VII CONGRESSO BRASILEIRO DE SOFTWARE • TEORIA E PRÁTICA

CBSOFT

MARINGÁ 2016

VI Workshop de Teses e Dissertações do CBSoft

CBSOFT.ORG

REALIZAÇÃO:

EXECUÇÃO:

ORGANIZAÇÃO:

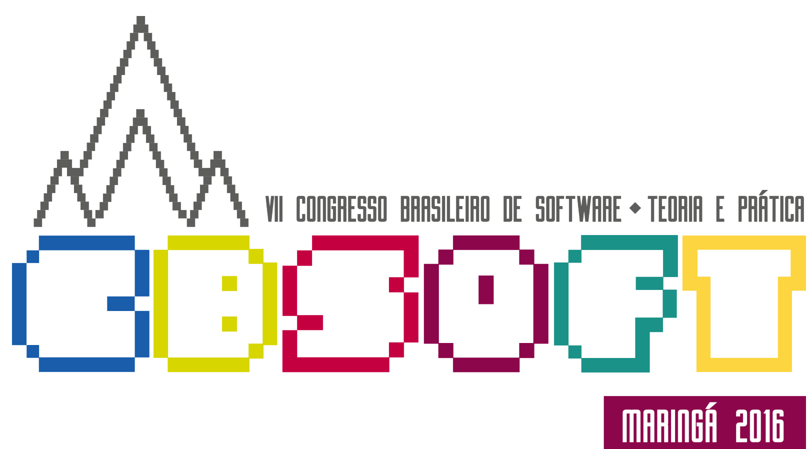
APOIO:

FOMENTO:

PATROCINADORES GIGA:

PATRCCINADORES MEGA:

ThoughtWorks



VI WORKSHOP DE TESES E DISSERTAÇÕES DO CBSOFT (WTDSOFT 2016)

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

ANAIS | *PROCEEDINGS*

Sociedade Brasileira de Computação – SBC

COORDENADOR DO COMITÊ DE PROGRAMA | *PROGRAM COMMITTEE CHAIR*
Uirá Kulesza (UFRN)

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

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

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

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

ISBN: 978-85-7669-339-0

Apresentação

É com grande satisfação que, em nome do Comitê de Programa e da Comissão Organizadora, saudamos os participantes do VI Workshop de Teses e Dissertações do CBSoft (WTDSoft 2016).

O WTDSoft é um evento promovido anualmente pelo CBSoft e este ano conta com uma série de artigos referentes a trabalhos de mestrado e doutorado na área. Foram submetidos 21 artigos ao WTDSoft, dos quais 11 foram selecionados para apresentação durante o evento e publicação em seus anais, 3 são de tese de doutorado e 8 de dissertações de mestrado. O processo de avaliação consistiu na revisão dos artigos por três revisores do comitê de programa, sendo também utilizada uma fase de discussão para minimizar discrepâncias nas avaliações.

Os trabalhos selecionados do workshop abrangem os seguintes temas: Arquitetura de Software, Testes e Otimização de Software, Ecossistemas e Linhas de Produto de Software, Auto-adaptação, Colaboração, e Visualização de Software.

Gostaríamos de agradecer a todos que contribuíram para a realização deste evento. A qualidade deste programa é resultado da dedicação dos membros do Comitê de Programa. Somos também imensamente gratos aos professores convidados, aos participantes, e todos os autores pelos trabalhos submetidos.

Desejamos a todos um excelente evento!

Maringá, setembro de 2016.

Uirá Kulesza (UFRN)
Coordenador do WTDSoft 2016

Foreword

It is with great pleasure that, on behalf of the Program and Organizing Committees, we welcome the participants of the VI Workshop on Theses and Dissertations of CBSoft (WTDSoft 2016).

WTDSoft is an annually promoted event by CBSoft and this year comprises a series of papers related to masters and doctoral work in the area. 21 papers were submitted to WTDSoft, 11 of which were selected for presentation during the event and published in its proceedings, 3 of PhD proposals and 8 of MSc projects. The review process ensured that each submission had three reviews and also used a discussion phase to minimize discrepancies in the ratings.

The selected papers of the workshop address the following topics: Software Architecture, Software Testing and Optimization, Software Ecosystems and Product Lines, Self-Adaptation, Collaboration, and Software Visualization.

We would like to thank everyone who contributed to the realization of this event. The quality of this program is the result of the dedication of the members of the Program Committee. We are also immensely grateful to the invited professors, participants, and all authors of submitted papers.

We wish everyone a great event!

Maringá, September 2016.

Uirá Kulesza (UFRN)
WTDSoft 2016 PC Chair

Comitê técnico | *Technical committee*

Coordenador de comitê de programa | *PC chair*

Uirá Kulesza (UFRN)

Comitê de programa | *Program committee*

Adenilso Simão (ICMC/USP)
Cecilia Rubira (UNICAMP)
Christina Chavez (UFBA)
Claudio Sant'Anna (UFBA)
Eduardo Figueiredo (UFMG)
Elisa Yumi Nakagawa (ICMC/USP)
Fabiano Ferrari (UFSCar)
Fernando Castor (UFPE)
Fernando Pereira (UFMG)
Francisco Carvalho-Junior (UFC)
Franklin Ramalho (UFCG)
Genaina Rodrigues (UnB)
Gledson Elias (UFPB)
Gustavo Soares (UFCG)
Ingrid Nunes (UFRGS)
Leonardo Tizzei (IBM Research)
Marcelo Maia (UFU)
Marco Túlio Valente (UFMG)
Martin Musicante (UFRN)
Patricia Machado (UFCG)
Raul Wazlawick (UFSC)
Roberta Coelho (UFRN)
Rodrigo Bonifacio (UnB)
Rohit Gheyi (UFCG)
Rosângela Penteado (UFSCar)
Silvia Virgilio (UFPR)
Thais Vasconcelos Batista (UFRN)
Tiago Massoni (UFCG)
Toacy Oliveira (COPPE/UFRJ)
Uirá Kulesza (UFRN)
Vander Alves (UnB)

Revisores externos | *External reviewers*

Cristiane Aparecida Lana (USP/SC)
Everton Cavalcante (UFRN)
Fischer Jonatas (PUC-Rio)
João Paulo Diniz (UFMG)
Tiago Volpato (USP/SC)

Artigos técnicos | *Technical papers*

Doutorado | *PhD*

A Reference Architecture for KDM-based Modernization Tools
Bruno M. Santos (UFSCar), Valter V. de Camargo (UFSCar) 1

Um Modelo de Percepção para Indicar o Contexto de Novas Tarefas em Projetos de Software
Edson Mello Lucas (UFRJ), Toacy Cavalcante de Oliveira (UFRJ) 10

Uma Abordagem para Prevenir a Erosão do *Design* baseada em Recomendações Sensíveis à Arquitetura
Marcos Barbosa Dósea (UFBA, UFS), Cláudio Nogueira Sant'Anna (UFBA) 19

Mestrado | *MSc*

Aprendizagem Contínua aplicada ao Problema de Seleção de Otimizações com Estimativa Estática
João Fabrício Filho (UTFPR, UEM), Anderson Faustino da Silva (UEM) 28

Automatização de oráculos de teste para o processamento de imagens médicas de modelos tridimensionais
Misael C. Júnior (USP/SC), Márcio E. Delamaro (USP/SC) 35

Detecção de Variabilidades e Comunalidades em uma Família de Produtos de Software
Denise Alves da Costa (UFPI), Pedro de Alcântara dos Santos Neto (UFPI) 42

Detecting Code Anomalies in Software Product Lines
Eduardo Fernandes (UFMG), Eduardo Figueiredo (UFMG) 49

Fatores Críticos na Manutenção da Arquitetura de TI: Uma Abordagem na Perspectiva de Ecossistemas de Software
Thaiana Maria Pinheiro Lima (UFRJ), Rodrigo Santos (UFRJ), Cláudia Werner (UFRJ) 56

Recomendações de Refatorações Arquiteturais Baseadas em Análise de Impacto no contexto da ADM
André de Souza Landi (UFSCar), Valter Vieira de Camargo (UFSCar) 63

Seamless and Adaptive Application-level Caching
Jhonny Mertz (UFRGS), Ingrid Nunes (UFRGS) 70

Um *framework* para visualização da evolução arquitetural de sistemas de *software* baseada em cenários de casos de uso
Leo Silva (UFRN, IFRN), Uirá Kulesza (UFRN) 77

A Reference Architecture for KDM-based Modernization Tools

Bruno M. Santos[1], Valter V. de Camargo[1]

[1]Departamento de Computação – Universidade Federal do São Carlos (UFSCar)
Caixa Postal 676 – 13565-905 – São Carlos – SP – Brazil

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

Abstract. *Software systems are in constant changes and this process usually makes the system more and more difficult to maintain. To help dealing with this context, OMG proposed the ADM blueprint, which advocates the use of software reengineering and MDA along the modernization of legacy systems. One of the contributions of ADM is a set of standard metamodels and the main metamodel is KDM, which supports the representation of several software artifacts. The main advantage of using KDM in tools is to develop algorithms that could be reused by other tools. The problem is that there is no reference in the literature that helps in the construction of KDM-based modernization tools. Thus, the main objective of this proposal is to develop a reference architecture to support the creation of modernization tools. To complete this task we intend to build a KDM-based modernization tool that applies refactorings in UML diagrams and reflect the modifications in the correspondent KDM There are several approaches in the literature that applies refactorings in UML models and in this phD proposal we intend to evaluate the costs of reusing them in the context of KDM-based modernization tools, that are tools built conform the ADM blueprint. This phD proposal will be evaluated in two steps, the first one will evaluate the algorithm of KDM-refactorings creation from the UML ones and the mechanism that will apply these refactorings in the models, to grant that they will work properly. The second evaluation will be the execution of a study case using an existing software system used at UFSCar.*

Nível: Doutorado
Ano de Ingresso: 03/2015
Época prevista de conclusão: 03/2019
Data de aprovação da proposta de dissertação: Proposta ainda não aprovada. Previsão para março de 2017.
Evento Relacionado: SBES e SBCARS

Keywords. *Architecture-Driven Modernization, Knowledge Discovery Metamodel, Unified Modelling Language, Modernization Tools, Refactorings, Reference Architecture, Reuse.*

1. Introduction and Problem Characterization

In 2003, the Object Management Group (OMG) created a task force called Architecture-Driven Modernization (ADM) with the goal of presenting a solution for the huge number of failed reengineering projects [Sneed, 2005]. According to OMG, the main problem was the lack of standard metamodels for representing all the details of complete systems, not only source code. This absence hinders the reusability of solutions among modernization tools, leading vendors to concentrate their effort on developing proprietary metamodels and solutions (metamodels, refactoring algorithms, transformation rules) that only work over their proprietary metamodels. Although UML is also a standard, it is not suitable for some kinds of modernization, for example, those that impact higher level elements [OMG, 2016].

As a response, ADM task force has proposed an ISO metamodel called Knowledge Discovery Metamodel (KDM) [ISO/IEC, 2012]. KDM was created with the intention of being able to represent all characteristics of a software system, ranging from low level details (source code, actions, etc), as is partially covered by UML, to higher level concepts not presented in programming languages (architectural modules, conceptual elements, business processes, infrastructure, and others).

The idea is that this metamodel becomes the standard way of representing software systems in the reengineering/modernization tools. In other words, if researchers and practitioners develop modernization solutions (mining and analysis algorithms, refactorings, transformations, analytics tools, metrics, etc) for KDM-represented systems, instead of proprietary metamodels or specific languages, these solutions could be more easily reused among all tools that recognize KDM as the main system representation.

Although OMG/ADM advocates the adoption of KDM as the base metamodel in modernization tools, it is not clear in the literature neither how to use this metamodel in tools nor what is the real advantages of it. This becomes even more obscure in modernization tools that the user apply refactorings on UML diagrams. This is a common situation, since KDM was not created for serving as base of diagrams. Therefore, up to this moment, there is no knowledge in the literature about how modernization tools should be designed to deal with UML and KDM simultaneously, neither how to benefit from the huge amount of UML resources available in the literature.

It is possible to find a number of solutions and approaches in the literature that apply refactoring on UML models [Misbhauddin and Alshayeb, 2015]. Most of these refactorings are transformation rules implemented in some transformation language, such as ATL. As it is possible to elaborate a mapping between KDM and UML, it seems also possible to automatically generate KDM refactorings from UML refactorings. In other words, we intend to reuse existing UML-refactorings in KDM models, for this, we are going to investigate how UML-refactorings could be applied in KDM because we know based on previous papers [Santos et al., 2014a and 2014b] that all the UML metaclasses could be represented by KDM metaclasses.

Therefore, in this thesis we intend to develop a reference architecture (RA) for modernization tools that allows the application of refactorings over UML diagrams. The

RA will consists of a set of guidelines, diagrams and schematic representations that shows the main modules of the tool, their relationships and internal mechanisms. It will be useful to support the development of new modernization tools or even to modernize existing tools that work exclusively with UML. An important characteristic of our tool is that it will allow the generation of KDM refactorings from UML refactorings.

Considering this context, this phD project aims at investigating: (i) the most suitable architecture for KDM-based modernizations tools that need to work with KDM and UML, keeping them synchronized and (ii) how to automatically generate KDM refactorings from existing UML refactorings.

The main contribution of this phD proposal will be a reference architecture to help modernization engineers while implementing modernization tools that reuses UML-refactorings in KDM models. We claim that a modernization tool created following this architecture reference would be more reusable and the chances of success would be raised. However, it is not clear in the literature how modernizations tool that uses KDM as a base-matamodel should be designed. Therefore, we claim that a reference architecture focused on model KDM-refactorings would help by improving the state of the art of model refactorings, thus, our main research question is: How a reference architecture could help in the creation of KDM-refactorings modernization tools?

2. Theorical Foundation

2.1. Architecture-Driven Modernization and Knowledge Discovery Metamodel

Architecture-Driven Modernization is a new trend of reengineering processes that considers standard models and MDA concepts along the process. What have influenced the creation of ADM the most is the high number of failed reengineering projects [Sneed, 2005]. According to OMG, the main reason of this problem is the lack of standardization, hindering the productivity of teams, preventing the reuse of algorithms and techniques and compromising the interoperability among modernization tools from different vendors.

The modernization process supported by ADM involves three phases and it is similar to a horseshoe: Reverse engineering, refactoring and forward engineering. In reverse engineering, the knowledge is extracted from legacy systems and it can be represented in different abstraction levels. During reverse engineering, refactorings are performed aiming to get a high-level representation of the system or to obtain an improved version of the system, independently of the adopted platform. In restructuring phase, it is possible to conduct refactoring, optimization and also insert new business rules in the system. The output is a new target model without the problems previously identified, which can be called "a modernized model". In forward engineering phase, wherein the models are resubmitted to a set of transformations to reach the source-code level again [Pérez-Castillo, Guzmán and Piattini, 2011a].

The main ADM metamodel is called KDM. KDM is a metamodel of common intermediate representation to existents systems and its operating environments. Using this representation it is possible exchange systems representation between platforms and languages aiming to analyze, to standardize and to transform existing systems [OMG,

2016]. The KDM metamodel represents physical and logical legacy software artifacts in different abstractions levels and it is formed by twelve packages organized in four layers: infrastructure, program elements, runtime resources and abstractions.

In the program elements layer there are the *Code* and *Action* packages that represent the source-code elements and their behavior. The *Code* package defines metaclasses that represent implementation level units and their associations. This package also includes metaclasses that represents common program elements supported by several programming languages, such as: classes, procedures, datatypes and templates. The Action package represents behavioral implementation level descriptions, such as: Declarations, operators and constraints. *Action* package completes the meaning of *Code* package and together they provide the implementation level abstraction in KDM models [Pérez-Castillo, Guzmán and Piattini, 2011b].

In this paper the KDM metamodel is used as a base metamodel to the entire approach, thus, the objectives successes which are the automatic generation of KDM refactorings and the reference architecture for modernization tools will be directly related to this metamodel.

2.2. Reference Architecture

Reference architecture is a structure that provides the characterization of software system functionality from a specific application domain [Eickelmann and Richardson, 1996]. Thus, reference architecture serves as a guideline to increase the chances of successfully develop a software system and it can be considered as a first essential step to the development of application frameworks.

Accordingly to Eickelmann and Richardson (1996), the proposition of reference architectures to software systems of a specific application domain is not a trivial task, this task involves depth knowledge about the domain that the reference architecture is been created. Besides, reference architecture provides the reuse of the architectural design and the understanding of a specific area [Nakagawa, 2006].

Designing reference architectures can be a challenging activity because there is a lack of frameworks that help in the creation process of tools that follows a particular reference architecture. Another problem is that there is a difficulty in generalize a reference architecture from a single concrete example, in other words, it would be necessary several architectural implementation of the same specific domain tool to provide the best reference architecture. In addition, there are no specific technics or methods to help in the specification of a reference architecture [Nakagawa, 2006].

There are several reference architectures available in the literature to the most different application domains, such as: Barber et al. (2002); Sandkuhl and Messer (2000). Another example is a reference architecture to e-commerce systems proposed by Bass et al. (2003). In software engineering area a reference architecture to test tools were proposed by Eickelmann and Richardson (1996). However, there is no reference architecture to modernization tools that uses standard metamodel [Nakagawa, 2006].

3. Contributions

We claim that this proposal has a technical and a conceptual contribution. The technical contribution is the development of a computational support to modernization software

system represented in UML diagrams. This computation support will have a refactoring generation module responsible for converting UML-refactoring in to KDM-refactorings. The conceptual contribution can be divided in two others. The first one is the development of a reference architecture to KDM-based modernization tools. The second conceptual contribution is the construction of a UML-KDM mapping. This mapping is a substantially important artifact because it opens a wide way of reusing UML proposals, such as the conversion of UML profiles in to KDM profiles.

4. Detailing the Proposal

The phD proposal is concentrated on the development of a reference architecture for ADM-based modernization tools. This reference architecture should serve as a guide for developing new tools or even for modernizing existing UML-based ones. To be more precise, we intend to focus on modernization tools in which the software engineer interact with UML class diagrams. Thus, we envisage a usage scenario where a software engineering applies refactorings on class diagrams, converting them to new, modernized class diagrams without changing the abstraction level. This scenario is the user view and accordingly to a systematic review from 2015 there are about 39 tools like that in the literature [Misbhauddin and Alshayeb, 2015].

The difference of our approach, when compared to existing tools is internal and not in the application level. This occurs because we intend to investigate how this type of tool should be internally designed, considering the existence of the KDM metamodel and the synergy that must exist with the UML models. Our research will be driven by these two questions: i) How should be designed the interaction between KDM and UML in ADM-based modernization tools ? and ii) How is the best design for behavior-preservation in this scenario? To answer the two points and acquire experience for elaborating the reference architecture, we intend to develop a tool that present the characteristics mentioned above. Most existing UML-based tools that allow the application of refactorings, perform the transformations just over the UML model [Arendt and Taentzer, 2012; Baar and Marković, 2007; Ruhroth et al., 2009; Stolc and Polasek 2010]. In our case, there is a second model (KDM) that needs to be synchronized with the UML model, which brings a significant challenge for the process.

One of the biggest challenge of this research, which impacts the two research questions mentioned above, is to create a suitable mapping between UML and KDM metamodels. This mapping is required to make evident which metaclasses in KDM are semantically related to the existings ones of UML. Therefore, they have some differences but also some similarities. For example, *Code* and *Action* packages of KDM are quite similar to UML, as they intend to represent code-level elements. On the other hand, KDM is much broader, covering other abstraction levels that is not well addressed by UML, such as Architecture, Conceptual, Data, Interfaces, etc.

In a previous work, we have advanced in the creation of this UML-KDM mapping [Santos et al., 2014a and 2014b], but just on the *Code* KDM Package. The existence of this mapping can support the development of several automatic solutions from UML to KDM. For example, one of the important focus of this work, which is impossible to conduct without such a mapping, is the development of a module for automatic transformation of UML-based refactorings into KDM-based ones.

In order to develop the reference architecture we will develop an ADM-based modernization tool that uses the UML-KDM mapping. This is necessary because this tool uses these two metamodels internally and they need to keep synchronized. Therefore, when a software engineer applies a specific refactoring in a class diagram and changes the UML model, the same modification must be propagated to the underlying KDM model that represent the system. Thus, it will be necessary a mechanism to manage that synchronization. Notice that in this proposal we do not present our reference architecture, it will be developed during the phD, after qualification exam, thus, we just present an architecture prototype of a KDM-based modernization tool.

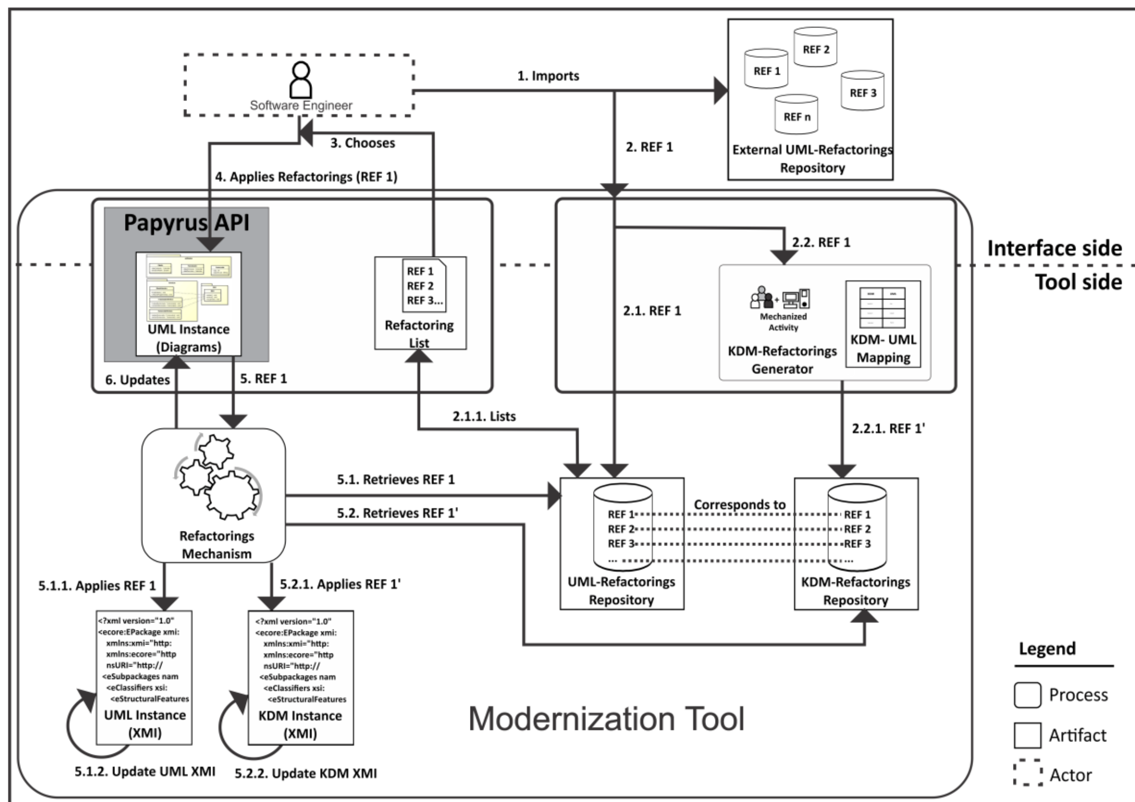


Figure 1 –KDM-based modernization tool

Another challenge here is to preserve the behavior of the system after a refactoring application. As mentioned, there are some works in the literature that preserve the behavior of the system after applying UML refactorings [Baar and Marković, 2007; Ruhroth et al., 2009; Stolc and Polasek 2010]. This is usually done changing the "actions" related to the modified UML element. Therefore, the mechanism that will be developed also needs to propagate the changes to the KDM model and also to its other packages related to actions.

Figure 1 shows a schematic representation of the ADM-modernization tool we intend to develop and how it should work. The software engineer can interact with the modernization tool in two moments. The first one is when he wants to take advantage of existing UML refactorings available in external repositories. In this case, the software engineer imports a specific refactoring (REF 1, for example) into the modernization tool. The refactoring is stored in the UML-Refactoring Repository and becomes

available to be used for users, i.e., it is ready to be applied over UML diagrams. Usually, refactorings like that, are a set of transformations rules in a specific transformation language such as ATL and QVT.

The UML refactoring imported also serve as an input to the KDM-Refactoring Generator. This generator uses the UML-KDM Mapping to generate a KDM-refactoring that is semantically equivalent to the UML refactoring provided. The refactorings repositories, UML and KDM, have a set of transformations rules. Therefore, in the UML-Refactorings each UML-refactoring has a corresponding KDM-refactoring version in the KDM-Refactorings Repository, such as: REF 1 corresponds to REF 1', REF 2 corresponds to REF 2', REF 3 corresponds to REF 3', and so on.

The second moment that the software engineer interacts with the modernization tool is when he wants to apply a refactoring over a class diagram. In this moment, a Refactoring Mechanism is activated to process this request. The Refactorings Mechanism is an algorithm that based on the chosen refactorings it retrieves the corresponding transformation rules in UML and KDM-Refactorings Repositories. Once the transformation rules are retrieved (REF 1 and REF 1') the algorithm executes them in their corresponding model instances and updates the XMI files. As a final step, the Refactorings Mechanism updates the UML Instance (Diagrams) view from the new refactored UML Instance (XMI). It is important remembering that the first UML and KDM instance will be generated by MoDisco API [Bruneliere et al., 2010] thus this API will be reused in our modernization tool.

5. Related Works

Based on a Systematic Mapping performed by Durelli et al. (2014a) we do not found any primary study that is directly related to our apporach. We are already updating this systematic mapping and actually we are on the full paper read selection phase but we also did not find any directly related paper. Nevertheless, Durelli et al. (2014b) proposed an approach that applies the Fowler's refactorings catalog in KDM metamodel. The approach enables a software engineer in the process of applying small refactorings in KDM instances. The main difference to our approach is that we are focusing on the creation of a reference architecture to KDM-based modernization tools and we are proposing the reuse of UML-based refactoring available in the literature [Misbhauddin and Alshayeb, 2015]. We claim that this is the main advantage of our approach since there are several papers that proposes this kind of research.

We intend to reuse the guidelines proposed in Santos et al. (2014a and 2014b) to build the UML-KDM mapping. Another set of related work is presented in Misbhauddin and Alshayeb (2015) because in their systematic literature review they present a set of 39 paper that propose model refactorings in UML diagrams by means of tool support. For example, Baar and Marković (2007) propose refactorings to UML class diagrams using OCL constraints. Another example can be seen in Arendt and Taentzer (2012), they propose a way to erase model smells by automatically suggesting appropriate model refactorings, and to get warnings in cases where new model smells come in by applying a certain refactoring in the UML class models.

6. Results Evaluation

The research questions that will guide the development of the approach are: (i) Does the algorithm generate the correct KDM refactorings from UML refactorings? ; (ii) Does the reference architecture improve the productivity of software engineer when creating modernization tools? To answer these questions we envision some evaluation approaches, described in the following paragraphs.

The evaluation process of a reference architecture covers the creation of several tools based on this RA, thus, if these tools were really effective we could claim that this RA is also effective. We are still analyzing alternative ways of evaluating our RA, but the first step is by creating the tool proposed in section 4. We will also investigate if a framework for creating this kind of tool based in our RA could be created, this would facilitate and speed up the tool creation process.

To evaluate the proposed tool, we anticipate two main evaluation scenarios; the first set the validation of the algorithm that generates the KDM-Refactorings transformations rule and the second on is the Refactorings Mechanism. We have to ensure that these two algorithms produce the same results as the UML ones because the model behavior should remain the same. The second evaluation scenario is the realization of a case study with a real software system used at Universidade Federal de São Carlos (UFSCar) that is called SIGA. This evaluation will help us to identify if the tool is producing the expected result and evaluate if this tool can be used in real software systems.

References

Arendt T, Taentzer G (2012). Integration of smells and refactorings within the Eclipse modeling framework. In: Proceedings of the Fifth Workshop on Refactoring Tools, Rapperswil, Switzerland.

Baar T, Marković S (2007). A graphical approach to prove the semantic preservation of UML/OCL refactoring rules. In: Virbitskaite I, Voronkov A (eds) Perspectives of systems informatics, vol 4378. Lecture Notes in Computer Science. Springer, Berlin, pp 70–83. doi:10.1007/978-3-540-70881-0_9.

Barber, K. S.; Holt, J.; Baker, G. (2002). Performance evaluation of domain reference architectures. In: Proc. if the 14th International Conference on Software Engineering and Knowledge Engineering, ACM Press, p. 225-232.

Bass, L.; Clements, P.; Kazman, R. (2003). Software architecture in practice. The SEI Series in Software Engineering, 2 ed. Addison-Wesley Publishing Company.

Bruneliere, H. et al. (2010).MoDisco: A generic and extensible framework for model driven reverse engineering. In: IEEE/ACM international conference on Automated software engineering, ACM New York, NY, USA. p. 173-174.

Durelli, R. S. et al. (2014a).A Mapping Study on Architecture-Driven Modernization. In: 15th IEEE International Conference on Information Reuse and Integration, 2014a, San Francisco, CA, USA. p 1-8.

Durelli, R. S. et al. (2014b). Towards a Refactoring Catalogue for Knowledge Discovery Metamodel. In: 15th IEEE International Conference on Information Reuse and Integration, San Francisco, CA, USA. p 1-8.

Eickelmann, N. S.; Richardson, D. J. (1996). An evaluation of software test environment architectures. In: Proc. Of the 18th Internationl Conference on Software Engineering, Berlin, Germany: IEEE Computer Society Press, p. 353-394.

ISO/IEC. ISO/IEC DIS 19506. (2012). Information technology - Architecture-Driven Modernization (ADM): Knowledge Discovery Meta-Model (KDM) http://www.iso.org/iso/catalogue_detail.1128htm?csnumber=32625, ISO/IEC. p. 331. 2012.

Misbhauddin, M ; Alshayeb, M. (2015). UML model refactoring: a systematic literature review. Journal of Empirical Software Engineering. p. 206–251.

Nakagawa, E. Y. (2006). Uma Contribuição ao Projeto Arquitetural de Ambientes de Engenharia de Software, p. 234. Thesis (PhD in Computer Science and Computational Mathmatics) – Instituto de Ciências Matemáticas e de Computaç]ao – ICMC/USP, São Carlos, São Paulo, Brasil, unpublished.

Object Management Group. (2016) OMG Specifications, June 2016. Documents omg/ http://www.omg.org/spec/.

Pérez-Castillo, R., Guzmán, I. G. E Piattini, M. (2011a). Architecture-Driven Modernization. Hershey, New York, v1.

