



Congresso Brasileiro
de Software: Teoria e Prática **BRASILIA.2013**

Congresso Brasileiro de Software: **Teoria e Prática**

29 de setembro a 04 de outubro de 2013



Brasília-DF

Anais

VEM 2013

I WORKSHOP BRASILEIRO DE VISUALIZAÇÃO, EVOLUÇÃO E MANUTENÇÃO DE SOFTWARE



VEM 2013

I Workshop Brasileiro de Visualização, Evolução e Manutenção de Software

29 de setembro de 2013
Brasília-DF, Brasil

ANAIIS

Volume 01
ISSN: 2178-6097

COORDENAÇÃO DO VEM 2013

Nabor das Chagas Mendonça (UNIFOR)
Renato Lima Novais (IFBA)
Manoel Gomes de Mendonça Neto (UFBA)

COORDENAÇÃO DO CBSOFT 2013

Genaína Rodrigues – UnB
Rodrigo Bonifácio – UnB
Edna Dias Canedo – UnB

REALIZAÇÃO

Universidade de Brasília (UnB)
Departamento de Ciência da Computação (DIMAp/UFRN)

PROMOÇÃO

Sociedade Brasileira de Computação (SBC)

PATROCÍNIO

CAPES, CNPq, Google, INES, Ministério da Ciência, Tecnologia e Inovação, Ministério do Planejamento, Orçamento e Gestão e RNP

APOIO

Instituto Federal Brasília, Instituto Federal Goiás, Loop Engenharia de Computação, Secretaria de Turismo do GDF, Secretaria de Ciência Tecnologia e Inovação do GDF e Secretaria da Mulher do GDF



VEM 2013

1st Brazilian Workshop on Software Visualization, Evolution and Maintenance

September 29, 2013
Brasília-DF, Brazil

PROCEEDINGS

Volume 01
ISSN: 2178-6097

VE M 2013 CHAIRS

Nabor das Chagas Mendonça (UNIFOR)
Renato Lima Novais (IFBA)
Manoel Gomes de Mendonça Neto (UFBA)

CBSOFT 2013 GENERAL CHAIRS

Genaína Rodrigues – UnB
Rodrigo Bonifácio – UnB
Edna Dias Canedo – UnB

ORGANIZATION

Universidade de Brasília (UnB)
Departamento de Ciência da Computação (DIMAp/UFRN)

PROMOTION

Brazilian Computing Society (SBC)

SPONSORS

CAPES, CNPq, Google, INES, Ministério da Ciência, Tecnologia e Inovação, Ministério do Planejamento, Orçamento e Gestão e RNP

SUPPORT

Instituto Federal Brasília, Instituto Federal Goiás, Loop Engenharia de Computação, Secretaria de Turismo do GDF, Secretaria de Ciência Tecnologia e Inovação do GDF e Secretaria da Mulher do GDF

Autorizo a reprodução parcial ou total desta obra, para fins acadêmicos, desde que citada a fonte

APRESENTAÇÃO

O I Workshop Brasileiro de Visualização, Evolução e Manutenção de Software (VEM 2013) é um dos eventos integrantes do IV Congresso Brasileiro de Software: Teoria e Prática (CBSOFT 2013), realizado em Brasília – DF, no período de 29 de setembro a 4 de outubro de 2013. O VEM tem como objetivo integrar as comunidades das áreas de visualização, manutenção e evolução de software, oferecendo um fórum em território brasileiro onde pesquisadores, estudantes e profissionais podem apresentar seus trabalhos e trocar idéias a respeito dos princípios, práticas e inovações recentes em suas respectivas áreas de interesse. O VEM surgiu a partir da junção de dois outros workshops brasileiros focados em temas relacionados que até então vinham sendo realizados de forma separada, a saber: Workshop de Manutenção de Software Moderna (WMSWM) e Workshop Brasileiro de Visualização de Software (WBVS).

O Comitê de Programa (CP) do VEM 2013 é formado por 30 pesquisadores atuantes nas áreas de visualização, manutenção e evolução de software, provenientes de diversas regiões do Brasil e de outros países da América Latina. Os membros do CP foram responsáveis pela seleção de 8 artigos completos para serem apresentados no VEM 2013, de um total de 11 artigos submetidos. Cada artigo submetido foi avaliado por pelo menos três membros do CP, com base nos critérios de originalidade, qualidade técnica e adequação ao escopo do workshop. Os artigos selecionadas abrangem diversos temas de interesse do evento, como gerência de configuração, métricas, análise arquitetural, e ferramentas de apoio à visualização, reutilização, manutenção, reengenharia e migração de software.

Além da apresentação dos oito artigos selecionados pelo CP, o programa técnico do VEM 2013 inclui ainda um palestra convidada e um painel onde os participantes do evento poderão discutir os problemas e soluções mais relevantes atualmente nas áreas de visualização, manutenção e evolução de software, bem como novas oportunidades de pesquisa e desenvolvimento tecnológico nessas áreas.

Para finalizar, gostaríamos de agradecer a todos os autores que submeteram artigos ao VEM 2013, pelo seu interesse, aos membros do CP, pelo esforço e valiosa colaboração durante o processo de seleção dos artigos, e aos organizadores e patrocinadores do CBSOFT 2013, pelo apoio na realização do evento.

Brasília, 29 de setembro de 2013

Nabor C. Mendonça

Renato L. Novais

Manoel Mendonça

Coordenadores do Comitê de Programa do VEM 2013

FOREWORD

The 1st Brazilian Workshop on Software Visualization, Evolution and Maintenance (VEM 2013) is part of the 4th Brazilian Congress on Software: Theory and Practice (CBSOFT 2013), held in Brasília – DF, from September 29 to October 4, 2013. Its main goal is to foster the integration of the software visualization, evolution and maintenance communities, providing a Brazilian forum where researchers, students and professionals can present their work and exchange ideas on the principles, practices and innovations related to their respective areas of interest. VEM was created from the fusion of two previous Brazilian workshops on related themes, namely Workshop on Modern Software Maintenance (WMSWM) and Brazilian Workshop on Software Visualization (WBVS).

