que tem por objetivo proporcionar a visualização macroscópica e genérica da arquitetura de um software [Ferreira 2011]. Proposto com base no modelo Bow-tie ([Broder et al. 2000] apud [Ferreira 2011]) que representa o macrografo da Web, o modelo *Little House* retrata a componentização do grafo de dependências de um software desenvolvidos em Java.

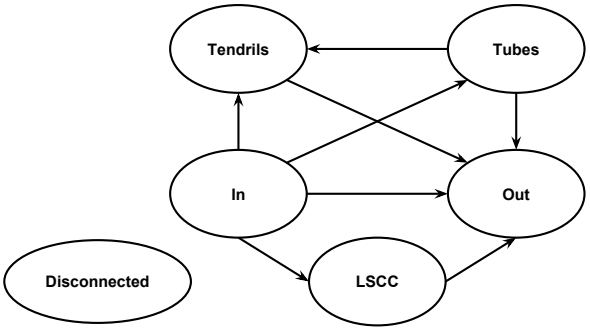


Figura 1. Modelo *Little House* [Ferreira 2011]

A Figura 1 ilustra os componentes do modelo. As classes de um sistema são agrupadas nos componentes de acordo com a seguinte regra: Uma classe A usa uma classe B, se em A existe uma referência a B. Os componentes de *Little House* são caracterizados conforme descrito a seguir:

Tubes: agrupa as classes que utilizam classes de *Tendrils* ou *Out* e são utilizadas por classes presentes no próprio componente ou em *In*.
Tendrils: agrupa classes que fazem uso de classes do componente *Out* e podem ser utilizadas por classes presentes em *In* ou *Tubes*.
In: reúne classes que usam classes de quaisquer outros componentes do modelo, exceto *Disconnected*, mas que não são utilizadas por classes existentes nos outros componentes.
Out: agrupa classes que não fazem uso de classes de outro componente e são utilizadas por classes presentes em qualquer outro componente do modelo, exceto *Disconnected*.
LSCC: é o maior componente fortemente conectado do modelo. As classes presentes neste componente podem ser alcançadas de forma direta ou indireta por quaisquer outras classes existentes neste componente.
Disconnected: aglomera classes que não são utilizadas ou fazem uso de quaisquer outras classes agrupadas nos demais componentes.

Estudos realizados com o modelo indicam que *Out* e *LSSC* são os componentes que: apresentam maior degradação ao longo da evolução do software do ponto de vista

de métricas [Ferreira et al. 2012a]; possuem piores valores de métricas quando comparados aos demais componentes [Ferreira et al. 2012b]; e são os componentes que agrupam classes com maior impacto de propagação de modificação [Ferreira et al. 2015]. Neste trabalho, busca-se identificar se os desenvolvedores apontados pelo algoritmo de cálculo de *truck factor* possuem de fato conhecimento dos componentes mais complexos do software.

3. Metodologia

O *dataset* inicial considerado utilizado neste trabalho foi o proposto por Avelino et al. [2016]. Ele é composto pelos 133 sistemas mais populares do GitHub, distribuídos em seis linguagens de programação: Java Script, Python, Ruby, C/C++, Java e PHP. Além disso, os sistemas possuem número de arquivos, número de desenvolvedores, tamanho e histórico de desenvolvimento significativos.

Os sistemas analisados neste trabalho atendem aos seguintes critérios: são escrito em Java; possuem seus *bytecodes* disponíveis ou disponibilizam mecanismos que facilitem a obtenção dos *bytecodes* (*i.e.*, presença do arquivo *pom.xml*). Esses critérios atendem às premissas de funcionamento da ferramenta *Connecta* [Ferreira 2011], que é responsável por gerar os dados do modelo *Little House*. Após aplicação dos critérios, foram selecionados quatro dos 22 sistemas Java disponíveis no *dataset* original: *Android Annotations*, um *framework* Android; *Dropwizard*, um *framework* web; *Titan*, um banco de dados de grafos; e *Glide*, um gerenciador de dependências.

Os dados relacionados ao *truck factor* (TF) foram obtidos através da ferramenta *Truck-Factor*[2] [Avelino et al. 2016]. Com essa ferramenta, foram coletados os seguintes dados: valor de TF do sistema, conjunto de desenvolvedores que formam o TF; e lista de arquivos nos quais esses desenvolvedores do TF têm autoria. A ferramenta *Connecta*[3] foi utilizada para obtenção dados referentes ao modelo *Little House* [Ferreira 2011]. A partir dos *bytecodes* do sistema, *Connecta* fornece uma lista contendo as classes do sistema e os componentes aos quais elas pertencem. Como última etapa, foi desenvolvido um *script* Java que concatena os dados das listas fornecidas pelas ferramentas *Truck-Factor* e *Connecta*. Além disso, o *script* é capaz de computar o percentual de classes que um desenvolvedor possui em um componente.

4. Resultados

Nesta seção são apresentados e analisados os resultados obtidos para os quatro sistemas estudados. Os dados obtidos estão apresentados em tabelas que reportam o total de classes por componente (Classes/Comp.), o número de classes que cada autor possui em cada componente (#Classes) e o percentual que esse número significa em relação ao total de classes do componente. Por questão de privacidade dos desenvolvedores, eles serão identificados pela sigla TFDev#.

Android Annotations

O sistema *Android Annottions* é um *framework* para desenvolvimento Android, possuindo *truck factor* 2. TFDev1 e TFDev2 são os desenvolvedores apontados como

[2]https://github.com/aserg-ufmg/Truck-Factor

[3]http://homepages.dcc.ufmg.br/ kecia/connecta_portugues.htm

críticos para o sistema, e possuem, respectivamente, 24,81% e 15,76% dos arquivos Java do sistema. A Tabela 1 apresenta os resultados para *Android Annotations*.

Tabela 1. Resultados *Android Annotations*

		TFDev1		TFDev2	
Componente	**Classes/Comp.**	**#Classes**	**%Autoria**	**#Classes**	**%Autoria**
LSCC	10	1	10%	9	90%
In	161	3	2%	89	55%
Out	10	2	20%	2	20%
Tendrils	71	10	14%	1	1%
Tubes	32	8	25%	1	3%
Disconnected	655	209	32%	46	7%

Embora TFDev1 seja indicado como o desenvolvedor cuja saída poderá gerar maior impacto para o sistema, ele não possui maior percentual de autoria nos componentes mais críticos do sistema. Isso fica a cargo do desenvolvedor TFDev2. Esse desenvolvedor possui maior percentual de autoria em dois dos três componentes mais críticos do sistema (LSCC e In). Sendo assim, ao considerar a importância das classes, o desenvolvedor cuja saída causaria maior impacto é TFDev2.

Titan

Titan é um banco de dados de grafos, otimizado para armazenar e consultar grafos com bilhões de vértices e arestas. Esse sistema possui *truck factor* 2. Os dois desenvolvedores críticos para esse sistema, TFDev3 e TFDev4, são responsáveis por 10% e 39% dos arquivos .java o sistema. Todavia, 53% dos arquivos de TFDev3 não são .java. Já no caso do desenvolvedor TFDev4, 99,7% dos arquivos dos quais ele possui autoria são .java. Conforme reporta a Tabela 2, esse sistema possui características análogas ao *Android Annotations*. Embora TFDev3 seja indicado como desenvolvedor com maior impacto no sistema, os dados indicam que TFDev4 possui maior percentual de autoria nos componentes mais críticos do sistema.

Tabela 2. Resultados *Titan*

		TFDev3		TFDev4	
Componente	**Classes/Comp.**	**#Classes**	**% Autoria**	**#Classes**	**%Autoria**
LSCC	74	1	1%	50	68%
In	214	66	31%	108	50%
Out	402	20	5%	163	41%
Tendrils	277	25	9%	48	17%
Tubes	57	4	7%	19	33%
Disconnected	371	27	7%	160	43%

Dropwizard e Glide

Os sistemas *Dropwizard* e *Glide* possuem *truck factor* 1. As Tabelas 3 e 4 apresentam, respectivamente, os resultados dos sistemas. Diferente dos resultados obtidos nos sistemas descritos *Android Annotations* e *Titan*, nos sistemas que possuem *truck factor* iguais a 1, observa-se que os desenvolvedores indicados pelo *truck factor* possuem maior percentual de autoria nos componentes mais complexos do sistema. No sistema

Dropwizard, o desenvolvedor TFDev5 possui maior percentual absoluto de autoria nos componentes LSCC e In e mais $1/4$ das classes do componente Out. Já no sistema *Glide*, o principal desenvolvedor, TFDev6, possui percentual de autoria significativa nos três componentes mais críticos do sistema.

Tabela 3. Resultados *Dropwizard*

		TFDev5	
Componente	**Classes/Comp.**	**#Classes**	**%Autoria**
LSCC	6	5	83%
In	46	24	52%
Out	106	40	38%
Tendrils	406	112	28%
Tubes	47	13	28%
Disconnected	293	89	30%

Tabela 4. Resultados *Glide*

		TFDev6	
Componente	**Classes/Comp.**	**#Classes**	**% Autoria**
LSCC	32	23	72%
In	5	2	40%
Out	220	126	57%
Tendrils	40	19	48%
Tubes	1	0	0%
Disconnected	93	44	47%

5. Discussão

Este trabalho investiga como é distribuído, na arquitetura do sistema, o conhecimento dos desenvolvedores que compõe seu *truck factor*. O objetivo deste estudo é identificar uma associação mais realista entre o conhecimento dos desenvolvedores e as classes presentes nos componentes mais críticos do sistema. Para isso, a arquitetura dos quatro sistemas Java avaliados foi representada pelo modelo *Little House*.

Embora a quantidade de software estudados seja pequena, os sistemas apresentados no estudo são relevantes. Foram utilizados quatro sistemas de código aberto, desenvolvidos em Java e cotados entre os mais populares do repositório GitHub. Além disso, a amostra conta com uma diversidade de domínios. Essas características são relevantes, uma vez que o estudo está relacionado ao conhecimento dos desenvolvedores sobre o código-fonte dos sistemas onde atuam.

Os resultados encontrados neste trabalho indicam conclusões importantes a respeito do cálculo de *truck factor*. Existem indícios de que, a importância das classes para o sistema seja um fator a ser considerado no cálculo da métrica. Como observado nos sistemas com *truck factor* 2, os desenvolvedores apontados como tendo o segundo maior impacto no sistema, foram aqueles que apresentam maior percentual de autoria nos componentes mais críticos sistema. Entretanto, a mesma afirmação não é válida para os projetos cujo *truck factor* é 1. Nesses sistemas, o desenvolvedor apontado pelo *truck factor* é de fato quem possui maior autoria (*i.e.,* conhecimento) nos componentes críticos.

A conclusão principal a que se chega a partir deste estudo é que os desenvolvedores responsáveis pelos componentes mais críticos da arquitetura do software são indicados dentre os desenvolvedores listados pelo *truck factor*. Portanto, a análise realizada pelo *truck factor* necessita ser aprimorada para que possa capturar com mais fidelidade a influência dos desenvolvedores no projeto, especialmente no que tange às questões arquiteturais.