Pérez-Castillo, R., Guzmán, I. G. E Piattini, M. (2011b). Knowledge Discovery Metamodel-ISO/IEC 19506: A standard to modernize legacy systems. Journal of Computer Standards & Interfaces, DOI: 10.1016/j.csi.2011.02.007.

Ruhroth T, Voigt H, Wehrheim H (2009). Measure, diagnose, refactor: a formal quality cycle for software models. Proceedings of the 35th Euromicro Conference on Software Engineering and Advanced Applications:360–367. doi:10.1109/seaa.2009.39

Sandkuhl, K.; Messer, B. (2000). Towards reference architectures for distributed groupware applications. In: 8th Euromicro Workshop on Parallel and Distributed Processing, Rhodes, Greece.

Santos, B. M. et al. (2014b). Investigating Lightweight and Heavyweight KDM Extensions for Aspect-Oriented Modernization. In: 11th Workshop on Software Modularity (WMod), Maceió, Brazil. p. 1-12.

Santos, B. M. et al. (2014a). KDM-AO: An Aspect-Oriented Extension of the Knowledge Discovery Metamodel. In: 28º Simpósio Brasileiro de Engenharia de Software (SBES), Maceió, Brazil: IEEE. p. 1-10.

Sneed, H. M. (2005). Estimating the costs of a reengineering project. In: Working Conference on Reverse Engineering, WCRE, p. 111–119.

Stolc M, Polasek I (2010). A visual based framework for the model refactoring techniques. In: IEEE 8th International Symposium on Applied Machine Intelligence and Informatics.

Um Modelo de Percepção para Indicar o Contexto de Novas Tarefas em Projetos de Software

Edson Mello Lucas e Toacy Cavalcante de Oliveira (Orientador)

PESC/COPPE - Programa de Engenharia de Sistemas e Computação
Cidade Universitária, Centro de Tecnologia, bloco H, sala 319
Rio de Janeiro - RJ - Brasil - Caixa postal: 68511 - CEP: 21941-972

{edmlucas, toacy}@cos.ufrj.br

Nível: Doutorado
Ingresso no Programa: Jun/2013
Previsão de conclusão: Dez/2017
Qualificação: 22/06/2016
Evento relacionado: SBES

Resumo. *Software é produzido por pessoas realizando tarefas que criam ou reutilizam artefatos, por isso cresce em tamanho e complexidade ao longo do tempo. Neste cenário, encontrar os artefatos para realizar as tarefas é um trabalho difícil e ao mesmo prejudica a produtividade dos desenvolvedores. Esta proposta apresenta um modelo de tarefas para indicar o contexto inicial de uma nova tarefa em projetos de software. O contexto de uma nova tarefa é definido como as suas tarefas similares, mais os seus artefatos associados com os seus respectivos especialistas. Esta proposta faz uso de algoritmos de linguagem natural e dos conceitos de produtor-consumidor. Para a identificação dos especialistas, também é proposto um modelo de percepção do conhecimento baseado nas intensidades de interação entre os desenvolvedores e destes sobre os artefatos. Os resultados preliminares foram satisfatórios para a determinação de especialistas. A expectativa é que este trabalho consiga evoluir no estado da arte na inferência de contexto inicial de tarefas em projetos de software.*

Palavras-chave: *percepção, contexto, especialista, conhecimento, colaboração*

1. Caracterização do Problema

O desenvolvimento de software é um processo centrado em pessoas que na maioria das vezes é conduzido de forma colaborativa, por isso é afetado pelas limitações humanas [Fuggetta and Di Nitto 2014; de Souza and Redmiles 2011; Mistrík et al. 2010]. Além disso, não é possível ter um plano rigidamente definido que possa ser seguido com sucesso, o que caracteriza o desenvolvimento de software como um processo intensivo em conhecimento [Qumer and Henderson-Sellers 2008; Di Ciccio et al. 2015]. Por isso é difícil prever todas as entradas e saídas de uma tarefa *a priori,* ficando a cargo do desenvolvedor a procura por artefatos para a realização das tarefas. Há evidências que o tempo gasto com a navegação é similar ao de edição sobre os artefatos, diminuindo assim a produtividade (Di Ciccio et al. 2015). Para contornar este problema, uma prática recorrente é a transferência de conhecimento entre os desenvolvedores com o objetivo de realizar tarefas [Whitehead et al. 2010; Bani-Salameh et al. 2010].

O desenvolvimento de software está inserido em um ambiente em que as interações entre os desenvolvedores e dos desenvolvedores sobre os artefatos digitais propiciam a transferência direta e indireta de conhecimento entre os envolvidos, principalmente quando organizados por processos ágeis [Fritz et al. 2010; Koskinen et al. 2003]. Consideramos que a transferência de conhecimento acontece quando as pessoas interagem diretamente ou através da observação dos artefatos produzidos com o objetivo de realizar tarefas. Nesta direção, é possível saber como uma tarefa foi realizada analisando os artefatos associados à tarefa e/ou interagindo com quem participou da sua realização. Por isso, ter a percepção de quem sabe mais sobre o quê ao longo do tempo é importante para apoiar o processo de transferência de conhecimento entre os desenvolvedores e, também para determinar o contexto inicial para uma nova tarefa.

A determinação do contexto para uma nova tarefa de desenvolvimento de software ainda é um desafio porque as avaliações das propostas apresentam uma precisão baixa [Cubranic et al. 2005; Thompson and Murphy 2014; Leano et al. 2014]. A Figura 1 ilustra um cenário de contexto. No passado, as tarefas realizadas possuem contextos reais porque é possível identificar os seus autores, os desenvolvedores que cooperaram e os artefatos associados às tarefas por interações de edição e/ou criação. No entanto, é difícil determinar o contexto real no início de uma nova tarefa, então o contexto inicial é composto por todos os desenvolvedores, artefatos e tarefas no projeto, como representado na Figura 1, no tempo zero. Em outras palavras, é difícil recomendar no tempo zero: *realize a tarefa usando aqueles artefatos e criando estes, e se necessário, interaja com estes desenvolvedores*. Além disso, desde o início da tarefa, o contexto sofre mudanças ao longo tempo e somente está fixado após o término da tarefa [Kersten and Murphy 2005].

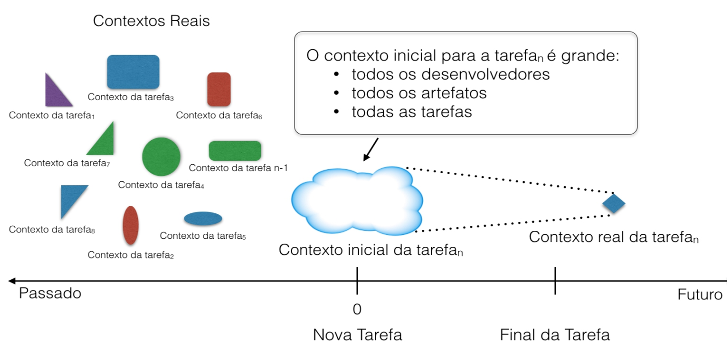


Figura 1: Contexto de tarefas

2. Fundamentação Teórica

Dourish e Bellotti (1992) definem percepção como *o entendimento das atividades dos outros que proporcionam um contexto para suas próprias atividades.* Onde contexto parte das necessidades individuais de ter conhecimento sobre o ambiente para melhor ajudar nos objetivos do grupo. E mais alinhado a um escopo, STOREY *et al.* (2005) observa que percepção consiste em saber quem está trabalhando no projeto, o que estão fazendo, que artefatos estão ou estavam manipulando, e como o trabalho individual pode afetar o trabalho dos outros. Isto indica que o contexto muda ao longo do tempo em virtude da inserção de novas tarefas e/ou pessoas.

De Souza e Redmiles (2011) definem rede de percepção como *a rede de atores cujas ações precisam ser monitoradas por um ator e aqueles a quem esse ator precisa fazer suas ações visíveis*. Esta definição coloca o ator com o papel central na origem e destino das informações na realização das tarefas individuais. Além de evidenciar que algumas informações precisam ser compartilhadas porque afetam a realização de outras tarefas. Por isso, a rede de percepção minimiza os conflitos de código.

Conhecimento pode ser descrito como a percepção, entendimento, ou informação que foi adquirida por experiência ou estudo[1]. Ele é um bem invisível, intangível e não pode ser diretamente observado [Hunt 2003]. O conhecimento tácito representa o conhecimento adquirido através da experiência, isto é, pontos de vista, compromissos, atitudes etc. Já o conhecimento explícito é o adquirido nas escolas e universidades e está registrado nos livros e manuais. Os dois tipos de conhecimento, tácito é explícito, estão presentes na realização de projetos, variando de proporção de projeto para projeto.

Processo de software pode ser definido como um conjunto coerente de políticas, estruturas organizacionais, tecnologias, procedimentos e artefatos que são necessários para conceber, desenvolver, implementar e manter um produto de software [Fuggetta 2000]. E mais especificamente, segundo [Osterweil 1987], processo é uma abordagem sistemática para a criação de um produto ou a realização de alguma tarefa, para realizar o trabalho ou alcançar um objetivo em uma forma ordenada. E de forma mais alinhada a esta proposta, processo é um conjunto de passos parcialmente ordenados destinado a atingir um objetivo [Feiler and Humphrey 1993].

O processo de desenvolvimento de software é intensivo em conhecimento porque é essencialmente uma atividade humana com suporte de ferramentas [Di Ciccio et al. 2015; Omoronyia 2008]. Assim, é possível capturar interações em nível básico com o objetivo de extrair conhecimento sobre os artefatos e desenvolvedores. Está é uma nova tendência das novas ferramentas de suporte para o desenvolvimento de software onde o monitoramento das interações é a fonte das descobertas [Kersten 2007; Robbes and Lanza 2008; Hattori and Lanza 2010].

3. Contribuições

A presente proposta tem como contribuições esperadas:

- O modelo de percepção do conhecimento centrado em tarefas baseado na intensidade de interação dos desenvolvedores. Modelo este que pode utilizar informações de ambientes de desenvolvimentos monitorados em tempo real ou as informações de repositórios de código (Git, SVN);
- Uma proposta para montar o contexto de tarefa indicando 1- os artefatos que serão utilizados na realização de uma nova tarefa de desenvolvimento de software; 2- as tarefas similares à nova tarefa; 3 – e os seus respectivos especialistas.

1 http://dictionary.cambridge.org/us/dictionary/english/knowledge

4. Estado Atual do Trabalho

4.1. Modelo de Percepção de Tarefas

O modelo de percepção de tarefas está organizado em quatro macro-atividades. A primeira macro-atividade é preparatória, "*Agrupar as tarefas do projeto por similaridade*", número 1 na Figura 2, ela organiza em grupos as tarefas do projeto por similaridade textual e, quando disponível, utiliza o histórico sobre a percepção dos especialistas. A similaridade textual faz uso das técnicas de processamento de linguagem natural [Cambria and White 2014]. A similaridade perceptiva é o histórico da aprovação da existência de similaridade entre duas tarefas declaradas pelos desenvolvedores que possui elevado grau de conhecimento, resultado da terceira macro-atividade. A similaridade perceptiva só está disponível quando a primeira macro-atividade é refeita com o objetivo de melhorar a eficácia do modelo. O resultado da primeira macro-atividade são grupos de tarefas similares.

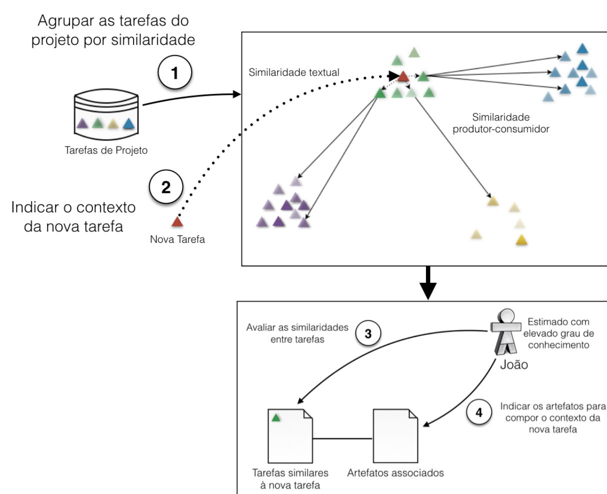


Figura 2: Modelo para indicar o contexto inicial de uma nova tarefa

A segunda macro-atividade, "I*ndicar o contexto da nova tarefa*", Figura 2-número 2, associa a nova tarefa ao grupo que mais se assemelha de acordo com a similaridade textual. Em seguida, lista as tarefas similares (textual) à nova tarefa dentro do grupo, mais as tarefas similares a estas dentro do projeto (relações de produtor-consumidor), juntamente com os seus artefatos. A similaridade por relações de produtor-consumidor é baseada na teoria da atividade [Barthelmess and Anderson 2002] Na terceira macro-atividade, "*Avaliar as similaridades entre tarefas*", Figura 2-número 3, os especialistas são convidados a declarar quais das tarefas descobertas pela segunda macro-atividade são similares à nova tarefa. E na quarta e última macro-atividade, "*Indicar os artefatos para compor o contexto da nova tarefa*", os mesmos especialistas podem também recomendar quais artefatos conectados às tarefas similares devem constar no contexto inicial da nova tarefa, Figura 2-número 4.

A segunda, terceira e quarta macro-atividades do Modelo são repetidas para cada nova tarefa do projeto. O histórico sobre a avaliação dos desenvolvedores sobre as similaridades entre tarefas é registrado para ajudar na organização das tarefas em grupos realizada pela primeira macro-atividade. Refazer a primeira macro-atividade com este histórico implica na construção de grupos de tarefas similares levando em consideração a percepção dos especialistas. Espera-se que a cada reconstrução dos grupos de tarefas, o Modelo possa indicar o contexto inicial para uma nova tarefa com uma melhor eficácia.

4.2. Modelo de Percepção do Conhecimento

No cenário de um projeto de desenvolvimento de software colaborativo, definimos o contexto $C(D_E, TS_S, A_R, V_V, P_C)$ como a união dos conjuntos de desenvolvedores (D_E), tarefas (TS_S), artefatos (A_R), Atividades (V_V) e Processos (P_C):

$C(D_E, TS_S, A_R, V_V, P_C) = D_E \cup TS_S \cup A_R \cup V_V \cup P_C$, onde para todo $a_r \in A_R$ existe pelo menos uma $ts_s \in TS_S$ tal que a_r contribui para a realização de ts_s; E, onde para todo $ts_s \in TS_S$ existe um $v_v \in V_V$ tal que ts_s se relaciona com v_v, e P_C é um conjunto de processos onde para todo $v_v \in V_V$ existe pelo menos um $p_c \in P_C$ tal que v_v está presente em p_c. Para o contexto $C(D_E, TS_S, A_R, V_V, P_C)$, inspirados em [Kersten 2007; Omoronyia 2008; Fritz et al. 2010], definimos as medidas:

$I_{da}(d_e, a_r, \Delta t)$ = medida da intensidade de interação de d_e sobre a_r durante um intervalo de tempo Δt , para $d_e \in D_E$ e $a_r \in A_R$;

$I_{dd}(d_e, d_f, ts_s, \Delta t)$ = medida da intensidade de interação entre dois desenvolvedores durante um intervalo de tempo Δt quando interagem por causa de ts_s, para $ts_s \in TS_S$.

Então, definimos 4 equações para medir o conhecimento explícito dos desenvolvedores registrados nos artefatos de um projeto de software colaborativo. A Equação 1, $K_a(d_e, a_r, t)$, expressa o conhecimento explícito do desenvolvedor sobre um artefato a partir da intensidade de interação do desenvolvedor sobre o artefato. A Equação 2, $K_{ts}(d_e, ts_s, t)$, calcula o conhecimento de um desenvolvedor sobre uma tarefa somando todo conhecimento do desenvolvedor sobre os artefatos associados à tarefa mais o somatório das intensidades de interação dos desenvolvedores envolvidos na sua realização. A Equações 3 e 4 são apresentadas em seguida com mais informações.

A Equação 3, $K_v(d_e, v_v, t)$, usa a Equação 2, $K_{ts}(d_e, ts_s, t)$, para calcular o conhecimento do desenvolvedor sobre um conjunto de tarefas ts_s que estão relacionadas com a atividade v_v, por exemplo, relacionadas por instância de uma atividade de processo ou similaridade textual. A Equação 3 tem duas partes, a primeira é n/m, onde n é o número de tarefas relacionadas com a atividade v_v que tem a participação do desenvolvedor d_e, e m o número todas de tarefas da atividade v_v. Então n/m expressa que quanto maior é o envolvimento do desenvolvedor d_e com as tarefas relacionadas com v_v, maior será o fator desta parcela, para n igual à m o fator é um, o que significa que o desenvolvedor d_e tem conhecimento sobre todas as tarefas do grupo v_v. A outra parte da Equação 3, soma o conhecimento do desenvolvedor sobre toda tarefa pertencente ao grupo v_v.

$$K_v(d_e, v_v, t) = \frac{n}{m} \cdot \sum K_{ts}(d_e, ts_s, t)$$

Equação 3: Medida de conhecimento de um desenvolvedor *d_e* sobre uma atividade *v_v* em um dado tempo *t*. Para toda *ts_s* que tem relação com *v_v*, onde *n* é o número de *ts_s* relacionadas com *v_v* associadas a *d_e* e *m* é o número total de *ts_s* relacionadas com *v_v*.

De forma parecida, a Equação 4, $K_p(d_e, p_c, t)$, mede o conhecimento do desenvolvedor sobre um processo a partir da Equação 3. Dependendo do ambiente, p_c é o processo definido que deu origens às instâncias de atividades, ou ser a união de todos

os grupos de tarefas similares. Assim, a primeira parte da Equação 4 (*r/s)* determina o fator de participação do desenvolvedor nos grupos de atividades formados, caso ele participe de todos os grupos, o fator será 1. A segunda parte da Equação 4 é o somatório do conhecimento do desenvolvedor sobre os grupos de atividades.

$$K_p(d_e, p_c, t) = \frac{r}{s} \cdot \sum K_v(d_e, v_v, t)$$

Equação 4: Medida de conhecimento de um desenvolvedor d_e sobre um processo p_c em um dado tempo t, onde toda atividade v_v está presente em p_c e r é o número de atividades com $K_v(d_e, v_v, t) > 0$ e s é o número total de atividades de p_c.

5. Descrição e Avaliação dos Resultados

Nesta fase inicial, definimos duas questões de pesquisa para avaliarmos se o conjunto de dados coletados para os experimentos representa a produção e o conhecimento dos desenvolvedores em um projeto de software. Se a conclusão for positiva, poderemos utilizá-lo para avaliar os modelos propostos na Seção 4. O projeto Mylyn[2] foi escolhido porque há registros da medida de interesse dos desenvolvedores quando eles realizam edições nos artefatos. Consideramos que essas medidas são uma representação da intensidade de interação que foi definida ma Seção 4.2, $I_{da}(d_e, a_r, \Delta t)$. Seguem as duas questões de pesquisa: Q1 – O conjunto de tarefas do projeto Mylyn que possui as medidas de interesse dos desenvolvedores sobre os artefatos, quando eles realizam edições, representa a <u>produção dos desenvolvedores</u> no projeto Mylyn? Q2 – É possível perceber o <u>conhecimento dos desenvolvedores</u> sobre o projeto Mylyn utilizando a intensidade de interação de edição dos desenvolvedores sobre os artefatos?

Tabela 1: A- Lista oficial do número de erros corrigidos por desenvolvedor publicada em 01/03/2016 no sítio do projeto Mylyn; B – Lista do número de erros corrigidos por desenvolvedor dos dados coletados do projeto Mylyn; C - Lista de percepção do conhecimento dos desenvolvedores sobre o projeto Mylyn em 15/01/2016, segundo a Equação 4, utilizando os dados coletados.

A

	Nome	Erros
1	Steffen Pingel	2053
2	Mik Kersten	1186
3	David Green	516
4	Frank Becker	385
5	Shawn Minto	303

B

	Nome	Erros
1	Steffen Pingel	765
2	Mik Kersten	225
3	Frank Becker	208
4	**Robert Elves**	181
5	Shawn Minto	150
6	David Green	88

C

	Nome	K(d, Mylyn,15/01/2016)
1	Steffen Pingel	479285
2	Shawn Minto	114457
3	**Robert Elves**	17384
4	Frank Becker	14990
5	Mik Kersten	13962
6	David Green	6094

A Tabela 1, Lista A, mostra a lista oficial dos cinco primeiros desenvolvedores que mais corrigiram erros no projeto Mylyn até 01/03/2016, consideramos que esta lista representa a estimativa de produção e do conhecimento dos desenvolvedores sobre o Projeto. A Lista B mostra os seis desenvolvedores que mais corrigiram erros nos dados coletados. O *Robert Elves* não aparece na lista oficial por engano porque é verdade que ele contribuiu com 871 erros corrigidos. Então a resposta para a Q1 é sim, o conjunto de dados coletados representa a <u>produção dos desenvolvedores</u> no projeto Mylyn. A Lista C mostra a lista dos seis desenvolvedores que mais expressaram conhecimento sobre o projeto Mylyn, segundo a Equação 4, utilizando o conjunto de dados coletados. Assim, a resposta para a Q2 também é sim, foi possível perceber o <u>conhecimento dos</u>

2 https://projects.eclipse.org/projects/mylyn

desenvolvedores utilizando a intensidade de interação dos desenvolvedores sobre os artefatos no conjunto de dados coletados do projeto Mylyn.

Concluímos então que o conjunto de dados selecionados representa a produção e o conhecimento dos desenvolvedores sobre o projeto Mylyn. Além disso, os dados apresentam uma boa associação entre artefatos e desenvolvedores na realização de tarefas. Os próximos passos incluem uma avaliação mais específica para o modelo de percepção do conhecimento envolvendo as equações 1, 2 e 3. Logo após iremos realizar uma avaliação do modelo de tarefas.

6. Comparação com Trabalhos Relacionados

Na literatura existem alguns trabalhos que suportam a criação de contexto de tarefas na Engenharia de Software. O Mylyn captura o interesse do desenvolvedor pelos artefatos a partir da interação do desenvolvedor com o ambiente [Kersten 2007]. No Mylyn, quando uma nova tarefa é iniciada, o contexto inicial é igual ao conjunto de artefatos disponíveis no projeto, à medida que o desenvolvedor interage com o ambiente, o Mylyn vai reduzindo a visualização do conjunto de artefatos para minimizar o esforço do desenvolvedor por busca de artefatos. MylynSDP estende o Mylyn com informações sobre a especificação de tarefas e artefatos [Portugal and de Oliveira 2014]. Desta forma, no início de uma nova tarefa, o MylynSDP faz uso desta especificação para disponibilizar no contexto inicial os artefatos e tarefas que seguem a mesma especificação. Mylyn e Mylyn-SDP não consideram o conhecimento colaborativo para a formação do contexto. O modelo de percepção de Omoronyia (2008), CRI, é similar ao modelo do Mylyn. CRI atua em um ambiente colaborativo para produzir medidas de influências entre desenvolvedores, artefatos e tarefas a partir das interações de visualização, alteração, criação e remoção de artefatos realizadas pelos desenvolvedores. CRI também permite visualizar a evolução das influências ao longo do tempo. CRI faz uso do conhecimento colaborativo, mas não o utiliza para inferir um contexto para uma nova tarefa.

Hipikat é uma ferramenta que recomenda artefatos relevantes para a realização de uma nova tarefa de desenvolvimento de software [Cubranic et al. 2005]. Hipikat utiliza um grande número de documentos disponíveis no projeto, tais como, código fonte, documentação, comunicações entre os desenvolvedores (e-mail, fóruns de discussão), relatório de erros e planos de teste. E faz recomendações a partir de similaridades textuais, ligações entre os artefatos disponíveis no projeto e por inferências de ligações. Hipikat foi avaliado em um estudo e obteve uma precisão de 11%. Thompson and Murphy (2014) fazem uso de similaridade de tarefas e de artefatos associados às tarefas para inferir um contexto inicial para uma nova tarefa de programação. O algoritmo proposto por eles para inferir o contexto inicial sobre os dados do Projeto Mylyn obteve uma precisão de 21%. Este trabalho é uma evolução do Hipikat [Cubranic et al. 2005], mais ainda apresenta um indício de precisão muito baixo.

A proposta de Leano *et al.* (2014) tem como objetivo identificar os artefatos que serão editados por uma nova tarefa de desenvolvimento de software. Eles combinam técnicas automatizadas com o conhecimento dos especialistas para validar um contexto inicial. Também fazem uso de técnicas de similaridades utilizando processamento de linguagem natural, grafo de envio de alterações de arquivos em repositório de código e CrowdSourcing. Não há estudos que indicam a validade deste trabalho. Fritz *et al.* (2014) propuseram o DOK (Degree Of Knowledge), um modelo para capturar o conhecimento dos desenvolvedores sobre o código. Este modelo é baseado na autoria do artefato somado ao DOI (Degree Of Interest), modelo de interação do Mylyn. DOK não faz uso das informações de interação entre desenvolvedores quando estes realizam

tarefas.

A nossa proposta é baseada na intensidade de interação dos desenvolvedores sobre os artefatos e na intensidade de interação entre os desenvolvedores, sendo que alguns trabalhos relacionados (Mylyn, CRI e DOK) só se baseiam na interação desenvolvedor-artefato. Os trabalhos Hipikat, [Thompson and Murphy 2014] e [Leano et al. 2014] utilizam similaridade textual de artefato para artefato enquanto a nossa proposta avança na formação de grupos baseando-se em similaridades. Igual a proposta de [Leano et al. 2014], refinamos as descobertas com o conhecimento tácito dos desenvolvedores. O nosso modelo de percepção do conhecimento, Seção 4.2., expressa o conhecimento dos desenvolvedores também sobre tarefas e grupos similares de tarefas utilizando a intensidade de interação focada na edição, diferente da modelo DOK que utiliza o DOI do Mylyn, que pode apresentar medidas de interação que não tem relação com uma interação que identifique conhecimento do desenvolvedor sobre artefato (eventos de predição e propagação de uma visualização).

Referências

Bani-Salameh H, Jeffery C, Al-Gharaibeh J (2010) A Social Collaborative virtual environment for software development. IEEE, pp 46–55

Barthelmess P, Anderson KM (2002) A View of Software Development Environments Based on Activity Theory. Comput Support Coop Work CSCW 11:13–37. doi: 10.1023/A:1015299228170

Cambria E, White B (2014) Jumping NLP Curves: A Review of Natural Language Processing Research [Review Article]. IEEE Comput Intell Mag 9:48–57. doi: 10.1109/MCI.2014.2307227

Cubranic D, Murphy GC, Singer J, Booth KS (2005) Hipikat: a project memory for software development. IEEE Trans Softw Eng 31:446–465. doi: 10.1109/TSE.2005.71

de Souza CRB, Redmiles DF (2011) The Awareness Network, To Whom Should I Display My Actions? And, Whose Actions Should I Monitor? IEEE Trans Softw Eng 37:325–340. doi: 10.1109/TSE.2011.19

Di Ciccio C, Marrella A, Russo A (2015) Knowledge-Intensive Processes: Characteristics, Requirements and Analysis of Contemporary Approaches. J Data Semant 4:29–57. doi: 10.1007/s13740-014-0038-4

Dourish P, Bellotti V (1992) Awareness and Coordination in Shared Workspaces. In: Proceedings of the 1992 ACM Conference on Computer-supported Cooperative Work. ACM, New York, NY, USA, pp 107–114

Feiler PH, Humphrey WS (1993) Software process development and enactment: concepts and definitions. In: Software Process, 1993. Continuous Software Process Improvement, Second International Conference on the. pp 28–40

Fritz T, Murphy GC, Murphy-Hill E, et al (2014) Degree-of-knowledge: Modeling a developer’s knowledge of code. ACM Trans Softw Eng Methodol 23:1–42. doi: 10.1145/2512207

Fritz T, Ou J, Murphy GC, Murphy-Hill E (2010) A degree-of-knowledge model to capture source code familiarity. ACM Press, p 385

Fuggetta A (2000) Software Process: A Roadmap. In: Proceedings of the Conference on The Future of Software Engineering. ACM, New York, NY, USA, pp 25–34

Fuggetta A, Di Nitto E (2014) Software Process. In: Proceedings of the on Future of

Software Engineering. ACM, New York, NY, USA, pp 1–12

Hattori L, Lanza M (2010) Syde: a tool for collaborative software development. ACM Press, p 235

Hunt DP (2003) The concept of knowledge and how to measure it. J Intellect Cap 4:100–113. doi: 10.1108/14691930310455414

Kersten M (2007) Focusing knowledge work with task context. The University of British Columbia

Kersten M, Murphy GC (2005) Mylar: A Degree-of-interest Model for IDEs. In: Proceedings of the 4th International Conference on Aspect-oriented Software Development. ACM, New York, NY, USA, pp 159–168

Koskinen KU, Pihlanto P, Vanharanta H (2003) Tacit knowledge acquisition and sharing in a project work context. Int J Proj Manag 21:281–290. doi: 10.1016/S0263-7863(02)00030-3

Leano R, Kasi B, Sarma A (2014) Recommending Task Context: Automation Meets Crowd. Hong Kong,

Mistrík I, Grundy J, van der Hoek A, Whitehead J (2010) Collaborative Software Engineering: Challenges and Prospects. In: Mistrík I, Grundy J, Hoek A, Whitehead J (eds) Collaborative Software Engineering. Springer Berlin Heidelberg, Berlin, Heidelberg, pp 389–403

Omoronyia I (2008) Enhancing awareness during distributed software development. PhD thesis, Department of Computer and Information Sciences, University of Strathclyde, Glasgow, UK

Osterweil L (1987) Software Processes Are Software Too. In: Proceedings of the 9th International Conference on Software Engineering. IEEE Computer Society Press, Los Alamitos, CA, USA, pp 2–13

Portugal IS, de Oliveira TC (2014) Using task contexts to improve software process execution. In: CIBSE 2014: Proceedings of the 17th Ibero-American Conference Software Engineering. pp 109–122

Qumer A, Henderson-Sellers B (2008) A framework to support the evaluation, adoption and improvement of agile methods in practice. J Syst Softw 81:1899–1919. doi: 10.1016/j.jss.2007.12.806

Robbes R, Lanza M (2008) SpyWare: a change-aware development toolset. ACM Press, p 847

Ryan S, O’Connor RV (2013) Acquiring and sharing tacit knowledge in software development teams: An empirical study. Inf Softw Technol 55:1614–1624.