The VEM 2013 Program Committee (PC) is composed of 30 active researchers in the areas of software visualization, evolution and maintenance, who come from several regions of Brazil as well as from other Latin American countries. The PC members selected 8 full articles to be presented at VEM 2013, from a total of 11 submissions. Each submission was evaluated by at least three PC members, based on their originality, technical quality and adequacy to the event's scope. The selected articles cover several themes of interest to the workshop, such as configuration management, metrics, architectural analysis, and tool support for software visualization, reuse, maintenance, reengineering and migration.

In addition to the eight full articles selected by the PC, the VEM 2013 technical program also includes an invited keynote talk and a panel where the event participants will discuss the main problems and solutions related to software visualization, evolution and maintenance, as well as new research and technological development opportunities in these areas.

Finally, we would like to express our deepest gratitude to all the authors who submitted their work to VEM 2013, for their interest, to the PC members, for their effort and invaluable collaboration during the article selection process, and to the CBSOFT 2013 organizers and sponsors, for their support and contribution.

Brasília, September 29, 2013

Nabor C. Mendonça

Renato L. Novais

Manoel Mendonça

VEM 2013 Program Committee Co-chairs

BIOGRAFIA DOS COORDENADORES / FEES 2013

CHAIRS SHORT BIOGRAPHIES

NABOR DAS CHAGAS MENDONÇA

Nabor Mendonça é Professor Titular da Universidade de Fortaleza (UNIFOR). Obteve o título de Doutor em Computação pelo Imperial College London, Reino Unido, em 1999. Suas principais áreas de pesquisa são manutenção e evolução de software, sistemas distribuídos, arquiteturas orientadas a serviço, e computação em nuvem. Desde 2009, é Bolsista de Produtividade em Pesquisa (Nível 2) do Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq).

Nabor Mendonça is a Titular Professor at Universidade de Fortaleza (UNIFOR), Brazil. He holds a Ph.D. degree in Computing from Imperial College London, United Kingdom. His main research areas are software maintenance and evolution, distributed systems, service-oriented architectures, and cloud computing. Since 2009, he holds a Productivity in Research Scholarship (Level 2) from Brazil's National Council for Scientific and Technological Development (CNPq).

RENATO LIMA NOVAIS

Renato Novais é professor efetivo do Instituto Federal de Educação, Ciência e Tecnologia da Bahia (IFBA). Obteve o título de Doutor em Ciência da Computação pela Universidade Federal da Bahia, em 2013, com período sanduíche no Fraunhofer Center for Experimental Software Engineering, MD, EUA. Suas principais áreas de pesquisa são visualização de software, evolução de software, engenharia de software experimental, manutenção e reengenharia de software, e compreensão de software.

Renato Novais is an effective professor at Instituto Federal de Educação, Ciência e Tecnologia da Bahia (IFBA). He holds a D.Sc. degree in Computer Science from Universidade Federal da Bahia. During his Doctorate, he spent a period as a visiting scientist in Fraunhofer Center for Experimental Software Engineering, MD, USA. His main research areas are software visualization, software evolution, experimental software engineering, software maintenance and reengineering, and software comprehension.

MANOEL GOMES DE MENDONÇA NETO

Manoel Mendonça é professor de Ciência da Computação da Universidade Federal da Bahia (UFBA). Prof. Mendonça obteve o Ph.D. em ciência da computação pela Universidade de Maryland em College Park (UMCP), o M.Sc. em Engenharia da Computação pela UNICAMP (Brasil), e o bacharelado em Engenharia Elétrica pela UFBA. De 1994 a 1997, ele atuou como cientista visitante e foi premiado com uma bolsa de doutoramento do IBM Toronto Laboratory's Centre for Advanced Studies. De 1997 a 2000, ele trabalhou como pesquisador associado na UMCP e como cientista do Centro Fraunhofer Maryland. Ingressou na Universidade de Salvador (Brasil) como Professor em 2000. Lá ele dirigiu Centro de Pesquisa em Computação da universidade e ajudou a criar o primeiro mestrado e doutorado em ciência da computação em seu estado natal, Bahia. De 2008 a 2009, foi presidente da Comissão Especial de Engenharia de Software da Sociedade Brasileira de Computação (SBC). Ingressou na UFBA em 2009. Lá, ele chefiou o programa de pós-graduação de ciência da computação e o Laboratório de Engenharia de Software. Desde 2012, ele é o diretor do Centro Fraunhofer para Engenharia de Software e Sistemas da UFBA. Dr. Mendonça publicou mais de 100 trabalhos técnicos. Seus principais interesses de pesquisa são em engenharia de software e visualização de informação. Ele é um membro da SBC e ACM, e um membro sênior do IEEE.

Manoel Mendonça is a Professor of Computer Science at the Federal University of Bahia (UFBA). Prof. Mendonça holds a Ph.D. in computer science from the University of Maryland at College Park (UMCP), a M.Sc. in computer engineering from UNICAMP (Brazil), and a bachelor in electrical engineering from UFBA. From 1994 to 1997, he was a visiting scientist and was awarded a doctoral fellowship from IBM Toronto Laboratory's Centre for Advanced Studies. From 1997 to 2000, he worked as a Faculty Research Associate at UMCP and as a scientist at the Fraunhofer Center Maryland. He joined Salvador University (Brazil) as a Professor in 2000. There he headed the university's Computing Research Center and helped to create the first computer science master and doctoral programs at his home state of Bahia. From 2008 to 2009, he was the president of the Special Commission for Software Engineering of the Brazilian Computer Society (SBC). He joined UFBA in 2009. There, he has headed the computer science graduate program and software engineering lab. Since 2012, he is the Director of the Fraunhofer Center for Software and Systems Engineering at UFBA. Dr. Mendonça has published over 100 technical papers. His main research interests are on software engineering and information visualization. He is a member of SBC and ACM, and a senior member of IEEE.