6. Ameaças à Validade

Validade Externa. Para a realização da avaliação empírica, foram utilizados quatro sistemas *opensource* desenvolvidos em Java. Embora a amostra não seja grande, foram utilizados sistemas relevantes, de alta complexidade e de diferentes domínios, o que propicia maior representatividade para o *dataset* utilizado. Como o estudo se baseou apenas em sistemas *opensource*, não é possível afirmar que os resultados aqui obtidos possam ser generalizados para sistemas proprietários.

Validade de Construção. Neste trabalho consideramos apenas os arquivos .java de um sistema. Entretanto, é comum a existência de arquivos com outras extensões em um sistema. Sendo assim, podem existir arquivos tão importantes quanto os arquivos Java, mas que não foram incluídos na análise.

7. Conclusões

Truck factor é uma métrica utilizada para identificar a distribuição de conhecimento em projetos de desenvolvimento de software. De maneira geral, a métrica tem como objetivo identificar o conjunto de desenvolvedores cuja saída faz com que o desenvolvimento ou a manutenção do software fiquem em risco. No entanto, os algoritmos existentes para o cálculo da métrica não levam em consideração a importância relativa das classes de um sistema. Este trabalho teve por objetivo identificar como o conhecimento dos desenvolvedores, apontados como *truck factor* do sistema, está distribuído na arquitetura do sistema.

Neste trabalho foi realizada uma análise empírica quatro sistema Java. Buscou-se identificar se os desenvolvedores apontados com *truck factor* são aqueles que possuem maior percentual de autoria nos componentes mais críticos da arquitetura do sistema. Para tanto, a arquitetura do sistema foi representada através no modelo *Little House*. Os resultados foram obtidos através da comparação entre os dados fornecidos pelo modelo e os dados fornecidos pela ferramenta *Truck-Factor*.

Android Annotations e *Titan*, apresentaram *truck factor* 2. Nesses sistemas, detectou-se que o desenvolvedor apontado pela ferramenta *Truck-Factor* como aquele cuja saída causaria maior impacto no sistema, não é aquele que de fato que possui maior conhecimento nos componentes mais críticos dos sistemas. Em ambos os sistemas, os dados obtidos para os segundos desenvolvedores indicam que eles são os desenvolvedores críticos no caso de arquivos .java. Já nos sistemas *Dropwizard* e *Glide*, com *truck factor* 1, identificou-se que os desenvolvedores indicados são de fato aqueles que possuem maior percentual de autoria nos componentes críticos da arquitetura utilizada para a avaliação. Esses resultados indicam que a análise do *truck factor* associada à importância da classe do ponto de vista arquitetural pode revelar informações mais precisas acerca da influência dos desenvolvedores no sistema.

Como trabalhos futuros são propostos: *survey* com os desenvolvedores dos sistema para validação dos dados obtidos no estudo; proposta de uma abordagem que faça uso da relevância das classes de um sistema no cálculo de *truck factor*.

Agradecimentos

Essa pesquisa foi apoiada pela FAPEMIG e CAPES.

Referências

Avelino, G., Passos, L., Hora, A., and Valente, M. T. (2016). A novel approach for estimating truck factors. In *24th International Conference on Program Comprehension (ICPC)*, pages 1–10.

Cosentino, V., Izquierdo, J., and Cabot, J. (2015). Assessing the bus factor of git repositories. In *22nd International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 499–503.

Ferreira, K. A. M. (2011). *Um modelo de predição de amplitude da propagação de modificações contratuais em software orientado por objetos*. PhD thesis, Universidade Federal de Minas Gerais.

Ferreira, K. A. M., Moreira, R. C. N., Bigonha, M. A. S., and Bigonha, R. S. (2012a). The evolving structures of software systems. In *3rd International Workshop on Emerging Trends in Software Metrics (WETSoM)*, pages 28–34.

Ferreira, K. A. M., Moreira, R. C. N., Bigonha, M. A. S., and Bigonha, R. S. (2012b). A generic macroscopic topology of software networks - a quantitative evaluation. In *26th Brazilian Symposium on Software Engineering (SBES)*, pages 161–170.

Ferreira, M. M., Ferreira, K. A. M., and Neto, M.-H. T. (2015). Mapping the potential change impact in object-oriented software. In *30th ACM Symposium on Applied Computing (ACM-SAC)*, pages 1654–1656.

Fritz, T., Murphy, G. C., Hill, M.-E., Ou, J., and Hill, E. (2014). Degree-of-knowledge: modeling a developer's knowledge of code. *Software Engineering and Methodology*, 23(2):14:1–14:42.

Hannebauer, C. and Gruhn, V. (2014). Algorithmic complexity of the truck factor calculation. In *Product-Focused Software Process Improvement*, pages 119–133. Springer.

Ricca, F., Marchetto, A., and Torchiano, M. (2011). On the difficulty of computing the truck factor. In *12th International Conference on Product-focused Software Process Improvement (PROFES)*, pages 337–351.

Torchiano, M., Ricca, F., and Marchetto, A. (2011). Is my project's truck factor low?: theoretical and empirical considerations about the truck factor threshold. In *2nd International Workshop on Emerging Trends in Software Metrics (WETSoM)*, pages 12–18.

Zazworka, N., Stapel, K., Knauss, E., Shull, F., Basili, V. R., and Schneider, K. (2010). Are developers complying with the process: an xp study. In *4th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 14:1–14:10.

Using Evowave for Logical Coupling Analysis of a Long-lived Software System

Rodrigo Magnavita[1], **Renato Novais**[1,2], **Bruno Silva**[3], **Manoel Mendoncça**[1]

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

[2]Instituto Federal da Bahia
Salvador, BA.

[3]Universidade Salvador - UNIFACS
Salvador, BA.

{rodrigo.magnavita, manoel.mendonca}@ufba.br,

renato@ifba.edu.br, bruno.carreiro@pro.unifacs.br

Abstract. *Logical coupling reveals implicit dependencies between program entities, by measuring how often they changed together during development. The comprehension of how the logical coupling property manifests itself in the project is a key activity. Software Evolution Visualization (SEV) has been a promising approach to this end. However, this is not trivial, since SEV has to handle different software entities and attributes, and still deals with the temporal dimension of evolution. In this sense, we have been working on the specification, development and evaluation of a novel SEV tool, called EVOWAVE. This novel visualization metaphor is able to visualize different types of data generated in software evolution using both overview-based and detail-based approaches. It can be applied to different software engineering tasks and domains. In this paper, we present a logical coupling study we conducted in order to evaluate this tool in this important domain.*

1. Introduction

As a software system is developed and maintained across time there is an increasing amount of effort to understand program entities and carry out changes on them. This phenomenon happens, besides other reasons, due to complex dependencies grown among program entities throughout the software evolution. Moreover, such dependencies may not be explicitly found by analyzing a single version of the source code. In that context, authors have recently investigated a property called Logical coupling, which reveals implicit dependencies between program entities, by measuring how often they changed together during development [Robbes et al. 2008]. Coupling has been proved useful in a variety of software daily activities, such as maintenance effort estimation [Gupta and Rohil 2012], program comprehension [Briand et al. 1999], design flaws detection [Marinescu 2004]. Due to its importance, logical coupling has been studied with different purposes, such as: detection of architectural weaknesses, poorly designed inheritance hierarchies, blurred interfaces of modules [Gall et al. 2003], and change impact analysis [Rolfsnes et al. 2016]. In all of those cases, the comprehension of how the logical coupling property manifests itself in the project is a key activity.

Software visualization (SoftVis) has been effectively used as a way to tackle with software comprehension activities using high amount of data. SoftVis supports engineers in understanding software and building knowledge through visual elements, with less complexity over a large amount of data which is often generated during software evolution [Diehl 2007]. The comprehension of logical coupling is particularly addressed in the field of Software Evolution Visualization (SEV), as the property of logical coupling is measured by analyzing how often software modules change together in their evolution.

In the last two years, we have been working on the specification, development and evaluation of a novel SEV tool, called EVOWAVE [Magnavita 2016]. EVOWAVE realizes a novel visualization metaphor [Magnavita et al. 2015]. It is able to visualize different types of data generated in software evolution using both overview-based and detail-based approaches. EVOWAVE is able to represent a huge number of historical events at a glance. Several mechanisms of interactions allow the user to explore the visualization in detail. EVOWAVE has one perspective, implemented in a circular layout. This open source tool can be applied to different software engineering tasks and domains. To validate its benefits, we conducted three exploratory studies using EVOWAVE in three different software engineering domains: software collaboration [Magnavita et al. 2015], library dependency, and logical coupling.

In this paper, we present the logical coupling study. As in the other two studies, we selected a close related work [D'Ambros et al. 2009], and replicated the study they conducted. Our goal was to show the benefits of EVOWAVE in a new domain (logical coupling) considering an already published work, and also to comparatively highlight the benefits of our tool.

Other works provide a similar visualization layout for software evolution analysis [Kula et al. 2014][D'Ambros et al. 2009][Burch and Diehl 2008]. In [Kula et al. 2014], the authors proposed an evolution-based visualization of the coupling relationship between system modules and their required libraries. Although providing a similar visual representation, they do not focus on logical coupling relations. D'Ambros et al. present a visualization-based approach, called Evolution Radar [D'Ambros et al. 2009], that integrates logical coupling information at different levels of abstraction (package level and file level). In addition, they illustrated a retrospective analysis driven by logical coupling in the context of the ArgoUML project involving its maintenance activities such as restructuring and re-documentation. Despite the fact EVOWAVE has been used in the same domain, we believe our tool has advantages against Evolution Radar since it is a fresh web-based open source tool, highly configurable, and supports visualization of events of any kind across time (besides software evolution properties).

In our context, we use the EVOWAVE, a multiple domain software evolution visualization metaphor, to perform the same exploratory study performed in [D'Ambros et al. 2009] on the analysis of ArgoUML's logical coupling information. Therefore, this paper contributes by showing the study on the ArgoUML's logical coupling history using a single perspective software evolution visualization.

This paper is organized as follow. Section 2 presents a brief description of EVOWAVE. Section 3 presents the study we conducted to evaluate EVOWAVE in the logical coupling domain. Finally, Section 4 concludes this work.

2. EVOWAVE in a Nutshell

EVOWAVE is a new visualization tool that enriches the analysis capabilities of software evolution. It is inspired on concentric waves with the same origin point in a container seen from the top. This section briefly presents the concepts related to the EVOWAVE. Figure 1 shows one example of EVOWAVE. It portrays a logical coupling data history of ArgoUML project. For the sake of understanding, this example is decorated (mockup) with labels pointing to the main concepts of the metaphor.