Storey M-AD, ubrani D, German DM (2005) On the use of visualization to support awareness of human activities in software development: a survey and a framework. ACM Press, p 193

Thompson CA, Murphy GC (2014) Recommending a starting point for a programming task: an initial investigation. ACM Press, pp 6–8

Whitehead J, Mistrík I, Grundy J, Hoek A van der (2010) Collaborative Software Engineering: Concepts and Techniques. In: Mistrík I, Grundy J, Hoek A, Whitehead J (eds) Collaborative Software Engineering. Springer Berlin Heidelberg, pp 1–30

Uma Abordagem para Prevenir a Erosão do *Design* baseada em Recomendações Sensíveis à Arquitetura

Aluno: Marcos Barbosa Dósea[1, 2]

Orientador: Cláudio Nogueira Sant'Anna[1]

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

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

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

Nível: Doutorado
Programa: Programa de Pós-Graduação em Ciência da Computação da Universidade Federal da Bahia
Ano de Ingresso no Programa: 2014
Época prevista para conclusão: Março de 2018
Época prevista para qualificação de tese: Dezembro 2016
Eventos Relacionados: SBES e SBCARS

Resumo. *Muitas estratégias automáticas têm sido propostas para identificar anomalias no design do código fonte. O objetivo é prevenir a erosão do design. Entretanto, para evitar a ocorrência de falsos positivos e falsos negativos, as abordagens precisam ser manualmente calibradas para considerar informações contextuais do código. Adicionalmente, muitas dessas abordagens apresentam tardiamente os problemas detectados no código desenvolvido. Nesse contexto, esse trabalho propõe uma nova técnica para extrair automaticamente regras de design do código de um projeto referência e utilizá-las para verificar o código de um projeto em desenvolvimento. As regras são extraídas e verificadas de acordo com o interesse arquitetural da classe e recomendações são enviadas para o desenvolvedor. Resultados iniciais mostram que os desenvolvedores realmente possuem dificuldades em utilizar as abordagens existentes para detectar anomalias no design. Também já possuímos evidências iniciais que o interesse arquitetural da classe deve ser considerado nas abordagens que utilizam métricas. Atualmente estamos evoluindo a técnica proposta a partir de uma prova de conceito realizada. Em seguida pretende-se realizar avaliações através de estudos de caso com desenvolvedores da academia e da indústria.*

Palavras-Chave. *Detecção de Anomalias no Código, Sistemas de Recomendação, Interesses Arquiteturais*

1. Caracterização do Trabalho

A revisão de código é uma prática comum utilizada para manter a qualidade do código e garantir que as decisões de *design* sejam cumpridas [Fagan 1976]. A erosão do *design* é um processo inevitável, mas boas práticas de desenvolvido incrementam a longevidade do sistema [van Gurp & Bosch 2002]. Um dos principais sintomas de erosão do *design* é a manifestação progressiva de anomalias no código.

Anomalias de código, também conhecidas como *code smells* [Fowler & Beck 1999], são sintomas no código fonte que podem indicar graves problemas de manutenibilidade. O envelhecimento do software e desenvolvedores inexperientes sempre foram considerados como as principais causas da erosão do *design* [Parnas 1994]. Entretanto, Lavallée & Robillard (2015) mostram que fatores organizacionais são os principais motivos para que mesmo bons desenvolvedores produzam código de baixa qualidade. Tufano et al. (2015) também exibem evidências que *code smells* são introduzidos por desenvolvedores experientes e que as práticas de revisão do código devem ser fortalecidas principalmente quando desenvolvedores estão trabalhando sobre pressão da organização. Adicionalmente mostram ainda que vários artefatos são afetados por *code smells* desde a sua criação e que as atividades de manutenção corretiva e evolutiva são as que mais introduzem anomalias.

Para auxiliar o processo de revisão manual do código, várias estratégias automáticas de revisão vêm sendo propostas [Marinescu 2004; Li & Zhou 2005; Arcoverde et al. 2012; Palomba et al. 2013]. A maioria dessas estratégias baseia-se em métricas coletadas do código. Um *code smell* é identificado quando os valores das métricas ultrapassam valores limiares fixados no início da análise. Essa abordagem leva frequentemente a um alto número de falsos positivos e falsos negativos que podem desmotivar sua utilização pelas equipes de desenvolvimento. Dessa forma, a determinação do valor limiar a ser utilizado em cada métrica torna-se um desafio para evitar um número excessivo de notificações para o desenvolvedor.

Alguns estudos propõem métodos para derivar valores limiares a partir da análise de código de um conjunto de sistemas [Ferreira et al. 2012; Fontana et al. 2015]. Entretanto, esses valores genéricos acabam desconsiderando decisões de *design* do sistema e precisam ser calibrados manualmente. Segundo Fontana *et al.* (2013) nem todos os *code smells* encontrados pelas abordagens automáticas são relevantes e a acurácia das ferramentas poderia ser melhorada caso utilizassem conhecimento do domínio e do design do sistema avaliado. Zhang *et al.* (2013) mostra evidências que informações do contexto como o domínio da aplicação, número de mudanças, tamanho e idade do sistema deveriam ser consideradas na utilização de métricas, por exemplo, para determinação de valores limiares a serem utilizados nas métricas.

Outro problema identificado é que muitas estratégias que identificam anomalias no código fonte precisam ser executadas manualmente pelo desenvolvedor. Se essa tarefa for postergada para o final da codificação da funcionalidade ou para o momento de gravação do código no repositório pode faltar tempo ou motivação para que os desenvolvedores realizem as correções sugeridas. Sistemas de recomendação em engenharia de software [Gasparic & Janes 2016] vêm se tornando uma alternativa viável para auxiliar desenvolvedores no processo de tomada de decisão. No processo de

codificação, poderiam auxiliar desenvolvedores a manterem-se informados das decisões de *design* existentes ou evoluções destas no sistema que precisarão modificar.

Nesse contexto, esse trabalho apresenta o estado atual de um estudo sobre uma nova técnica para prevenir a erosão do *design* através da recomendação de anomalias de *design* que consideram o interesse arquitetural da classe analisada. Nosso objetivo de pesquisa é investigar se a extração automática dessas regras de *design,* considerando o interesse arquitetural das classes, melhora a acurácia das recomendações propostas em relação às abordagens existentes. O interesse arquitetural da classe é capturado a partir do design do código fonte do sistema avaliado. As anomalias de design que pretendemos considerar são às detectadas através de métricas obtidas do código. Para alcançar esse objetivo estamos realizando estudos empíricos para responder as seguintes questões de pesquisa:

(Q1) O que os desenvolvedores pensam sobre as estratégias disponíveis para detecção de anomalias no código fonte?

(Q2) Existem diferenças estatisticamente significantes entre as características do código fonte de classes com interesses arquiteturais distintos?

(Q3) É possível extrair regras de *design* relevantes de um código fonte que segue essas mesmas regras e utilizá-las para avaliação de um código fonte em desenvolvimento?

(Q4) Recomendar anomalias de *design* sensíveis ao contexto arquitetural ajudam a prevenir a erosão do *design*?

Este artigo detalha o método adotado para responder as questões de pesquisa propostas e está organizado da seguinte forma: a Seção 2 apresenta a fundamentação teórica do trabalho; a Seção 3 destaca as principais contribuições do estudo; a Seção 4 discute trabalhos relacionados ao estudo realizado; na Seção 5 detalhamos o estado atual da pesquisa em desenvolvimento; finalmente a Seção 6 discute a abordagem que será utilizada para avaliação.

2. Fundamentação Teórica

A. *Interesse Arquitetural da Classe*

O interesse arquitetural de uma classe define a principal responsabilidade da classe dentro do design do código avaliado. Para identificar o interesse arquitetural propomos a utilização de duas informações: (1) o papel arquitetural da classe e (2) o design do código da aplicação. Em sistemas que seguem arquiteturas referências [Medvidovic & Taylor 2010], o papel arquitetural da classe é normalmente atribuído através de herança, anotação ou implementação de uma interface provida por essa arquitetura referência. Por exemplo, em aplicações que seguem a arquitetura referência ASP.NET MVC[1], uma classe possui o papel de Controller quando estende uma classe abstrata com o mesmo nome. Já aplicações que seguem a arquitetura referência Spring WEB MVC[2], essa atribuição é realizada através da inserção da anotação @Controller.

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

Entretanto, decisões de design podem das responsabilidades distintas a classes com o mesmo papel arquitetural. Além das responsabilidades definidas pela arquitetura referência, podem ser atribuídas novas responsabilidades a uma classe com um determinado papel arquitetural. Por exemplo, gerenciar regras de negócio no Controller é uma decisão de design que pode ser tomada para simplificar a arquitetura. Se compararmos classes Controllers que possuem essa responsabilidade adicional com outras que não possuem, podemos levar para o desenvolvedor um número grande de problemas que na verdade não precisam ser resolvidos. Nossa proposta considera essas decisões de design presentes no código com o objetivo de melhorar a acurácia das estratégias de detecção de anomalias no código.

B. *Identificação Automática de Anomalias no Código*

Automatizar o processo de detecção de anomalias no código ajuda a evitar a erosão do *design* do código. Entretanto, segundo Fontana et al. (2012) diferentes ferramentas podem prover resultados distintos quando eles analisam o mesmo sistema pelas seguintes razões: (i) a ambiguidade na definição de *code smells* e como consequência as diferentes interpretações dadas pelos desenvolvedores das ferramentas; (ii) as diferentes técnicas utilizadas pelas ferramentas para detectar os *smells;* (iii) os diferentes valores limiares utilizados mesmo quando as técnicas utilizadas são análogas ou idênticas.

Outra dificuldade é que a avaliação dessas ferramentas é superficial, realizada com poucos *smells* e em sistemas pequenos. Alguns exemplos de ferramenta são o Checkstyle[3], PMD[4] e JDeodorant[5]. Nem todos os *code smells* são detectados por essas ferramentas. Mas alguns *smells* como *Long Method* e *Large Class* são normalmente disponibilizados por todas as ferramentas.

Finalmente, nem todas as abordagens automatizadas para detecção de anomalias estão integradas ao ambiente de desenvolvimento do código. Dessa forma, a execução externa ao ambiente dificulta a navegação através do código e o acesso às anomalias detectadas.

C. *Recomendação em Engenharia de Software*

Desenvolvedores precisam ter consciência das decisões de *design* que devem ser obedecidas durante a implementação do código. A falta de conhecimento pode levar o desenvolvedor a navegar por grandes quantidades de código ou documentos em busca dessas informações de *design*. Sistemas de recomendação tornam-se uma alternativa para ajudar desenvolvedores a tomar decisões onde falta experiência ou há dificuldade para obter toda informação necessária para continuidade do trabalho [Robillard et al. 2010].

Segundo Gasparic & Janes (2016) os sistemas de recomendação em engenharia de software existentes precisam investir na melhoria da apresentação dos resultados. A maioria das ferramentas exibe alguns poucos artefatos e deixa para o desenvolvedor descobrir o que fazer. Isso justificaria a baixa utilização prática dos sistemas de recomendação produzidos pela comunidade de engenharia de software. Outro desafio é a utilização de informações contextuais ainda pouco utilizadas por esses sistemas e que poderia melhorar as recomendações fornecidas.

[3] http://checkstyle.sourceforge.net/
[4] http://pmd.github.io/
[5] https://marketplace.eclipse.org/content/jdeodorant

3. Contribuições

As principais contribuições desse trabalho são listadas a seguir:

i) Oferecer um panorama sobre a opinião dos desenvolvedores de código sobre as estratégias utilizadas para revisão do código.

ii) Avaliar se o interesse arquitetural da classe influencia de forma estatisticamente significante na distribuição dos valores das métricas.

iii) Definir e avaliar uma técnica para recomendar anomalias de *design* contextuais extraindo informação de um código com os mesmos interesses arquiteturais do código avaliado.

iv) Avaliar se a recomendação de anomalias de *design* considerando o interesse arquitetural da classe reduz a erosão do *design*.

4. Trabalhos Relacionados

Uma variedade de abordagens propõem evitar a erosão do *design* a partir de estratégias de detecção de anomalias no código baseadas principalmente em métricas. Alguns estudos discutem a importância de considerar o contexto na avaliação do código fonte. Marinescu (2006) mostra como a acurácia na detecção dos *code smells Data Class* e *Feature Envy* pode ser melhorada quando são utilizadas na análise informações do *design* da aplicação. O estudo revelou que considerar o papel arquitetural teve um grande impacto na eliminação de falsos positivos das estratégias clássicas de detecção. Nossa proposta também considera o papel arquitetural, mas propomos também a utilização de informações do design do código fonte, extraídas de forma automática e utilizadas para recomendar anomalias de design para o desenvolvedor.

Guo *et al.* (2010) defende que as regras usadas para detecção de *code smells* devem considerar informações do domínio da aplicação analisada. Nas abordagens baseada em métricas, a configuração adequada de valores limiares é crítico para detecção de *smells* específicos do domínio analisado. Através da discussão com especialistas alguns valores limiares são calibrados para adequar ao domínio da aplicação, por exemplo, classes geradas para controlar a interface com o usuário possuem muitos métodos públicos e algumas podem ser consideradas como *God Class*. Nosso trabalho também utiliza informações do domínio da aplicação, mas estas são extraídas automaticamente. Dessa forma pretendemos evitar ajustes manuais para cada domínio de aplicação que for analisar.

Macia *et al.* (2013) explora a ligação entre a estrutura arquitetural com os *code smells*. No estudo, é proposto um conjunto de estratégias de detecção que utilizam métricas sensíveis à arquitetura. A avaliação realizada indica um incremento de 50% no número de anomalias relevantes detectadas. Entretanto, a abordagem utiliza valores limiares genéricos extraídos de sistemas distintos que precisaram ser calibrados por desenvolvedores da aplicação. Nossa abordagem difere porque nós consideramos apenas um sistema como referência de design para extração de valores limiares e esses valores não são genéricos para toda aplicação. Eles dependem do interesse arquitetural da classe avaliada.

5. Metodologia e Estado Atual do Trabalho

A metodologia para execução deste trabalho foi organizada para responder as quatro questões de pesquisa definidas na Seção 1. A Figura 1 apresenta o processo metodológico organizado como uma sequência de estudos empíricos que estamos conduzindo para definição e avaliação da técnica proposta.

Figura 1. Metodologia de Execução do Trabalho

A primeira questão de pesquisa (Q1) está sendo respondida através da execução das duas primeiras atividades exibidas no processo da Figura 1. Para saber o que os desenvolvedores pensam sobre as técnicas existentes para evitar erosão do *design*, inicialmente realizamos um mapeamento sistemático para ter uma visão ampla sobre as abordagens existentes. O mapeamento serviu de base para o planejamento da segunda atividade do percurso metodológico. O *survey* já foi planejado e disponibilizado na Web para os participantes no período de 21 de dezembro de 2014 a 13 de Fevereiro de 2015. Um total de 375 respondentes iniciaram o *survey* e 321 (85,6%) completaram todas as questões mandatórias. O *survey* possuía 12 questões, das quais cinco questões foram sobre informações demográficas. Atualmente já realizamos a análise dos resultados e estamos escrevendo um artigo para divulgação.

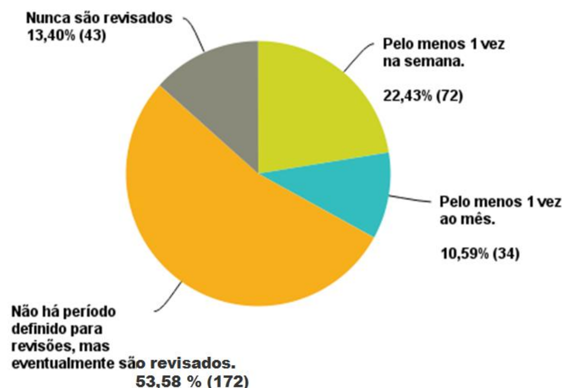


Figura 2. Periodicidade de Revisão do Código Fonte

A Figura 2 apresenta o resultado de uma das questões realizadas no *survey* que questiona sobre a periodicidade de revisão do código. Apesar de 86,6% dos respondentes afirmarem realizar revisões no código, somente 22,43% realizam a revisão pelo menos uma vez por semana. Quanto maior a periodicidade de revisão maior será o custo e tempo alocado para realizar revisão e aplicar as correções no código desenvolvido. A utilização das abordagens automáticas poderia diminuir o número de anomalias detectados por uma revisão manual. Entretanto o número de falsos positivos e falsos negativos gerados desmotivam sua utilização. Nossa hipótese é que os resultados poderiam ser melhorados caso o interesse arquitetural da classe fosse considerado.

A avaliação dessa hipótese corresponde à segunda questão de pesquisa (Q2) e a terceira atividade do percurso metodológico. Para responder a essa questão avaliamos a distribuição de valores de quatro métricas relacionadas a métodos e verificamos se existem diferenças estatisticamente significantes quando consideramos os interesses arquiteturais das classes. Foram avaliados 15 sistemas reais escolhidos de forma sistemática do repositório GitHub e pertencentes a três domínios distintos: aplicações Web, aplicações Mobile e *plug-ins* do Eclipse. Os valores obtidos foram avaliados

através do teste estatístico não-paramétrico Kruskal-Wallis para verificar se existem diferenças estatisticamente significativas no valor de cada métrica. Os resultados obtidos mostraram que os interesses arquiteturais devem ser considerados, indicando que ao invés de definir valores limiares genéricos para todo sistema, esses valores deveriam ser definidos para cada interesse arquitetural analisado no sistema.

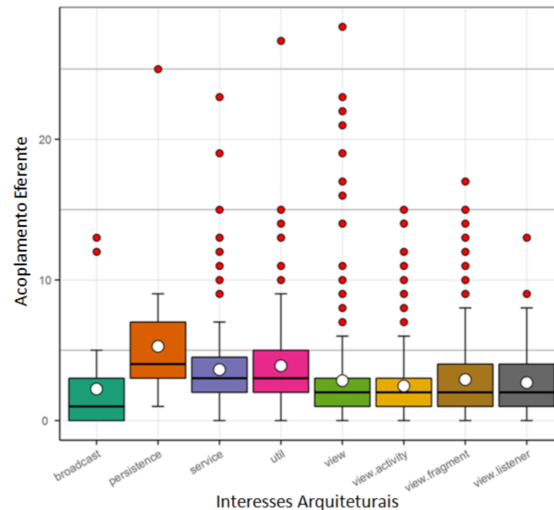


Figura 3. Distribuição de Valores da Métrica Acoplamento Eferente em Interesses Arquiteturais Distintos

A Figura 3 exibe graficamente a distribuição de valores em um dos sistemas avaliados para métrica de acoplamento eferente em oito interesses arquiteturais. Percebe-se que o acoplamento eferente dos métodos no interesse arquitetural *persistence* é muito maior que a do interesse *broadcast*. Pelo menos 75% dos métodos de *broadcast* possui acoplamento eferente inferior aos menores valores encontrados nos métodos do interesse arquitetural *persistence*.

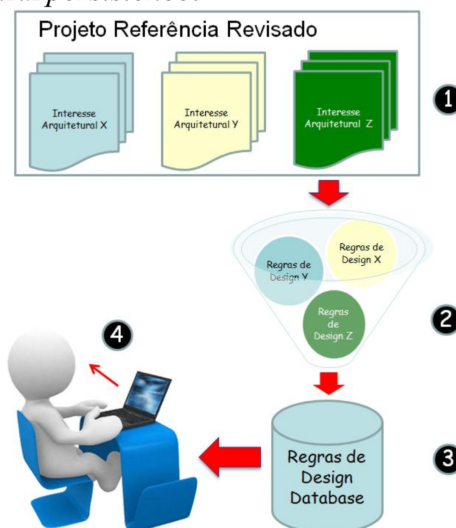


Figura 4. Técnica para Recomendação de Anomalias de Design Sensível à Arquitetura.

A Figura 4 mostra as etapas da técnica de recomendação de anomalias de design sensíveis à arquitetura que propomos para responder a terceira questão de pesquisa (Q3). A primeira etapa é selecionar um projeto que possua decisões de *design* semelhantes às que serão verificadas no projeto em desenvolvimento. Esse projeto referência pode ser uma versão anterior revisada do software que será analisado ou ainda um projeto que possua decisões de *design* semelhantes. Na segunda etapa, é realizada a identificação do interesse arquitetural de cada classe do projeto referência. Essas classes são agrupadas de acordo com o seu interesse arquitetural e regras de

design são extraídas de cada grupo e armazenadas na terceira etapa em meio persistente. Alguns exemplos de regras de design são o tamanho e complexidade dos métodos.

O quarto e último passo do método é identificar o interesse arquitetural das classes do projeto em desenvolvimento. Para cada classe, classificada em um determinado interesse arquitetural, são verificadas se as regras de design do interesse arquitetural correspondente estão sendo obedecidas. A avaliação da proposta apresentada corresponde à quarta questão de pesquisa apresentada a seguir.

6. Metodologia de Avaliação

Para realizar a avaliação e responder a quarta questão de pesquisa (Q4) inicialmente desenvolvemos uma prova de conceito com uma versão inicial da técnica proposta. Foi definida uma técnica que identifica interesses arquiteturais automaticamente a partir de um projeto referência e identifica a anomalia de design métodos longos. Para dar suporte ao método foi desenvolvido um *plug-in* para o Eclipse. Quando as definições para o tamanho do método são desobedecidas, o ambiente recomenda para o desenvolvedor que aquele método, considerando o interesse arquitetural da classe, pode ser um método longo. O método foi avaliado em dez versões do sistema MobileMedia e os resultados estão sendo descritos para publicação.

Pretendemos evoluir o método para identificar outras anomalias de *design* e realizar avaliações através de dois estudos experimentais, um com estudantes e outro no ambiente da indústria. No estudo com estudantes, pretendemos avaliar a efetividade do uso de uma ferramenta de recomendação para melhoria da qualidade do código. Nosso objetivo é separar os alunos em dois grupos e propor a realização de atividades de manutenção no código. O primeiro grupo utilizando a nossa técnica e o segundo grupo sem utilizar. Em seguida faremos uma avaliação qualitativa e quantitativa sobre a influência da técnica na qualidade do código resultante.

No estudo a ser realizado no ambiente da indústria, pretendemos avaliar a acurácia da nossa abordagem em relação as existentes para detecção de anomalias. Inicialmente pretendemos que os desenvolvedores, utilizando sistemas reais, identifiquem possíveis *anomalias de código* em um conjunto de classes selecionadas. Em seguida identificaremos as anomalias com a nossa abordagem e as existentes. Solicitaremos aos desenvolvedores para avaliar os resultados obtidos. Através de uma análise quantitativa e de um questionário entregue aos participantes, pretendemos avaliar se a abordagem contribuiu para encontrar *anomalias de código* mais relevantes.

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

References

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

Fagan, M.E., 1976. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3), pp.182–211.

Ferreira, K. a. M.M. et al., 2012. Identifying thresholds for object-oriented software metrics. In *Journal of Systems and Software*. Elsevier Inc., pp. 244–257.

Fontana, F.A. et al., 2015. Automatic metric thresholds derivation for code smell

detection. 6th *International Workshop on Emerging Trends in Software Metrics*.

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*.

Fontana, F.A., Braione, P. & Zanoni, M., 2012. Automatic detection of bad smells in code: An experimental assessment. *Journal of Object Technology*, 11(2), pp.1–38.

Fowler, M. & Beck, K., 1999. *Refactoring: improving the design of existing code*, Addison-Wesley.

Gasparic, M. & Janes, A., 2016. What recommendation systems for software engineering recommend: A systematic literature review. *Journal of Systems and Software*, 113, pp.101–113.

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. *37th IEEE International Conference on Software Engineering*.

Li, Z. & Zhou, Y., 2005. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. *ACM SIGSOFT Software Engineering Notes*, 1(5), pp.306–315.

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*.

Medvidovic, N. & Taylor, R.N., 2010. Software architecture. In *32nd ACM/IEEE International Conference on Software Engineering*. New York.

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

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

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

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*.

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

Aprendizagem Contínua aplicada ao Problema de Seleção de Otimizações com Estimativa Estática

João Fabrício Filho
joaof@utfpr.edu.br

Orientador: Anderson Faustino da Silva
anderson@din.uem.br

Mestrado

Programa de Pós-graduação em Ciência da Computação
Universidade Estadual de Maringá

Ano de ingresso no programa: 2015

Previsão de conclusão: Fevereiro/2017

Data da aprovação da qualificação: 30 de Março de 2016

Resumo: Os compiladores modernos aplicam otimizações aos códigos, na tentativa de melhorar a qualidade dos códigos gerados. O problema de seleção de otimizações consiste na escolha de uma sequência de otimizações que seja capaz de gerar um código de boa qualidade. Nesse contexto, o espaço de busca é amplo, consequentemente uma busca exaustiva é impraticável. A aplicação da aprendizagem contínua de longo prazo mostra-se uma alternativa viável e prática para reduzir o custo de uma busca exaustiva. Contudo, o custo de se obter uma sequência de otimizações ainda demanda várias avaliações – compilar, executar e medir o tempo de execução do programa a cada sequência encontrada –, o que inviabiliza a aplicação de estratégias desse porte para usuários finais. O objetivo deste trabalho é aplicar a aprendizagem contínua de longo prazo ao problema de seleção de otimizações, avaliando os códigos gerados sem a necessidade de uma execução real. Espera-se contribuir no intuito de aproximar tal estratégia a usuários finais, reduzindo o seu custo computacional.

Palavras-chave: Compiladores, Otimizações de código, Aprendizagem Contínua, Estimativas Estáticas de Código.

SBLP

Aprendizagem Contínua aplicada ao Problema de Seleção de Otimizações com Estimativa Estática

João Fabrício Filho [1], Anderson Faustino da Silva [1]

[1]Programa de Pós-Graduação em Ciência da Computação
Departamento de Informática
Universidade Estadual de Maringá (UEM)
Avenida Colombo 5790 – Bloco C56 – CEP 87020900 – Maringá/PR – Brasil

anderson@din.uem.br, joaof@utfpr.edu.br

1. Caracterização do Problema

Compiladores [Aho et al. 2006] são programas que traduzem códigos de uma linguagem-fonte para uma linguagem-alvo . A tradução do código para a linguagem alvo não é uma tarefa trivial, já que existem diferentes instruções para traduzir os comandos da linguagem fonte.

Durante o processo de tradução, existe uma etapa de otimização, na qual há uma tentativa de melhorar a tradução realizada, mantendo a semântica do código. Nessa etapa, algoritmos de transformação são aplicados ao código de entrada de forma a tentar melhorá-lo. Tais algoritmos, embora sejam chamados de otimizações de código, são heurísticas, podendo em certos casos acarretar perda de desempenho.

Várias características influenciam o desempenho obtido pelo código gerado, após a aplicação de uma otimização. Tais como: instruções utilizadas no código; configuração da máquina em que se vai executá-lo; otimizações aplicadas anteriormente; e parâmetros de compilação. Isso indica que a seleção de otimizações é um problema específico para cada programa em particular, como também para cada arquitetura de *hardware*.

O problema de seleção de otimizações consiste na obtenção das melhores otimizações e seus respectivos parâmetros para aplicar a um código de entrada de forma que o código final tenha um bom desempenho.

Os trabalhos pesquisados de técnicas da área de seleção de otimizações não possuem custo computacional e tempo de execução que sejam razoáveis para obter respostas em tempo hábil para o usuário final. Compiladores modernos, como CLANG [Lattner and Adve 2004] e GCC [Stallman and DeveloperCommunity 2009], não implementam soluções para tal problema, mas possuem sequências pré-fixadas para serem ativadas por meio de parâmetros de compilação (-O1, -O2 e -O3).

A aprendizagem contínua de longo prazo [Tartara and Reghizzi 2013] se mostra como uma alternativa viável para o problema de seleção de otimizações pelo atrativo de fazer com que cada experiência adquirida seja utilizada posteriormente. Além disso, a aprendizagem de máquina combina estratégias que vasculham seletivamente o espaço de busca.

As avaliações de código necessárias na aprendizagem contínua de longo prazo se realizam por meio da execução do código, o que aumenta o tempo de resposta do sistema otimizador.

Estimativas estáticas são uma alternativa à execução. Nesse contexto, o trabalho de Wu e Larus (1994) propôs uma técnica que se mostrou eficiente para estimar a frequência relativa de execução de blocos básicos e de chamadas de funções. Portanto, o trabalho de Wu e Larus (1994) tem grande potencial para reduzir o tempo de resposta do sistema.

O objetivo deste trabalho é melhorar uma técnica para solucionar o problema de seleção de otimizações, em que se aplique a aprendizagem contínua, e que não haja necessidade de executar os códigos gerados para realizar avaliações.

Para alcançar tal objetivo, objetivos específicos são estabelecidos a seguir:

- Melhorar um algoritmo de aprendizagem contínua no qual haja conhecimento acumulado para utilização em novas execuções do algoritmo;
- Aplicar o conhecimento acumulado de forma a guiar a busca por melhores soluções; e
- Utilizar uma técnica de estimativa de custos para avaliação estática de códigos.

2. Fundamentação Teórica

Diversas técnicas já foram aplicadas ao problema de seleção de otimizações, em diferentes situações.

O trabalho de Martins *et al* (2014) reduziu o espaço de busca utilizando verificação de similaridade de código para aplicar uma busca por agrupamento. A busca ocorre em duas etapas: (1) verificação de similaridade para reduzir o número de conjuntos da busca e (2) aplicação de uma estratégia de busca local [Martins et al. 2014].

A verificação de similaridade é baseada em uma comparação, na qual cada código possui uma representação por cadeia de *strings*, que é chamada de DNA. Cada caractere dessa cadeia é chamado de gene e representa uma instrução do código. Um código é dito similar a outro quando seus DNAs contém genes similares. Assim, se reduz a busca a apenas conjuntos em que códigos mais similares obtiveram bons resultados. A estratégia de busca local realiza-se por técnicas simples, que exploram os conjuntos sem maiores custos computacionais.

O trabalho de Purini e Jain (2013) aplicou uma redução da amostragem para cada diferente classe de programa em uma fase de treinamento. Na execução, extrai as dez melhores sequências de otimizações dessa amostragem, simplificando a busca pela melhor sequência. A ideia é obter uma cobertura de boas sequências para todas as classes de programas, com um espaço de busca simples para aplicação de estratégias menos custosas [Purini and Jain 2013].

Com uma estratégia de busca mais elaborada, técnicas de aprendizagem de máquina aplicadas ao problema de seleção de otimizações se apresentam como uma alternativa viável para a redução do espaço de busca, além da coleta de informações de execuções anteriores, que contribui para uma busca intuitiva. Porém, a maioria dos algoritmos que aplicam essa técnica requer uma fase de treinamento antes da utilização da solução. A fase de treinamento geralmente é uma fase custosa, que pode levar dias para ser completada.