COMITÊS TÉCNICOS / TECHNICAL COMMITTEES

COMITÊ DIRETIVO / STEERING COMMITTEE

Aline Vasconcelos, IFF
Cláudia Werner, COPPE/UFRJ
Dalton Serey, UFCG
Glauco Carneiro, UNIFACS
Heitor Costa, UFLA
Leonardo Murta, UFF
Manoel Mendonça, UFBA
Marco Antônio Pereira Araújo, IFSEMG
Nabor Mendonça, UNIFOR
Renato Novais, IFBA
Rosana Braga, ICMC/USP
Sandra Fabbri, UFSCar

COMITÊ DE PROGRAMA / PROGRAM COMMITTEE

Aline Vasconcelos, IFF
Cláudia Werner, COPPE/UFRJ
Claudio Sant'Anna, UFBA
Dalton Serey, UFCG
Eduardo Figueiredo, UFMG
Elisa Huzita, UEM
Glauco Carneiro, UNIFACS
Guilherme Travassos, COPPE/UFRJ
Gustavo Rossi, Universidad Nacional de La Plata
Heitor Costa, UFLA
Humberto Marques, PUC Minas
Joberto Martins, UNIFACS
Jorge César Abrantes de Figueiredo, UFCG
Kecia Ferreira, CEFET-MG
Leonardo Murta, UFF
Lincoln Souza Rocha, UFC/Quixadá
Marcelo Pimenta, UFRGS
Marco Antônio Pereira Araújo, IFSEMG
Marco Aurelio Gerosa, IME/USP
Maria Istela Cagnin, UFMS
Nabor Mendonça, UNIFOR (Coordenador / Chair)
Paulo Henrique Mendes Maia, UECE
Renato Novais, IFBA (Coordenador / Chair)
Ricardo Argenton Ramos, UFVSF

Rogério de Carvalho, IFF
Rosana Braga, ICMC/USP
Rosângela Penteado, UFSCar
Sandra Fabbri, UFSCar
Simone Vasconcelos, IFF
Valter Camargo, UFSCar

COMITÊ ORGANIZADOR / ORGANIZING COMMITTEE

COORDENAÇÃO GERAL

Genaína Nunes Rodrigues, CIC, UnB

Rodrigo Bonifácio, CIC, UnB

Edna Dias Canedo, CIC, UnB

COMITÊ LOCAL

Diego Aranha, CIC, UnB

Edna Dias Canedo, FGA, UnB

Fernanda Lima, CIC, UnB

Guilherme Novaes Ramos, CIC, UnB

Marcus Vinícius Lamar, CIC, UnB

George Marsicano, FGA, UnB

Giovanni Santos Almeida, FGA, UnB

Hilmer Neri, FGA, UnB

Luís Miyadaira, FGA, UnB

Maria Helena Ximenis, CIC, UnB

ÍNDICE DE ARTIGOS / TABLE OF CONTENTS

ARTIGOS COMPLETOS / FULL PAPERS

ST1 – CONFIGURAÇÃO, TIPOS E ANÁLISE ARQUITETURAL

WHAT IS GOING ON AROUND MY REPOSITORY? 14

Cristiano Cesario (UFF), Leonardo Murta (UFF)

TIPAR OU NÃO TIPAR? COMPREENDENDO QUAIS FATORES INFLUENCIAM A ESCOLHA POR UM SISTEMA DE TIPOS 22

Carlos Souza (UFMG), Eduardo Figueiredo (UFMG), Marco Túlio Valente (UFMG)

VERIFICAÇÃO DE CONFORMIDADE ARQUITETURAL COM TESTES DE DESIGN – UM ESTUDO DE CASO 30

Izabela Melo (UFCG), João Arthur Brunet Monteiro (UFCG), Dalton Serey (UFCG), Jorge César Abrantes de Figueiredo (UFCG)

ST2 – VISUALIZAÇÃO E REUTILIZAÇÃO

CHANGEMINER: UM PLUG-IN VISUAL PARA AUXILIAR A MANUTENÇÃO DE SOFTWARE 38

Francisco Rodrigues Santos (IFS), Methanias Colaço Júnior (UFS), Manoel Mendonça (UFBA)

VISUALIZAÇÃO DE SOFTWARE COMO SUPORTE AO DESENVOLVIMENTO CENTRADO EM MÉTRICAS ORIENTADAS A OBJETOS 46

Elidiane Pereira dos Santos (IFBA), Sandro Andrade (IFBA)

REUSEDASHBOARD: APOIANDO STAKEHOLDERS NA MONITORAÇÃO DE PROGRAMAS DE REUTILIZAÇÃO DE SOFTWARE 54

Marcelo Palmieri (COPPE/UFRJ), Marcelo Schots (COPPE/UFRJ), Claudia Werner (COPPE/UFRJ)

ST3 – ADAPTAÇÃO E REENGENHARIA

UMA ABORDAGEM BASEADA EM EVENTOS PARA ADAPTAÇÃO AUTOMÁTICA DE APLICAÇÕES PARA A NUVEM 62

Michel Vasconcelos (UNIFOR), Davi Barbosa (CUE-FIC), Paulo Henrique Maia (UECE), Nabor Mendonça (UNIFOR)

UMA ABORDAGEM DE REENGENHARIA DE SOFTWARE ORIENTADA POR MÉTRICAS DE QUALIDADE 70

Giuliana Bezerra (UFRN)

What is going on around my repository?

Cristiano M. Cesário, Leonardo G. P. Murta

Instituto de Computação
Universidade Federal Fluminense (UFF) – Niterói, RJ – Brazil
`{ccesario, leomurta}@ic.uff.br`

Abstract. Software development using distributed version control systems has become more frequent recently. Such systems bring more flexibility, but they also bring greater complexity to administer and monitor the multiple existing repositories as well as the proliferation of several branches, which requires frequent merges. In this paper we propose DyeVC, an extensible tool to assist the developer in identifying dependencies among the distributed repositories in order to help to understand what is going on around one's repository.