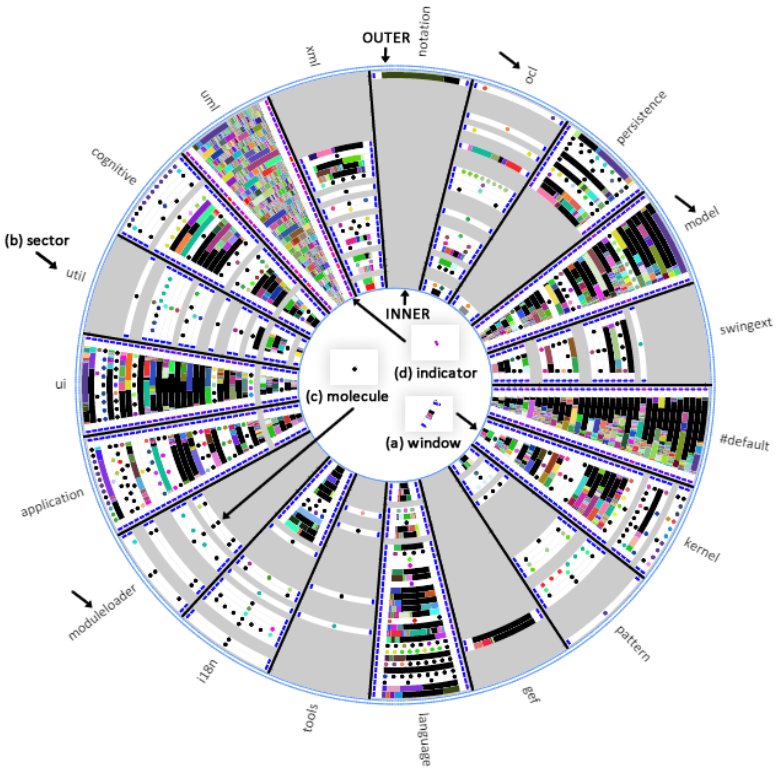


Figure 1. EVOWAVE visualization with all changes separated by modules from 2003 to 2005 with the "uml" package in focus.

EVOWAVE has a circular layout with two circular guidelines (***inner*** and ***outer***). The path between them represents a software life cycle period between two selected dates. This period, named ***timeline***, gives an overview of the software history. It is comprised by a series of short periods with the same periodicity (e.g., ten days, two hours, one month).

A ***window*** is a group of consecutive short periods (Figure 1-a). It is circular in shape and its length depends on the number of grouped periods. It can be used to compare a subset of short periods, making it possible to carry out a detailed analysis regarding the overall context. A ***sector*** is a visual element drawn between the two circular guidelines according to its angle (Figure 1-b). It may have different angles. This concept is used to group events that share some characteristic (e.g., classes of the same packages).

Molecules are depicted as circular elements inside sectors and windows (Figure 1-c). Each molecule has an event associated to it. When we have more molecules than the display size limits, we gathered and drawn them as a quadrilateral polygon that fills the

region where the molecules are. Molecules can represent any change on software history, such as file changes, team changes or bug reports. The molecules' color can be used to map software properties as an event categorization or a numerical property range. The number of molecules ***indicator*** is drawn as a rectangle located in the frontier of the sectors for each window (Figure 1-d). Its color varies from red to blue, where the reddest indicator has the largest number of molecules and the bluest has lowest number of molecules. The indicator can refer to the local sector or to all sectors. If set to local, its color will take in consideration the other windows inside the sector. Otherwise, if set to all sectors (global), its color will take in consideration all windows present in the visualization.

EVOWAVE provides a set of ***mechanisms of interaction***. They allow users, for example, to navigate (zoom in) into sectors of interest or to highlight a window across all sectors. In this last case, it mitigates possible problems that occurs when we have to much data and many used colors.

3. An Exploratory Study on Logical Coupling Domain

This section presents the study we conducted to validate EVOWAVE on the analysis of logical coupling between software modules during its history. Using the GQM (Goal-Question-Metric) paradigm [Basili and Rombach 1988], the goal of this study was: **To analyze** the EVOWAVE metaphor; **With the purpose of** characterize; **Regarding** logical coupling evolution between software modules; **From the point of view** of EVOWAVE researchers; **In the context of** a retrospective analysis made by D'Ambros [D'Ambros et al. 2009] in a real world large open source system called ArgoUML.

3.1. Study Settings

ArgoUML is the most well-known open source UML modeling tool written in Java with more than 17 years of development and at least 200,000 lines of code distributed in 4,222 classes.

Setup. We developed a parser to read Subversion logs and identify the files that were modified together within the "org.argouml.uml" package (which is the root package of the software). For each commit, we extracted the following data: the changed files and their package; when the change occurred; and the commit id. With this data we setup the EVOWAVE as follows: **The period of analysis** involved 173 months, from September 4th, 2000, to January 11th, 2015; **Sectors** are source code packages; **Windows** represent six months; **Molecules** represent changes in a Java file; and **Molecule Colors** represent the commit of the change. When a package is selected the commits in black represent revisions that did not make changes in the source code.

Tasks. We followed the same approach of [D'Ambros et al. 2009], which uses the logical coupling between modules to understand implicit dependencies in the system. The main goal is to make a retrospective analysis of the logical coupling evolution in the ArgoUML project.

3.2. Study Execution

First, we made an overview analysis in order to understand how the changes were spread among the system modules (i.e. Java packages). Figure 2-A helps us to identify packages with most changes. The root package "org.argouml.uml" underwent most of the changes

during fourteen years of development, followed by “#default” and “kernel” respectively. In addition, only one window in “org.argouml.uml” is empty (gray color arc in the package sector), while “#default” has two and ”kernel” has three. This means that, within the period of 173 months analyzed, there was only one period of 6 months without changes in the classes of the root package. Therefore, we decided to analyze the logical coupling of such package (“org.argouml.uml”) due to: 1) the amount of changes in it; 2) it was intensively changed from the beginning to the present date (almost fifteen years); and 3) it is an important package as it contains core classes for the UML tool.

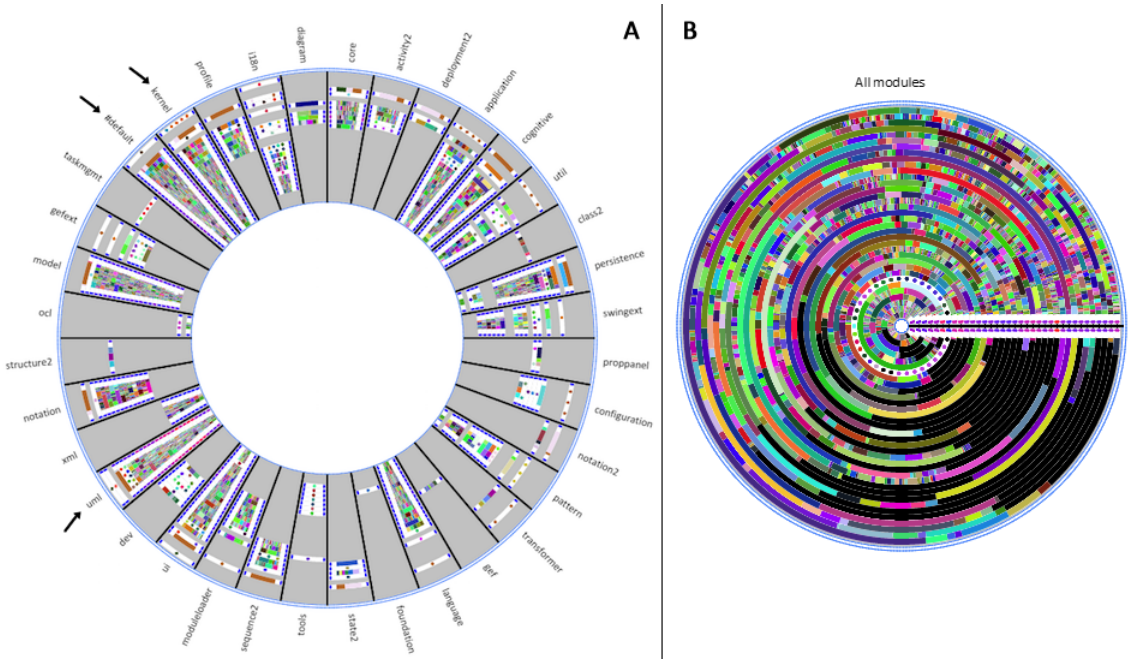


Figure 2. A) EVOWAVE visualization with all module changes during fifteen years of development; B) EVOWAVE visualization with all module changes from 2003 to 2005, focused on the “uml” package.

We filtered the period limiting it from January 2nd, 2003, to December 30th, 2005, in order to analyze the logical coupling of the package “uml” during the period with most changes. Additionally, for more detailed information, we changed the period of the window from six to one month. Figure 2-B illustrates the visualization with this filter and aggregates all modules into one sector to get a global picture of the logical coupling of the “uml” package. It is possible to notice that most commits caused changes in the “uml” package, as the black color has low presence in the visualization. Another worth observing information from this global picture is the number of commits with several files undergoing changes together. This can be noticed by looking at the size of the arcs with the same color. This may be interpreted as a problem to be further investigated in a project because the higher the number of co-changed files, the higher is the logical coupling. Therefore, one possible consequence of such high degree of interdependency among source files is an increasing occurrence of side effects during maintenance activities.

To investigate in more details, we analyzed the logic coupling of the “uml” package with other system modules, supported by the visualization illustrated in Figure 1.

Due to space restriction, we selected four packages (*moduleloader*, *ocl*, *model*, and *uml*), to explain how someone could recover more information using EVOWAVE and thus improving comprehension and discovering knowledge about the project.

The package "moduleloader" has a low logical coupling with the "uml" package. Nevertheless, in the fifth early month in the "moduleloader" package there were three changes in the same commit that changed almost all modules. Looking deeper into the comment of this commit and into the changes, we identified they were related to the copyright style and impacted 1,065 files. In terms of source code those changes had no impact, but let us know that probably every single file must change when there is a new copyright information.

The package "ocl" is highly logical coupled with the "uml" package. This is observed because there is almost no black color in the "ocl" sector. Among the changes, there are two commit highlights in the forth newer window: the green and red colors. The commit related to the green color is the copyright change that we demonstrate while analyzing the "moduleloader" package. The commit represented by the red color seems to impact many packages. While analyzing the comment and the changes from this commit, we identified a major change in the system. The class responsible to be the facade for all communications with the "model" package changed its signature from "ModelFacade" to "Model.getFacade()". This is a big change in the system because it breaks the signature used in many packages (e.g., ocl, persistence, uml). The logical coupling between the "ocl" package and the "uml" package for this commit is a consequence of their coupling with the model package.