O trabalho de Tartara e Reghizzi (2013) aplicou aprendizagem de máquina ao problema de seleção de otimizações, introduzindo aprendizagem de longo prazo, sem a

necessidade de fase de treinamento. Evitar essa fase é de grande valia, já que pode-se iniciar a busca por soluções imediatamente. A aprendizagem inicia nas primeiras execuções do *software* otimizador, e ao longo do tempo, se estabelecem métricas para avaliar as otimizações utilizadas. As métricas utilizadas para comparação de códigos são dados de blocos básicos do código de entrada, que inicialmente são construídas aleatoriamente, e se modificam conforme a evolução ou mutação da população de soluções.

3. Comparação com Trabalhos Relacionados

A aprendizagem contínua de Tartara e Reghizzi (2013) é a base de partida deste trabalho, com abordagens adicionais ao problema de seleção de otimizações. A estimativa estática é o principal diferencial, já que o trabalho de [Tartara and Reghizzi 2013] avalia os códigos gerados por meio da execução.

Uma desvantagem observada na aprendizagem contínua é que não há o retorno de um conjunto sem a necessidade de avaliações, mesmo que haja na base de conhecimento um código semelhante ao de entrada. Desse modo, uma abordagem deste trabalho diferente em relação ao de [Tartara and Reghizzi 2013] é a utilização de uma segunda base de dados para verificar similaridade entre códigos, e reduzir o espaço de busca para a melhor sequência utilizada para um código similar. A similaridade entre códigos tem base no trabalho de Martins *et al* (2014),mas diferentemente desse trabalho, não são aplicados algoritmos de busca em sequências de códigos similares. A busca é reduzida a uma sequência de otimizações, que é retornada sem a preocupação de estimar seu custo.

O trabalho de Purini e Jain (2013) aplicou redução da amostragem de sequências em uma fase de treinamento, a qual não haverá neste trabalho. Além disso, o trabalho de Purini e Jain (2013) elenca as melhores otimizações por classe de programas, enquanto este trabalho propõe classificações por (1) similaridade de código e (2) características de blocos básicos.

Outro diferencial deste trabalho em relação aos três citados é o fato de focar o processo de seleção de otimizações analisando as funções de maior custo do programa, também chamadas de funções quentes.

4. Contribuições

A proposta deste trabalho é atacar o problema de seleção de otimizações de forma a diminuir o custo da busca por soluções. Espera-se contribuir com a aproximação da aplicação das técnicas de solução do problema de seleção de otimizações a usuários finais, reduzindo o seu custo computacional e consequentemente o tempo de resposta do sistema. Demais contribuições deste trabalho são listadas a seguir:

- Reduzir o custo das avaliações de código, fazendo com que ao invés da execução do programa do usuário, o sistema estime o custo dos códigos gerados;
- Diminuir o tempo de resposta do sistema, consequência tanto da exploração intuitiva do espaço de busca quanto da redução do custo das avaliações;
- Propor um sistema de seleção de otimizações que considere as funções de quentes; e
- Reduzir o número de avaliações, por meio da exploração intuitiva de códigos similares.

O Algoritmo 1 apresenta a estratégia de busca utilizada neste trabalho, que está baseada na aprendizagem contínua de longo prazo [Tartara and Reghizzi 2013], no uso de funções de quentes e na avaliação por meio de técnicas de estimativa estática de código [Wu and Larus 1994].

Algorithm 1: Proposta mesclando aprendizagem contínua, características de funções quentes e estimativa estática

Input: src, código-fonte do programa a ser compilado
Output: bin, código gerado com a melhor sequência de otimizações

```
f ← Função de maior custo em src
c ← Código mais similar a f em B_0
if Similaridade( c, f ) < S then
    C ← Melhor conjunto utilizado em c
    bin ← Compilar( src, C )
else
    bin_0 ← Compilar( src, C̄ )
    custo_0 ← CustoEstimado( bin_0 )
    R ← {⟨bin_0, C̄, custo_0⟩}
    for i ← 1 to numGeracoes do
        Elite ← MelhoresCandidatosPorHeuristicas( h, B_1 )
        NovosCandidatos ← EvoluirOuMutar( B_1, p, m )
        E ← Elite ∪ NovosCandidatos
        foreach C_i ∈ E do
            bin_i ← Compilar( src, C_i )
            custo_i ← CustoEstimado( bin_i )
            R ← R ∪ {⟨bin_i, C_i, custo_i⟩}
        bin ← MelhorBinarioDaMelhorConfiguracao( R )
        B_0, B_1 ← AtualizarBaseConhecimento( R )
return bin
```

Os parâmetros de configuração do algoritmo são listados a seguir:

- $numGeracoes$: o número de gerações da aprendizagem contínua.
- B_0: base de dados com resultados, melhores sequências e códigos já otimizados para avaliação de similaridade.
- B_1: base de conhecimento com estimativas de custos, heurísticas e sequências de otimizações aplicadas.
- S: nível de similaridade a ser atingido para considerar um código de B_0 similar ao código de entrada.
- $\bar{C}$: sequência padrão de otimizações tida como *baseline*.
- p: probabilidade de utilizar novos candidatos ao invés de candidatos da base de conhecimento B_1.
- m: probabilidade de se ativar a mutação ao evoluir um candidato.

As funções utilizadas para representar o algoritmo são:

- `Compilar`: aplica a sequência de otimizações de entrada ao código-fonte, e retorna o executável.

- `CustoEstimado`: estima estaticamente o custo do código gerado.
- `Similaridade`: mede o grau de similaridade entre dois códigos.
- `MelhoresCandidatosPorHeuristicas`: coleta as heurísticas mais frequentes, os melhores candidatos para o código de h e as melhores heurísticas da base de conhecimento B_1.
- `EvoluirOuMutar`: seleciona um número de candidatos para completar a população e, por meio de uma probabilidade evolui ou faz o candidato escolhido sofrer mutação.
- `MelhorBinarioDaMelhorConfiguracao`: encontra o menor custo estimado nas tuplas de entrada, retornando o código executável correspondente.
- `AtualizarBaseConhecimento`: grava os dados da tupla de entrada nas bases de conhecimento.

Estimar o custo de um código gerado implica em estabelecer o custo da execução de seu grafo de fluxo de controle. Isso é realizado por meio de uma variante do trabalho proposto por Wu e Larus (1994). Tal variante utiliza o custo de cada instância do bloco básico, para estimar o custo do código gerado. Dessa forma, é possível aumentar a acurácia da estimativa. O custo de uma instrução é calculado considerando a tradução de código e o *hardware* final, pelo fato da estimativa ser realizada em código intermediário.

A proposta deste trabalho é realizar uma verificação de similaridade, tendo como parâmetro de similaridade um grau S, que se alcançado indica que o código de entrada é similar a um código visto anteriormente. Ao se ter um código similar, o conjunto retornado é o melhor conjunto utilizado pelo código da base. A medida de similaridade terá por base a técnica de comparação de cadeias utilizando o algoritmo de Needleman-Wunsch [Needleman and Wunsch 1970], que indica dois códigos similares como os que possuem o maior escore.

5. Descrição e Avaliação dos Resultados

Os resultados serão comparados com a implementação original da aprendizagem contínua de longo prazo proposta por [Tartara and Reghizzi 2013], a fim de avaliar se a estratégia proposta se trata de uma melhoria da técnica em questão. Desse modo, os experimentos utilizarão o *benchmark suite* cBench [Fursin and Temam 2010], tendo como *baseline* o nível de otimização `-O3`, os mesmos utilizados na avaliação experimental do referido trabalho.

A avaliação dos resultados deve levar em conta que o atual programa de entrada não deve ser conhecido pelo *software* otimizador, e que este deve melhorar as sequências encontradas ao longo do tempo, já que há aprendizagem acumulada a cada execução do sistema. Dessa forma, serão realizados experimentos com os *benchmarks* do cBench como entrada ao longo de uma simulação da vida do *software* otimizador.

Cada *benchmark* deverá ser avaliado N vezes no *software* otimizador. Em cada experimento, o número de programas diferentes já executados pelo sistema terá variação de 1 até N, assim pode-se simular a vida útil do *software* otimizador, e obter a confirmação de que está havendo aprendizagem ao longo das execuções.

6. Estado Atual do Trabalho

O trabalho se divide nas seguintes etapas:

1. Revisão da literatura da área de seleção e ordenação de otimizações, a fim de aprofundar a argumentação teórica da proposta do trabalho.
2. Implementação da aprendizagem contínua de longo prazo como proposto por [Tartara and Reghizzi 2013].
3. Implementação de uma variante do algoritmo proposto por Wu e Larus (1994).
4. Implementação do algoritmo proposto.
5. Avaliação experimental.
6. Escrita da dissertação para defesa de mestrado.

A atividade 1 está sendo desenvolvida durante todo o tempo do trabalho de mestrado, enquanto as atividades 2 e 3 estão em fase de conclusão.

Referências

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools*. Prentice Hall, Boston, MA, USA, 2 edition.

Fursin, G. and Temam, O. (2010). Collective optimization: A practical collaborative approach. *ACM Transactions on Architecture and Code Optimization (TACO)*, 7(4):20:1–20:29.

Lattner, C. and Adve, V. (2004). Llvm: a compilation framework for lifelong program analysis transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86.

Martins, L. G., Nobre, R., Delbem, A. C., Marques, E., and Cardoso, J. a. M. (2014). Exploration of compiler optimization sequences using clustering-based selection. *ACM SIGPLAN Notices - LCTES '14*, 49(5):63–72.

Needleman, S. B. and Wunsch, C. D. (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443 – 453.

Purini, S. and Jain, L. (2013). Finding good optimization sequences covering program space. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):56:1–56:23.

Stallman, R. M. and DeveloperCommunity, G. (2009). *Using The Gnu Compiler Collection: A Gnu Manual For Gcc Version 4.3.3*. CreateSpace, Paramount, CA.

Tartara, M. and Reghizzi, S. C. (2013). Continuous learning of compiler heuristics. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):46:1–46:25.

Wu, Y. and Larus, J. R. (1994). Static branch frequency and program profile analysis. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, MICRO 27, pages 1–11, New York, NY, USA. ACM.

Automatização de oráculos de teste para o processamento de imagens médicas de modelos tridimensionais

Misael C. Júnior e Márcio E. Delamaro (Orientador)

Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional (PPGCCMC)
Instituto de Ciências Matemáticas e de Computação – Universidade de São Paulo (USP)
Avenida Trabalhador São-Carlense, 400 - Centro CEP: 13566-590 - São Carlos - SP

misaeljr@usp.br, delamaro@icmc.usp.br

Nível: Mestrado
Ano de ingresso no programa: 03/2015
Época prevista de conclusão: 03/2017
Data da aprovação da proposta de dissertação: 26/04/2016
Eventos relacionados: SBES

Resumo: Oráculos de teste determinam se uma execução de um SUT (do inglês *System Under Test*) está correta ou não. Em um cenário de teste automatizado, oráculos de teste são mecanismos essenciais para a produtividade e eficácia dos testes. Entretanto, dependendo do domínio de saída do SUT, a automatização dos oráculos pode se tornar um desafio. Dependendo da natureza dos dados produzidos pelo sistema, o SUT é conhecido como sistema de saída complexa. Sistemas com saídas gráficas/áudio, objetos tridimensionais e aplicações Web são exemplos de sistemas com saídas complexas. O presente trabalho de mestrado visa à configuração e à avaliação de oráculos automatizados de teste para sistemas cujas saídas consistem em imagens médicas sintéticas tridimensionais. Para alcançar tal objetivo, será explorado o *framework* O-FIm/CO (do inglês *Oracle for Images and Complex Outputs*) que utiliza conceitos de Recuperação de Imagem Baseada em Conteúdo (do inglês *Content-Based Image Retrieval* - CBIR). Além de adaptações e extensões do *framework*, será desenvolvido um catálogo de *plug-ins* que representem extratores de características de imagens médicas tridimensionais de vasos sanguíneos. Resultados de trabalhos anteriores mostram que tal técnica contribui para o aumento da produtividade do teste, mitigando e complementando os esforços manuais. Por meio de experimentos com imagens tridimensionais produzidas por sistemas reais, espera-se medir as vantagens e desvantagens da técnica e contribuir para a redução de tempo e esforços gerados por abordagens manuais (oráculo humano) durante a avaliação da qualidade de sistemas geradores de imagens médicas tridimensionais.

Palavras-chave: teste de software, oráculos de teste, angiografia, redes vasculares sintéticas tridimensionais.

1. Caracterização do Problema

O teste de software é um conjunto de atividades dinâmicas que consistem na execução do programa em teste com algumas entradas específicas, visando a verificar se seu comportamento é condizente com sua especificação [Delamaro et al. 2007]. Dentro do contexto de teste, oráculos são os mecanismos utilizados para julgar a correção da saída ou o comportamento de uma execução de um determinado programa [Oliveira et al. 2009].

Um problema recorrente associado à automatização de oráculos de teste ocorre quando a saída do SUT (do inglês *System Under Test*) configura um domínio complexo de dados [Oliveira et al. 2014]. Nesse cenário, a alta complexidade das saídas produzidas pelo SUT dificulta a aplicação das atividades de teste [Oliveira et al. 2014]. Sistemas com saídas gráficas/áudio, objetos tridimensionais, interfaces gráficas com o usuário e alguns aplicativos da Web são exemplos contemporâneos de sistemas com saídas complexas.

Este trabalho visa a cooperar com a área de ES (Engenharia de Software), mais precisamente, com a área de Teste de Software, por meio da configuração e avaliação de oráculos automatizados de teste para sistemas cujas saídas consistem em imagens médicas sintéticas tridimensionais. Tais imagens são fundamentais no auxílio à área médica, contribuindo na antecipação do diagnóstico de determinadas patologias [Galarreta-Valverde et al. 2013]. Devido a problemas recorrentes de falta de qualidade em sistemas que geram ou manipulam imagens sintéticas tridimensionais, em consequência da falta de estratégias de teste automatizado durante o seu processo de desenvolvimento, considera-se necessária aplicações específicas para o suporte adequado ao teste de sistemas dessa natureza [Galarreta-Valverde et al. 2013].

Para tanto, será explorado o *framework* O-FIm/CO (do inglês *Oracle for Images and Complex Outputs*), ferramenta que apoia o teste de programas com processamento gráfico por meio da automatização de mecanismos de oráculo. A estratégia vem sendo explorada para apoiar o teste em imagens médicas bidimensionais como, por exemplo, esquemas CAD (do inglês *Computer Aided Design*) de imagens mamográficas [Delamaro et al. 2013]. No entanto, imagens bidimensionais são limitadas e impossibilitam uma análise minusiosa da imagem e com maior precisão. Dessa forma, considera-se necessária a adaptação do *framework* O-FIm/CO para imagens médicas tridimensionais como, por exemplo, imagens sintéticas tridimensionais de vasos sanguíneos.

2. Fundamentação Teórica

Nesta seção são apresentados conceitos relevantes para o entendimento do presente projeto de mestrado em andamento. Dessa forma, são abordados conceitos relacionados à oráculos de teste, CBIR, do *framework* O-FIm/CO e de Redes vasculares tridimensionais.

2.1. Oráculos de Teste e CBIR

O mecanismo que se utiliza para julgar a correção da saída ou o comportamento esperado da execução de um determinado programa é conhecido como "oráculo" [Hoffman 2001]. Uma particularidade dos testes automatizados e manuais é a necessidade de oráculos de teste. Diferentes recursos técnicos podem ser explorados para a automatização de oráculos como: (1) modelos formais executáveis; (2) versões anteriores confiáveis do SUT; (3) documentações UML (do inglês *Unified Modeling Language*); (4) relações metamórficas; (5) saídas esperadas de casos de teste previamente computados; e (6) o próprio testador verificando as saídas do SUT, configurando um *oráculo humano*.

O Problema de oráculo (do inglês *Oracle problem*) é configurado em casos nos quais, utilizando meios práticos, é impossível ou muito difícil julgar a correção de saídas geradas [Weyuker 1982]. Dependendo do oráculo, os seguintes casos podem ocorrer:

- **Falsos positivos:** o resultado do oráculo de teste é considerado inválido (*fail*), entretanto o sistema funciona de acordo com suas especificações; e
- **Falsos negativos:** o resultado do oráculo de teste é considerado válido (*pass*), entretanto o sistema não funciona de acordo com suas especificações.

Neste projeto de mestrado procura-se aproveitar conceitos de Recuperação de Imagens Baseada em Conteúdo (CBIR – do inglês *Content-Based Image Retrieval*) para auxiliar na atividade de teste, permitindo a automatização de oráculos para sistemas que produzam saídas complexas como, por exemplo, uma imagem sintética tridimensional. CBIR é uma técnica oriunda da área de Processamento de Imagem e Reconhecimento de Padrões que permite que um sistema recupere um conjunto de imagens de uma base, tendo como chave uma imagem de referência [Datta et al. 2008]. Com o grande aumento na geração de imagens, vêm-se desenvolvendo novas metodologias para busca de imagens. Assim, para se obter as imagens de interesse são usadas características inerentes tais como cor, textura e forma.

2.2. *Framework* O-FIm/CO

O *framework* O-FIm/CO[1] emprega conceitos de CBIR para configurar oráculos para apoiar testes em sistemas com saídas gráficas ou complexas como, por exemplo, uma imagem médica processada ou uma GUI (do inglês *Graphical User Interface*). O O-FIm/CO apoia à comparação de objetos por meio da similaridade, particular a cada objeto, e automatiza a função de oráculos para domínios complexos de modo flexível e replicável. Na prática, os domínios de saída suportados pelo O-FIm/CO têm sido avaliados por oráculos humanos por meio de suas capacidades sensoriais como, por exemplo, visão e audição [Delamaro et al. 2013, Oliveira et al. 2014].

A estrutura genérica do O-FIm/CO permite ao testador interagir com o núcleo do *framework* diretamente, por meio de comandos específicos, ou indiretamente por meio de um *Wizard*. O núcleo tem a tarefa de reconhecer e instalar *plug-ins* (extratores ou funções de similaridade) para uso posterior na composição de oráculos. Dessa forma, por intermédio da utilização do O-FIm/CO, testadores obtêm um programa escrito em Java capaz de julgar se saídas/objetos complexos (imagens, sons, etc) produzidos pelo SUT são similares ou não, de acordo com as características e restrições inseridas diante do contexto do trabalho.

A Figura 1 ilustra um exemplo de definição de um oráculo gráfico para o *framework* O-FIm/CO que é traduzido por um *parser* e indica ao *framework* como um oráculo gráfico deve realizar uma comparação durante a execução de determinado teste. Um oráculo é criado usando dois extratores, **MyExtractor** e **OurExtractor**. O primeiro extrator possui uma propriedade chamada "color" que será definida com o valor (uma String) "red" e uma propriedade "alpha" cujo valor é um inteiro longo, 78. O segundo possui uma propriedade "scale" cujo valor será inicializado com o valor double, 1.33. Ambos possuem uma propriedade chamada "rectangle" para a qual um vetor de inteiros deve ser usado.

[1]Link para o projeto: http://ccsl.icmc.usp.br/pt-br/projects/o-fim-oracle-images

```
--------------------------------------------------------------------------------
similarity Euclidean

extractor MyExtractor { color = "red" alpha = 78 rectangle = [100 100 30 40] }
extractor OurExtractor { rectangle = [0 0 128 64] scale = 1.33 }

precision = 0.46
--------------------------------------------------------------------------------
```

Figura 1. Exemplo de um oráculo gráfico para o *framework* O-FIm/CO (Figura adaptada de [Oliveira 2012])

2.3. Redes Vasculares Tridimensionais

Sistemas que manipulem imagens sintéticas tridimensionais de redes vasculares são fundamentais no auxilio à área médica contribuindo na antecipação do diagnóstico de determinadas patologias [Galarreta-Valverde et al. 2013]. Dessa forma, é essencial que as imagens processadas tenham alto grau de confiabilidade, fornecendo detalhes da imagem claros facilitando o trabalho do profissional da área médica. No entanto, sistemas que geram ou manipulam imagens sintéticas tridimensionais têm problemas recorrentes de qualidade devido à falta de estratégias de teste automatizado. Assim, é essencial o uso de ferramentas automatizadas que auxiliem o profissional de saúde no diagnóstico de possíveis anomalias ou patologias [Galarreta-Valverde et al. 2013].

A Figura 2 apresenta uma série de imagens de quatro vasos sintéticos tridimensionais gerados pela estratégia definida em [Galarreta-Valverde et al. 2013]. As imagens são apresentadas primeiramente de modo completo (parte superior da Figura 2) e depois de modo segmentado – sem o fundo e somente com o "mapa" vascular representado (parte inferior da Figura 2).

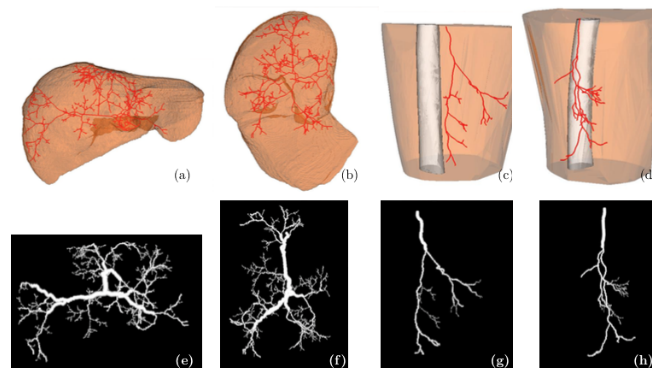


Figura 2. Vasos sintéticos tridimensionais – antes e depois da segmentação (Figura adaptada de [Galarreta-Valverde et al. 2013])

3. Contribuições

O escopo deste projeto é a automatização de oráculos de teste para sistemas que manipulem imagens sintéticas tridimensionais de angio-RM (Angiografia por Ressonância Magnética) ou por angio-TC (Tomografia Computadorizada), permitindo uma análise minuciosa das redes vasculares [Galarreta-Valverde et al. 2013].

Dessa forma, espera-se que os estudos realizados contribuam para estabelecer métodos de avaliação da qualidade de sistemas cujas saídas são imagens médicas tri-

dimensionais. A estratégia deve contribuir para aliviar os esforços humanos em testar sistemas de tal natureza, contribuindo para a melhora na qualidade de algoritmos e sistemas que processem imagens tridimensionais de redes vasculares reais ou sintéticas.

Em um contexto amplo, os resultados desse trabalho podem contribuir para sua ampla utilização em sistemas de auxílio ao diagnóstico médico, haja vista que a confiança nesses sistemas é severamente limitada por questões de qualidade. Dentre outras contribuições relacionadas ao projeto, espera-se: (1) disponibilizar um catálogo de extratores sob licenças de software livre; (2) estabelecer um arcabouço para que novos sistemas tridimensionais sejam inseridos no *framework* O-FIm/CO; e (3) incentivar o desenvolvimento de *frameworks* que auxiliem em atividades de validação em sistemas com saídas complexas.

4. Estado Atual do Trabalho

Para se obter uma noção global do estágio atual do trabalho é necessário entender as sete etapas nas quais o trabalho foi dividido:

1. **Mapeamento sistemático (MS):** pesquisa bibliográfica visando a identificar estudos empíricos acerca de atividades de teste e de avaliações da qualidade de sistemas biomédicos;
2. **Implementação de extratores:** esta etapa visa à identificação de características que possam ser extraídas das imagens tridimensionais. Tais características serão base para definição do conjunto de extratores a serem implementados;
3. **Estudo e implementação de extensões no O-FIm/CO:** desenvolver possíveis adaptações que podem ser realizadas no O-FIm/CO para que oráculos de teste de saídas que sejam do tipo de imagens tridimensionais possam ser gerados;
4. **Implementação de oráculos de teste:** atividades técnicas envolvendo o *framework* O-FIm/CO, extratores de características para imagens tridimensionais e funções de similaridade. Ajuste e parametrização de extratores de características para diferentes modelos tridimensionais configuram o objetivo principal dessa atividade, além da escolha de modelos e SUTs a serem explorados;
5. **Definição e condução de estudos experimentais:** condução de estudos empíricos com SUTs reais como, por exemplo, o apresentado em [Galarreta-Valverde et al. 2013]. Além disso, pretende-se identificar novos sistemas *open-souce* (acadêmicos ou industriais) para realização de estudos de caso;
6. **Escrita de artigos científicos:** serão escritos artigos científicos visando à divulgação adequada dos resultados; e
7. **Escrita do documento de dissertação:** escrita do documento final de dissertação.

A partir da realização do MS (Mapeamento Sistemático), foi possível identificar características que possam ser extraídas por meio do desenvolvimento dos extratores de densidade, bifurcação e de segmentos. Tais extratores foram validados em um conjunto de imagens sintéticas de vasos sanguíneos. Em seguida, espera-se a identificação de outras características que possam viabilizar o desenvolvimento de novos extratores a serem inseridos no contexto do *framework* O-FIm/CO. Atualmente, o trabalho está na fase de implementação de extratores e estudo e implementação de extensões no O-FIm/CO.

5. Descrição e Avaliação dos Resultados

No contexto do presente trabalho, um MS foi realizado objetivando identificar pesquisas relacionadas a abordagens e estratégias de teste de software e avaliação de qualidade em sistemas biomédicos. Como uma estratégia de seleção e busca de estudos, duas Questões de Pesquisa (QP) foram definidas. A primeira QP tem como finalidade sumarizar estudos que apresentam abordagens informais ou *ad-hoc* e sistemáticas ou automatizadas de avaliação de qualidade ou teste de software. A segunda QP objetivou a identificação de procedimentos experimentais utilizados na validação de sistemas biomédicos. Nesse cenário, 83 estudos foram selecionados a partir das QPs definidas.

Dessa forma, foram identificadas características que possam ser extraídas pelos extratores de: (1) densidade; (2) bifurcação; e (3) segmentos. O extrator de densidade fornece um indicador de quanto o vaso sanguíneo preenche a imagem como um todo. O extrator de bifurcação identifica quantas divisões, em dois ramos, há no vaso sanguíneo. Por sua vez, o extrator de segmentos identifica quantas regiões podem ser identificadas no vaso sanguíneo. Tais extratores serão inseridos no contexto do *framework* O-FIm/CO, podendo ser utilizados como fonte de informação para os oráculos. Além disso, as informações fornecidas pelos extratores serão utilizadas no contexto do *framework* O-FIm/CO em diversas imagens, como imagens modelo e imagens do SUT.

Além dos extratores já citados, outros extratores foram identificados como, por exemplo: (1) tortuosidade; (2) diâmetro de vasos; (3) ângulos de bifurcação; (4) curva média dos contornos; (5) curvaturas mínimas dos contornos; e (6) curvaturas máximas dos contornos. Importante ressaltar que alguns dos extratores citados já foram desenvolvidos, sendo necessária apenas a adaptação ao contexto do *framework* O-FIm/CO.

Após a inclusão dos extratores no *framework* O-FIm/CO e implementação dos oráculos de teste, pretende-se avaliar: (i) a capacidade de detecção de falhas/imperfeições dos oráculos automatizados; (ii) o número de falsos positivos e falsos negativos dos oráculos; (iii) o número de verdadeiros positivos e verdadeiros negativos dos oráculos; e (iv) o custo-benefício e *tradeoffs* do uso da técnica de oráculos automatizados.

6. Comparação com Trabalhos Relacionados

De um modo geral, o MS evidenciou a carência de abordagens de validação automatizadas em sistemas biomédicos que gerem ou manipulem imagens, evidenciando o ineditismo do projeto. Por sua vez, 25% (21/83) dos estudos apresentam abordagens para sistemas que geram ou processam imagens médicas como, por exemplo, exames de ultra-sonografia, imagens radiológicas, imagens radiográficas, etc. Tal predominância, demonstra um grande interesse da literatura na validação de tais sistemas. Além disso, 85,7% (18/21) dos estudos sobre imagens discutem a complexidade do processo de validação nesse domínio de sistema, demonstrando a carência de processos que validem as saídas produzidas.

Trabalhos como Filho et al. (2014) e Gibson et al. (2001) avaliam a qualidade de imagens médicas por meio de abordagens automáticas, demonstrando que tais abordagens são mais eficientes que o uso de métodos manuais [Filho et al. 2014, Gibson et al. 2001]. Nesse cenário, Delamaro et al. (2013) demonstra que a utilização de oráculos automatizados pode apontar falhas em imagens de referência, permitindo a sua melhoria.

Diante de um problema recorrente associado à automatização de testes e de

oráculos, o presente trabalho procura aproveitar conceitos de CBIR para automatizar atividades de teste de sistemas cujas saídas são imagens médicas de modelos tridimensionais.

Referências

Datta, R., Joshi, D., Li, J., and Wang, J. Z. (2008). Image Retrieval: Ideas, Influences, and Trends of the New Age. *CSUR 2008*, pages 1–60.

Delamaro, M. E., Maldonado, J. C., and Jino, M. (2007). *Introdução ao teste de software*. Elsevier.

Delamaro, M. E., Nunes, F. L. S., and Oliveira, R. A. P. (2013). Using concepts of content-based image retrieval to implement graphical testing oracles. *STVR 2013*, pages 171–198.

Filho, A. C. S. S., Rodrigues, E. P., Junior, J. E., and Carneiro, A. A. O. (2014). A computational tool as support in b-mode ultrasound diagnostic quality control. *Revista Brasileira de Engenharia Biomédica*, pages 402–405.

Galarreta-Valverde, M. A., Macedo, M. M. G., Mekkaoui, C., and Jackowski., M. P. (2013). Three-dimensional synthetic blood vessel generation using stochastic L-systems. In *MIC 2013*, pages 86691I–86691I–6.

Gibson, N. M., Dudley, N. J., and Griffith, K. (2001). A computerised quality control testing system for b-mode ultrasound. *Ultrasound in medicine & biology*, pages 1697–1711.

Hoffman, D. (2001). Using oracles in test automation. *PNSQC 2001*, pages 90–117.

Oliveira, R. A. P. (2012). Apoio à automatização de oráculos de teste para programas com interfaces gráficas. Master's thesis, Instituto de Ciências Matemáticas e de Computação (ICMC) – Universidade de São Paulo (USP), São Carlos/SP.

Oliveira, R. A. P., Delamaro, M. E., and Nunes, F. L. S. (2009). O-FIm – oracle for images. In *SBES 2009*, pages 1–6.

Oliveira, R. A. P., Gil, V. N., Nunes, F. L. S., and Delamaro, M. E. (2014). An extensible framework to implement test oracle for “non-testable programs”. In *SEKE 2014*, pages 199–204.