1 Introduction

Version Control Systems (VCS) date back to the 70s, when SCCS emerged [Rochkind 1975]. Their primary purpose is to keep software development under control [Estublier 2000]. Along these almost 40 years, VCSs evolved from a centralized repository with local access, as in SCCS and RCS [Tichy 1985], to a client-server approach, as in CVS [Cederqvist 2005] and Subversion [Collins-Sussman et al. 2011]. More recently, distributed VCSs (DVCS) arose, allowing clones of the entire repository in different locations, as in Git [Chacon 2009] and Mercurial [O'Sullivan 2009a]. According to a survey conducted among the Eclipse community [Eclipse Foundation 2012], Git and Github usage increased from 13% to 27% between 2011 and 2012 (a growth greater than 100%). Between 2010 and 2011, this growth had already been around 78% [Eclipse Foundation 2011]. This clearly shows momentum and a strong tendency in the adoption of DVCSs among the open source community.

According with [Walrad and Strom 2002], creating branches is essential to software development, because it enables concurrent development, allowing the maintenance of different versions of a system, the customization to different platforms and to different customers, among other features. DVCSs include better support to work with branches [O'Sullivan 2009b], turning the branch creation into a recurring pattern, no matter if this creation is explicitly done by executing a “branch” command or implicitly, when a repository is cloned, a commit is issued based on an old revision, or updates are pushed to or pulled from other repositories. All these branches, whether explicit or not, eventually will be reintegrated to their origin by means of merge operations, reflecting the changes made.

The proliferation of branches, the increasing growth of development teams, and their distribution along distant locations – even different continents –, introduce additional complexity to notice actions performed in parallel by different developers. According to [Perry et al. 1998], concurrent development increases the number of defects in software. Besides, [Da Silva et al. 2006] say that development tools control concurrent development with the assistance of VCSs, which work in majority with the concept of each developer having a private workspace. This postpones the perception of conflicts that result from changes made by co-workers. These conflicts are noticed only after a pull

or a push in the context of DVCS. Moreover, [Brun et al. 2011] shows that, even using modern DVCSs, conflicts during *merges* are frequent, persistent, and appear not only as overlapping textual edits (i.e., physical conflicts) but also as subsequent build (i.e., syntactic conflicts) and test failures (i.e., semantic conflicts).

By enabling repository clones, DVCSs expand the branching possibilities exposed by [Appleton et al. 1998], allowing several repositories to coexist with fragments of the project history. This may lead to complex topologies where changes can be sent to or received from any repository. This scenario generates traffic similar to that of peer-to-peer applications. With this diversity of topologies, the administration of the evolution of a complex system becomes a tough task, making it difficult to find an answer to questions like: “Which clones were created from a repository?” or “What are the dependencies between different clones?”, among others.

Most of existing works deal with the last two issues, giving to the developers the perception of concurrent changes. Palantir [Sarma and Van der Hoek 2002], Lighthouse [Da Silva et al. 2006], CollabVS [Dewan and Hegde 2007], Safe-Commit [Wloka et al. 2009], Crystal [Brun et al. 2011], WeCode [Guimaraes and Silva 2012], and Polvo [Santos and Murta 2012] are examples of this kind of work. Among these, the only one that deals with multiple branches is Polvo, establishing metrics that assist in determining the merge effort between branches. However, it has a strict focus in Centralized Version Control Systems (CVCS), which are much less prone to branches if compared to DVCS.

This paper introduces DyeVC¹, which is a novel visualization infrastructure for DVCS. The main goal of DyeVC is to increase the developer knowledge of what is going on around his repository and the repositories of his teammates. DyeVC is an extensible tool that gathers information about different clones of a repository and presents them visually to the user. This allows one to perceive how his repository evolved over time and how this evolution compares to the evolution of other repositories in the project.

The rest of this paper is organized as follows. Section 2 presents a motivational example to this work. Section 3 presents the approach used in DyeVC, along with its current state of development. Section 4 discusses some related work and in Section 5 we conclude the paper and present future work.

2 Motivational Example

Figure 1 shows a scenario with some developers, each one having a clone of a repository originally created at Xavier Institute. Xavier Institute acts like a central repository, where code developed by all teams is integrated, tested, and released to production. There is a team working in Xavier Institute, leaded by Professor X, and a remote developer (Storm) that periodically receives updates from the Institute. Outside the Institute, Wolverine leads a remote team located in a different continent, which is constantly synchronized with the Institute. Arrows in Figure 1 indicate the direction in which updates are sent. Thus, for example, Rogue can pull updates from Gambit, and both Gambit and Beast can pull updates from Rogue.

¹ Dye is commonly used in cells to observe the cell division process. As an analogy, DyeVC allows developers to observe how a Version Control repository evolved over time.

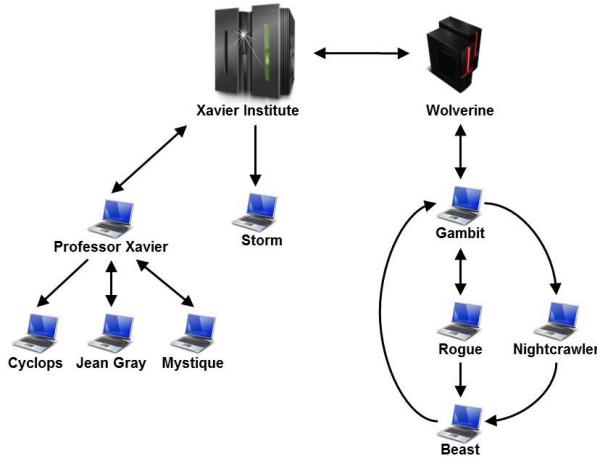


Figure 1 – A development scenario involving some developers

Each one of the developers has a complete copy of the repository and is able to send and receive updates to or from any other developer. Considering the existence of n developers, we could reach a total of $n * (n - 1)$ different possibilities of communication. In practice, however, this limit is not reached: while interaction among some developers is frequent, it may happen that others have no idea about the existence of some coworkers, as it occurs with Mystique and Nighthcrawler, where there is no direct communication.