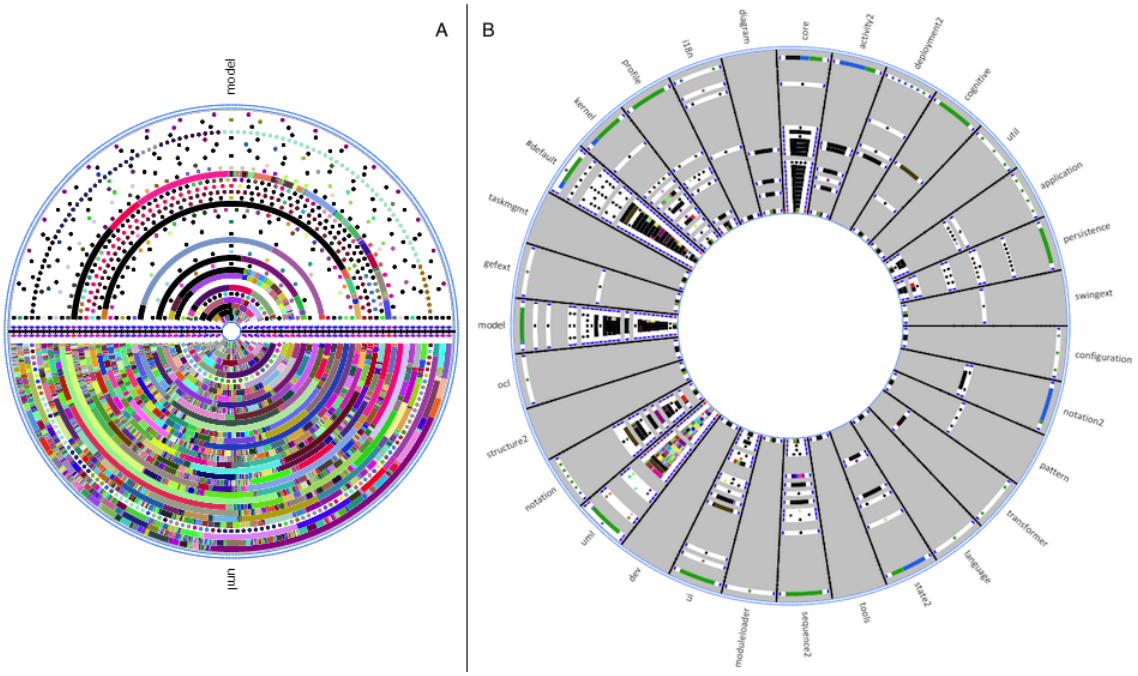


Figure 3. A) EVOWAVE visualization to analyze the logical coupling between the "uml" and "model" packages. B) EVOWAVE visualization from January 2, 2010 to February 8, 2013

The "model" package holds a lot of data because it changed many times during

this period. In this case, we filtered the visualization to show only the "uml" package and the "model" package in order to better understand their logical coupling. Figure 3-A represents the visualization with this filter. The first point to notice is the amount of changes in those packages. Clearly the "uml" package had more changes than the "model" package during this period. Another important point is the frequency of black and other colors. Black molecules are present in practically all months. Nevertheless, there are many colors (i.e. many different commits) present in windows. This leads to the conclusion that many commits were submitted in that month, and they revealed logical coupling with the "uml" package. The eleventh newest window (January, 2005) has the highest number of colors, and was one of the months with most changes. Looking deeper into the commits submitted in that month, we found that those packages are coupled by an architecture decision to have a Facade design pattern to access the "model" package. The Facade pattern can be defined as an entry point to features joined in a module to reduce the need for one module to known another module completely. Having this design pattern as the reason for the coupling of those two packages is a good sign of good architectural decisions throughout the software development.

Additionally, we filtered the period limiting it from January 2nd, 2010 to February 8th, 2013 in order to analyze the logical coupling of the package "uml" in more recent data. Figure 3-B illustrates the visualization for this period. The first thing we noticed was the reduction of the logical coupling among the "uml" package and the rest of the system. The reason for this might have been refactoring activities performed between 2005 and 2010 to reduce the coupling between the "uml" package and the rest of the system. Nevertheless, the green commit at the second newest window stands out in this visualization. This commit impacts practically all active modules in the system. Looking more deeply into the commit and its changes, we identified that the reason was a change in the logging library from log4j to the native Java logging API.

4. Final Remarks

Logical coupling reveals implicit dependencies between program entities. It is useful in a variety of software daily activities. A challenge is to comprehend how the logical coupling property manifests itself in the project. To this end, Software Evolution Visualization is a promising approach.

In this work, we used the EVOWAVE, a multiple domain software evolution visualization metaphor, to perform a exploratory study on the analysis of ArgoUML's logical coupling information. This study showed we were able to evaluate the EVOWAVE in another domain (i.e. logical coupling). We used the same study procedures as in [D'Ambros et al. 2009]. In addition to the fact it is one more domain, we can highlight other outcomes of our tool as it is an fresh open source tool, web-based, and highly configurable.

The exploratory study conducted helped us to see the outcomes and shortcomings of EVOWAVE. As outcomes, it indicated the EVOWAVE usefulness on the logical coupling analysis of a large software system. As shortcomings, it indicated the need of tool improvement. More mechanisms of interaction are necessary to speedup the analysis, specially by end users. The exploratory study itself has limitations, since it is conducted by the authors' perspective. As future work, we plan to conduct other experimental stud-

ies in order to evaluate EVOWAVE in the context of logical coupling domain, however considering end users.

References

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

[Briand et al. 1999] Briand, L. C., Daly, J. W., and Wüst, J. K. (1999). A unified framework for coupling measurement in object-oriented systems. *IEEE Trans. Softw. Eng.*, 25(1):91–121.

[Burch and Diehl 2008] Burch, M. and Diehl, S. (2008). Timeradartrees: Visualizing dynamic compound digraphs. *Computer Graphics Forum*, 27(3):823–830.

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

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

[Gall et al. 2003] Gall, H., Jazayeri, M., and Krajewski, J. (2003). CVS release history data for detecting logical couplings. In *Proceedings of the 6th International Workshop on Principles of Software Evolution*, IWPSE '03, pages 13–, Washington, DC, USA. IEEE Computer Society.

[Gupta and Rohil 2012] Gupta, N. K. and Rohil, M. K. (2012). *Object Oriented Software Maintenance in Presence of Indirect Coupling*, pages 442–451. Springer Berlin Heidelberg, Berlin, Heidelberg.

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

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

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

[Marinescu 2004] Marinescu, R. (2004). Detection strategies: metrics-based rules for detecting design flaws. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 350–359.

[Robbes et al. 2008] Robbes, R., Pollet, D., and Lanza, M. (2008). Logical coupling based on fine-grained change information. In *2008 15th Working Conference on Reverse Engineering*, pages 42–46.

[Rolfsnes et al. 2016] Rolfsnes, T., Alesio, S. D., Behjati, R., Moonen, L., and Binkley, D. W. (2016). Generalizing the analysis of evolutionary coupling for software change impact analysis. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 201–212.

Inferência de Tipos em Ruby: Uma comparação entre técnicas de análise estática e dinâmica

Sergio Miranda[1], **Marco Tulio Valente**[1], **Ricardo Terra**[2]

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

[2]Universidade Federal de Lavras, Lavras, Brasil

{sergio.miranda,mtov}@dcc.ufmg.br, terra@dcc.ufla.br

Resumo. *Informações sobre tipos que variáveis podem assumir é importante para desenvolvedores de software, uma vez que os apoiam no entendimento do código, aumentam a confiança para realizar atividades de refatoração, auxiliam na detecção de erros, etc. No entanto, como característica intrínseca de linguagens dinamicamente tipadas, não há definições explícitas de tipos de variáveis. Quando necessário, a inferência de tipos é realizado por técnicas de análise dinâmica, as quais requerem a execução do sistema e possuem alto custo computacional. Diante disso, este artigo avalia o uso de análise estática—em comparação com análise dinâmica—na inferência de tipos em linguagens dinamicamente tipadas. Uma análise em seis sistemas de código aberto demonstrou que, em média, um algoritmo elementar por análise estática já é capaz de inferir 44% dos tipos. Uma análise qualitativa apontou sete melhorias (duas simples, duas moderadas e três complexas) que devem ser aplicadas ao algoritmo elementar para se obter resultados equivalentes aos obtidos por análise dinâmica.*

1. Introdução

Linguagens dinâmicas não impõem restrições de tipos durante o desenvolvimento, ou seja, execuções corretas do sistema de software não são impedidas de forma prematura. Apesar de agilizar o desenvolvimento, essa característica traz certas desvantagens [Agesen et al. 1995]. Informações de tipo podem apoiar positivamente diversas tarefas de manutenção, evolução e compreensão de software, pois (a) tornam o código mais legível e auto-documentável; (b) facilitam a implementação de ferramentas de análise de código; (c) tornam refatorações mais seguras e confiáveis; (d) permitem a construção de melhores IDEs (por exemplo, com suporte a recursos de auto-completar); e (e) permitem a detecção antecipada de erros, garantindo que todas mensagens enviadas sejam entendidas [Palsberg and Schwartzbach 1991].

Informações sobre os tipos das variáveis não são fáceis de serem obtidas em linguagens dinâmicas em comparação com linguagens estaticamente tipadas. Quando necessário, a inferência de tipos ocorre por meio de técnicas de análise dinâmica, as quais requerem a execução do sistema e possuem alto custo computacional [Agesen and Holzle 1995]. Isso dificulta o trabalho de ferramentas que auxiliam desenvolvedores em atividades de melhoria da qualidade do sistema de software. Por exemplo, ferramentas para garantia de conformidade arquitetural, ferramentas para realizar refatoração automaticamente, etc., podem se beneficiar de informações sobre tipos. Especificamente, ArchRuby – um verificador de conformidade arquitetural para a linguagem Ruby – utiliza uma heurística *estática* de inferência de tipos [Miranda et al. 2016, Miranda et al. 2015].

Neste artigo, a *questão de pesquisa* se resume em verificar os desafios do uso de análise estática para inferência de tipos em linguagens dinamicamente tipadas. Portanto, este artigo compara as informações de tipos coletadas através de análise estática com informações coletadas através de análise dinâmica. Para análise estática, utilizou-se um algoritmo para inferência de tipos com uma heurística simples [Miranda et al. 2016]. Já para análise dinâmica, foram utilizados dados coletados durante a execução do sistema de software. O objetivo é comparar as duas situações e prover melhorias para que a análise estática possa gerar resultados equivalentes àqueles proporcionados por técnicas de análise dinâmica. Uma análise quantitativa em seis sistemas de código aberto demonstrou que, em média, um algoritmo elementar por análise estática é capaz de inferir 44% dos tipos. Complementarmente, uma análise qualitativa apontou sete melhorias que devem ser aplicadas para se obter resultados equivalentes aos obtidos por análise dinâmica.

O restante deste artigo está organizado como a seguir. A Seção 2 detalha a heurística de inferência de tipos utilizada. A Seção 3 apresenta detalhes da avaliação conduzida, tais como questões de pesquisa propostas, projeto do estudo e resultados encontrados. A Seção 4 apresenta trabalhos relacionados. Por fim, a Seção 5 conclui.