Weyuker, E. J. (1982). On testing non-testable programs. *The Computer Journal*, pages 465–470.

Detecção de Variabilidades e Comunalidades em uma Família de Produtos de Software

Denise Alves da Costa e Pedro de Alcântara dos Santos Neto (Orientador)

Programa de Pós-Graduação em Ciência da Computação (PPGCC)
Departamento de Computação
Universidade Federal do Piauí (UFPI)
CEP: 64.049-550 – Teresina – PI – Brasil
denise.alvesss@gmail.com
pasn@ufpi.edu.br

Nível: Mestrado

Ano de ingresso no Programa: 03/2015
Previsão de conclusão: 03/2017
Data da aprovação da proposta de dissertação: prevista para Agosto/2016
Sigla do Evento Relacionado: SBES

Resumo: Linha de Produto de Software (LPS) é uma nova forma de construção de produtos de Software que tem chamado grande atenção de muitas empresas. Embora sejam diversas as vantagens de se utilizar uma LPS, também muitos são os desafios para a sua adoção. Empresas que já possuem diversas variantes de um produto, customizadas para clientes específicos, possuem dificuldade em adotar LPS, devido aos custos envolvidos. Este trabalho propõe um método para apoiar as empresas nesse processo de transição, partindo de vários produtos similares para uma linha de produto, por meio da detecção de comunalidades e variabilidades existentes nos produtos que fazem parte do seu portfólio.

Palavras-chave*: Linha de Produto de Software, Abordagem Extrativa, Detecção de Variabilidades.*

1. Introdução

Empresas de software desenvolvem produtos para um determinado nicho de mercado, como sistemas acadêmicos, financeiros ou administrativos. Dentro desse nicho, apesar de serem semelhantes, os produtos normalmente possuem funcionalidades específicas para cada cliente.

A fim de evitar construir aplicações a partir do zero a cada novo cliente, geralmente as fábricas adotam o reuso de artefatos nos grupos de clientes com demandas semelhantes. Uma das práticas de reuso mais utilizadas consiste em copiar e colar unidades de software [Sommerville 2011]. No entanto, o aumento no número de produtos torna a manutenção cada vez mais custosa para a fábrica de software. Dessa forma, são necessárias metodologias e ferramentas apropriadas que tornem mais eficiente o gerenciamento desses produtos. Nesse sentido, profissionais da área vêm buscando na Linha de Produto de Software (LPS), uma solução efetiva para o problema [Pohl et al. 2005].

Este trabalho de mestrado visa propor um método para auxiliar as empresas no processo de implantação de uma LPS. A Seção 2 apresenta alguns conceitos relacionados a LPS necessários para o entendimento do trabalho. Na Seção 3 é apresentado o problema a ser abordado. A Seção 4 apresenta a solução proposta por este trabalho. A Seção 5 relaciona os principais trabalhos encontrados relacionados ao tema. A Seção 6 descreve o estado atual da pesquisa. E por fim, na Seção 7 é discutido acerca da avaliação do método.

2. Fundamentação Teórica

LPS é um paradigma de desenvolvimento de sistemas no qual para um grupo de softwares semelhantes (chamado de família), existe uma plataforma comum para reuso (artefatos em comum) e componentes variáveis de acordo com a necessidade de cada cliente [Pohl et al. 2005]. A adoção da LPS em contraste com o paradigma tradicional (um projeto diferente a cada novo cliente), traz diversas vantagens para a fábrica de software, como a eliminação de códigos duplicados e menores custos com manutenção.

Existem três abordagens de implantação de uma LPS em uma fábrica de software: a pró-ativa, a reativa e a extrativa [Krueger 2001]. Na pró-ativa, a LPS é planejada e construída para cobrir todo o escopo dos produtos que farão parte dela. Na abordagem reativa, a LPS se inicia com um, ou alguns produtos e cresce gradativamente à medida que novas aplicações são requisitadas. E por fim, na abordagem extrativa, explorada neste trabalho, a LPS é construída a partir de vários produtos pré-existentes.

Entretanto, a adoção da LPS nas empresas de desenvolvimento pode ser custosa devido ao tempo e ao trabalho envolvidos no processo de implantação [Dura and Yilmaz 2009]. Pesquisadores, por sua vez, investigam métodos para auxiliar essas empresas nesse processo. Da mesma forma, os autores deste trabalho também buscam uma forma de apoiar adoção de LPS na indústria.

3. Caracterização do Problema

É muito comum encontrar empresas de software que possuem diversos clientes com produtos semelhantes, mas com distinção em partes dos seus componentes, decorrente das diferentes necessidades de seus clientes. Nesse contexto, o uso de LPS poderá ser iniciado

a partir da abordagem extrativa, que prevê a criação da LPS a partir da identificação das comunalidades e variabilidades existentes entre os produtos da linha. [Clements 2002].

Para a extração da linha a partir das aplicações já existentes é necessário executar as tarefas relacionadas a seguir além daquelas próprias da construção de qualquer LPS:

- Identificar as comunalidades e as variabilidades dos sistemas. Ou seja, é necessário identificar os artefatos que pertencem a todas as aplicações e os artefatos que estarão presentes em apenas uma parte das aplicações.
- Levantamento de *features*. Todas as funcionalidades pertencentes a cada um dos sistemas devem ser identificadas.
- Mapeamento de *features*. Associação entre os artefatos e as *features* identificadas.

Tais atividades demandam grande consumo de tempo alto esforço em suas execuções [Krueger and Jungman 2009]. Nos últimos anos, isso levou muitos pesquisadores da engenharia de software a se voltarem a desenvolver pesquisas em LPS para impulsionar a adoção na indústria de software. Existem diversas ferramentas que oferecem suporte ao gerenciamento da LPS [de Lima Júnior 2008], no entanto ainda são poucas as técnicas e ferramentas para apoiar o processo de transição dos sistemas únicos para a linha, objeto de estudo deste trabalho.

4. Contribuições

Este trabalho de mestrado tem como objetivo apoiar a criação de uma LPS via abordagem extrativa. Como objetivo inicial, este trabalho se propõe a gerar o que denominamos de Estrutura de Gerenciamento de Variabilidades (EGV). Essa estrutura permitirá a identificação das partes comuns e variáveis entre os produtos, de forma a se manter todo o código dos mesmos em uma estrutura única, além de permitir a manutenção do código utilizando os conceitos de LPS e assim facilitando seu desenvolvimento. Essa estrutura também possibilitará a geração de produtos com diversas configurações.

O método proposto segue os passos mostrados na Figura 1. No primeiro passo, deverão ser definidos todos os sistemas que farão parte da futura LPS. Para o segundo passo, uma ferramenta deverá encontrar de maneira automatizada as variabilidades e as comunalidades entre os sistemas da família. Isso ocorrerá apenas a nível de código, não de *features*; essas poderão ser posteriormente mapeadas às porções de código detectadas. Nesse passo também será feita a associação das variabilidades encontradas com cada um dos produtos analisados. Por fim, no terceiro passo é construída a EGV.

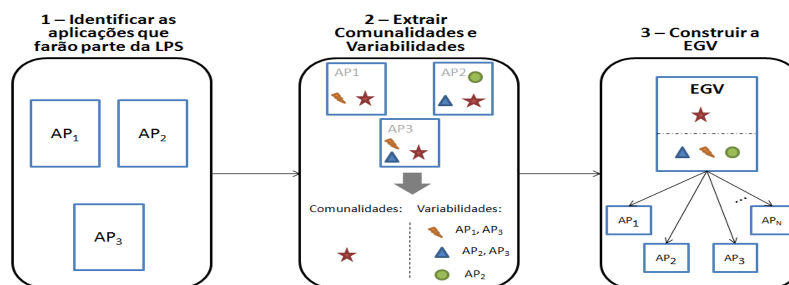


Figura 1. Visão geral do método proposto.

As variabilidades de que trata este trabalho podem ser desde pacotes, até blocos de inicializações de variáveis. Conceitualmente, a análise é feita em uma estrutura de árvore,

na qual o conjunto de nós é formado hierarquicamente por pacotes, classes, atributos, métodos, estruturas de controle e até simples comandos.

Considerando essa estrutura, a comparação entre as aplicações se inicia do nível mais alto e vai em direção ao mais baixo da árvore. Se em um nível N existe um elemento que não se repete em todas as demais aplicações, no mesmo nível, então ele é considerado uma variabilidade e a análise não precisa descer nesse ramo. Por outro lado, se um elemento do nível N se repete no mesmo nível em todas as aplicações, com os mesmos antepassados, ele é considerado uma comunalidade. Neste caso, sabe-se que o nó é comum, mas ainda nada pode ser dito a respeito de seu conteúdo, então é necessário continuar a análise descendo na árvore e comparar os seus elementos filhos. Se alguma diferença for encontrada, uma nova variabilidade será identificada. Ao mesmo tempo em que o método encontra as variabilidades, ele deverá também identificar a quais aplicações essa variabilidade pertence. Todas essas informações deverão ser mantidas dentro da EGV.

Para classificar um pacote ou uma classe, apenas seu nome e seus antepassados na árvore são considerados, não o seu interior. Caso o elemento exista em todas os produtos, mas o seu conteúdo seja diferente em cada um desses, o mesmo ainda será classificado como uma comunalidade, e os seus filhos serão classificados como variabilidades (aqueles que não forem comuns). De maneira similar, um conjunto de métodos será considerado uma comunalidade se eles tiverem a mesma assinatura e pertencerem à mesma classe e ao mesmo pacote nas diferentes aplicações, nesse momento o seu corpo não será considerado. Caso o método possua comandos diferentes, esses serão posteriormente classificados como variabilidades.

Com base nessas informações, no terceiro passo é criada uma infraestrutura onde a equipe poderá: visualizar as diferenças entre os produtos e as porções de código comum; mapear *features*; fazer alterações no código e gerar produtos de software. Tais atividades deverão ser viabilizadas por meio de ferramenta apropriada, para gerenciamento da EGV.

Para empresas com software de médio a grande porte, entende-se que o passo 2 do método é bastante oneroso se feito de forma manual, sem o auxílio de uma ferramenta que automatize o processo. Então, com o método e a ferramenta propostos neste trabalho, esperamos que a fábrica poupe esforços nesse processo de transição para LPS. Para fins de ilustração, a seguir é apresentado um exemplo de aplicação do método proposto.

Suponha que uma empresa desenvolvedora de soluções para Universidades pretenda adotar os conceitos de LPS para facilitar seu trabalho. O primeiro passo é então identificar as aplicações já existentes que deverão fazer parte da linha. A Figura 2 mostra a estrutura (pacotes e classes) das respectivas aplicações.

O próximo passo é descobrir as comunalidades e variabilidades das aplicações. Para isso, inicia-se uma comparação no nível 0, no qual observa-se que todas as raízes são iguais, então deve-se continuar a busca, a fim de saber se o restante da árvore é também igual, ou se possui variabilidades. A comparação prossegue então para o nível 1. Nesse exemplo verifica-se que todas as aplicações possuem os pacotes *autenticacao* e *administracao*, logo esses pacotes são classificados como comuns. No entanto, a verificação deverá continuar nos níveis inferiores a fim de verificar se seus conteúdos também são comuns. No nível 2, verifica-se que existem três variabilidades abaixo do pacote *autenticacao*: *Usuario.java*, *Login.java* e *Biometria.java*, uma em cada projeto. Prosseguindo

no mesmo nível, encontra-se o pacote *academico* que também é uma comunalidade e o pacote *patrimonial*, que é comum a apenas duas aplicações, sendo portanto uma variabilidade. Por fim, no nível 3 são encontradas duas classes comuns a todas as aplicações (*Aluno.java* e *Professor.java*) e outra (*Tombo.java*) pertencente à variabilidade detectada no nível anterior (*patrimonial*). À medida que análise avança, as variabilidades são mapeadas às suas respectivas aplicações: autenticacao.Usuario.java - {Universidade1}; autenticacao.Login.java - {Universidade2}; autenticacao.Biometria.java - {Universidade3}; administracao.patrimonial - {Universidade2, Universidade3}.

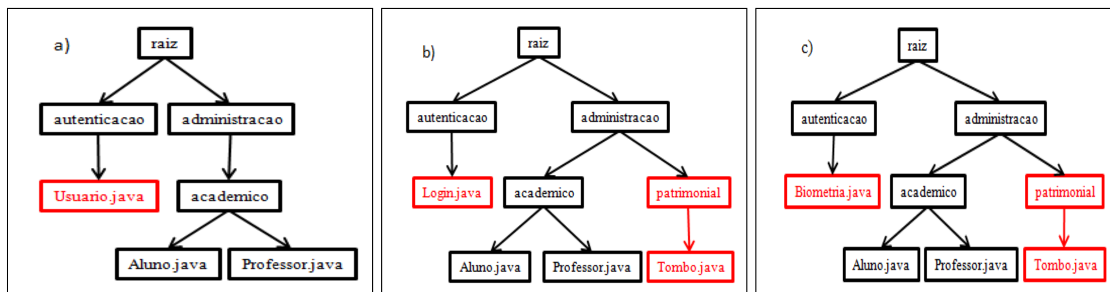


Figura 2. Exemplo de aplicações que devem formar uma LPS: a) Universidade1, b) Universidade2, c) Universidade3.

A abordagem escolhida neste trabalho aplica-se a um contexto no qual um sistema foi inicialmente construído e outros foram criados a partir de uma cópia dele (ou de outra cópia) seguida de modificações, ou adaptações, permitindo assim que o método encontre as comunalidades. No entanto, para sistemas onde as estruturas de pacotes, classes ou mesmo os métodos e atributos tenham sido renomeados, o método não se aplica.

5. Comparação com Trabalhos relacionados

Assim como este, o trabalho de [Nunes et al. 2012] também busca por similaridades e variações em produtos membros da mesma família de software, escritos na linguagem Java. Além das comparações entre as aplicações variantes, também são comparadas as diferentes versões de cada uma, utilizando heurísticas sensíveis a história. O método recebe como entrada os mapeamentos entre as *features* e o código de cada uma das aplicações e então detecta as *features* comuns e as opcionais. Diferentemente do trabalho de Nunes, o tratado neste artigo não considera versões anteriores dos sistemas, além de não exigir o mapeamento das *features* para identificar as variabilidades e comunalidades, isso poderá ser feito posteriormente com o auxílio da ferramenta de gerenciamento da EGV.

[Klatt et al. 2013] apresenta um método para identificar pontos de variação em produtos semelhantes para apoiar engenheiros na construção da LPS. Talvez a maior limitação desse trabalho seja o fato de que o método é aplicável apenas em produtos cujas variabilidades dos códigos sejam alternativas, ou seja, se um produto A possui uma variabilidade V, então o produto B tem necessariamente uma variante dessa mesma variabilidade V. O trabalho deste mestrado, por sua vez abrange também esse tipo de variabilidade, pois detecta qualquer variabilidade que exista no código-fonte.

Os autores em [Martinez et al. 2015] apresentam um conjunto de princípios para a construção de um framework com o objetivo de apoiar de ponta-a-ponta a adoção de LPS utilizando a abordagem extrativa. Eles foram estabelecidos para abranger o processo

de transição a partir de qualquer tipo de artefato, diferentemente deste trabalho, o qual analisa apenas código-fonte. Os autores criaram um framework para avaliar a proposta. Os resultados do experimento apresentaram uma boa eficácia em todo o processo.

Em [Ajila 2005] é apresentado um framework conceitual para a construção de uma LPS para uma família de software. O framework abrange a identificação das features em cada produto, a construção da arquitetura da linha de produto e até a derivação de produtos individuais a partir dessa arquitetura, construindo o modelo de features de acordo com a FODA (Feature-Oriented Domain Analysis). O framework proposto ainda não foi implementado, mas sua ferramenta é apresentada como trabalho futuro pelo autor.

Além de trabalhos empenhados na problemática da implantação, outros focam na detecção das comunalidades, chamadas de clones. [Baxter et al. 1998] em seu artigo apresentam um método para detectar clones em sistemas de software utilizando Árvore Sintática Abstrata. Diferentemente do trabalho aqui apresentado, os autores realizam uma análise bottom-up. Além de clones idênticos, o trabalho também investiga ”clones próximos”, que são trechos de código quase idênticos. A abordagem foi avaliada por meio de um experimento com 19 sistemas escritos em C. Dentre outras conclusões o experimento mostrou que os sistemas mais recentes eram os que possuíam maior números de clones e não foi encontrada correlação entre o tamanho dos softwares com a quantidade de clones encontrados.

6. Estado Atual do Trabalho

O trabalho encontra-se atualmente na fase de desenvolvimento da ferramenta para a detecção das comunalidades e variabilidades dos sistemas, específica para a linguagem Java. Hoje, a ferramenta já consegue identificar as semelhanças e diferenças entre as aplicações com a granularidade a nível de atributos e métodos; no entanto pretende-se afinar mais a granularidade, a nível de instruções ou estruturas de controle.

Uma outra linha do trabalho é o levantamento de ferramentas que fazem gerenciamento de LPS. Isso é necessário para a análise da melhor forma de se construir a EGV e sua ferramenta de gerenciamento. Uma empresa parceira possui intenção de utilizar os resultados do trabalho para melhorar sua estrutura de manutenção de produtos.

7. Avaliação dos Resultados

As contribuições e o desempenho efetivos do método proposto serão avaliados por meio de um estudo de caso que será realizado em uma empresa que desenvolve produtos de software para gestão em saúde.

Uma avaliação inicial já foi realizada no intuito de guiar este trabalho. Foi utilizado como base um Sistema para Gestão de Cooperativas Médicas, o qual possui três versões principais para todos os seus clientes, chamadas de Versão 1, 2 e 3. Os resultados são mostrados na Figura 3. Pelos resultados, foi possível observar que a versão 3 é mais destoante de todas e que existem mais variabilidades compartilhadas entre as versões 1 e 2 do que nos demais pares. Futuramente será analisada a acurácia dos dados encontrados.

Embora ainda esteja em seu início, já foi possível exibir informações não conhecidas pela equipe do produto. Além disso, o detalhamento dos objetivos deste trabalho foram apreciados pela empresa que demonstrou interesse nos resultados da pesquisa.

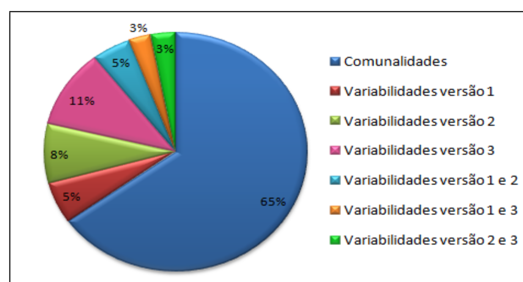


Figura 3. Comunalidades e variabilidades encontradas no sistema de Gestão de Cooperativas médicas, comparando-se as versões 1, 2 e 3.

Referências

Ajila, S. A. (2005). Reusing base product features to develop product line architecture. In *IRI -2005 IEEE International Conference on Information Reuse and Integration, Conf, 2005.*, pages 288–293.

Baxter, I. D., Yahin, A., Moura, L., Sant'Anna, M., and Bier, L. (1998). Clone detection using abstract syntax trees. In *ICSM - 1998. Proceedings of the International Conference on Software Maintenance*, pages 368–377. IEEE.

Clements, P. (2002). Being proactive pays off. *IEEE Software*, 19(4):28–30.

de Lima Júnior, R. A. (2008). Comparação entre ferramentas para linha de produtos de software.

Dura, Ö. and Yilmaz, A. E. (2009). Software product line development: A review on practical issues and challenges. In *Computer and Information Sciences, 2009. ISCIS 2009. 24th International Symposium on*, pages 736–742. IEEE.

Klatt, B., Küster, M., and Krogmann, K. (2013). A graph-based analysis concept to derive a variation point design from product copies. In *International Workshop on Reverse Variability Engineering*, pages 1–8.

Krueger, C. (2001). Easing the transition to software mass customization. In *Software Product-Family Engineering*, pages 282–293. Springer.

Krueger, C. W. and Jungman, M. N. (2009). Software customization system and method. US Patent 7,543,269.

Martinez, J., Ziadi, T., Bissyandé, T. F., Klein, J., and Le Traon, Y. (2015). Bottom-up adoption of software product lines: a generic and extensible approach. In *Proceedings of the 19th International Conference on Software Product Line*, pages 101–110. ACM.

Nunes, C., Garcia, A., Lucena, C., and Lee, J. (2012). History-sensitive heuristics for recovery of features in code of evolving program families. In *Proceedings of the 16th International Software Product Line Conference - Volume 1*, SPLC '12, pages 136–145, New York, NY, USA. ACM.

Pohl, K., Böckle, G., and Linden, F. J. v. d. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Sommerville, I. (2011). *Engenharia de Software*. Pearson Education do Brasil, 9 edition.

Detecting Code Anomalies in Software Product Lines

Eduardo Fernandes and Eduardo Figueiredo (Supervisor)

Postgraduate Program in Computer Science (PPGCC)
Software Engineering Laboratory (LabSoft) – Department of Computer Science (DCC)
Federal University of Minas Gerais (UFMG)
Minas Gerais – MG – Brazil

{eduardofernandes, figueiredo}@dcc.ufmg.br

Degree: Master's in Computer Science
Ingression Year: 2015
Expected Conclusion: February 2017
Dissertation proposal approved by post-graduate program committee in May 10, 2016
Related events: SBCARS and SBES

Abstract. *A Software Product Line (SPL) is a set of software systems that share common features designed to a specific domain. Although the design of SPL aims to support reuse with minimization of development costs, an inappropriate implementation of SPLs may lead to code anomalies. These code anomalies may affect negatively costs and management of SPL development. Few studies investigate anomalies in SPL with focus on detection strategies and automated detection of these anomalies. Moreover, recent studies indicate the need of proposing new anomalies and detection strategies, as the development of supporting detection tools. In this study, we investigate anomalies in SPL to (i) comprehend the state of the art on anomalies and detection strategies from literature, (ii) propose new detection strategies for well-known anomalies in SPL and, eventually, propose new anomalies and respective detection strategies, (iii) evaluate the proposed strategies, (iv) propose a supporting tool for the proposed detection strategies, and (v) evaluate the proposed tool. Preliminary results of the study indicate that anomalies in SPL is an interesting research topic with many opportunities for scientific contribution.*

Keywords: *software product lines, code anomalies, detection strategies, detection tools*

1. Introduction

A SPL is a set of software systems that share common features designed to a specific software domain [Pohl et al. 2005]. Each feature is an increment of functionality or property in a system [Batory 2005; Kästner et al. 2007]. There are two types of features in a SPL: commonalities and variabilities [Pohl and Metzger 2006b]. Commonalities are common features reused in the development of new software products of the SPL [Pohl et al. 2005]. In turn, variabilities are specific features that differentiate a product to other depending on the needs of a given client [Pohl and Metzger 2006b]. In general, we use feature models to represent the variability of a SPL [Kang et al. 1990]. A feature model represents the configurable products from a given a product line [Weiss 1999].

Code anomalies are symptoms of deeper quality problems in the system design or code [Fowler 1999]. For instance, we may consider duplicated code as a threat for software maintenance, since a change in one code part may require various changes in other code parts [Fowler 1999]. Even small software systems may contains several code anomalies [Macia et al. 2012]. As all software systems, artifacts of a SPL may contain code anomalies. However, to the best of our knowledge, few studies investigate the manifestation of anomalies in SPL [Apel et al. 2013]. Vale et al. (2014) discuss that new anomalies may be proposed, considering the inherent complexity of SPL design. Moreover, anomalies proposed in other software settings, such as the Fowler's code anomalies [Fowler 1999], may be applied in the SPL context.

A detection strategy is a quantifiable expression of a rule for identification of code fragments that follows that rule [Marinescu 2004]. In this context, we may apply software metrics in the composition of detection strategies [Lanza and Marinescu 2007]. Detection strategies may support the detection of code anomalies [Marinescu 2004], including in the development of automated tools [Fernandes et al., 2016].

This paper presents a proposal of master thesis in Computer Science. We aim to (i) to investigate anomalies in SPL, (ii) to propose detection strategies for anomalies in SPL, and (iii) to propose supporting tools for automated detection of these anomalies. For this purpose, we designed six study steps including literature review on detection of code anomalies, empirical investigation of detection strategies, definition of new detection strategies for well-known anomalies in SPL and, eventually, new anomalies and detection strategies for them, and the proposal and evaluation of a supporting tool. Our main goal is to provide more effective detection strategies for well-known code anomalies and, eventually, identify new anomalies not investigated in previous work.

In this paper, we also present contributions and preliminary results of this study. First, we conducted an *ad hoc* literature review [Oliveira et al. 2015] and a systematic literature review [Fernandes et al. 2016] on detection tools for code anomalies. Second, we performed a comparative study of tools in terms of recall, precision, and agreement. Third, we conducted a comparative study of methods for threshold derivation of software metrics as an extension of a previous study [Vale and Figueiredo 2015].

The remainder of this paper is organized as follows. Section 2 provides background information. Section 3 describes the study design, including the steps to guide our study. Section 4 presents preliminary results of the study. Section 5 discusses related work. Finally, Section 6 concludes the paper with ongoing and future work.

2. Background

This section provides background information to support the study comprehension. Section 2.1 presents software product lines (SPL) and variability. Section 2.2 discusses anomalies in source code and the occurrence of anomalies in SPL.

2.1. Software Product Lines

A SPL is a set of software systems that share common features designed to attend requirements from a specific market segment [Pohl and Metzger 2006a]. Each feature represents an increment of functionality or property in a system [Batory 2005; Kästner et al. 2007]. In general, we use a tree known as feature model [Kang et al. 1990] to represent variability of an SPL. This feature model represents all possibilities to configure a product from a given a product line [Weiss 1999].

Two types of features compose a SPL: commonalities and variabilities. Commonalities are common features reused in the development of new SPL products. In turn, variabilities are specific features that differentiate a product to other depending on the needs of each client [Pohl et al. 2005]. By using SPL to support software reuse, it is expected a minimization of development costs, in a medium-to-long term, increase the quality of software products, since the constant assessment of reused features, and a minimization of time-to-market [Pohl et al. 2005].

2.2. Code Anomalies

A code anomaly consists of any symptom that may indicate a deeper quality problem in the system design or code [Fowler 1999]. Code anomalies are an important factor affecting the quality and maintainability of a software system [Hall et al. 2014]. For instance, we may consider Duplicated Code as a threat for software maintenance tasks, since a change in one part of the code may require various changes in other parts all over the source code [Fowler 1999]. In turn, a class with many responsibilities that lacks internal relationships between its methods, i.e., a God Class, may hinder software maintenance and evolution [Lanza and Marinescu 2007].

Artifacts of an SPL may contain anomalies, as in software systems developed using other techniques. Vale at al. (2014) conducted a systematic literature review to identify the main types of anomalies investigated in the context of SPL. The authors found 70 different anomalies divided in three groups: (i) *code anomalies* that are fine-grained anomalies related to the source code; (ii) *architectural anomalies* that are coarse-grained anomalies in higher levels of abstraction; and (iii) *hybrid anomalies* that are similar to code anomalies with some architecture-level aspects.

There are two types of code anomaly detection: manual or automated analyses [Moha et al. 2010]. Considering automated approaches, we conducted a systematic literature review to identify tools that support detection of code anomalies [Fernandes et al., 2016]. In this study, we have found 29 tools that are available online for download. These tools aim to detect 61 different anomalies in programming languages, such as Java, C, and C++. However, after a comparative study, we observed low rates of recall for the tools, in contrast with high rates of precision. Because of that, we may observe an opportunity to propose new detection strategies that support the implementation of more precise and effective tools, for instance.

3. Master Thesis Planning

The main goals of our study are to (i) investigate the state of the art on code anomalies in SPL, (ii) propose new detection strategies for anomalies in SPL, and (iii) propose supporting tools for automated detection of these anomalies. For this purpose, we designed six steps of study as illustrated in Figure 1. These steps cover a period from March 2015, month to start the Master's activities, to February 2017, the expected month for completion of the study. Each step is described as follows.

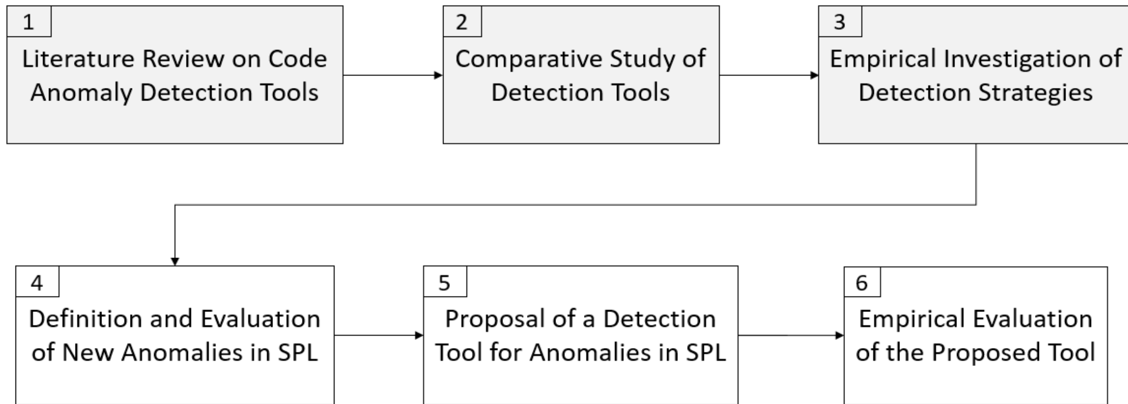


Figure 1. Study steps

In Step 1, we aim to comprehend the state of the art on methods and technologies to support the detection of source code anomalies in general; that is, we do not focus on SPLs at this point. This is an important step for an overall knowledge acquisition with respect to the studied research topic. We expect to understand how researches have been investigating code anomaly detection in any development context. After that, we then focus on the context of SPL development. In Step 2, we intend to compare the tools identified in Step 1 to observe advantages and drawback of the tools available in literature. We also expect to understand how developers provide the detection for users in terms of applicability of the tools (provided features, for instance).

In Step 3, we aim to investigate different detection strategies for code anomalies. Since many detection strategies have been proposed in the literature, we expect to understand different approaches for anomaly detection. In Step 4, we expect to define and evaluate new code anomalies in SPL and the respective detection strategies. Step 5 targets on the development of an automated tool to detect the new proposed anomalies. Finally, in Step 6 we propose an empirical evaluation of the tool to assess effectiveness on code anomaly detection in SPLs in terms of recall and precision of the tool.