Taking Beast as an example, at a given moment, how can he know if there are commits in Rogue or in Nighthcrawler that were not yet pulled? Alternatively, would be the case that there are local commits pending to be pulled by Gambit? Beast could certainly periodically pull changes from his partners, to check if eventually there were updates, but this would be a manual procedure, prone to be forgotten. What if a tool had the knowledge of Beast's partners, and constantly monitored those, warning Beast of any local or remote updates that had not been synchronized yet?

3 DyeVC

The approach we propose with DyeVC involves continuously monitoring a group of interrelated repositories, based on repositories registered by the user. The implementation uses Java Web Start² Technology, and focus on monitoring Git repositories. The gathering of information from repositories is accomplished using JGit³ library, which allows the user to use DyeVC without having a Git client installed. DyeVC presents a visual log using JUNG⁴ library, from which it inherits the ability to extend existing layouts and filters to create new ones, which can be dynamically attached to the graphs it presents.

DyeVC gathers information in different levels of detail, presenting it in a visual style, and supplying notifications whenever there are changes in any of the registered repositories or in its partners, which are repositories that a given repository communicates with, as shown in Figure 2. The period between subsequent monitor runs is configurable and defaults to 5 minutes.

² <http://docs.oracle.com/javase/6/docs/technotes/guides/javaws/>

³ <http://www.eclipse.org/jgit/>

⁴ <http://jung.sourceforge.net/>

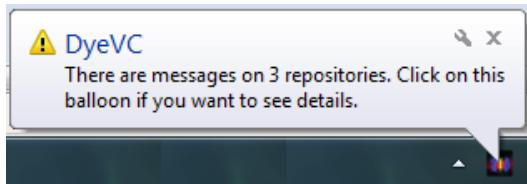


Figure 2 – DyeVC showing notifications in the notification area

The levels of detail defined in DyeVC include: presenting the status of a given repository against its partners (Level 1); zooming into the branches the repository, showing the status of each local branch that tracks a remote branch (Level 2); and zooming into the commits of the repository, showing a visual log with information about each commit (Level 3).

In Level 1, DyeVC presents the status of a repository against its partners. The status presented by DyeVC can be one of those presented in Table 1.

Table 1 – Possible status of a repository

Status	Description
?	DyeVC has not analyzed the repository yet.
✓	Repository is synchronized with all partners.
↑	Repository has changes that were not sent yet to its partners (it is ahead its partners).
↓	Partners have changes that were not sent yet to the repository (it is behind its partners).
↑↓	Repository is both ahead and behind its partners.
✗	Invalid repository. This happens when DyeVC cannot access the repository. The reason is presented to the user.

The evaluation of the status individually considers the existing commits in each repository. Each commit maps a group of changes in various artifacts and it is uniquely identified in the repository. Moreover, due to the nature of DVCS, where old data is never deleted and commits are cumulative, if a commit N is created over a commit N – 1, the existence of commit N in a given repository implies that commit N – 1 also exists in the repository. Thus, by examining the existence of commits in the local repository not yet replicated to the remote repository, and vice-versa, it is possible to come to one of the situations presented in Table 2.

Table 2 – Status of a local repository with regard to a remote one, based on the existence of non-replicated commits in each one of them

Existence of non-replicated commits		Local Status
Local Repository	Remote Repository	
Yes	Yes	↑↓ Ahead and Behind (needs push and pull)
Yes	No	↑ Ahead (needs push)
No	Yes	↓ Behind (needs pull)
No	No	✓ Synchronized

To illustrate how this approach works, let us assume that each commit is represented by an integer number. At a giving moment, the local repository of some of the developers from Figure 1 have the commits shown in Table 3.

Table 3 – Existing commits in each repository

Repository	Wolverine	Gambit	Rogue	Nightcrawler	Beast
<i>Commits</i>	10 11	10 11	10 12	10 11 13	10

Considering just the synchronizations presented in Figure 1, which depend on the direction of the arrows, the perception of each developer regarding to his known partners is shown in Table 4. Notice that the perceptions are not symmetric. For instance, as Gambit does not pull updates from Nightcrawler, there is no sense in giving him information regarding Nightcrawler.

Table 4 - Status of each repository based on known remote repositories

Repository	Wolverine	Gambit	Rogue	Nightcrawler	Beast
Wolverine	-	✓	-	-	-
Gambit	✓	-	↑↓	-	↑
Rogue	-	↑↓	-	-	-
Nightcrawler	-	↑	-	-	-
Beast	-	-	↓	↓	-

The main window of DyeVC presents Level 1 information, as shown in Figure 3. Upon a mouse over on a repository, DyeVC informs the number of commits that are ahead or behind in each branch that tracks a remote branch (Level 2 information).