2. Inferência de Tipos

Existem diversos algoritmos para realizar inferência de tipos em linguagens dinâmicas [Palsberg and Schwartzbach 1991, Agesen and Holzle 1995], sendo que alguns necessitam de anotações para auxiliar no processo [Furr et al. 2009]. Consequentemente, a necessidade de anotar o código dificulta o entendimento e a adoção desses algoritmos. Este artigo, portanto, parte de um algoritmo proposto para a linguagem Ruby que é baseado em análise estática [Miranda et al. 2016, Miranda et al. 2015]. Embora dinamicamente tipada, é possível inferir em Ruby parte dos tipos de variáveis e parâmetros formais. O algoritmo escolhido utiliza-se de uma heurística – mais especificamente, uma simplificação da heurística formalizada por Furr et al. [Furr et al. 2009] – que visa construir um conjunto `TYPES` de triplas $[\texttt{method}, \texttt{var\_name}, \texttt{type}]$, onde `type` é um dos possíveis tipos inferidos para a variável ou parâmetro formal `var_name` do método `method`. Esse conjunto é construído de acordo com a seguinte definição recursiva:

i) **Base**: Para cada inferência direta (e.g., instanciação) de um tipo `T` de uma variável `x` em um método `f`, então $[\texttt{f}, \texttt{x}, \texttt{T}] \in \texttt{TYPES}$.

ii) **Passo recursivo**: Se $[\texttt{f}, \texttt{x}, \texttt{T}] \in \texttt{TYPES}$ e existir em `f` uma chamada $\texttt{g}(\texttt{x})$, então $[\texttt{g}, \texttt{y}, \texttt{T}] \in \texttt{TYPES}$, onde `y` é o nome do parâmetro formal de `g`. Esse passo é aplicado até um ponto fixo ser atingido, i.e., até nenhum novo elemento ser adicionado ao conjunto de `TYPES`.

A Listagem 1 ilustra o funcionamento da heurística proposta [Miranda et al. 2016]. Ao executar o passo base do algoritmo, `TYPES` é inicializado com $[\texttt{A::f}, \texttt{x}, \texttt{Foo}]$, $[\texttt{A::f}, \texttt{b}, \texttt{B}]$, $[\texttt{A::f}, \texttt{self}, \texttt{A}]$, $[\texttt{B::g}, \texttt{c}, \texttt{C}]$ e $[\texttt{C::h}, \texttt{d}, \texttt{D}]$ já que são as inferências diretas. Na primeira aplicação do passo recursivo, as triplas $[\texttt{B::g}, \texttt{x}, \texttt{Foo}]$ e $[\texttt{B::g}, \texttt{z}, \texttt{A}]$ são incluídas em `TYPES`, uma vez que se conhece o tipo das variáveis `x` e `self` na chamada de `g`. Na segunda aplicação do passo recursivo, as triplas $[\texttt{C::h}, \texttt{y}, \texttt{Foo}]$ e $[\texttt{C::h}, \texttt{y}, \texttt{A}]$ são incluídas em `TYPES`, uma vez que se conhece o tipo das variáveis `x` e `z` na chamada de `h`. Na terceira aplicação do passo recursivo, as triplas $[\texttt{D::m}, \texttt{k}, \texttt{Foo}]$ e $[\texttt{D::m}, \texttt{k}, \texttt{A}]$ são incluídas em `TYPES` (onde `k` é o nome do parâmetro em `D`), uma vez que se

conhece o tipo da variável `y` na chamada de `m`. A aplicação do passo recursivo se repete até que nenhuma nova tripla seja adicionada ao conjunto.

```
1 class A                    class B                    class C
2   def f                      def g(x z)                 def h(y)
3     x = Foo.new                c = C.new                d = D.new
4     b = B.new                  c.h(x)                   d.m(y)
5     b.g(x,self)                c.h(z)                  end
6   end                        end                      end
7 end                        end
```

Listagem 1. Código para ilustração da heurística de inferência de tipo

Nesse exemplo, é importante notar que o parâmetro formal `y` do método `C::h` pode ser do tipo `A` ou `Foo`. Isso indica que o mecanismo de propagação de tipos deve considerar todos os potenciais tipos de uma variável ou parâmetro formal.

3. Estudo Comparativo

Nesta seção é avaliado o algoritmo descrito na seção anterior utilizando seis sistemas de código aberto desenvolvidos em Ruby. A avaliação compara informações de tipos geradas através da execução dos testes automatizados dos sistemas (i.e., análise dinâmica) com as informações de tipos geradas pelo algoritmo descrito na Seção 2 (i.e., análise estática).

3.1. Questões de Pesquisa

Este estudo tem como objetivo responder às seguintes questões de pesquisa:

- ***QP #1*** – Qual a acurácia de abordagens estáticas de inferência de tipos em comparação com as dinâmicas?
- ***QP #2*** – Como é possível ampliar a acurácia de abordagens estáticas de inferência de tipo?

3.2. Projeto do estudo

Dataset: Avaliou-se o algoritmo de inferência de tipos em seis sistemas de código aberto desenvolvidos em Ruby. Os critérios de escolha foram: (i) possuir testes automatizados e (ii) ser executável com a versão 2.1 ou superior da linguagem Ruby, pois a máquina virtual da linguagem só oferece maneiras de inspecionar o *runtime* a partir de tal versão. A Tabela 1 apresenta os sistemas escolhidos juntamente com informações sobre a porcentagem de cobertura de testes[1] e quantidade de linhas de código.

Oráculo: Considerou-se como oráculo os tipos inferidos pelos testes automatizados dos sistemas. Embora essa decisão possa gerar falsos negativos, procurou-se minimizar tal problema selecionando sistemas com elevada porcentagem de cobertura de testes (média de 89%, conforme reportado na Tabela 1).

Métricas: As informações de tipos foram coletadas de dois modos: (i) pela execução dos testes automatizados dos sistemas (i.e., inspecionando o *runtime* e armazenando todos os tipos gerados durante a execução) e (ii) pela execução do algoritmo estático descrito na Seção 2. Foram gerados pares ordenados para representar os tipos coletados. Suponha que

[1]Foi utilizada a ferramenta SimpleCov, disponível em: https://github.com/colszowka/simplecov.

Tabela 1. Sistemas avaliados

Sistema	Cobertura de Testes	LOC
Capistrano	94%	1.779
CarrierWave	81%	2.636
Devise	97%	3.683
Resque	84%	2.315
Sass	95%	13.609
Vagrant	87%	8.429
Média	**89%**	**5.408**

uma variável `x` é do tipo `A`, logo a representação é dada pelo par ordenado $(\texttt{x}, \{\texttt{A}\})$. Dessa maneira, são propostas duas métricas para auxiliar na comparação dos resultados. A primeira, denominada $Recall_1$ e formalizada na Equação 1, tem como objetivo mensurar a quantidade de tipos que o algoritmo analisado conseguiu inferir de forma exata quando comparado com os tipos obtidos durante a execução dos testes automatizados.

$$Recall_1 = \frac{\mid Static \cap Dynamic \mid}{\mid Dynamic \mid} \tag{1}$$

Para ilustrar $\texttt{Recall}_1$, assuma as seguintes tuplas coletadas durante a execução dos testes automatizados, $(\texttt{a}, \{\texttt{X}\})$, $(\texttt{b}, \{\texttt{Y}\})$ e $(\texttt{c}, \{\texttt{W}, \texttt{Z}\})$. Em seguida, assuma que as tuplas $(\texttt{a}, \{\texttt{X}\})$, $(\texttt{b}, \{\texttt{Y}\})$ e $(\texttt{c}, \{\texttt{W}\})$ foram obtidas pelo algoritmo estático em avaliação. Consequentemente, ao realizar o cálculo de $\texttt{Recall}_1$, o valor encontrado é 67% (2/3), pois apenas as tuplas $(\texttt{a}, \{\texttt{X}\})$ e $(\texttt{b}, \{\texttt{Y}\})$ estão presentes na interseção entre as duas abordagens.

De forma complementar, também é calculado uma segunda métrica denominada $Recall_2$ e formalizada na Equação 2. Essa métrica tem como objetivo analisar a quantidade de variáveis que o algoritmo estático conseguiu coletar, mesmo que parcialmente. A função $variáveis$ retorna o conjunto de variáveis que tiveram tipos obtidos.

$$Recall_2 = \frac{\mid variáveis(Static) \cap variáveis(Dynamic) \mid}{\mid Dynamic \mid} \tag{2}$$

Reconsidere o exemplo anterior para ilustrar o cálculo do valor de $\texttt{Recall}_1$. É importante notar que o algoritmo avaliado inferiu para a variável `c` apenas o tipo $\{\texttt{W}\}$. Dessa maneira, o cálculo de $\texttt{Recall}_2$ seria 100% (3/3), já que o algoritmo avaliado conseguiu inferir pelo menos um tipo para todas as três variáveis (`a`, `b` e `c`).

3.3. Resultados

QP #1 – Qual a acurácia de abordagens estáticas de inferência de tipos em comparação com as dinâmicas?

Após a execução dos testes automatizados e do algoritmo estático em avaliação foram calculados os valores para as métricas descritas na Seção 3.2. A Tabela 2 apresenta os resultados encontrados para os seis sistemas avaliados. Na média, o valor para $\texttt{Recall}_1$ é de 44,4%, já para $\texttt{Recall}_2$ esse valor sobe para 58,1%. Os números entre parênteses representam os valores para o numerador e denominador das fórmulas de $\texttt{Recall}_1$ e $\texttt{Recall}_2$. O maior $\texttt{Recall}_1$, 50,8%, foi para o sistema `CarrierWave`, em que 62 das 122 variáveis foram inferidas pela abordagem estática. Já para $\texttt{Recall}_2$, o sistema `Sass` obteve o maior resultado (58,4%), em que 135 variáveis foram inferidas pela abordagem estática pelo menos parcialmente.

Tabela 2. Resultados

Sistema	$Recall_1$	$Recall_2$
Capistrano	39,0% (30/77)	45,5% (35/77)
CarrierWave	50,8% (62/122)	57,4% (70/122)
Devise	43,8% (155/354)	56,5% (200/354)
Resque	45,0% (9/20)	55,0% (11/20)
Sass	46,0% (500/1.088)	58,4% (635/1.088)
Vagrant	40,8% (220/539)	55,7% (300/539)
Média	**44,4%**	**58,1%**

A Listagem 2 ilustra uma situação onde o algoritmo analisado não foi capaz de inferir o tipo de uma variável para o sistema `Vagrant`.

```
1  def filter(command)
2    command_filters.each do |c|
3      command = c.filter(command) if c.accept?(command)
4    end
5    command
6  end
7  def create_command_filters
8    [].tap do |filters|
9      @@cmd_filters.each do |cmd|
10       require_relative "command_filters/#{cmd}"
11       class_name = "VagrantPlugins::CommunicatorWinRM::
12                     CommandFilters::#{cmd.capitalize}"
13       filters << Module.const_get(class_name).new
14     end
15   end
16 end
```

Listagem 2. Código de exemplo do sistema Vagrant

O *loop* da linha 2 percorre o arranjo populado no método `create_command_filters` (linha 7) para atribuir um valor à variável `command` (linha 3). Ainda, o método `create_command_filters` utiliza técnicas de reflexão para atribuir valores ao arranjo (linhas 11-13), ou seja, os valores só serão conhecidos em tempo de execução. Logo, pode ser complexo ou ate mésmo inexequível para uma análise estática inferir o tipo que será atribuído a tal variável.