4. Preliminary Results

In Figure 1, we colored Steps 1 to 3 in grey because they were completed until the submission date of this paper, i.e., finished until June 2016. Steps 4 to 6 are ongoing studies and we expect to conclude these steps until January 2017 for thesis defense in February 2017. We discuss the preliminary results as follows.

In Steps 1, we conducted an *ad hoc* literature review on detection tools for a specific code anomaly, Duplicated Code [Oliveira et al. 2015]. We obtained a set of 20 tools, 11 of them are available online for download. We also compared tools in terms of

applicability, such as features provided by the tools (data export, user interface, etc.). After, we conducted a systematic literature review on detection tools for any code anomaly [Fernandes et al. 2016]. We have found 84 different tools and 29 of them are available for download. We also conducted a comparative study of tools to compute recall, precision, and agreement. In general, our results suggest that the available tools provide redundant results, high precision, and low recall.

In Step 3, we proposed detection strategies for two code anomalies in SPL, God Class and God Method [Fowler 1999]. Note that, as an example, there are differences between a God Class in a traditional Java system and a SPL. In case of feature-oriented SPLs, classes and refinements compose a system, and these refinements extend basic classes. In addition, we conducted a comparative study of detection strategies. As a result, we observed low recall and high precision for the proposed strategies. We intend to publish the results of this study soon. In parallel to Step 3, we conducted a study to compare methods for derivation of thresholds for software metrics. For this purpose, we used a SPL-based benchmark and a set of Java systems from Qualitas Corpus[1]. This study was an extension of precious work [Vale and Figueiredo 2015] and we intend to publish the respective results soon. Finally, Steps 4 to 6 are under development.

In parallel with Step 4, we are investigating the co-occurrence of code anomalies in the same part of source code from SPLs. Our goal is to investigate whether the occurrence of multiple code anomalies in a part of the code may affect the quality of the software system. The study design was defined and our study in under development. After finishing this study, we expect to publish the results of this study.

5. Related Work

Apel et al. (2013) discuss feature-oriented SPLs, including the development process of product lines. Moreover, the authors discuss various SPL implementation techniques such as conditional compilation, aspect-oriented programming, and feature-oriented programming. They also discuss refactoring and supporting tools for analysis of product lines. Andrade et al. (2014) present a study to investigate architectural anomalies in SPLs, i.e., the authors focus on the analysis of design-level aspects of systems.

Vale at al. (2014) present a systematic literature review on anomalies in SPLs. The authors found 70 different anomalies, some of them also defined for non-SPL systems. They observe that, considering the inherent complexity of designing SPLs, new anomalies may be proposed. Moreover, they state that some previously defined anomalies from other contexts may be applicable to SPL. In addition, we may propose new detection strategies to improve the effectiveness in detection of anomalies.

Considering the discussed related work, we propose an investigation of code anomalies in SPL, including the proposal of new code anomalies with the respective detection strategies, and automated tools to support the detection of these anomalies.

[1] http://qualitascorpus.com/

6. Conclusion and Future Work

This paper presents a proposal of master thesis in Computer Science that focuses on the detection of code anomalies in SPL. Based on the lack of studies to investigate detection strategies and supporting tools for this purpose, we aim to (i) propose new detection strategies for well-known code anomalies in SPL, (ii) eventually, propose new anomalies and detection strategies to detect them, and (iii) propose automated tools to support the new detection strategies. We present the study design for the two expected years of a Master's and also discuss some preliminary results of previous studies.

As future work, we intend to (i) conclude the last study steps, (ii) publish the study results as papers for the community, (iii) complete the study in parallel with Step 4, and (iv) finish this Master's is February 2017 with the thesis defense.

7. Acknowledgments

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

References

Andrade, H., Almeida, E., and Crnkovic, I. (2014). Architectural Bad Smells in Software Product Lines: An Exploratory Study. In *Proceedings of the 11th Working Conference on Software Architecture (WICSA)*, pp. 12:1-12:6.

Apel, S., Batory, D., Kästner, C., and Saake, G. (2013). *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer Science & Business Media.

Batory, D. (2005). *Feature Models, Grammars, and Propositional Formulas*. Springer.

Fernandes, E., Oliveira, J., Vale, G., Paiva, T., and Figueiredo, E. (2016). A Review-based Comparative Study of Bad Smell Detection Tools. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pp. 18:1-18:12.

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

Hall, T., Zhang, M., Bowes, D., and Sun, Y. (2014). Some Code Smells have a Significant but Small Effect on Faults. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, pp. 33:1-33:39.

Kang, K., Cohen, S., Hess, J., Novak, W., and Peterson, A. (1990). *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical report, DTIC Document.

Kästner, C., Apel, S., and Batory, D. (2007). A Case Study Implementing Features Using AspectJ. In *Proceedings of the 11th International Conference on Software Product Line (SPLC)*, pp. 223-232.

Lanza, M. and Marinescu, R. (2007). *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer Science & Business Media.

Macia, I., Garcia, J., Popescu, D., Garcia, A., Medvidovic, N., and von Staa, A. (2012). Are Automatically-Detected Code Anomalies Relevant to Architectural Modularity?:

An Exploratory Analysis of Evolving Systems. In *Proceedings of the 11th International Conference on Aspect-Oriented Software Development (AOSD)*, pp. 167-178.

Marinescu, R. (2004). Detection Strategies: Metrics-based Rules for Detecting Design Flaws. In *Proceedings of the 20th International Conference on Software Maintenance (ICSM)*, pp. 350-359.

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

Oliveira, J., Fernandes, E., and Figueiredo, E. (2015). Evaluation of Duplicated Code Detection Tools in Cross-Project Context. In *Proceedings of the 3rd Workshop on Software Visualization, Evolution, and Maintenance (VEM)*, pp. 49-56.

Pohl, K. and Metzger, A. (2006a). Software Product Line Testing. *Communications of the ACM*, 49(12), pp. 78-81.

Pohl, K. and Metzger, A. (2006b). Variability Management in Software Product Line Engineering. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, pp. 1049-1050.

Pohl, K., Böckle, G., and van Der Linden, F. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer Science & Business Media.

Vale, G. and Figueiredo, E. (2015). A Method to Derive Metric Thresholds for Software Product Lines. In *Proceedings of the 29th Brazilian Symposium on Software Engineering (SBES)*, pp. 110-119.

Vale, G., Figueiredo, E., Abilio, R., and Costa, H. (2014). Bad Smells in Software Product Lines: A Systematic Review. In *Proceedings of the 8th Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)*, pp. 84-94.

Weiss, D. (1999). *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison-Wesley Professional.

Fatores Críticos na Manutenção da Arquitetura de TI: Uma Abordagem na Perspectiva de Ecossistemas de Software

Aluna: Thaiana Maria Pinheiro Lima[1]
Orientadores: Rodrigo Santos e Cláudia Werner
Nível: Mestrado

Programa de Engenharia de Sistemas e Computação
COPPE/UFRJ – Universidade Federal do Rio de Janeiro
CEP 21945-970 – Rio de Janeiro, RJ, Brasil

{thaiana,rps,werner}@cos.ufrj.br

Ano de ingresso: 2015
Previsão de conclusão: Maio de 2017
Data de aprovação da dissertação (qualificação): 30/05/2016
Evento relacionado: SBES e SBCARS

Resumo: *As organizações, sejam elas fornecedoras ou adquirentes de software, utilizam diversos processos para organizar e selecionar seus produtos e serviços de software. O processo de aquisição, em particular, interfere na arquitetura de TI da empresa, pois nele são decididas a adoção, alteração ou descontinuidade das tecnologias que dão suporte aos seus produtos e serviços. Entretanto, essas organizações não estão isoladas em suas fronteiras institucionais e, sim, compõem um Ecossistema de Software (ECOS), a partir do qual uma rede socio-técnica pode ser explorada para auxiliar a modificação da arquitetura de TI. Para alcançar melhores resultados na seleção de tecnologias, é necessário um conjunto de informações acerca da base de tecnologias utilizadas e de seus relacionamentos, mas também sobre os elementos do seu ECOS. O objetivo deste trabalho é desenvolver uma abordagem para suporte à tomada de decisão no processo de modificação da arquitetura de TI de uma organização, explorando informações sobre a tecnologia candidata, seus fornecedores e os impactos gerados para a organização e para a sua rede socio-técnica. Nesse sentido, são investigados fatores críticos para manutenção da arquitetura de TI, bem como indicadores e métricas para avaliar tais fatores. Essa abordagem deve permitir às organizações reavaliarem a sua arquitetura de TI visando reduzir impactos negativos, tais como custos adicionais em projetos de desenvolvimento existentes ou dependências da organização em relação aos atores do ECOS, e.g., fornecedores ou desenvolvedores externos.*

Palavras-chave: *Ecossistemas de Software, Manutenção da Arquitetura de TI, Redes Socio-técnicas.*

[1] Com auxílio financeiro da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES)

1. Caracterização do Problema

Organizações fornecedoras ou adquirentes de software, em geral, possuem uma arquitetura de TI, de modo a planejar e estabelecer que tecnologias elas adotam ou padronizam para dar suporte a suas aplicações (produtos e serviços). Uma arquitetura de TI contribui para atender as demandas de negócio da organização por meio de um conjunto de determinações técnicas (Weils & Ross, 2004). Modificações na arquitetura podem envolver a adoção de uma nova tecnologia, remoção de parte da arquitetura, ou alteração de tecnologias utilizadas. Mudanças nessa arquitetura não são triviais, pois afetam o desenvolvimento ou aquisição de novas aplicações, uma vez que estas devem estar aderentes à arquitetura de TI. Além disso, podem refletir em necessidades de treinamento de equipe de desenvolvimento na nova tecnologia, aversão a mudanças e custos com troca de licenças ou mesmo de aplicações que dependiam da tecnologia descontinuada, entre outros (Lagerström et al., 2014). Devido à rápida evolução tecnológica, as organizações necessitam atualizar e reavaliar a sua arquitetura de TI com mais frequência. Avaliar a tecnologia em relação a dados pré-estabelecidos, gerenciáveis e fundamentados confere ao processo maior transparência, pois uma listagem de critérios utilizados pode ser conferida. Em Lagerström et al. (2014), uma das ações de maior sucesso apontada pelas empresas consiste em utilizar um procedimento bem definido na aquisição de TI. Parte da definição do processo passa por definir critérios de avaliação da tecnologia candidata (Shafia et al., 2015).

Nesse contexto, surge o problema de identificar, obter e analisar dados que influenciam na manutenção da arquitetura de TI de uma organização. Além disso, é necessário saber também quais são os impactos que essa modificação traz ao Ecossistema de Software (ECOS) da organização em termos de fatores como custos gerados, problemas de integração e de dependências de tecnologias e diferentes tipos de licença. O problema tratado neste trabalho se refere à identificação desses fatores críticos e formas de mensurá-los ao avaliar as tecnologias candidatas a modificar a arquitetura de TI de uma organização, considerando que esta integra um ECOS que envolve objetivos, aplicações, tecnologias e atores (e.g. desenvolvedores, fornecedores, usuários). Portanto, existem fatores críticos para a manutenção da arquitetura de TI, sociais ou técnicos, não identificados ou utilizados em conjunto no contexto da organização, que ainda não estão bem definidos quanto a sua descrição, possíveis indicadores e métricas, ou análises qualitativas. É preciso analisar a rede socio-técnica, que representa as relações de uma tecnologia no ECOS para que, de posse desses fatores, o gestor decida claramente que tecnologia beneficiará mais a organização no momento da aquisição e/ou no futuro próximo.

Nesse sentido, este trabalho visa desenvolver uma abordagem para auxiliar os gestores e arquitetos na manutenção da arquitetura de TI de suas organizações, a partir da perspectiva de ECOS. Um apoio ferramental para execução dos procedimentos definidos na abordagem também é considerado uma contribuição desta pesquisa. O presente artigo está estruturado da seguinte forma: na Seção 1, o problema foi caracterizado; na Seção 2, a fundamentação teórica dos temas abordados é apresentada; na Seção 3, as principais contribuições deste trabalho são expostas; na Seção 4, a abordagem proposta e o seu estado atual são apresentados; na Seção 5, é descrito como o trabalho será avaliado; por fim, na Seção 6, os trabalhos relacionados são discutidos.

2. Fundamentação Teórica

Nesta seção, são apresentadas as principais áreas que fundamentam a busca por uma solução para a temática e o problema abordados neste trabalho. São elas: Ecossistemas de Software, Redes Socio-técnicas e Manutenção da Arquitetura de TI.

2.1. Ecossistemas de Software

Uma vez que uma organização 'abre' a sua base de ativos de software (i.e., conjunto de aplicações e tecnologias) para além de seus limites, passando a disponibilizá-la sob uma plataforma tecnológica e a interagir com atores externos à sua realidade de trabalho, forma-se um ECOS (Bosch, 2009). Existe um conjunto de elementos que configura os ECOSs, sendo eles, principalmente: (a) a *plataforma central* de apoio ao desenvolvimento de software, e.g., a plataforma Java oferecendo suporte ao desenvolvimento de aplicativos no ECOS Android; (b) os *atores envolvidos*, sejam eles internos ou externos à organização, e.g., fornecedores, usuários etc.; e (c) os *ativos de ECOS*, e.g., as aplicações desenvolvidas, documentação etc. Estes elementos são tratados levando-se em consideração as interações entre si. Dado que os ecossistemas naturais inspiraram os ECOSs, estes também exploram algumas propriedades, como a saúde (Jansen, 2014).

Uma propriedade fundamental para a manutenção e crescimento de um ECOS é a saúde. Durante o ciclo de vida da plataforma central, determina-se a saúde de um ECOS a partir dos seguintes medidas (Jansen, 2014): (1) *robustez*, mede como o ecossistema se recupera de perturbações em sua estrutura ou na rede de atores; (2) *produtividade*, mede o nível de atividade do ECOS; e (3) *criação de nicho*, refere-se à capacidade de criação de oportunidades para os membros do ECOS (antigos e novos). Algumas características do ECOS podem ser utilizadas como indicadores da rede e de seus elementos. Por exemplo, a medida de robustez pode ser indicada por características como o número de projetos ativos ou a conectividade da rede após uma perturbação. Por sua vez, a produtividade pode ser medida pela quantidade de produtos e serviços de software produzidos em certo período de tempo. Por fim, a criação de nicho pode ser verificada pela diversidade de contribuidores e de projetos, i.e., número de papéis no ECOS e variedade de projetos criados/mantidos.

2.2. Redes Socio-técnicas

Redes, de forma geral, são construídas para mapear as interações entre seus elementos. Redes sociais representam diferentes tipos de relacionamentos entre pessoas conectadas por uma ou por várias relações, de forma a partilharem informações e valores, por exemplo. Como a definição de ECOS reforça a questão dos atores e de suas interações, as redes sociais podem ser aplicadas na representação da rede que emerge em um ECOS, conforme discutido em (Barbosa et al., 2013).

Segundo (Santos, 2016), a comunicação e a interação entre esses atores são realizadas por meio dos artefatos que eles compartilham. Pelo fato dos atores terem uma rotatividade maior do que os artefatos na rede, cria-se uma identidade para o artefato, transformando-o em um "cidadão de primeira classe" ao explorar o recorte da rede que detém grande parte da informação, não contemplada antes pelas redes puramente sociais. A partir das interações entre os elementos do ECOS, isto é, pessoas, organizações e artefatos técnicos, torna-se possível mapear a rede que representa tais relações, sendo esta

denominada rede socio-técnica (Lima et al., 2015). Uma maneira de representar um ECOS é por meio de Grafos, onde existem diversos tipos de *nós* como elementos do ECOS (e.g., atores, organizações, tecnologias, aplicações) e *arestas* como as relações existentes entre eles (e.g., compra e venda de tecnologia, comunicação entre desenvolvedores, *download* e *upload* de software). Nesta representação, é possível utilizar os conceitos e algoritmos da Teoria dos Grafos para calcular tamanho, centralidade, conectividade, entre outros atributos que podem ser analisados como características de ECOS e trazer um significado relevante em termos de propriedades como saúde, sustentabilidade e diversidade (Santos, 2016).

2.3. Manutenção da Arquitetura de TI

Arquitetura de TI é um conjunto de decisões técnicas integradas visando guiar a organização no atendimento de suas necessidades de negócio (Weils & Ross, 2004). Nesse sentido, a arquitetura de TI é utilizada para a padronização de processos e tecnologias que dão suporte às aplicações mantidas pela organização. Portanto, as modificações desejadas devem ser executadas com planejamento e atenção, porque alteram os padrões da organização. Ao selecionar uma tecnologia a partir de um conjunto de candidatas, a organização está optando por se envolver no ECOS desta tecnologia. Este é um processo complexo e traz implicações para decisões futuras da organização (Jansen, 2014), e.g., futuros desenvolvimentos podem necessitar de uma tecnologia mais nova ou de versões diferentes daquela definida pela arquitetura de TI. Portanto, o conjunto de informações a ser utilizado pode variar dependendo do perfil da organização, e.g., pública ou privada, bem como da flexibilidade de sua arquitetura (Lagerström et al., 2014). Identificar cenários específicos pode auxiliar na escolha de quais fatores serão úteis e devem ser apresentados.

Existem ainda outras restrições organizacionais que podem interferir na inserção ou remoção de uma tecnologia na arquitetura de TI, tais como (Shafia et al., 2015): *políticas e normas da organização* – incentivar software *open source*, fornecedores nacionais ou não aceitar determinados tipos de licença proprietária; *legislação* – principalmente em casos de empresas públicas, a legislação do país pode interferir na seção de tecnologias candidatas em questões relacionadas a taxações ou embargos econômicos; *questões econômicas* – orçamento fixado para o período da modificação ou situação econômica do país; *problemas da cultura organizacional* – aversão a mudanças de paradigma, barreiras de tecnologias desenvolvidas com reutilização, ou rejeição a determinados fornecedores.

3. Contribuições

Ao modificar a arquitetura de TI, a organização também modifica o seu ECOS com base nos ECOSs em que ela está inserida, por utilizar tecnologias e aplicações desenvolvidas fora de seus 'limites'. Esta não é uma tarefa trivial e requer informações sobre a tecnologia candidata com relação à sua comunidade, aderência à arquitetura de TI, qualidade etc., bem como do ECOS da organização, que interfiram na seleção da tecnologia, e.g., equipes com conhecimento na tecnologia candidata, políticas de governo, restrições de normas e padrões etc. A perspectiva de ECOS traz então as informações das relações externas à organização e da comunidade que a cerca. Isto é importante porque relacionamentos com fornecedores, suporte da comunidade de usuários e outros fatores externos à organização podem determinar o sucesso da aquisição e adoção/alteração/descontinuidade da tecnologia.

A principal contribuição deste trabalho está no suporte à modificação de arquitetura de TI baseada em fatores críticos que levem em consideração a rede do ECOS e que apoiem decisões de gestores e arquitetos, ou outro *stakeholder* que tenha autoridade para manipular a arquitetura de TI da organização. Identificar e centralizar as informações necessárias em uma ferramenta para utilização da organização é outra contribuição do trabalho. Por fim, a construção de cenários pela abordagem reflete o perfil do ECOS e torna a análise mais específica e aderente ao contexto organizacional. A organização também ganha ao poder personalizar o peso das informações utilizadas, pois auxilia na medição mais apropriada, variando o peso de acordo com os seus objetivos organizacionais e de negócios.

4. Estado Atual do Trabalho

Este trabalho se propõe a desenvolver uma abordagem que possa guiar a organização no processo de modificação da arquitetura de TI ao adicionar, remover ou alterar tecnologias de suporte às suas aplicações. As repercussões de tal escolha interferem no planejamento de TI das empresas, pois o ECOS da organização também depende de parte dos elementos do ECOS da tecnologia. Este suporte será realizado por meio de um conjunto de métricas técnicas e sociais que permitam a comparação de tecnologias. As informações necessárias para a análise e comparação das tecnologias candidatas serão elencadas em três níveis de classificação: fatores críticos, indicadores e métricas. Estas informações serão levadas em consideração pelo time de gestores e arquitetos de TI ao avaliar as tecnologias candidatas, de forma a identificar as ameaças e oportunidades de adoção/alteração/descontinuidade de tecnologias pela organização. Os elementos fundamentais dessa abordagem, i.e., fatores críticos e seus indicadores e métricas, são descritos a seguir: (a) **fatores críticos:** elementos que devem ser observados de modo a produzir menor impacto no processo de trabalho da organização e manter a integridade das aplicações, tecnologias e processos. Por exemplo: avaliação/satisfação ou suporte; (b) **indicadores**: para cada fator crítico, é necessário um conjunto de indicadores que representem como o fator é interpretado. Por exemplo: avaliação do ECOS ou comunidade ativa; e (c) **métricas**: indicadores são associados a um conjunto de métricas visando obter um valor comparável entre as tecnologias. Por exemplo: número de *bugs* relatados ou número de *commits* realizados.

As fontes de informação podem ser os repositórios de software da organização, repositórios de projetos *open source*, comunidade(s) da tecnologia (e.g., fóruns, *sites* oficiais), de modo que os dados sejam inseridos manualmente ou mesmo de relatórios de consultoras em tecnologias (e.g., Gartner, Forrester). O resultado é informado por meio das métricas e de uma comparação entre as tecnologias. Assim como cenários diferentes são criados, as métricas podem ter pesos diferentes, que podem ser inseridos pelos usuários. Os dados oriundos das fontes de coleta serão utilizados como insumos para o cálculo das métricas selecionadas, que será apresentado para os responsáveis pela decisão de modificar a arquitetura de TI da organização. A ferramenta irá focar em automatizar a obtenção dos dados usando, por exemplo, mineração de repositórios *open source* para capturar métricas da tecnologia e da comunidade existente. Para o cálculo final estar melhor aderente aos objetivos da organização, poderão ser atribuídos pesos e/ou desmarcados elementos da abordagem a fim de denotar prioridades. Na Figura 1, é apresentada a proposta.

O objetivo do *módulo 1* é apoiar a organização na escolha das tecnologias que ela deseja comparar na execução. O *módulo 2* visa especificar o cenário mais apropriado, a

escolha das métricas de interesse e a atribuição de pesos. O *módulo 3* ajuda a obter e armazenar os dados das métricas. O *módulo 4* compila os valores das métricas para cada tecnologia a fim de compará-las em uma escala adequada. O *módulo 5* dá suporte à representação dos resultados de modo gráfico, ou por outro método selecionado. O *módulo 6* apresenta a comparação entre as candidatas com base na pontuação fornecida no *módulo 4*. O *módulo 7* tem por objetivo complementar os resultados com orientações importantes (recomendações) para a seleção da tecnologia, mas que não podem ser quantificadas. Esses guias podem indicar ameaças e oportunidades, deixando a análise a cargo da organização.

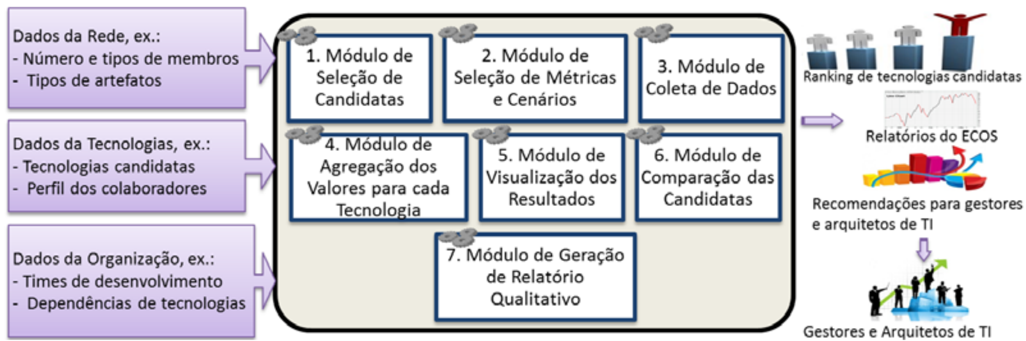


Figura 1. Arquitetura conceitual da proposta

Algumas restrições organizacionais, como as identificadas em Shafia et al. (2015), podem influenciar as métricas, e.g., cultura organizacional. O estado atual do trabalho está na identificação de fatores, indicadores e métricas a partir de relatos de casos reais e da literatura, para serem utilizados no desenvolvimento do *módulo 2*. Isto está sendo feito por meio de revisão da literatura e participação em um mapeamento sistemático sobre arquitetura de TI em ECOS no contexto de uma tese de doutorado. Uma proposta de métricas estudada é utilizar as três forças identificadas em Wareham et al. (2014) para governança em ecossistemas de tecnologias. São elas: padronização *vs.* variação; controle *vs.* autonomia; e individualidade *vs.* coletividade. Esses parâmetros seriam julgados pelos gestores e arquitetos de TI para cada indicador marcado e para as tecnologias em avaliação.

5. Descrição e Avaliação dos Resultados

Os principais resultados obtidos pelo trabalho serão: (1) a listagem de fatores críticos e seus indicadores e métricas; (2) métodos de obtenção dos dados de repositórios; (3) análise e comparação das tecnologias durante a modificação da arquitetura de TI; (4) descrição de cenários para diferentes tipos de organização; (5) realização de estudos experimentais para concepção e avaliação da abordagem; e (6) ferramenta de visualização de dados para auxiliar a organização na tomada de decisão no contexto de ECOS. Os fatores críticos e seus indicadores e métricas serão obtidos por uma busca *ad hoc* e a partir de um mapeamento sistemático. Esses elementos serão validados por especialistas da indústria e da academia por meio de um *survey online* no contexto de alguns cenários. A partir destes resultados, a abordagem será finalizada. Por fim, a ferramenta será desenvolvida para apoiar a abordagem proposta e será avaliada por meio de *focus group* com especialistas.

6. Trabalhos Relacionados

Em (Choi & Scacchi, 2001), é apresentada uma abordagem que auxilia a modelagem e simulação de arquiteturas na aquisição de software. A simulação possui dados limitados da

organização e tecnologia e não fornece informações para o responsável pela tomada de decisões. Em (van Lingen et al., 2013), alguns fatores de saúde de ECOS *open source* são identificados e utilizados para comparar casos reais como Drupal, WordPress e Joomla. Vários desses fatores refletem o perfil da tecnologia e de sua comunidade, porém não é discutido como utilizar essas informações na manutenção da arquitetura de TI. Em (Jansen, 2014), as medidas de saúde produtividade, robustez e criação de nicho são detalhadas em métricas mais específicas, mas o autor não foca na questão das tecnologias ou da arquitetura de TI de uma organização. Em (Pereira & Almeida, 2014), é discutida a arquitetura de TI e o que é considerado pelos *frameworks* de apoio à sua modelagem e instanciação. No entanto, o trabalho não considera a modificação de tecnologias e não utiliza dados da comunidade. Por fim, em (Knodel et al., 2014), arquitetura para ECOS é discutida, provendo alguns fatores e valores envolvidos, mas o estudo se concentra na arquitetura de software e não na listagem de tecnologias padrão da organização.

7. Referências

Barbosa, O., Santos, R., Alves, C., Werner, C., Jansen, S. (2013) "A Systematic Mapping Study on Software Ecosystems through a Three-dimensional Perspective". In: Jansen, S., Cusumano, M., Brinkkemper, S. (eds.) Software Ecosystems: Analyzing and Managing Business Networks in the Software Industry, 59-81. Edward Elgar Publishing.

Bosch, J. (2009) "From Software Product Lines to Software Ecosystem". In: Proceedings of the 13th International Software Product Line Conference, San Francisco, USA, 1-10.

Choi, S. J., Scacchi, W. (2001) "Modeling and Simulating Software Acquisition Process Architectures". The Journal of Systems and Software 59(3):343-354.

Jansen, S. (2014) "Measuring the Health of Open Source Software Ecosystems: Beyond the Scope of Project Health". Information and Software Technology 56(11):1508-1519.

Knodel, J., Naab, M., Rost, D. (2014) "Supporting Architects in Mastering the Complexity of Open Software Ecosystems". In: Proceedings of the 8th ECSAW, Vienna, Austria.

Lagerström, R., Baldwin, C., MacCormack, A., Dreyfus, D. (2014) "Visualizing and Measuring Software Portfolio Architectures: A Flexibility Analysis". Harvard Business School, WP 14-083.

Lima, T., Santos, R., Werner, C. (2015) "A Survey on Socio-technical Resources for Software Ecosystems". In: Proc. of the 7th ACM MEDES, Caraguatatuba, Brasil, 72-79.

Pereira, D.C., Almeida, J.P.A. (2014) "Representing Organizational Structures in an Enterprise Architecture Language". In: Proc. of the FOMI, Rio de Janeiro, Brazil, 7-15.

Santos, R. (2016) "Managing and Monitoring Software Ecosystem to Support Demand and Solution Analysis". PhD Thesis. COPPE/UFRJ, Rio de Janeiro, Brazil, 246p.

Shafia, M.A., Rabadi, N.J., Babakhan, A.R. (2015) "The Strategies and the Factors that Influence Technology Acquisition Channels. Case study: Iranian die–making industries". Intl. Journal of Manufacturing Technology and Management 29(1-2):48-65.

van Lingen, S., Palomba, A., Lucassen, G. (2013) "On the Software Ecosystem Health of Open Source Content Management Systems". In: Proceedings of the 5th International Workshop on Software Ecosystems, Potsdam, Germany, 45-56.

Wareham, J., Fox, P.B., Giner, J.L.C. (2014) "Technology Ecosystem Governance". Organization Science 25(4), 1195-1215.

Weill, P., Ross, J.W. (2004) "IT Governance: How Top Performers Manage IT Decision Rights for Superior Results". Harvard Business Press.