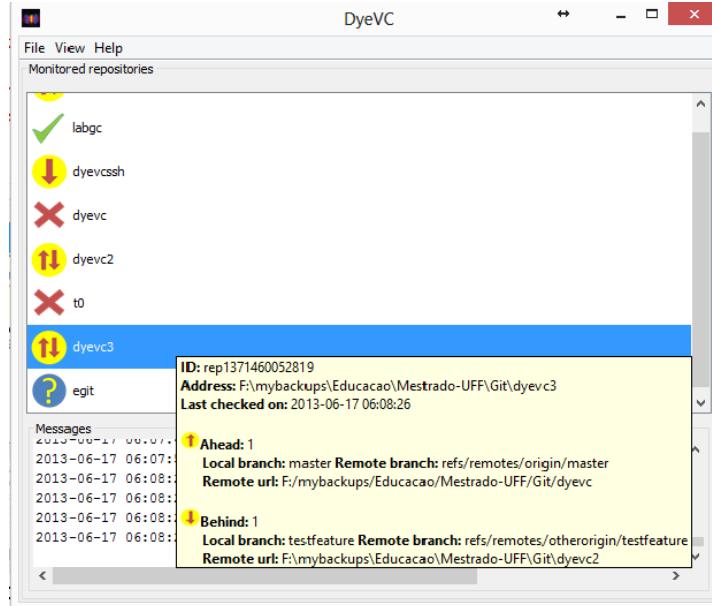


Figure 3 – DyeVC Main Screen

Level 3 information is shown as a visual log of the repository (Figure 4). Each node in the graph represents a commit, and receives a color according with its type, which can be one of: regular node (cyan), branch node (red), merge node (green), branch and merge node (yellow), start node (black) or branch head (last commit in a branch, in dark gray). Nodes are drawn according to its precedence order. Thus, if a commit N is created over a commit N – 1, then commit N will be located in the right hand side of commit N – 1. DyeVC presents a tooltip with some information about each commit, upon a mouse over a node.

The log window can also be zoomed in or out, whether the user wants to see details of a particular area of the log or an overview of the entire history. The line style can be one of cubic curves, straight or quad curves. By changing the window mode from *transforming* to *picking*, it is possible to select a group of nodes and collapse them into one node that represents them, or simply drag them into new positions to have a better understanding of an area where there are too many crossing lines.

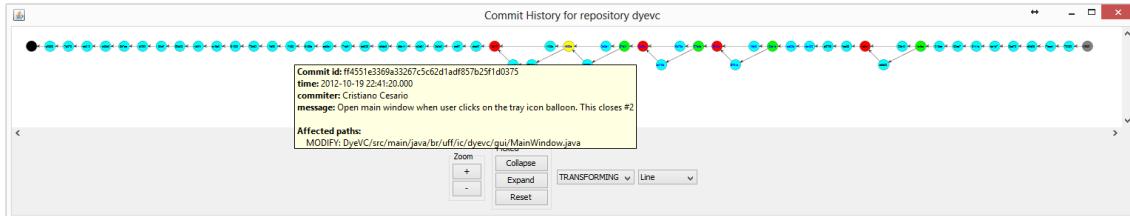


Figure 4 – Log Window (commit history)

4 Related Work

DyeVC relates primarily with studies that aim at improving the perception of developers that work with distributed software development (awareness tools). A recent work by [Steinmacher et al. 2012] presents a systematic review of such studies. We can group these works in two categories: those that notify commit activities (conflict avoidance) and those that detect conflicts (conflict detecting).

Approaches that notify commit activities, such as *SVN Notifier*⁵, *SCM Notifier*⁶, *Commit Monitor*⁷, *SVN Radar*⁸ and *Hg Commit Monitor*⁹ focus on avoiding conflicts by increasing the developer's perception of concurrent work. However, they fail to identify related repositories and do not provide information in different levels of details, such as status, branches, and commits. DyeVC provides these different levels of details, as shown in Section 3.

Approaches that detect conflicts, such as *Polvo* [Santos and Murta 2012] *Palantir*, [Sarma and Van der Hoek 2002], *CollabVS* [Dewan and Hegde 2007], *Crystal* [Brun et al. 2011], *Safe-Commit* [Wloka et al. 2009], *Lighthouse* [Da Silva et al. 2006], and *We-Code* [Guimarães and Silva 2012] not only give the developer awareness of concurrent changes, but also inform if any conflicts were detected. Among these works, only *Crystal* works with DVCSs. It detects physical, syntactic, and semantic conflicts (provided that the user informs the compile and test commands), but does not deal with repositories that pull updates from more than one partner and demands having a Git client installed. On the other hand, *Polvo* presents metrics that quantify merging complexity between involving Subversion branches, which are calculated reactively (upon user request). DyeVC can be seen as a supporting infrastructure that can be combined with such approaches to allow conflicts and metrics analysis over DVCS.

⁵ <http://svnnotifier.tigris.org/> (2012)

⁶ <https://github.com/pocorall/scm-notifier> (2012)

⁷ <http://tools.tortoisessvn.net/CommitMonitor.html> (2013)

⁸ <http://code.google.com/p/svnradar/> (2011)

⁹ <http://www.fsmpl.uni-bayreuth.de/~dun3/hg-commit-monitor> (2009)

5 Conclusions and Future Work

Collaborative software development is a great challenge. If, on the one hand, parallelism in activities brings scale gains, on the other hand it tends to increase the concurrency, causing rework and productivity loss. The proliferation of branches and the distribution of repositories makes it difficult to realize parallel actions made by different developers. In this paper, we presented DyeVC, a tool that identifies the status of a repository in contrast with its partners, which are dynamically found in an unobtrusive way.

A number of research topics arise from this approach. The ability to discover partners of a repository induces that it is possible to find all existing clones of a repository, showing them as a network topology of interrelated nodes that communicate with each other, sending and receiving updates. A global commit history view could be drawn, showing all existing commits in all nodes in the network, allowing one to see which commits exist somewhere, and which ones have not been propagated to all nodes yet. The ability to attach new layouts and filters allows the development of new visualizations, in order to present different metrics and views of the repository (e.g. which repositories or which people changed a specific artifact or group of artifacts, which commits introduced high amount of changes in the code, which branches would cause a conflict if merged, among others).

Acknowledgments

The authors would like to thank CNPq and FAPERJ for the financial support.