De forma similar, a Listagem 3 ilustra um segundo exemplo onde não foi possível detectar o tipo de uma variável para o sistema Capistrano.

```
1  def fetch(key, default=nil, &block)
2    value = fetch_for(key, default, &block)
3    while callable_without_parameters?(value)
4      value = set(key, value.call)
5    end
6    return value
7  end
```

Listagem 3. Código de exemplo do sistema Capistrano

A variável `value` é inicializada na linha 2. Com o valor retornado pelo método `fetch_for`. No entanto, a variável `value` pode ter seu valor alterado no *loop* execu-

tado entre as linhas 3-5. Similarmente à situação apresentada no exemplo anterior, a variável `value` também tem seu valor definido de forma dinâmica, impondo dificuldade para detecção através de análise estática de código.

Por último, a Listagem 4 demonstra uma situação no sistema Devise, onde o valor de uma variável é obtido através de acesso a estrutura de dados `Hash`. A linha 5 atribui, à variável `skip`, o valor que está armazenado na chave `:skip_helpers` do *hash* `options`. Uma análise estática deveria armazenar os valores que são passados como chave e valor para tal estrutura para ser capaz de detectar qual o tipo atribuído à variável `skip`.

```
1  def default_used_helpers(options)
2    singularizer = lambda { |s| s.to_s.singularize.to_sym }
3    if options[:skip_helpers] == true
4      @used_helpers = @used_routes
5    elsif skip = options[:skip_helpers]
6      @used_helpers = self.routes - Array(skip).map(&singularizer)
7    else
8      @used_helpers = self.routes
9    end
10 end
```

Listagem 4. Código de exemplo do sistema Devise

QP #2 – Como é possível ampliar a acurácia de abordagens estáticas de inferência de tipo?

Uma análise qualitativa dos resultados indicam possíveis mudanças que podem auxiliar na melhora dos resultados obtidos. Essas mudanças são classificadas de acordo com a sua complexidade:

- *Simples*: Considerar a funcionalidade de `include` e `extend` oferecida pela linguagem Ruby, as quais implementam o suporte a *mixins* para implementar herança múltipla [Bracha and Cook 1990]. Assim, algoritmos de análise estática deveriam simular o funcionamento dessas funcionalidades armazenando todos os métodos definidos em módulos que são incluídos em classes através da utilização de `include` e `extend`.
- *Moderada*: Considerar a execução e retorno de blocos, os quais são considerados funções de primeira ordem em Ruby, podendo ser armazenados em variáveis e passados como parâmetros. Assim, o processo de análise estática deveria acompanhar as mudanças que são realizadas pela execução de blocos.
- *Complexa*: Considerar valores armazenados em estruturas de dados, tais como `Array`, `Hash`, `Set`, etc. Como Ruby inclui reflexão e avaliação dinâmica de código (e.g., eval), o algoritmo deveria ser capaz de analisar instruções dinâmicas. Por exemplo, `instance_eval`, `class_eval` e `define_method` são funcionalidades que alteram o programa em tempo de execução. Deve-se também considerar a atribuição de valores a variáveis de instância de objetos, o que requer que a análise estática tenha ciência do contexto de execução e do fluxo de atribuições.

A Tabela 3 reporta os resultados de uma análise manual do impacto das mudanças propostas nos sistemas avaliados.

Tabela 3. Análise de impacto das mudanças propostas no valor de `Recall`$_1$

	$Recall_1$								
	Trivial	**Simples**		**Moderada**		**Complexa**			
Sistema	**base**	**include**	**extend**	**retorno blocos**	**execução blocos**	**valores em estrutura**	**instr. dinâmica**	**fluxo de execução em variáveis de instância**	**Total**
Capistrano	39,0%	5,2%	1,3%	14,3%	13,0%	15,6%	1,3%	10,3%	**100%**
CarrierWave	50,8%	6,6%	1,6%	11,5%	13,1%	9,8%	2,5%	4,1%	**100%**
Devise	43,8%	2,8%	1,4%	12,7%	15,5%	9,0%	6,3%	8,5%	**100%**
Resque	45,0%	0%	0%	20,0%	15,0%	10,0%	0%	10,0%	**100%**
Sass	46,0%	0,6%	0,6%	13,8%	18,4%	12,9%	3,6%	4,1%	**100%**
Vagrant	40,8%	2,8%	0,4%	13,0%	22,3%	11,1%	2,2%	7,4%	**100%**
Média	**44,4%**	**2,0%**	**0,7%**	**13,4%**	**18,4%**	**11,7%**	**3,5%**	**5,9%**	**100%**

Note que com a implementação das modificações consideradas simples e moderadas é possível elevar o valor médio de `Recall`$_1$ para 78,9% (44,4+2,0+0,7+13,4+18,4), um aumento considerável para a análise estática. Ainda, adicionando as modificações consideradas complexas esse valor seria elevado para 100% (78,9+11,7+3,5+5,9). Como resultado mais significativo, é importante mencionar que, se implantadas as melhorias de complexidade moderada (retorno e execução de blocos), o `Recall`$_1$ elevaria de 44,4% para 76,2%. Logo, seriam essas as melhorias que trariam o maior benefício imediato a algoritmos estáticos de inferência de tipo.

4. Trabalhos Relacionados

Outros trabalhos investigaram o problema de se aplicar análise estática para realizar inferência de tipos em linguagens dinâmicas. Em um dos primeiros trabalhos, Palsberg e Schwartzbach [Palsberg and Schwartzbach 1991] investigam o uso de análise estática de tipos em uma linguagem dinâmica inspirada em Smalltalk. Por ser uma linguagem fictícia, construída para ilustrar os conceitos apresentados no trabalho, várias funcionalidades de linguagens dinâmicas são desconsideradas (e.g., reflexão, funções de primeira ordem, etc.). Neste trabalho, por outro lado, utilizou-se uma linguagem real sem desconsiderar quaisquer de suas funcionalidades.

Em um estudo recente com Ruby, Furr et al. avaliam o uso de algoritmo de inferência de tipos de forma estática, juntamente com o auxílio de anotações, para auxiliar desenvolvedores a capturarem erros sem a necessidade de executar o programa [Furr et al. 2009]. Neste artigo, o algoritmo estático avaliado é uma versão simples e não invasiva (sem uso de anotações) do algoritmo proposto pelos autores.

Morison [Morrison 2006] desenvolveu um algoritmo de inferência de tipos para Ruby que foi integrado ao IDE RadRails, utilizada por programadores que utilizam o *framework* Rails. Em linhas gerais, o algoritmo realiza análise de fluxo para sugerir os possíveis métodos que podem ser invocados por um determinado objeto. Tal análise é similar à ideia proposta pelo algoritmo apresentado neste artigo, porém se diferencia por analisar fluxos de atribuições que são possíveis de serem feitos a variáveis de instância.

5. Conclusão

Linguagens dinâmicas não impõem restrições de tipos, porém a informação dos tipos que um objeto pode assumir auxilia os mantenedores de software em diversas tarefas, tais como refatorações, correções de *bugs* e detecção de violações arquiteturais. Diante disso, neste trabalho foram comparadas abordagens estática e dinâmica na inferência de tipos em seis sistemas de código aberto obtendo os seguintes resultados: (i) o algoritmo de análise estática foi capaz de inferir, na média, 44,4% dos tipos obtidos pela análise dinâmica e (ii) foram também sugeridas sete melhorias que, se aplicadas ao algoritmo de análise estática, obteria resultados equivalentes aos obtidos por análise dinâmica.

Como trabalho futuro, planeja-se executar os testes em um conjunto maior de sistemas para se obter uma maior confiabilidade estatística na extrapolação dos resultados. Posteriormente, pretende-se implementar as melhorias simples e moderadas, o que elevaria a acurácia apresentada pelo algoritmo de análise estática para 78,9%. Ainda, avaliar o desempenho do algoritmo estático e conduzir o mesmo estudo para outras linguagens dinâmicas – tais como Python e JavaScript – , cujas comunidades podem se beneficiar com informações de tipos.

Agradecimentos: Este trabalho foi apoiado pela FAPEMIG, CAPES e CNPq.

Referências

[Agesen and Holzle 1995] Agesen, O. and Holzle, U. (1995). Type feedback vs. concrete type inference: A comparison of optimization techniques for object-oriented languages. In *10th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 91–107.

[Agesen et al. 1995] Agesen, O., Palsberg, J., and Schwartzbach, M. I. (1995). Type inference of self: Analysis of objects with dynamic and multiple inheritance. *Software: Practice and Experience*, 25(9):975–995.

[Bracha and Cook 1990] Bracha, G. and Cook, W. (1990). Mixin-based inheritance. In *5th Conference on Object-Oriented Programming: System Languages, and Applications (OOPSLA)*, pages 303–311.

[Furr et al. 2009] Furr, M., An, J. D., Foster, J. S., and Hicks, M. (2009). Static type inference for Ruby. In *24th Symposium on Applied Computing (SAC)*, pages 1859–1866.

[Miranda et al. 2016] Miranda, S., Rodrigues, E., Valente, M. T., and Terra, R. (2016). Architecture conformance checking in dynamically typed languages. *Journal of Object Technology*, 15(3):1–34.

[Miranda et al. 2015] Miranda, S., Valente, M. T., and Terra, R. (2015). Conformidade e visualização arquitetural em linguagens dinâmicas. In *18th Ibero-American Conference on Software Engineering (CIbSE), Software Engineering Technologies (SET) Track*, pages 137–150.

[Morrison 2006] Morrison, J. (2006). Type inference in Ruby. In *Google Summer of Code Project*.

[Palsberg and Schwartzbach 1991] Palsberg, J. and Schwartzbach, M. I. (1991). Object-oriented type inference. In *6th Conference on Object-Oriented Programming: System Languages, and Applications (OOPSLA)*, pages 146–161.

From Formal Results to UML Model
A MDA Tracing Approach

Vinícius Pereira[1], **Rafael S. Durelli**[2], **Márcio E. Delamaro**[1]

[1]Instituto de Ciências Matemáticas e de Computação (ICMC)
Universidade de São Paulo (USP) — São Carlos – SP – Brazil

[2]Departamento de Ciência da Computação (DCC)
Universidade Federal de Lavras (UFLA) — Lavras – MG – Brazil.

{vpereira,delamaro}@icmc.usp.br, rafael.durelli@dcc.ufla.br

Abstract. *The Unified Modeling Language (UML) is the de-facto industrial standard for modeling object-oriented software systems. Nevertheless, the UML has some limitations such as: (i) ambiguity, (ii) semantic unclear, and (iii) lack of formal semantics. To deal with this, researchers propose UML transformations for formal models. These models are precise but difficult to be analyzed by people with no knowledge of formalism. This paper describes a model-driven approach which enable traceability of counterexamples from formal results back into UML Model. The traceability is made via a mapping between existing elements in the formal results and the elements present in the UML model. Using our approach, it is feasible to analyze and understand the formal results even without a thorough knowledge of formal methods, and consequently fix the UML model where needed.*

1. Introduction

Nowadays UML (Unified Modeling Language) is widely used among professionals from different areas of computer science [Hutchinson et al. 2011]. However, a system modeled by means of just UML is not complete in the sense of missing relevant information such as formal notation. For example, considering a car-collision avoidance system (CCAS), one can use UML to model such system, however, it is not possible to accurately describe situations where the brakes are triggered automatically. This occurs because the UML does not have a semantic free of ambiguities – i.e., UML does not have a formal notation. One possible way to fix this drawback in UML is using a temporal logic. Therefore, CCAS could be modeled using formal notation, free from ambiguity and with a clear semantics.