Recomendações de Refatorações Arquiteturais Baseadas em Análise de Impacto no contexto da ADM

Aluno: André de Souza Landi[1]

Orientador: Valter Vieira de Camargo[1]

[1]Programa de Pós-Graduação em Ciências da Computação (PPG-CC)
Departamento de Computação (DC) – Universidade Federal de São Carlos (UFSCar)
Caixa Postal 676 – 13.565-905 – São Carlos – SP – Brasil

{andre.landi,valter}@dc.ufscar.br

Resumo. *A Modernização Dirigida a Arquitetura (ADM) é uma nova tendência para padronizar os tradicionais processos de reengenharia. A base dessa proposta é um metamodelo ISO chamado Knowledge Discovery Metamodel (KDM) cujo objetivo é representar todas as características de um sistema legado e ser o metamodelo base de ferramentas de modernização. Um cenário existente em processos de modernização é reestruturar a arquitetura de um sistema no sentido de reajustar sua arquitetura deteriorada à arquitetura que foi planejada no início do seu desenvolvimento - conhecido como reconciliação arquitetural. Neste sentido, este projeto visa desenvolver uma abordagem de reconciliação arquitetural baseada em análise de impacto no contexto da ADM. A abordagem tem como entrada um conjunto de desvios arquiteturais identificados previamente por uma ferramenta existente e prossegue com o objetivo de refatorar a arquitetura do sistema eliminando os desvios encontrados. O primeiro passo da abordagem será gerar diferentes recomendações de refatorações para cada desvio encontrado. Essas recomendações serão então classificadas de acordo com seus impacto (novos desvios, métricas, etc.) que sua execução gerarão na arquitetura do sistema. Por fim, as refatorações serão aplicadas e uma nova versão do sistema será obtida. Um plug-in para o Eclipse será desenvolvido no sentido de apoiar a aplicação da abordagem. Um diferencial da proposta é o uso do metamodelo KDM como metamodelo base para a representação do sistema a ser refatorado. O uso deste metamodelo fará com que os algoritmos desenvolvidos sejam independentes de plataforma, linguagem e sejam reusáveis em outras ferramentas de modernização que seguem os princípios da ADM. A proposta será avaliada com o objetivo de averiguar se a abordagem desenvolvida apoia de forma efetiva um processo de reconciliação arquitetural.*

Nível: Mestrado
Ano de Ingresso: 2015
Época prevista de conclusão: 04/2017
Data de aprovação da proposta de dissertação: 29/06/2016
Evento Relacionado: SBES e SBCARS

Palavras-chave: ADM, KDM, recomendação de refatoração, reconciliação arquitetural, desvio arquitetural.

1. Caracterização do Problema

A Reconciliação Arquitetural (RA) é um processo que visa corrigir os desvios presentes na arquitetura de um sistema. O objetivo é fazer com que um sistema volte a ter a arquitetura que foi planejada no início do desenvolvimento e, consequentemente, satisfaça os atributos de qualidade esperados inicialmente (AVGERIOU; GUELFI; PERROUIN, 2005; GRUNBACHER *et al.*, 2003). A RA envolve dois grandes passos; a Checagem de Conformidade Arquitetural (CCA) e a Refatoração Arquitetural (RFA). O objetivo da CCA é identificar os desvios arquiteturais existentes no sistema, já o objetivo da RFA é eliminar esses desvios identificados por meio de refatorações (RFs).

Existem diversas técnicas de CCA disponíveis na literatura (BITTENCOURT *et al.*, 2010; CHAGAS *et al.*, 2016; DEITERS *et al.*, 2009), porém a que será utilizada neste projeto foi desenvolvida no laboratório de pesquisa AdvanSE do DC/UFSCar chamada ArchKDM (CHAGAS *et al.*, 2016). Essa técnica foi escolhida por dois principais motivos: (i) ter sido desenvolvida pelo mesmo laboratório de pesquisa do autor desta proposta; e (ii) utilizar como base o metamodelo KDM. A RFA é a realização de uma refatoração (RF) ou de um conjunto de RFs em um nível maior de abstração ou visando solucionar seus problemas (BUSCHMANN *et al.*, 1996; MO *et al.*, 2015). Entretanto, pesquisas acerca de RFAs no processo de RA ainda são escassas, dessa forma sendo o foco deste projeto.

Um ponto crítico de processos de RA é decidir quais RFAs devem ser aplicadas para solucionar um dado desvio arquitetural. Isso decorre da quantidade de alternativas existentes para solucionar um desvio. Entretanto, a escolha da RFA correta é algo importante, pois sua execução pode levar a problemas inesperados ou novos desvios.

Na literatura especializada encontram-se algumas abordagens que apresentam soluções voltadas à RFAs, entretanto nesses trabalhos as mesmas são escolhidas e/ou aplicadas pelo próprio engenheiro de software, fazendo com se tenha pouco apoio computacional, além de estarem sujeitos à escolhas incorretas levando ao aparecimento de novos desvios (AVGERIOU; GUELFI, 2005; BOURQUIN; KELLER, 2007; ENCKEVORT, 2009; HANNEMANN; MURPHY; KICZALES, 2005; IVKOVIC; KONTOGIANNIS, 2006). Com base em um levantamento bibliográfico foram encontradas apenas quatro abordagens de RA capazes de sugerir RFAs (ENCKEVORT, 2009; HANNEMANN; MURPHY; KICZALES, 2005; HEROLD; MAIR, 2014; TERRA *et al.*, 2012). Nota-se nestes trabalhos que as sugestões são baseadas na complexidade da RFA, não se preocupando com os impactos gerados no sistema. Isto é, as abordagens não fornecem uma análise de impacto das recomendações, possibilitando novos problemas ou desvios arquiteturais.

Em uma linha de pesquisa paralela, mas relacionada, destaca-se a Modernização Dirigida a Arquitetura (ADM), iniciada pelo OMG em 2003. A ADM visa padronizar o uso dos processos de reengenharia com o apoio de metamodelos padrões ISO e conceitos da *Model-Driven Architecture* (MDA). O principal metamodelo da ADM é o *Knowledge Discovery Metamodel* (KDM), criado para ser capaz de representar todas as características de um sistema de software, como: código-fonte, eventos, plataforma de implantação, detalhes arquiteturais e regras de negócio. O processo de RA é um tipo de modernização existente, mas ainda pouco pesquisada dentro do contexto ADM.

Uma observação importante dos trabalhos citados acima é que nenhum foi concebido no contexto da ADM e não usam ou citam o metamodelo KDM. Portanto, ainda não existem indícios na literatura da adequabilidade do metamodelo KDM como apoio a realização de RFAs e ao processo de RA como um todo. Dessa forma, essa proposta visa trazer contribuições tanto para a área de RFAs quanto para a ADM. No primeiro caso, objetiva-se desenvolver uma abordagem de RFAs que seja capaz de recomendar e classificar RFAs com base no impacto que sua execução causará no sistema. No segundo caso, o objetivo é averiguar a adequabilidade do metamodelo KDM como base para a execução de RFAs no processo de Reconciliação Arquitetural.

2. Fundamentação Teórica

A ADM visa padronizar o uso dos processos de reengenharia de software com base em metamodelos padrão ISO e conceitos da MDA. O principal metamodelo da ADM é o KDM que foi criado para ser capaz de representar praticamente todos os detalhes de um sistema de software. O principal objetivo é que esse metamodelo tenha grande aceitação da comunidade e seja empregado/adotado como metamodelo base em ferramentas de modernização. Caso isso ocorra, várias soluções poderiam ser reusadas/compartilhadas entre essas ferramentas, por exemplo: algoritmos de mineração e análise, métricas, transformações específicas de domínio, etc. O KDM pode ser considerado uma família de metamodelos, uma vez que é composto por metamodelos menores. Por exemplo, o pacote *Structure* é composto por metaclasses que representam a arquitetura de um sistema legado, composta de, por exemplo, componentes, camadas e subsistemas.

A CCA visa identificar os desvios arquiteturais existentes no sistema e pode ser definida como sendo uma das atividades principais de controle de qualidade de software. Como comentado, a ArchKDM é a ferramenta de CCA que será utilizada e seu funcionamento se dá por três etapas: (i) especificação da arquitetura planejada; (ii) geração automática de uma instância do KDM que representa o sistema legado e seu mapeamento para os elementos arquiteturais da arquitetura planejada; e (iii) realização da CCA entre ambas instâncias citadas. Como resultado obtém-se uma instância do pacote *Structure* do metamodelo KDM que contém apenas os desvios encontrados durante a CCA. Esses desvios são, por exemplo, chamadas de métodos, declarações de variáveis, criações de objetos, extensões de classe, implementações de interfaces, etc.

A segunda etapa da RA é composta pelas RFAs e é o enfoque maior desta proposta. Segundo Fowler (2014), refatoração (RF) é um processo que visa reestruturar, redesenhar ou reimplementar um sistema ou partes do mesmo. Uma RF não precisa, necessariamente, ser só em código-fonte, sendo assim existem diversos artefatos de um sistema que também podem ser refatorados. Sendo assim, conforme citado anteriormente a RFA é a realização da RF ou de um conjunto de RFs em um nível maior de abstração ou visando solucionar problemas desse maior nível.

3. Caracterização da Contribuição

Este projeto pretende trazer contribuições em dois aspectos: (i) em nível de aplicação do que será desenvolvido; e (ii) outro mais técnico e voltado ao funcionamento interno da abordagem. O primeiro refere-se a auxiliar engenheiros de software durante RFAs de desvios identificados e o segundo concentra-se em investigar o emprego do KDM como

metamodelo base no funcionamento interno da ferramenta que será desenvolvida. Na Figura 1 é apresentada a abordagem proposta. A legenda explica os símbolos utilizados.

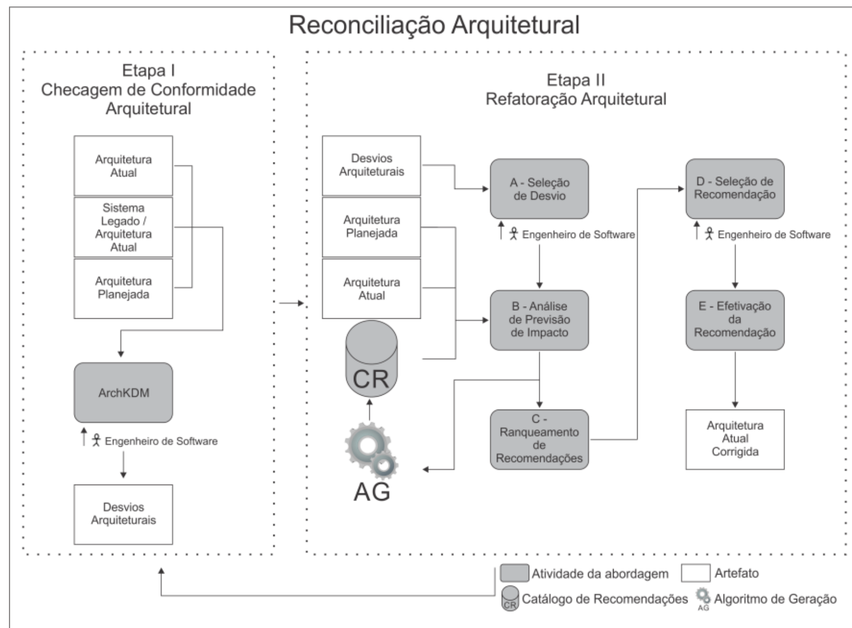


Figura 1 - Passos da abordagem proposta

A abordagem proposta é dividida em duas grandes etapas. A primeira é denominada de "Checagem de Conformidade Arquitetural" e para sua execução será usada a ArchKDM. Sua saída é uma instância KDM, no formato de um arquivo XMI que representa o sistema e um conjunto de desvios arquiteturais encontrados na CCA.

Após a primeira etapa, inicia-se a segunda etapa que é a abordagem proposta denominada de "Refatoração Arquitetural". Essa etapa é dividida em cinco atividades principais para seu devido funcionamento. A primeira é denominada "A–Seleção do Desvio". Nessa atividade deve-se selecionar um determinado desvio de uma lista apresentada pela ferramenta. Isso é necessário para marcar exatamente qual é o desvio que será analisado nas próximas etapas do processo. Essa seleção será apoiada por uma anotação (*annotation*) realizada por meio de uma transformação no KDM no sentido de adiciona-la no desvio selecionado.

Após a seleção do desvio a ferramenta irá executar internamente as próximas duas atividades denominadas de "B–Análise de Previsão de Impacto" e "C–Ranqueamento de Recomendações". Para o correto funcionamento destas atividades um dos pontos importantes é a existência de uma base de conhecimento denominada "CR–Catálogo de Recomendações". Essa base relaciona tipos de desvios com conjuntos candidatos de RFAs para corrigi-los.

Como forma de adicionar novas recomendações ao catálogo, será elaborado um algoritmo chamado "AG–Algoritmo de Geração" que tem como principal objetivo montar novas recomendações quando não são encontradas recomendações. Sendo assim, com base no catálogo, o algoritmo de previsão de impacto a ser elaborado, conseguirá realizar as recomendações cabíveis para a correção do desvio selecionado. O algoritmo de previsão de impacto deverá realizar uma análise de previsão da maioria, se não de todos, os pontos de impacto arquitetural e algumas métricas ainda a serem definidas em

um sistema caso seja efetivada uma determinada recomendação. Com as recomendações geradas e os dados da análise, será realizado um ranqueamento das recomendações conforme critérios selecionados como prioridade. Ao termino de ambas as atividades tem-se uma lista composta pelas recomendações ranqueadas.

Geradas as recomendações, inicia-se a atividade denominada "D–Seleção de Recomendação". Nessa atividade deve-se selecionar uma recomendação para ser aplicada e corrigir o desvio selecionado. Nesta atividade cada recomendação apresentará o conteúdo da RFA e os impactos que sua aplicação causará, compondo então três informações bases: (i) uma breve descrição textual da recomendação; (ii) o conjunto de RFs que compõe a RFA; e (iii) os impactos que serão ocasionados caso ela seja aplicada. Desta forma, com essas informações o engenheiro poderá decidir pela melhor recomendação para solucionar um determinado desvio arquitetural.

Ao final, depois de selecionada uma recomendação pode-se realizar a atividade denominada "E–Efetivação da Recomendação". Essa atividade realizará a execução da recomendação, isto é, a RFA propriamente dita. Assim, o engenheiro de software obterá uma versão do sistema (instância KDM) livre do desvio arquitetural selecionado. Após a execução da abordagem proposta, fecha-se o ciclo da RA para o desvio selecionado possibilitando iniciar a RA para outro desvio do sistema.

Assim sendo, os principais objetivos desta proposta são: (i) desenvolver uma abordagem de RFA completa usando o metamodelo KDM fornecido pela ADM; (ii) investigar a adequabilidade do KDM para a realização do processo de RFA bem como; e (iii) investigar a adequabilidade do KDM no processo de RA como um todo.

4. Estado Atual do Trabalho

Até o momento foi feita a primeira etapa de uma Revisão Sistemática (RS) que tem como questão principal: "Quais são as Abordagens existentes de reconciliação / refatoração arquitetural?". A partir da aplicação das strings de busca, 117 artigos foram selecionados para a etapa de extração de dados, a qual ainda está em andamento. Algo que já pode ser afirmado é que nenhum trabalho foi encontrado utilizando como base o metamodelo KDM. Além disso, também foi identificado que apenas o trabalho de Hannemann *et al.* (2005) realiza análise de impacto da RFA, mas de forma bastante preliminar. Em paralelo com a RS, foram realizadas as seguintes atividades principais: (i) estudo do metamodelo KDM; (ii) estudo aprofundado da API do MoDisco; (iii) estudo aprofundado do ArchKDM; e (iv) definição formal de um desvio arquitetural e atividade "A" da segunda etapa desenvolvida.

5. Trabalhos relacionados

Conforme mencionado anteriormente, esta proposta é focada em dois itens principais: (i) recomendações de RFAs; e (ii) análise de impacto de cada recomendação no sistema. Na literatura é possível encontrar abordagens que oferecem apoio ao primeiro item. Dentre essas abordagens as de Terra *et al.* (2012), Herold *et al.* (2014), Enckevort *et al.* (2009) e Hannemann *et al.* (2005) são mais relacionadas a presente proposta, pois propõem ou utilizam alguma ferramenta para identificar desvios arquiteturais e realizam recomendações para os desvios encontrados. É interessante notar que nestas abordagens, com exceção da de Terra *et al.*, é fornecida apenas uma opção de recomendação.

Também vale ressaltar que cada abordagem utiliza métodos diferentes para realizar tais recomendações. Herold *et al.*, por exemplo, baseia-se em anotações de um metamodelo proprietário, Enckevort *et al.* baseia-se em sugestões fixadas no inicio da abordagem ao se gerar regras de checagem de desvios em OCL. Já os trabalhos de Terra *et al.* e Hannemann *et al.* baseiam-se em catálogos de RFs, mesmo Hannemann *et al.* conseguindo gerar apenas uma opção de recomendação. Outros pontos importantes a serem observados são: (i) o trabalho de Terra *et al.* oferece mais de uma recomendação, portanto elas necessitam ser priorizadas. Essa priorização é realizada baseando-se na complexidade de execução de cada RFA, ou seja, recomendações que são mais fáceis de serem aplicadas recebem uma priorização mais alta. Os (ii) trabalhos de Herold *et al.* e Terra *et al.* não são aplicáveis a todos os tipos de desvios ou de RF/RFAs recomendadas, ou seja, são restritos a apenas alguns. Por fim, (iii) os trabalhos de Enckevort *et al.* e Hannemann *et al.* não oferecem um apoio completo ao processo de RF/RFAs estando sujeitos à escolhas ou aplicações incorretas.

Outras pesquisas da literatura também apresentam o uso de recomendações de RF/RFAs, porém não oferecem detalhes suficientes que permitam um entendimento mais aprofundado. Avgeriou *et al.* (2005), Ivkovic *et al.* (2006) e Allier *et al.* (2011) são exemplos de pesquisas realizadas com este déficit de informações. Por esse motivo, pode-se deduzir que os trabalhos podem não estar completos e/ou não oferecem um apoio computacional completo e/ou são manuais. Sendo assim, é possível concluir que para essas abordagens, a descrição desta etapa não é importante.

Já referente ao item de análise de impacto, é interessante notar que com exceção do trabalho de Hannemann *et al.*, nenhum dos trabalhos se preocupa com possíveis impactos de suas recomendações no sistema. Entretanto, mesmo Hanneman *et al.* apresentando uma análise de impacto, a mesma não oferece muitos detalhes e concentra-se em uma análise pontual para cada instrução de RF das recomendações. Essa análise tem o intuito de diminuir possíveis problemas da RF, por exemplo, verificação de nomes para evitar conflitos com nomes já existentes. Além disso, não fornece uma análise de impacto da recomendação como um todo.

Em uma linha de pesquisa relacionada, a experiência de execução de RFAs também é importante. Um exemplo é o de Bourquin *et al.* (2007) que apresenta os resultados da experiência de refatorar um grande sistema industrial. Esse tipo de pesquisa é importante para salientar que trabalhos deste aspecto auxiliam pesquisadores na realização de RF/RFAs e de RAs além de exemplificar a importância de um apoio computacional para este tipo de atividade.

6. Avaliação dos Resultados

Uma avaliação aplicada a esta proposta será um experimento controlado cujo objetivo será averiguar se a ferramenta desenvolvida apoia de forma efetiva uma atividade de RA. A ideia principal é utilizar dois grupos de sujeitos e dois sistemas com desvios arquiteturais conhecidos. Ambos os grupos deverão conduzir um processo de RA ora com a abordagem proposta ora com outra abordagem escolhida. O objetivo é traçar parâmetros de comparação e medi-los, como tempo de realização da atividade e efetividade da correção das violações. A abordagem escolhida para comparação foi a

proposta por Terra *et al.* (2012), devido a abordagem estar disponível para *download* e ser a abordagem que mais se assemelha a presente proposta.

Também será conduzido um estudo exploratório pelo próprio autor deste projeto no sentido de averiguar a adequabilidade do metamodelo KDM durante a aplicação das RFAs. Vislumbra-se que essa avaliação se concentrará em avaliar três aspectos: (i) se o KDM possui todas as metaclasses suficientes para a aplicação das RFAs; (ii) se a complexidade das RFAs (códigos em ATL) é elevada quando comparada com RFAs escritas para outros modelos; e (iii) se as dependências internas desse metamodelo permite analisar os impactos que uma determinada RFA causará.

Referências

ALLIER, S. *et al.* From object-oriented applications to component-oriented applications via component-oriented architecture. **Proceedings - 9th Working IEEE/IFIP Conference on Software Architecture, WICSA 2011**, p. 214–223, 2011.

AVGERIOU, P.; GUELFI, N.; PERROUIN, G. Evolution through architectural reconciliation.**Electronic Notes in Theoretical Computer Science**,p.165–181, 2005.

BITTENCOURT, R. A. *et al.* Improving automated mapping in reflexion models using information retrieval techniques. **Proceedings - Working Conference on Reverse Engineering, WCRE**, p. 163–172, 2010.

BOURQUIN, F.; KELLER, R. K. High-impact refactoring based on architecture violations. **Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR**, p. 149–158, 2007.

BUSCHMANN, F. *et al.* Pattern-Oriented Software Architecture Volume 1: A System of Patterns. **Wiley**, v. Vol. 1, p. 476, 1996.

CHAGAS, F. et al. KDM as the Underlying Metamodel in Architecture-Conformance Checking. **Simpósio Brasileiro de Engenharia de Software (SBES)**, p.1–10, 2016.

DEITERS, C. *et al.* Rule-based architectural compliance checks for enterprise architecture management. **Proceedings - 13th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2009**, p. 183–192, 2009.

ENCKEVORT, T. VAN. Refactoring UML Models. **Conference Companion on Object oriented programming systems languages and applications**, p. 635, 2009.

FOWLER, M. **Refatoração: Aperfeiçoando o Projeto de Código Existente**. [s.l.] Bookman, 2014.

GRUNBACHER, P. *et al.* Reconciling software requirements and architectures with intermediatemodels. **Software and Systems Modeling**, v. 3, n. 3, p. 235–253, 2003.

HANNEMANN, J.; MURPHY, G. C.; KICZALES, G. Role-based refactoring of crosscutting concerns. **Proceedings of the 4th international conference on Aspect-oriented software development - AOSD '05**, p. 135–146, 2005.

HEROLD, S.; MAIR, M. **Recommending refactorings to re-establish architectural consistency**. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). **Anais**...2014

IVKOVIC, I.; KONTOGIANNIS, K. A framework for software architecture refactoring using model transformations and semantic annotations. **European Conference on Software Maintenance and Reengineering, CSMR**, p. 135–144, 2006.

MO, R. *et al.* Hotspot Patterns: The Formal Definition and Automatic Detection of Architecture Smells. **Proceedings - 12th Working IEEE/IFIP Conference on Software Architecture, WICSA 2015**, p. 51–60, 2015.

TERRA, R. *et al.* Recommending refactorings to reverse software architecture erosion. **European Conference on Software Maintenance and Reengineering**, 2012.

Seamless and Adaptive Application-level Caching

Jhonny Mertz and Ingrid Nunes (Orientador)

Programa de Pós-Graduação em Computação (PPGC)
Departamento de Informática Aplicada – Instituto de Informática
Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brazil

{jmamertz,ingridnunes}@inf.ufrgs.br

Nível: Mestrado
Ano de ingresso no programa: 03/2015
Época prevista de conclusão: 02/2017
Data da aprovação da proposta de dissertação: Não aplicável
Eventos relacionados: SBES e SBCARS

Abstract. *Latency and cost of Internet-based services are encouraging the use of application-level caching to reduce the user-perceived latency and Internet traffic, and improve the scalability and availability of origin servers. However, this type of caching typically depends on specific details of the application and is designed and implemented in an ad-hoc way. Moreover, it is often tangled and spread all over the code. Although a well designed and implemented cache can achieve desired non-functional requirements, application performance may decay over time due to changes in the application domain, workload characteristics and access patterns, thus requiring constantly tuning the cache settings to cope with the changing dynamics of the domain, which implies extra time dedicated to maintenance. This work addresses these challenges imposed to developers by proposing a new framework to implement application-level caching, which aims to detach cache concerns from application code while leveraging application specificities to adapt cache to gradual changes at runtime.*

Keywords: *application-level caching, self-adaptive systems, separation of concerns, web application development*

1. Introduction

Dynamically generated web content represents a significant portion of web requests, and the rate at which dynamic documents are delivered is often orders of magnitudes slower than static documents [Ravi et al. 2009]. To continue satisfying users' demands and also reducing the workload on content providers, over the past years many optimization techniques have been proposed. The ubiquitous solution to this problem is some form of *caching* [Podlipnig and Böszörmenyi 2003]. Recently, latency and cost of Internet-based services are driving the proliferation of *application-level caching*, which can decrease the user perceived latency, and improve the scalability and availability of origin servers. Therefore, this type of caching has become a popular technique for enabling highly scalable web applications.

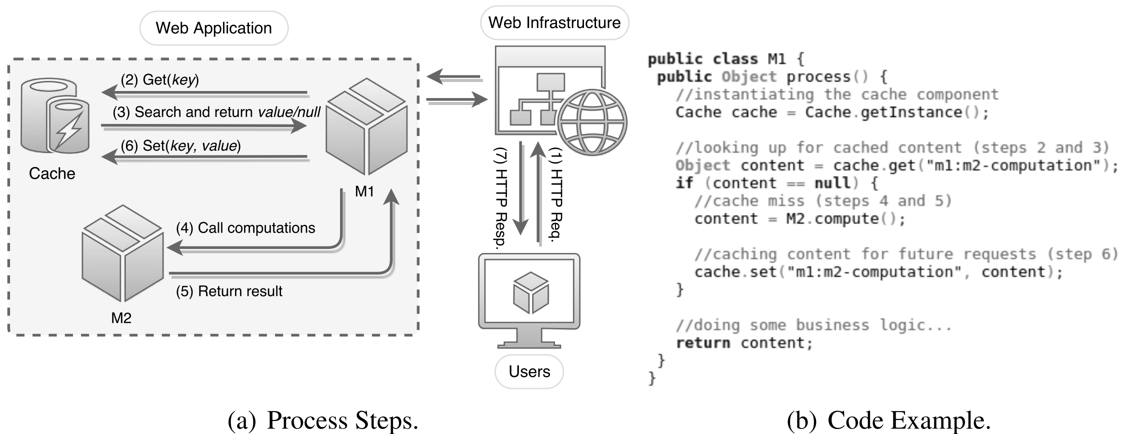
Despite its popularity, the implementation of application-level caching is not trivial and demands high effort, since it is typically implemented in an *ad hoc* way. It involves four key challenging issues: determining *what* data should be cached, *when* the data selected should be cached or evicted, *where* the cached data should be placed, and *how* to maintain the cache efficient. The main problem is that solutions for these issues usually depend on application-specific details and are manually designed and implemented by developers, which is time-consuming, error-prone and, consequently, a common source of bugs [Ports et al. 2010, Gupta et al. 2011]. Moreover, maintenance is compromised because caching logic is *tangled* with the business logic, and *spread* all over the code.

Although a well designed and implemented cache can achieve desired non-functional requirements, application performance may decay over time due to changes in the application domain, workload characteristics and access patterns. Thus, to leverage caching to improve application performance, it is required a frequent adjust of caching decisions, which implies extra time dedicated to maintenance [Radhakrishnan 2004]. This shortcoming motivates the need for adaptive caching solutions, which could automatically improve themselves to cope with the changing dynamics of the application while minimizing the challenges it poses for developers.

In Section 2 we give background on application-level caching and describe our problem. We describe research question and goals in Section 3, then present an overview of the proposed seamless and adaptive caching solution in Section 4. We describe the work in progress and outcomes expected from this study in Section 5. Finally, we discuss related work in Section 6 and, in Section 7, we conclude.

2. Background and Problem Statement

Figure 1(a) illustrates an example where an application-level cache is used to lower the application workload. First, the application receives an HTTP request (Step 1) from the web infrastructure. This HTTP request is eventually treated by an application component called M1, which in turn depends on M2. However, M2 can be an expensive operation (i.e. request the database, call a service or process that deals with a large amount of data). Therefore, M1 explicitly implements in M1's code a caching layer, which verifies whether the necessary processing is already in the cache before calling M2 (Step 2), as shown in Figure 1(b). Then, the cache component performs a look up for the requested data and returns either the cached result or a not found error (Step 3). If data is found, it means a cache *hit* and M1 can continue its execution without calling M2. If, however, a not found



(a) Process Steps. (b) Code Example.

Figure 1. Application-level Caching Overview.

error is returned, it means a cache *miss* happened, then M2 needs to be invoked (Steps 4 and 5). The newly fetched result of M2 can be cached to serve future requests faster (Step 6) and eventually a response is sent to the user (Step 7).

As already discussed in the introduction, the development of application-level caching involves four key issues: determining *how*, *what*, *when*, and *where* to cache data. The first issue is related to the fact that the cache system and the underlying source of data are not aware of each other, as illustrated in Figure 1(a). The implementation and maintenance of application-level caching are thus a challenge because its direct management leads to a concern spread all over the system—mixed with business code—which leads to increasing complexity and time of maintenance, as presented in Figure 1(b). Furthermore, the extra logic also requires additional testing and debugging time, which can be expensive in some scenarios. Moreover, web applications are usually not conceived to use caching since the beginning. As the system grows, complexity or usage analysis are performed and may lead to requests for improvements. Thus, developers must refactor data access logic to encapsulate cache data into the proper application-level object. This on-demand and manual caching process can be very time-consuming, error-prone and, consequently, a common source of bugs [Wang et al. 2014].

The second and third issues refer to deciding the right content to cache and the best moment of caching or clearing the cache to avoid cache thrashing and stale content. These decisions involve (a) choosing the granularity of cache objects, (b) translating between raw data and cache objects, and (c) maintaining cache consistency, which might not be trivial in complex applications. The last issue is related to the maintenance of the cache system, which involves concerns such as determining replacement policies and cache size.

3. Research Question and Goals

Given the issues of application-level caching (hereafter referred to simply as "caching") mentioned in introduction, our research question consists of *how to seamlessly support developers in integrating caching concerns into the web applications and leverage application-specific information to achieve an adaptive cache management?*

This work addresses the challenges imposed on developers in the context of a new seamless and adaptive framework to detach caching concerns from the application

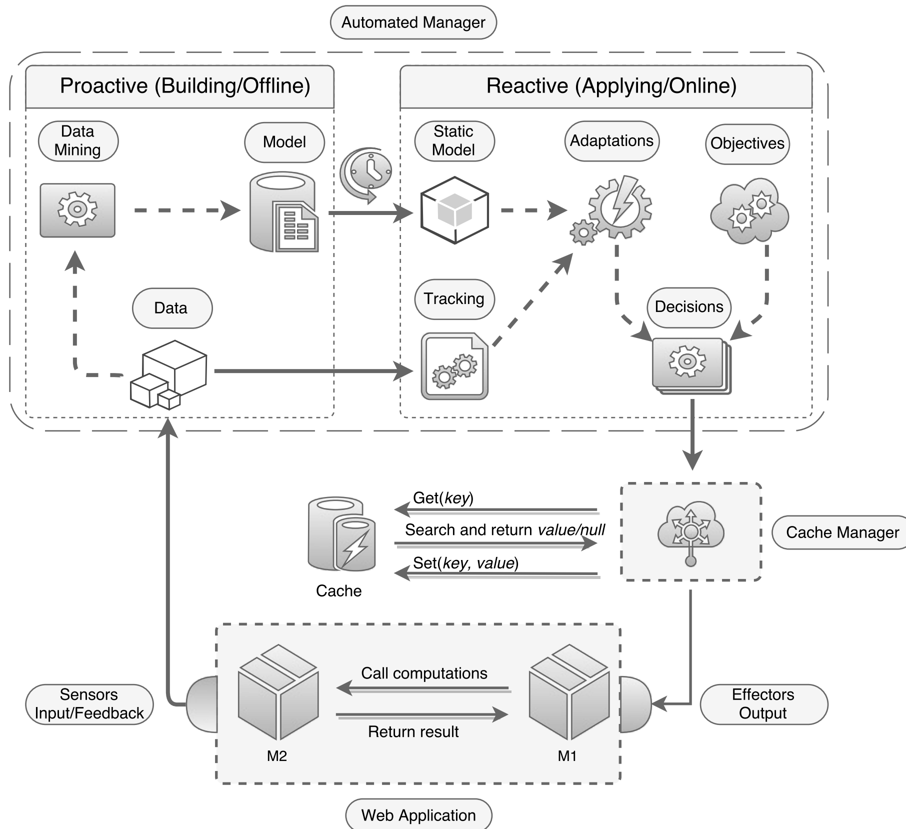


Figure 2. Seamless and Adaptive Application-level Caching Approach.

and adapt the cache management to gradual changes at runtime. Such adaptive solution would require minimal effort and input from developers, letting the caching abstractions decide the granularity of caching as opposed to providing a simple key-value store that developers must manually manage and tune when a supposed bottleneck is detected. In addition, it aims at an optimal utilization of the infrastructure, in particular of the caching system.