References

- Appleton, B., Berczuk, S., Cabrera, R. and Orenstein, R. (Aug 1998). Streamed lines: Branching patterns for parallel software development. In *Proceedings of the 1998 Pattern Languages of Programs Conference*. , PLoP 1998. ACM.
- Brun, Y., Holmes, R., Ernst, M. D. and Notkin, D. (Sep 2011). Proactive detection of collaboration conflicts. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. , ESEC/FSE '11. ACM.
- Cederqvist, P. (2005). *Version Management with CVS*. Free Software Foundation.
- Chacon, S. (2009). *Pro Git*. 1. ed. Berkeley, CA, USA: Apress.
- Collins-Sussman, B., Fitzpatrick, B. W. and Pilato, C. M. (2011). *Version Control with Subversion*. Stanford, CA, USA: .
- Da Silva, I. A., Chen, P. H., Van der Westhuizen, C., Ripley, R. M. and Van der Hoek, A. (Oct 2006). Lighthouse: coordination through emerging design. In *Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange*. , eclipse '06. ACM.
- Dewan, P. and Hegde, R. (Sep 2007). Semi-synchronous conflict detection and resolution in asynchronous software development. In *Proceedings of the 10th European Conference on Computer-Supported Cooperative Work*. , ECSCW 2007. Springer London.
- Eclipse Foundation (Jun 2011). The Open Source Developer Report - 2011 Eclipse Community Survey.

- Eclipse Foundation (Jun 2012). The Open Source Developer Report - 2012 Eclipse Community Survey.
- Estublier, J. (May 2000). Software configuration management: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*. , ICSE '00. ACM.
- Guimarães, M. L. and Silva, A. R. (Jun 2012). Improving early detection of software merge conflicts. In *Proceedings of the 2012 International Conference on Software Engineering*. , ICSE 2012. IEEE Press.
- O'Sullivan, B. (2009a). *Mercurial: The Definitive Guide*. 1. ed. O'Reilly Media.
- O'Sullivan, B. (Sep 2009b). Making sense of revision-control systems. *Communications of the ACM*, v. 52, n. 9, p. 56–62.
- Perry, D. E., Siy, H. P. and Votta, L. G. (Apr 1998). Parallel changes in large scale software development: an observational case study. In *Proceedings of the 20th International Conference on Software engineering*. , ICSE '98. IEEE Computer Society.
- Rochkind, M. J. (Dec 1975). The source code control system. *IEEE Transactions on Software Engineering. (TSE)*, v. 1, n. 4, p. 364–470.
- Santos, R. and Murta, L. G. P. (2012). Evaluating the Branch Merging Effort in Version Control Systems. In *Proceedings of the 26th Brazilian Symposium on Software Engineering (SBES)*. , SBES '12. IEEE Computer Society.
- Sarma, A. and Van der Hoek, A. (Aug 2002). Palantir: coordinating distributed work-spaces. In *Computer Software and Applications Conference, 2002. COMPSAC 2002. Proceedings. 26th Annual International*.
- Steinmacher, I., Chaves, A. and Gerosa, M. (May 2012). Awareness Support in Distributed Software Development: A Systematic Review and Mapping of the Literature. *Computer Supported Cooperative Work (CSCW)*, p. 1–46.
- Tichy, W. (1985). RCS: A system for version control. *Software - Practice and Experience*, v. 15, n. 7, p. 637–654.
- Walrad, C. and Strom, D. (Sep 2002). The importance of branching models in SCM. *Computer*, v. 35, n. 9, p. 31 – 38.
- Wloka, J., Ryder, B., Tip, F. and Ren, X. (May 2009). Safe-commit analysis to facilitate team software development. In *Proceedings of the 31st International Conference on Software Engineering*. , ICSE '09. IEEE Computer Society.

Tipar ou não tipar? Compreendendo quais Fatores Influenciam a Escolha por um Sistema de Tipos

Carlos Souza, Eduardo Figueiredo, Marco Túlio Oliveira Valente

¹Departamento de Ciência da Computação, UFMG, Brasil

carlos.garcia@dcc.ufmg.br, figueiredo@dcc.ufmg.br, mtov@dcc.ufmg.br

Abstract. One of the most important features to be taken into account when choosing a programming language is its typing system, static or dynamic. This question has become increasingly more important due to the recent popularization of dynamic languages such as Ruby and JavaScript. This paper studies which are the most influencing factors for a programmer when choosing between typing systems. An analysis of the source code of over a thousand projects written in Groovy, a programming language where one can choose, for each declaration, either to type it or not, shows in which situations programmers prefer a typing system over the other. Results of this study suggest that the previous experience of the programmer, project size, complexity of modules, scope and visibility of statements are some of the most important factors in this decision.

Resumo. Uma das características mais importantes a serem consideradas na hora de se escolher uma linguagem de programação é o seu sistema de tipos, estático ou dinâmico. Essa questão tem se tornado cada vez mais importante graças à popularização recente de linguagens dinamicamente tipadas como Ruby e JavaScript. Este trabalho apresenta um estudo sobre quais fatores mais influenciam a escolha de um programador por um sistema de tipos. Uma análise do código fonte de mais mil projetos escritos em Groovy, uma linguagem de programação onde se pode escolher, para cada declaração, usar tipos ou não, permite visualizar em quais situações um sistema de tipos é preferido. Resultados deste estudo apontam que a experiência prévia do programador, tamanho do projeto, complexidade dos módulos, escopo e visibilidade das declarações são alguns dos fatores mais importantes para essa decisão.

1. Introdução

Ao se escolher uma linguagem de programação para um projeto, um desenvolvedor deve considerar algumas características desta linguagem, sendo uma das mais importantes o sistema de tipos. Este, que pode ser estático ou dinâmico, define em que momento o tipo de uma declaração deve ser definido [Pierce 2002]. Declarações em linguagens com tipagem estática, como Java e C#, devem ser acompanhadas pela definição de um tipo, que pode ser usado pelo compilador para checar a corretude do código. Já em linguagens dinamicamente tipadas, como Ruby e JavaScript, a definição do tipo só é realizada em tempo de execução.