A formal model is almost the opposite model to UML. As formal models do not have the UML problems, they are more difficult to create and reuse than UML models. Furthermore, the developer must be a formalism expert to create, read, and understand a formal model. Consequently, it is more common to find developers working with UML than developers dealing with formal models. For this reason, over the years, researchers seek ways to unite the strengths of both formal models and UML. Usually the formalization of UML semantics found in the literature uses only one type of diagram [Eshuis 2006, Lund and Stolen 2006, Bouabana-Tebiel 2009, Micskei and Waeselynck 2011]: (i) State Diagram or (ii) Activity Diagram.

These researchers provide solutions that involve different formal techniques, but can be summarized by the following process: (i) it is necessary to formalize semantically

one or more UML diagrams, (ii) these diagrams are then converted to a formal model, (iii) a model checker formally checks this formal model, and (iv) a analysis is performed and the results are provided by the model checker. Although they are valid approach, them assume that the developer has sufficient knowledge of formalism to interpret the formal results, analyze them and make the necessary corrections in the UML model, and then repeat this whole process. However, if the developer has sufficient knowledge to perform this kind of analysis, he probably would use the formal model directly, instead. To the best of our knowledge, up to this moment, there is no research concentrated on analyzing and understanding the formal results even without a thorough knowledge of formal methods, and consequently fix the UML model where needed.

Therefore, this paper presents a generic Model-Driven Architecture (MDA) that defines a meta-model for the traceability of formal results within the UML model regardless of the formalization used in the UML model. By using this approach, the developer might not possess knowledge in formalism (or have little knowledge) and yet he will be able to follow step-by-step the formal results given by the model checker. The main contributions of this paper are threefold: (i) we show a new meta-model for describe formal results, (ii) we demonstrate the feasibility of our meta-model by implementing it as two DSLs (Mapping and Trace) in the Eclipse environment, (iii) we also show a complete example of how to use our meta-model and its DSLs.

2. Related Studies

Studies that formalize the UML semantics generally propose formalization of only one UML diagram, like State Diagram or Activity Diagram [Forster et al. 2007, Kaliappan and Konig 2012]. Other studies formalize two or more types of diagrams [Konrad et al. 2004, Broy et al. 2006], where can be cited more specifically the proposals of [Graw et al. 2000] and [Baresi et al. 2012] that formalize four and five diagrams, respectively. However, these proposals assume that the user can read and understand the formal results. mas All cited studies can have their formalization of UML semantics processes divided into three distinct stages: (i) **Modeling**, wherein the system to be developed is modeled using UML and the semantic concepts of each proposal; (ii) **Transformation**, wherein the UML model (and the assigned semantics) are transformed into a formal model according to the syntax and semantics of the formal language chosen by the proposal; and (iii) **Verification**, wherein the formal model is analyzed by a model checker that shows the formal results of this verification.

The difference between these studies and the MDA proposed in this paper is that the latter seeks a viable way to represent the information of the formal results (Traceability stage) regardless of which UML diagram was formalized. Also, there are two major details: the MDA represents the formal results in the semi-formal system model (UML model) itself without creating other UML diagrams for the results; and it might be used with any kind of formalization of UML semantics.

3. MDA Tracing Approach

Initially, to make it possible to represent the formal results within the UML model, two artifacts are required: (i) UML Model itself; and (ii) Formal Results, created by the model checker responsible for analyzing the formal model. Using this two artifacts, the approach has available both the formal environment and the UML semi-formal graphical

environment. However, the Transformation stage (see Section 2) is very particular to each type of formalization. To overcome this problem, we defined a third artifact called UML Mapping [Pereira et al. 2015] that properly connect both environments.

Our UML Mapping is compose of three elements: (i) ID-UML, a unique identifier to each UML element; (ii) ID-Formal, the identifier which is given to each UML element when the UML model is transformed to a formal model; and (iii) typeElement, which holds the element type at the UML model (e.g. Class, State, Message, etc.) Using the ID-UML, it is possible to identify the equivalent formal element. Similarly, using the ID-formal, it is also possible to identify the UML element by their respective ID-UML. We argue that as any UML graphical element owns an ID-UML in the UML Editor, then this mapping can be applied to any formalized UML diagram.

The UML Mapping is a file generated by an interface that need to be used by the transformation tool of a formalization of UML semantics process. The interface creates a ".mapping" file which respect a Domain-specific Language (DSL) called Mapping DSL. Our MDA then uses another DSL (named as Trace) to parse the Formal Results – together with the Mapping DSL – to represents the information inside the UML Model. In the next subsections we discuss both DSL.

3.1. Mapping DSL

The Mapping DSL grammar defines the elements that need to exist inside the mapping file. Each line of the file contains a tuple with three elements: ID-UML, ID-Formal, and typeElement. The Figure 1 shows the grammar defined for the Mapping DSL. The grammar shows that a Mapping might have N Definition (line 5). Each Definition holds a tuple with the three elements – *one* of each – separate by a comma (line 7). Finally, each element – FormalElement, UMLElement, and TypeElement – has its name (lines 9, 11, and 13).

```
1  grammar br.traceability.mapping.dsl.Mapping with org.eclipse.xtext.common.Terminals
2  generate mapping "http://www.traceability.br/mapping/dsl/Mapping"
3
4  Mapping:
5      definitions+=Definition*;
6  Definition:
7      formalElement=FormalElement ',' umlElement=UmlElement ',' typeElement=TypeElement;
8  FormalElement:
9      name=ID;
10 UmlElement:
11     name=ID;
12 TypeElement:
13     name=ID;
```

Figure 1. Mapping DSL Grammar

By using the Xtext framework[1], our grammar generate a plug-in to Eclipse IDE[2]. Then, the interface builds the mapping file with the tuples every time that a Transformation stage is performed, and the Eclipse checks if the mapping file respect the Mapping grammar.

3.2. Interface

Our MDA depends of an interface that create the mapping file during the Transformation stage. This stage is the only moment in the formalization process where both environments – UML and formal model – exists together. Our interface is a piece of Java code

[1]https://eclipse.org/Xtext/
[2]https://eclipse.org/

which intercepts the UML element being transformed and write on a file its ID-UML, its typeElement (Class, State, Message, Transition, etc.), and the ID-Formal that is given to it. Algorithm 1 shows a pseudo code of our interface.

```
   Input: UMLModel umlModel, File mapping
1  begin
2      foreach StateDiagram std in umlModel.getStateDiagrams() do
3          foreach State state in std.getStates() do
4              FormalState fstate = new FormalState(state);
5              Predicate pred = fstate.getPredicate();
6              mapping.write(pred.getName() + "," + state.getUmlId() + "," +
               state.getType());
7          end
8          foreach Transition trans in std.getTransitions() do
9              FormalTransition ftrans = new FormalTransition(trans);
10             Predicate pred = ftrans.getPredicate();
11             mapping.write(pred.getName() + "," + trans.getUmlId() + "," +
               trans.getType());
12         end
13     end
14 end
```

Algorithm 1: Interface - Getting data for Mapping

The core idea at Algorithm 1 is collecting the required data from State Diagrams for our mapping. During the Transformation stage, our method gather the three required data for each element inside the State Diagram (Lines 2 to 13). At Line 2, the method iterate with each State Diagram present in the UML model being formalized. In Lines 3 to 7, our method manipulate all states inside a State Diagram. At Line 3, our method gather each State in the State Diagram. Then, in Line 4 the method instantiate a $FormalState$ – a semantic version of State – with the given state. $FormalState$ depends on the type of formalization being used. Line 5 instantiate a $Predicate$ which holds the formal information for the state. Finally, the data are written in the mapping file, as can be seen in Line 6, in order to be used later in our MDA. A similar process is done with all transitions inside a State Diagram (Lines 8 to 12). The same logic is applied for Class, Object, Sequence, and Interaction Overview Diagrams.

3.3. Trace DSL

This DSL imports the Mapping DSL to use its tuples grammar inside the Trace grammar. Different from the Mapping DSL, which is not used directly by the user, the Trace DSL is written by the user. The Trace grammar defines what happens to each Formal Element present at the Formal Result. The Figure 2 shows the Trace grammar without the enum types.

The FormalResult might have a name and be a collection of TimeNode or FormalElement (lines 5 to 9). A TimeNode has its own name and a collection of FormalElement (lines 10 to 13). At FormalElement occurs the first use of the Mapping DSL. At line 15, after the use of a keyword, the formalElement variable receives a $map :: FormalElement$. The map makes reference to the Mapping DSL imported at line 3 and $FormalElement$ is the reference to the formal element defined at Mapping DSL. Then at line 16, the grammar shows that a FormalElement has one Element.

```
1  grammar br.traceability.trace.dsl.Trace with org.eclipse.xtext.common.Terminals
2  generate trace "http://www.traceability.br/trace/dsl/Trace"
3  import "http://www.traceability.br/mapping/dsl/Mapping" as map
4
5  FormalResult:
6      'FormalResults' name=STRING '{'
7          ((timenodes+=TimeNode (',' timenodes+=TimeNode)*) |
8          (formalElements+=FormalElement (',' formalElements+=FormalElement)*))
9      '}';
10 TimeNode:
11     'TimeNode' name=STRING '{'
12         formalElements+=FormalElement (',' formalElements+=FormalElement)*
13     '}';
14 FormalElement:
15     'FormalElement' formalElement=[map::FormalElement] '{'
16         element=Element
17     '}';
18 Element:
19     'RefersToElement' typeElement=[map::TypeElement] '{'
20         'WithID' umlElement=[map::UmlElement]
21         'BelongsTo' (diagram=Diagram | model=Model)
22          transformationController=TransformationController
23     '}';
24 Diagram:
25     kind=DiagramKind 'from' model=Model;
26 Model:
27     kind=ModelKind;
28 TransformationController:
29     kind=TransformationKind 'HasTransformationTo' (color=STRING | size=INT);
```

Figure 2. Trace DSL Grammar

The Element has the other two element of the Mapping tuple. First, at line 19 the typeElement variable receives the $TypeElement$ from the map – the import alias for the Mapping DSL. Then, at line 20 the $UmlElement$ from map is also assigned to a variable. At lines 21-22 the Element is associated to a Diagram or a Model and the TransformationController is called. For the UML, all elements must have a Diagram associated to it (lines 24-25). On the other hand, a BPMN (Business Process Model and Notation) element do not need a Diagram, so it's associated with a Model. The Trace DSL has validators to verified if the correct elements are associated with their right Diagram or Model. Finally, then enum types are: (i) DiagramKind – all the diagrams supported by the MDA; (ii) ModelKind – all models supported by the MDA; and (iii) TransformationKind – all transformation model-to-model supported by the MDA.