4. Proposed Approach

Our approach is composed of two key parts: a meta-model for obtaining application-specific information, and a runtime control loop to deal with caching concerns. The meta-model will capture declarative information about the cache and application, which will give hints to the autonomic system that in turn manages the cache at runtime autonomously. Figure 2 presents our approach, consisting of two complementary asynchronously running cycles: (a) a proactive (offline) model building, which monitors and analyzes the behavior of the application at runtime leveraging information from the meta-model instance, and (b) a reactive (online) model applying, responsible for executing adjustments accordingly to meet certain desirable objectives. Static or dynamic information about the underlying system is acquired through a seamless caching framework, which is being implemented using Aspect-oriented Programming (AOP) to provide separation of concerns.

To evaluate our proposed solution, we will perform a comparative analysis of

caching approaches: (a) manually designed and implemented by developers in existing open-source web applications (baseline), and (b) such applications with the proposed seamless and adaptive approach (without the originally implemented caching concerns). These caching setups will be evaluated regarding well-known cache metrics such as latency saving ratio and hit ratio, which will be measured by the execution of synthetic workloads, generated based on parameterized and empirical distributions.

5. Preliminary Results and Expected Contributions

An essential step towards such seamless and adaptive approach is to *understand*, *extract*, *structure* and *document* the application-level caching knowledge implicit and spread in existing web applications. Given this context, we performed a qualitative study, which involved the investigation of ten web applications (open-source and commercial) with different characteristics, to identify patterns and guidelines to support developers while designing, implementing and maintaining application-level caching. This qualitative research reveals that developers made use of supporting libraries and frameworks in order to prevent code tangling and raise the abstraction level of caching. Such supporting components are distributed cache systems and libraries that can act locally. Despite there are existing tool-supported approaches that can help developers to implement caching with minimal impact on the application code, cache decisions such as determining cacheable content and the right moment of caching or clearing the cache content are still designed and maintained manually, based on common sense, development blog consults, or online searches for tips.

These remaining issues motivate our adaptive application-level caching approach, which initially focuses on identifying and caching important methods (i.e. methods that would improve application performance if cached). The knowledge and metrics to distinguish important from less important methods were also derived from our qualitative study. In summary, the expected contributions of this work are: (a) a caching approach focused on integrating caching into web applications in a seamless and easy way, (b) a framework that detaches caching concerns from application code, and (c) an approach that provides an adaptive performance-driven selection and management of cacheable content by leveraging application-specific information captured in a meta-model.

6. Related Work

[Ports et al. 2010] and [Gupta et al. 2011] address implementation issues by providing high-level caching abstractions, in which developers can designate cacheable content, and the proposed system automatically caches and invalidates data. Aspect-oriented programming, which aims to increase modularity by allowing the separation of cross-cutting concerns, have been shown as an option to ease the caching implementation [Bouchenak et al. 2006]. Furthermore, popular web frameworks such as Spring Framework[1] and Rails[2] already provide solutions to help developers with caching implementation issues.

Although these approaches can raise the abstraction level of caching and prevent adding much cache-related code to the base code, their limitation is that they still require

[1]http://docs.spring.io/spring/docs/current/spring-framework-reference/html/cache.html
[2]http://edgeguides.rubyonrails.org/caching_with_rails.html

reasoning about caching to decide whether to cache or not content. Our approach can minimize this problem by discovering cacheable spots, based on dynamic monitoring and analysis of the application and cache behaviors. [Della Toffola et al. 2015] addressed the identification of caching opportunities. Based on an analysis of profiling logs and predefined thresholds, the proposed approach identifies and suggests performance fixes (i.e. caching opportunities). However, developers should review the suggestions and refactor the code manually, inserting cache logic into the application.

These are static caching strategies, which may become less efficient in the light of unpredicted or unobserved workloads or access patterns. To supply these limitations, automatic and intelligent caching approaches have been proposed to dynamically configure strategies like admission and replacement policies [Podlipnig and Böszörmenyi 2003, Ali et al. 2011]. However, these proposed methods mainly address the context of web pages (at proxy level), providing good results in general but are less efficient where complex logic and personalized web content are processed within the application [Wang et al. 2014]. Nevertheless, these approaches can still inspire and provide insights to application-level caching adaptation.

Adaptive approaches towards application-level caching have been proposed to leverage replacement algorithms [Ma et al. 2014, Negrão et al. 2015], consistency management [Pohl 2005, Huang et al. 2010], and admission policies [Einziger and Friedman 2014]. Some approaches [Radhakrishnan 2004, Hu et al. 2015] focus on the cache infrastructure by exploring the size and fault-tolerance adaptations in cache servers. Even though these approaches focuses on providing adaptations at application-level caches, none of them takes into account application specificities to autonomously manage their target. Thus, they ignore cache meta-data expressed by application-specific characteristics, which are closely related to application computations. Consequently, caching design, implementation and maintenance shortcomings call for new methods and techniques, which can support developers to reach a good trade-off between cache development and application performance, leveraging application-specific details.

7. Conclusion

Our work focuses on supporting developers while designing, implementing and maintaining application-level caching in their applications. We expect to provide a better overall experience with caching for developers by automatically monitoring a web application and adapting the caching logic according to the changing dynamics. This work is the first step towards a context-aware approach that can manage the cache and application adaptively, minimizing the reasoning required from developers while reaching an optimal performance of the caching service on time.

References

Ali, W., Shamsuddin, S. M., and Ismail, A. S. (2011). A survey of Web caching and prefetching. *International Journal of Advances in Soft Computing and Its Applications*, 3(1):18–44.

Bouchenak, S., Cox, A., Dropsho, S., Mittal, S., and Zwaenepoel, W. (2006). Caching Dynamic Web Content: Designing and Analysing an Aspect-Oriented Solution. *Pro-*

ceedings of the ACM/IFIP/USENIX International Conference on Middleware, 4290:1–21.

Della Toffola, L., Pradel, M., and Gross, T. R. (2015). Performance problems you can fix: a dynamic analysis of memoization opportunities. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 607–622, New York, New York, USA. ACM Press.

Einziger, G. and Friedman, R. (2014). TinyLFU : A Highly Efficient Cache Admission Policy. In *Proceedings of the 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 146–153, Torino. IEEE.

Gupta, P., Zeldovich, N., and Madden, S. (2011). A trigger-based middleware cache for ORMs. In *Proceedings of the 12th ACM/IFIP/USENIX International Middleware Conference*, volume 7049 LNCS, pages 329–349, Lisbon, Portugal. Springer Berlin Heidelberg.

Hu, X., Wang, X., Li, Y., Zhou, L., Luo, Y., Ding, C., Jiang, S., Ding, C., and Wang, Z. (2015). LAMA : Optimized Locality-aware Memory Allocation for Key-value Cache. In *Proceedings of the USENIX Annual Technical Conference*, pages 57–69.

Huang, J., Liu, X., Zhao, Q., Ma, J., and Huang, G. (2010). A browser-based framework for data cache in web-delivered service composition. In *Proceedings of the IEEE International Conference on Service-Oriented Computing and Applications, SOCA 2010*, pages 1–8. IEEE.

Ma, H., Liu, W., Wei, B., Shi, L., Bao, X., Wang, L., and Wang, B. (2014). Paap. In *Proceedings of the International ACM SIGIR Conference on Research & Development in Information Retrieval*, pages 983–986, New York, New York, USA. ACM Press.

Negrão, A. P., Roque, C., Ferreira, P., and Veiga, L. (2015). An adaptive semantics-aware replacement algorithm for web caching. *Journal of Internet Services and Applications*, 6(1):4.

Podlipnig, S. and Böszörmenyi, L. (2003). A Survey of Web Cache Replacement Strategies. *ACM Computing Surveys*, 35(4):374–398.

Pohl, C. (2005). *Adaptive caching of distributed components*. PhD thesis, Dresden University of Technology.

Ports, D. R. K., Clements, A. T., Zhang, I., Madden, S., and Liskov, B. (2010). Transactional Consistency and Automatic Management in an Application Data Cache. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, pages 279–292, CA, USA. USENIX Association.

Radhakrishnan, G. (2004). Adaptive application caching. *Bell Labs Technical Journal*, 9(1):165–175.

Ravi, J., Yu, Z., and Shi, W. (2009). A survey on dynamic Web content generation and delivery techniques. *Journal of Network and Computer Applications*, 32(5):943–960.

Wang, W., Liu, Z., Jiang, Y., Yuan, X., and Wei, J. (2014). EasyCache: a transparent in-memory data caching approach for internetware. In *Proceedings of the 6th Asia-Pacific Symposium on Internetware on Internetware*, pages 35–44, New York, New York, USA. ACM Press.

Um *framework* para visualização da evolução arquitetural de sistemas de *software* baseado em cenários de casos de uso

Leo Silva[1 2], Uirá Kulesza[1]

[1]Departamento de Informática e Matemática Aplicada (DIMAp) - Universidade Federal do Rio Grande do Norte (UFRN)
Natal – RN – Brasil

[2]Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Norte (IFRN)
Natal – RN – Brasil

leo.silva@ifrn.edu.br, uira@dimap.ufrn.br

Nível: Mestrado
Ano de ingresso no programa: 2015
Data prevista da aprovação da proposta de dissertação: outubro de 2016
Época prevista de conclusão: abril de 2017
Eventos Relacionados: SBES

Abstract. *The software systems are widely integrated into the society through various devices. Many of them are complex due to their large scale, making their architecture also complex to understand and maintain. The software visualization area uses techniques whose goal is to improve the understanding complex software artifacts in order to improve the productivity of the software process. This paper proposes to use these techniques to assist the analysis of software architecture evolution. A visualization tool is being developed to show the use case scenarios that have degraded their performance quality attribute between software versions. This tool will be evaluated through an empirical study in a real software development environment.*

Keywords: *software architecture, software visualization, software evolution*

Resumo. *Os softwares estão cada vez mais inseridos na sociedade mediante vários dispositivos. Muitos deles são complexos devido a sua larga escala, tornando a sua arquitetura também complexa para entender e manter. A área de visualização de software utiliza técnicas cujo objetivo é melhorar o entendimento e tornar mais produtivo o processo de desenvolvimento do software. Este trabalho propõe usar essas técnicas para auxiliar a análise da evolução arquitetural de softwares. Para isso, está sendo desenvolvida uma ferramenta que indica os cenários de casos de uso que tem seu atributo de qualidade de desempenho degradado entre versões. Ela será avaliada através de um estudo empírico em um ambiente real de desenvolvimento de software.*

Palavras-chave*: arquitetura de software, visualização de software, evolução de software*

1. Introdução

A importância dos *softwares* hoje em dia é cada vez mais percebida, seja através dos celulares e *tablets*, dos computadores, *smart* TVs ou até mesmo dos carros. Caserta e Zendra (2011) dizem que um *software* se torna rapidamente complexo quando o seu tamanho aumenta, trazendo dificuldades para o seu entendimento, manutenção e evolução. Além disso, muitos desses *softwares* podem ter requisitos de qualidade críticos como segurança e desempenho. Todos esses sistemas possuem uma arquitetura que serve de base para o seu desenvolvimento.

A área de arquitetura de *software* (AS) propõe técnicas, métodos e ferramentas para auxiliar no projeto do *software* e análise de seus atributos de qualidade. Embora as realizações da área sejam notáveis, existem sempre desafios inerentes ao desenvolvimento dos sistemas modernos, tais como, a complexidade devido a sua larga escala, os altos custos na mudança e a erosão do projeto [Jansen e Bosch 2005].

Além disso, os *softwares* são virtuais e intangíveis, sendo difícil criar representações mentais totais ou parciais deles [Roy e Graham 2008]. Nesse contexto, a área de visualização de *software* (VS) é definida como o uso de representações visuais para melhorar o entendimento e compreensão de diferentes aspectos de um sistema de *software* [Carpendale e Ghanam 2008].

Um dos principais tópicos na área de visualização da evolução de *software* é a visualização da evolução de arquiteturas de *software* [Caserta e Zendra 2011]. Ter uma visão da evolução de todo o sistema é considerada fundamental, porque ela pode explicar e documentar o estado atual do projeto do *software* [Roy e Graham 2008]. Uma vez que um *software* está em constante mudança para atender a novos requisitos, adaptar-se a novas tecnologias ou reparar erros, a inexistência de atividades para entender a evolução da sua arquitetura pode levar a sua degradação [D'Ambros e Lanza 2009], fazendo com que atributos de qualidade inicialmente definidos deixem de ser atendidos. É especialmente difícil visualizar a evolução da arquitetura pelo fato de ser adicionada a variável tempo, fazendo com que a quantidade de dados envolvidos aumente uma vez que todas as versões do *software* passam a ser levadas em consideração [Caserta e Zendra 2011] [Khan et al. 2012]. Com relação à aspectos da VS, alguns autores sugerem que questões como usabilidade [Caserta e Zendra 2011], efetividade, interação com o usuário, complexidade visual e técnicas de navegabilidade [Carpendale e Ghanam 2008] devem ser considerados. Algumas soluções existentes pecam por mostrar a evolução da arquitetura sob vários aspectos, tornando-as, em alguns casos, pouco eficientes.

Esta proposta de trabalho objetiva aplicar técnicas de VS para acompanhar a evolução arquitetural de um *software*. Pretende-se usar a técnica de cenários de casos de uso [Kazman et al. 1996] para analisar arquiteturas de sistemas *web*. A ferramenta busca apontar quais desses cenários degradaram ou melhoraram o atributo de qualidade de desempenho. Tal atributo foi escolhido por se tratar de uma propriedade crítica para a maioria dos sistemas de *software* atuais. A ferramenta proposta visa prover um melhor entendimento da arquitetura de um *software* por parte dos arquitetos e desenvolvedores, auxiliando-os a: (i) perceber quais cenários de casos de uso degradaram ou melhoraram o seu desempenho; (ii) identificar exatamente qual parte do código-fonte foi o responsável por uma dada degradação; e (iii) identificar quando e qual desenvolvedor realizou mudanças relacionadas a degradação.

2. Fundamentação Teórica

Conceitos de visualização de *software* (VS) e arquitetura de *software* (AS) são importantes para este trabalho. Esta seção oferece breve definições sobre esses temas. A VS é uma parte da visualização da informação. É usada para tornar a estrutura, comportamento e evolução do *software*, como a organização do código-fonte, o estado, e os *bugs*, mais compreensíveis [Diehl 2007]. Diehl (2007) enfatiza que o objetivo da VS é ajudar a compreender sistemas de *software* e aumentar a produtividade do processo de desenvolvimento.

Bass (2007) define AS como sendo "a estrutura ou estruturas do sistema, que compreendem os elementos de *software*, as propriedades externamente visíveis desses elementos e as relações entre eles". Adicionalmente, de acordo com Taylor et al. (2009), a arquitetura de um *software* incorpora todas as decisões de projeto tomadas pelos seus arquitetos, que podem afetar muitos dos seus módulos, incluindo sua estrutura e atributos de qualidade [Taylor et al. 2009]. A arquitetura é essencial para o desenvolvimento de um *software*, visando atender adequadamente os seus requisitos relacionados com os atributos de qualidade [Kazman et al. 2001], tais quais: desempenho, confiabilidade, segurança e portabilidade [Bass 2007]. No *framework* proposto, a arquitetura do *software* será analisada com o objetivo de coletar informações sobre o atributo de qualidade de desempenho.

Para essa análise, será usada uma estratégia comum de avaliação de arquiteturas: a técnica de cenários de casos de uso. Os cenários são definidos pela interação entre os *stakeholders* e o sistema. De acordo com Kazman et al. (2001), existem três tipos de cenários que podem ser usados nesse processo: cenários de casos de uso, cenários de crescimento e cenários exploratórios. No âmbito deste trabalho, será usado o primeiro tipo, que é definido como uma interação completa do usuário com o sistema em execução. Esses cenários de caso de uso serão executados a fim de determinar se a arquitetura satisfaz às restrições impostas pelo atributo de qualidade de desempenho.

3. Contribuições Esperadas

As seguintes contribuições são esperadas ao término do desenvolvimento deste trabalho:

a) Desenvolver um *framework* para visualização da evolução de AS utilizando a abordagem baseada em cenários;

b) Instanciar o *framework* para desenvolvimento de uma ferramenta que indica que cenários degradaram ou melhoraram o atributo de qualidade de desempenho durante a evolução arquitetural de um dado sistema;

c) Avaliar o *framework* no contexto de equipes reais de desenvolvimento de sistemas *web,* através da condução de estudos empíricos que tragam evidências da utilidade do *framework* proposto.

4. Estado Atual do Trabalho

De acordo com a literatura, existem várias ferramentas de VS que tratam da visualização da sua evolução arquitetural, cada uma com suas particularidades. Caserta e Zendra (2011) apontam que essas ferramentas apresentam a evolução arquitetural a partir de: (i) como a arquitetura global é alterada a cada versão, incluindo mudanças no código fonte; (ii) como os relacionamentos entre os componentes evoluem; e (iii) como

as métricas evoluem a cada *release*. Nas pesquisas realizadas até o momento da escrita deste artigo, não foram encontradas soluções de visualização da evolução arquitetural de um sistema a partir da perspectiva de cenários de casos de uso, com foco no atributo de qualidade de desempenho.

Após a revisão literária, foi idealizado um *framework* cujo objetivo é mostrar a evolução arquitetural de um *software* desenvolvido em Java através da realização de análises estáticas e dinâmicas baseadas em cenários de casos de uso. Uma vez feitas as análises nas versões desejadas, serão usadas técnicas de VS para mostrar a evolução com foco no atributo de qualidade de desempenho. A linguagem Java foi escolhida por: (i) ser atualmente uma das mais utilizadas no mundo, de acordo com a empresa TIOBE (http://tiobe.com/tiobe-index/); (ii) estar presente no desenvolvimento de *software standalone*, *web* e móvel, e; (iii) embora nos últimos anos tenham se popularizado linguagens como Python e Ruby, o Java é bem aceito no mercado brasileiro e mundial, em especial para o desenvolvimento *web*, fazendo com que existam vários sistemas desse tipo como alvos potenciais para a avaliação da ferramenta proposta.

Ferramentas de *profiling* para Java possuem análise de desempenho, porém com características diferentes. A VisualVM (http://visualvm.java.net) exibe o tempo de execução dos métodos em tempo real e o usuário pode realizar *snapshots*, entretanto, sem comparação entre eles ou de versões anteriores do *software*. Já a JProfiler (https://www.ej-technologies.com/products/jprofiler/overview.html), ferramenta paga, pode exibir o *call graph* dos métodos em tempo real, com seus respectivos tempos de execuções. *Snapshots* de determinados momentos da execução podem ser registrados e comparados. Entretanto, a comparação não é feita entre as versões anteriores do *software*. A YourKit Java Profiler (https://www.yourkit.com/java/profiler/features/) possui funcionalidades semelhantes a JProfiler. Entretanto, também é uma ferramenta paga e não realiza comparação entre versões do sistema.

Como mencionado, o *framework* proposto visa mostrar a evolução da arquitetura com o foco no atributo de qualidade de desempenho entre as diferentes versões do *software*. A análise da arquitetura requer a execução de funcionalidades do sistema que, por sua vez, executam vários módulos. Essa execução é realizada para cada uma das versões do *software* envolvidas na análise. Dessa forma, quando há degradação ou melhoria no atributo de qualidade em questão para determinada funcionalidade, nem sempre todos os módulos analisados são responsáveis por essa variação. Percebe-se que a análise de desempenho é distinta da análise de propriedades estáticas do *software*, como tamanho, coesão e acoplamento. O *framework* é composto pelos 5 (cinco) módulos descritos a seguir.

No *módulo de engenharia reversa*, a partir do código-fonte do sistema, é realizada uma engenharia reversa com o objetivo de obter as classes, atributos, métodos e relacionamentos. Para auxiliar o desenvolvimento deste módulo, a biblioteca de engenharia reversa PlantUML (http://plantuml.com) foi selecionada, por se tratar de (i) um componente que não está acoplado a nenhuma ferramenta de modelagem UML ou IDE; (ii) ser gratuita; e (iii) possuir uma linguagem própria para definir os diagramas. Os dados obtidos pela engenharia reversa serão usados quando os usuários desejarem extrair mais detalhes sobre os cenários analisados. A implementação deste módulo está finalizada.

Já no m*ódulo de integração com VCS (do inglês version control system)*, o código-fonte analisado será obtido e manipulado. Uma vez identificado um cenário que degradou ou melhorou, o usuário poderá obter informações sobre o *commit* da alteração, bem como qual *issue* está relacionada àquele código, se existir. A implementação deste módulo está finalizada.

O *módulo de integração com analisador*, por sua vez, visa integrar o *framework* a uma ferramenta de análise arquitetural baseada em técnicas de análise estática e dinâmica que foi desenvolvida dentro do grupo de pesquisa por Pinto (2015). Uma análise dinâmica do desempenho de cenários de interesse executados pelo sistema é realizada de forma automatizada. A integração se faz necessária uma vez que serão usados os resultados gerados por essa análise para obter os dados necessários para a visualização. A implementação deste módulo está em andamento.

No m*ódulo core*, são feitas as solicitações para a engenharia reversa, as integrações, a manipulação do banco de dados, o processamento que for necessário e o envio dos dados ao módulo de visualização. A implementação deste módulo está finalizada.

Por fim, é no *módulo de visualização* que serão usadas as técnicas de VS para mostrar a evolução arquitetural. A partir dos resultados coletados do analisador, será exibida em uma página *web* a visualização de (i) pacotes do sistema, destacando, através de cores, quais deles tiveram o atributo de qualidade de desempenho degradados ou melhorados. Os pacotes representam um conjunto de classes e podem ser usados para mostrar a estrutura e organização dos módulos ou componentes de um sistema. O usuário poderá aprofundar a visualização clicando no pacote desejado e, então, serão exibidas as classes daquele pacote ou seus sub-pacotes. Nesse nível de visualização, os métodos que afetaram o atributo de qualidade serão, também, destacados por cores. O usuário poderá visualizar o código-fonte do método ou obter informações do VCS. O *framework* permitirá, ainda, a visualização de (ii) cenários de casos de uso. Nessa visualização, serão mostrados os cenários analisados, destacados com cores para indicar os que afetaram o atributo de qualidade analisado. O usuário poderá escolher um dos cenários para aprofundar a visualização. Feito isso, será exibido o grafo de chamadas dos métodos executados, também diferenciados através de cores os que afetaram o cenário. A partir desse grafo, o usuário poderá escolher o método a ser visualizado e, depois, será exibida a visualização da classe a qual o método pertence.

Vale salientar que, como o *framework* (e os módulos) está em constante evolução, podem surgir novas funcionalidades e novas visualizações que não foram inicialmente previstas ou planejadas, no entanto, sem modificar drasticamente o seu funcionamento global descrito nesta seção.

5. Trabalhos Relacionados

Telea et al. (2008) propõem uma técnica de visualização, chamada *Code Flows*, que mostra a evolução do código-fonte através de uma linha contendo todas as versões. Essa técnica permite rastrear visualmente a evolução de um dado fragmento de código.

Uma metáfora com cidades foi usada por Wettel e Lanza (2007) para mostrar a evolução de arquiteturas de *software*: *CodeCity*. A ferramenta exibe a evolução em uma visualização 3D analisando o histórico do sistema enquanto que a cidade se atualiza

para refletir a versão atual. Para resolver um problema com relação ao desaparecimento de classes ou métodos no *CodeCity*, Wettel e Lanza (2008) criaram uma visualização de *timeline* para mostrar a evolução de uma classe. Essa técnica se baseia na metáfora de um edifício, onde cada prédio representa uma classe e cada tijolo um método.

Pinzger et al. (2005) propõem uma técnica de visualização chamada *RelVis*. Essa técnica pode exibir várias métricas de *software* pertencentes aos módulos e seus relacionamentos, utilizando grafos simples e diagramas *Kiviat* para representar os valores de métricas. As versões do *software* são mostradas em ordem de lançamento, cada um com uma cor. Um dos problemas dessa abordagem é que as faixas coloridas, muitas vezes, se sobrepõem, tornando difícil de visualizar o valor de determinadas métricas [Caserta e Zendra 2011].

Lanza (2001) propôs uma técnica de visualização que exibe a evolução de todo o sistema em uma imagem: *The Evolution Matrix*. Nessa técnica são combinadas a visualização e as métricas do *software* em caixas bidimensionais, onde o número de métodos é representado na largura e o número de atributos na altura. Cada versão do *software* é mostrada em uma coluna e cada linha representa uma classe diferente.

Os trabalhos relacionados aqui mostrados se diferenciam do proposto pelo fato de este propor uma visualização da evolução da arquitetura de um *software* a partir do uso de técnicas de análise estática e dinâmica. Além disso, este trabalho foca especificamente na análise da arquitetura baseada em cenários com foco no atributo de qualidade de desempenho. O resultado dessa análise é mostrado usando técnicas de visualização descritas na seção 4 deste artigo.

6. Avaliação dos Resultados

A avaliação do *framework* proposto nessa dissertação será conduzida da seguinte forma:

- Instanciação do *framework* para sistemas *web* com o intuito de analisar a evolução arquitetural da perspectiva do atributo de qualidade de desempenho;
- Conduzir um estudo empírico de utilização da ferramenta em um ambiente real de desenvolvimento de sistemas *web*. O objetivo será analisar os benefícios da instância do *framework*, assim como identificar eventuais carências e melhorias a serem realizadas. Tal estudo pretende apresentar resultados coletados automaticamente pela instância do *framework* a respeito de cenários reais de evolução da arquitetura, assim como coletar *feedback* de desenvolvedores responsáveis pelo sistema, na forma de uma avaliação qualitativa. Essa metodologia já foi utilizada com sucesso por um outro trabalho de pesquisa conduzida pelo grupo de pesquisa [Pinto et al. 2015].

Referências

Bass, L. (2007). "Software architecture in practice". Pearson Education India.

Carpendale, S.; & Ghanam, Y. (2008). "A survey paper on software architecture visualization". Technical report, University of Calgary.

Caserta, P.; & Zendra, O. (2011). "Visualization of the static aspects of software: a survey". *Visualization and Computer Graphics, IEEE Transactions on*, *17*(7), 913-933.

D'Ambros, M.; & Lanza, M. (2009). "Visual software evolution reconstruction". *Journal of Software Maintenance and Evolution: Research and Practice*, *21*(3), 217-232.

Diehl, S. (2007). "Software visualization: visualizing the structure, behaviour, and evolution of software". Springer Science & Business Media.

Jansen, A.; & Bosch, J. (2005). "Software architecture as a set of architectural design decisions". Em *Software Architecture, 2005. WICSA 2005. 5th Working IEEE/IFIP Conference on* (pp. 109-120). IEEE.

Kazman, R.; Abowd, G.; Bass, L; Clements, P. (1996). "Scenario-Based Analysis of Software Architecture". *IEEE Software*, *13*(6), 47-55.

Kazman, R.; Klein, M.; & Clements, P. (2001). "Evaluating Software Architectures-Methods and Case Studies". Addison-Wesley, Reading, Mass., 2001.

Khan, T.; Barthel, H.; Ebert, A.; & Liggesmeyer, P. (2012). "Visualization and evolution of software architectures". Em *OASIcs-OpenAccess Series in Informatics* (Vol. 27). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Lanza, M. (2001). "The evolution matrix: Recovering software evolution using software visualization techniques". Em *Proceedings of the 4th international workshop on principles of software evolution* (pp. 37-42). ACM.

Pinto, F. A. P. (2015). "An Automated Approach for Performance Deviation Analysis of Evolving Software Systems". 2015. 154f. Tese (Doutorado) – Programa de Pós-Graduação em Sistemas e Computação (PPgSC) - UFRN, Natal/RN. 2015.

Pinto, F.; Kulesza, U.; & Treude, C. (2015). "Automating the performance deviation analysis for multiple system releases: An evolutionary study". *IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM), 2015*, pp. 201-210. IEEE Computer.

Pinzger, M.; Gall, H.; Fischer, M.; & Lanza, M. (2005). "Visualizing multiple evolution metrics". Em *Proceedings of the 2005 ACM Symposium on Software Visualization*, pp. 67-75...

Roy, B.; Graham, T. C. N. (2008). "Methods for Evaluating Software Architecture: A Survey". Technical *report* No. 2008-545, School of Computing, Queen's University at Kingston. Ontario, Canada.

Taylor, R. N.; Medvidovic, N.; Dashofy, E. M. (2009). "Software architecture: foundations, theory, and practice". Wiley Publishing.

Telea, A.; Auber, D. (2008). "Code flows: Visualizing structural evolution of source code". Em *Computer Graphics Forum* (Vol. 27, No. 3, pp. 831-838). Blackwell Publishing Ltd.

Wettel, R.; Lanza, M. (2007). "Visualizing software systems as cities". Em *Visualizing Software for Understanding and Analysis, 2007. VISSOFT 2007. 4th IEEE International Workshop on* (pp. 92-99). IEEE.

Wettel, R.; Lanza, M. (2008). "Visual exploration of large-scale system evolution". Em *Reverse Engineering, 2008. WCRE'08. 15th Working Conference on* (pp. 219-228). IEEE.