By using the Trace DSL, the user might defines which formal elements he wants to see inside the UML. One could also defines the type of transformation of each element such as color transformation (background, line, and font).

4. Example

The CCAS example is used to illustrate the use of our MDA. In this case, the UML model has the MADES UML semantic [Baresi et al. 2012] and it was converted to a LISP[3] model – the formal model to the MADES approach. We used the MADES UML because it is the approach that formalize most UML diagrams so far. The CCAS examples is one example from the MADES Project[4]. Given a property that holds (or not) at the formal model, then a formal result is generated by Zot Model Checker [Pradella 2009]. Figure 3 shows a short example of a formal result from Zot.

As can be seen in Figure 3 the formal result is not easy to read and understand. One need to identify the problem in the formal result, find the UML element equivalent to the LISP code, and then re-factor the UML model to try to fix the problem. Our MDA

[3]https://common-lisp.net/
[4]http://www.mades-project.org/

```
------ time 1 ------
  $OBJ_BRAKES_STD_STATEMACHINE1_STATE_IDLE
  $OBJ_BUS_OP_SENDSENSORDISTANCE
  MESSAGE_IJQC4AOAEEKTXBQZTILH3G_END
  MESSAGE_IJQC4AOAEEKTXBQZTILH3G_START
  IOD__PNSFKAN_EEKTXBQZTILH3G_SENDSENSORDISTANCE_START
  $SD_SENDSENSORDISTANCE
  $OBJ_CTRL_STD_STATEMACHINE1_STATE_NOACTION
$SD_SENDSENSORDISTANCE_PARAM_DISTANCE = -4.0
```

Figure 3. Formal Result from Zot

approach aims facilitate the user, by "executing" the formal results inside the UML model. In this context, user might see the step-by-step execution and find more easily and rapidly where to re-factor the model.

Our MDA approach uses both DSL as the Platform Independent Model (PIM) and an ATL (ATL Transformation Language)[5] to perform a Model-to-Model (M2M) transformation. The Figures 4 and 5 show both DSL – Mapping and Trace, respectively – instances for the CCAS example.

```
OBJ_brakeS_STD_StateMachine1_STATE_idle,
_6IhhoAOCEeKTXbQztILh3g,
STATE

OBJ_brakeS_STD_StateMachine1_STATE_braking,
_63an8AOCEeKTXbQztILh3g,
STATE

OBJ_ctrl_STD_StateMachine1_STATE_noAction,
_M8hzMAODEeKTXbQztILh3g,
STATE

OBJ_ctrl_STD_StateMachine1_STATE_braking,
_QrIy8AODEeKTXbQztILh3g,
STATE
```

Figure 4. Mapping DSL generated by interface

```
FormalResults "Test" {
    FormalElement OBJ_brakeS_STD_StateMachine1_STATE_braking {
        RefersToElement STATE {
            WithID _63an8AOCEeKTXbQztILh3g
            BelongsTo StateDiagram from UMLModel
            BackgroundColor HasTransformationTo "Blue"
        }
    },
    FormalElement OBJ_brakeS_STD_StateMachine1_STATE_idle {
        RefersToElement STATE {
            WithID _6IhhoAOCEeKTXbQztILh3g
            BelongsTo StateDiagram from UMLModel
            BackgroundColor HasTransformationTo "Blue"
        }
    }
}
```

Figure 5. Trace DSL written by user

After obtain the PIM instance, a M2M transformation is performed following the ATL code written to parse the PIM through a Platform Specific Model (PSM). The PSM reflects the modeling environment (e.g. UML, BPMN, etc.). So the PIM that compose our MDA is generic enough to represent any type of formal result – given the formalization process uses our interface. The use of a PSM for each modeling environment allows the transformation of a generic formal result to a specific graphic model.

Finally, with the PSM our MDA code is generated as a Eclipse plug-in and the user could use it for execute the formal result and see the data-flow inside the original model – the UML in this example. The Figure 6 shows an Eclipse plug-in running our MDA with the Zot formal results and the MADES UML model.

Figure 6 highlights the following boxes: (1) Formal Results View – that presents to user the formal result and where the user could click at each formal element to see its correspondent UML element; (2) Model View – where the user could see the hierarchy of the UML element; and (3) Editor View – the graphical representation of the model where the user might edit the UML element.

[5]https://eclipse.org/atl/

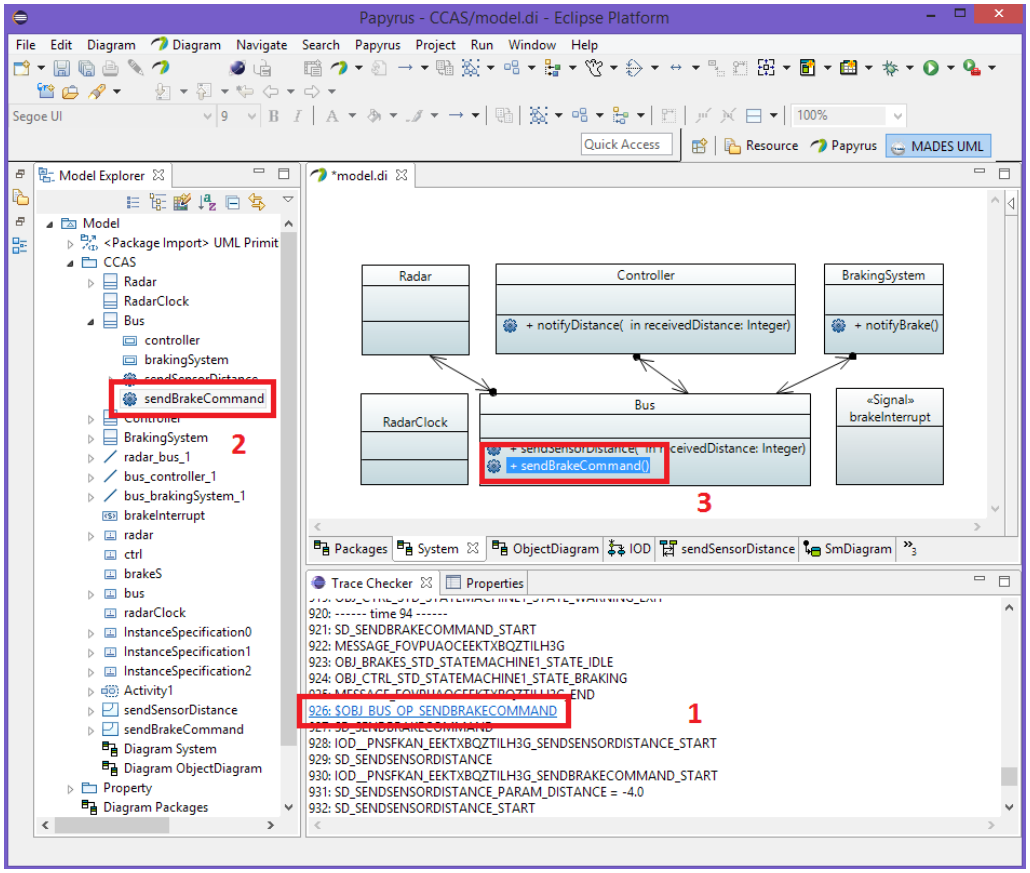


Figure 6. Plugin example

5. Conclusion

This paper presented a generic model-driven approach that is able to represent formal results back – from model checkers – inside the UML model. The MDA manage to do this by using a PIM which composed by two DSL. The DSLs define the structure of our mapping and a language to write which formal element the user might want to see at UML and which transformation it will have. The paper describes how both DSL were defined, the requirements to use it, as well as an example using MADES UML formalization as the test environment. Our MDA approach aims to fill the gap in how to trace formal results back into UML model. We achieve this by guiding the user through the analysis of formal results generate by model checkers. This enhances the users' understanding level about the results and thereby one can find possible defects more easily, fixing them and improving the UML model.

Although we have presented a example using MADES UML, our approach is generic enough to work with different formalization types of UML semantics, due to the way that our Mapping DSL works. In addition, the our approach works with every type of UML diagram since the transformation tool uses our mapping interface and the UML Editor provides access to ID-UML. As a future work, we aim to: ((i)) improve our approach and its plugin; (ii) carry out case studies and experiments; and (iii) analyze the possibility of using the our approach with other modeling languages, such as SysML and BPMN.

Acknowledgements

Vinicius Pereira would like to thanks the financial support provided by CAPES (DS-7902801/D) and CNPq (245715/2012-6)

References

Baresi, L., Morzenti, A., Motta, A., and Rossi, M. (2012). Towards the UML-based formal verification of timed systems. In *FMCO'12*, volume 6957 of *LNCS*, pages 267–286. Springer Berlin/Heidelberg.

Bouabana-Tebiel, T. (2009). Semantics of the interaction overview diagram. In *IRI'09*, pages 278–283, Piscataway, NJ, EUA. IEEE Press.

Broy, M., Crane, M. L., Dingel, J., Hartman, A., Rumpe, B., and Selic, B. (2006). 2nd UML 2 semantics symposium: formal semantics for UML. In *MoDELS'06*, pages 318–323. Springer-Verlag.

Eshuis, R. (2006). Symbolic model checking of UML activity diagrams. *ACM TOSEM*, 15:1–38.

Forster, A., Engels, G., Schattkowsky, T., and Straeten, R. V. D. (2007). Verification of business process quality constraints based on visual process patterns. In *First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 197–208.

Graw, G., Herrmann, P., and Krumm, H. (2000). Verification of UML-based real-time system designs by means of cTLA. In *3rd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, pages 86–95.

Hutchinson, J., Whittle, J., Rouncefield, M., and Kristoffersen, S. (2011). Empirical assessment of MDE in industry. In *ICSE'11*, pages 471–480.

Kaliappan, P. and Konig, H. (2012). On the formalization of UML activities for component-based protocol design specifications. In *SOFSEM'12*, volume 7147 of *LNCS*, pages 479–491. Springer Berlin-Heidelberg.

Konrad, S., Cheng, B. H. C., and Campbell, L. (2004). Object analysis patterns for embedded systems. *IEEE Transactions on Software Engineering*, 30(12):970–992.

Lund, M. S. and Stolen, K. (2006). A fully general operational semantics for UML 2.0 sequence diagrams with potencial and mandatory choice. In *FM'06*, volume 4085 of *LNCS*, pages 380–395.

Micskei, Z. and Waeselynck, H. (2011). The many meanings of UML 2 sequence diagrams: a survey. *Software and Systems Modeling*, 10:489–514.

Pereira, V., Baresi, L., and Delamaro, M. E. (2015). Mapping formal results back to uml semi-formal model. In *Proceedings of the 17th International Conference on Enterprise Information Systems (ICEIS'15)*, volume 2, pages 320 – 329. SCITEPRESS – Science and Technology Publications.

Pradella, M. (2009). An user's guide to zot. Disponível em: *http://home.dei.polimi.it/pradella/Zot/*.