A discussão sobre qual sistema de tipos é melhor tem se tornado cada vez mais relevante nos últimos anos graças à rápida popularização de linguagens dinamicamente tipadas. De acordo com o TIOBE Programming Community Index[[tio](#)], um conhecido

ranking que mede a popularidade de linguagens de programação, 27% das linguagens de programação adotadas na indústria possuem tipagem dinâmica. Em 2001, esse número era de apenas 17%. Entre as 10 linguagens no topo do ranking, 4 possuem sistemas de tipos dinâmicos: JavaScript, Perl, Python e PHP. Em 1998, nenhuma dessas linguagens estava entre as 10 primeiras do ranking.

Diversos fatores podem ser considerados na escolha por uma linguagem de programação com sistema de tipos dinâmico ou estático. Linguagens com tipagem dinâmica, por serem mais simples, permitem que programadores executem suas tarefas de desenvolvimento mais rapidamente [Pierce 2002]. Ainda, ao removerem o trabalho de declarar os tipos das variáveis, estas linguagens permitem que seus usuários foquem no problema a ser resolvido, ao invés de se preocuparem com as regras da linguagem [Tratt 2009].

Por outro lado, sistemas de tipos estáticos, também possuem suas vantagens. Estes conseguem prevenir erros de tipo em tempo de compilação [Lamport and Paulson 1999]. Declarações de tipos aumentam a manutenibilidade de sistemas pois estas atuam como documentação do código, informando ao programador sobre a natureza de cada variável [Cardelli 1996, Mayer et al. 2012]. Sistemas escritos a partir destas linguagens tendem a ser mais eficientes, uma vez que não precisam realizar checagem de tipo durante sua execução [Bruce 2002, Chang et al. 2011]. Por fim, ambientes de desenvolvimento modernos, tais como Eclipse e IDEA, quando possuem conhecimento sobre o tipo de uma declaração, são capazes de auxiliar o programador através de funcionalidades como documentação e complemento de código [Bruch et al. 2009].

Este artigo apresenta um estudo com o objetivo de entender quais dos fatores descritos acima influenciam de fato a escolha de um programador por tipar ou não as suas declarações. A fim de obter resultados confiáveis, essa questão foi estudada tendo como base código desenvolvido por programadores no contexto de suas atividades cotidianas através da análise de uma massa de dados composta por mais de mil projetos. Esses projetos foram escritos em Groovy, uma linguagem com sistema de tipos híbrido, que permite escolher, para cada declaração, tipá-la ou não. Assim, através de uma análise estática dessa massa de dados, é possível visualizar quando programadores escolhem cada sistema de tipos e, a partir daí, entender quais são os fatores que influenciam essa decisão.

O restante deste artigo está organizado da seguinte forma. A seção 2 introduz os principais conceitos da linguagem de programação Groovy. As seções 3 e 4 descrevem a configuração do estudo e seus resultados. Ameaças à validade deste trabalho são discutidas na seção 5 enquanto alguns trabalhos relacionados são apresentados na seção 6. Por fim, a seção 7 conclui este trabalho e levanta alguns trabalhos futuros.

2. A Linguagem Groovy

Groovy é uma linguagem de programação orientada a objetos projetada para ser executada sobre a plataforma Java, mas com características dinâmicas semelhantes às de Ruby e Python. Sua adoção tem crescido de maneira notável nos últimos anos e, apesar de ter sido lançada há apenas 6 anos, Groovy já é a 36ª linguagem mais popular da indústria de software [tio].

Em Groovy, um programador pode escolher tipar suas declarações ou não. Tipagem estática e dinâmica podem ser combinadas no mesmo código livremente. No algo-

ritmo 1, por exemplo, o tipo do retorno do método é definido, enquanto os parâmetros e a variável local são tipados dinamicamente.

Algoritmo 1 Um método escrito em Groovy

```
1. Integer add(a, b) {  
2.     def c = a + b  
3.     return c  
4. }
```

A maior parte da linguagem Java também é válida em Groovy e código Groovy pode interagir diretamente com código Java e vice-versa. Esses fatores tem atraído um grande número de programadores Java que desejam utilizar suas funcionalidades sem ter que aprender uma linguagem completamente diferente ou mudar a plataforma de execução de seus sistemas. Outros recursos interessantes de Groovy são suporte nativo a coleções, possibilidade de se escrever scripts e metaprogramação.

3. Configuração do Estudo

Este trabalho analisa em quais declarações programadores preferem utilizar tipagem estática ou dinâmica a fim de entender quais fatores influenciam nessa escolha. Abaixo são descritos a massa de dados e o analisador estático usados para tal.

3.1. Massa de Dados

Os projetos utilizados neste estudo foram obtidos do GitHub, um serviço de controle de versão baseado em Git. Utilizando a API do GitHub, foi possível obter o código fonte de quase dois mil projetos Groovy. Após descartar projetos privados e duplicados, restaram 1112 projetos com um total de 1,67 milhões de linhas de código, considerando apenas a última versão de cada projeto. A distribuição do tamanho destes projetos é mostrada na figura 1. Estes sistemas foram desenvolvidos por um total de 926 programadores.

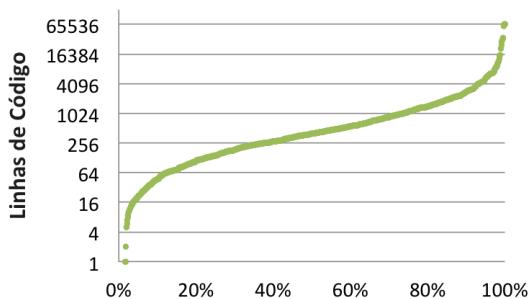


Figura 1. Distribuição do tamanho dos projetos

3.2. Analisador de código estático

O analisador de código estático utilizado neste trabalho é baseado na biblioteca de metaprogramação de Groovy. Com esta biblioteca, é possível criar uma árvore sintática abstrata(AST) a partir de código fonte utilizando uma das diversas fases do próprio compilador de Groovy. A fase escolhida foi a de conversão, que possui informação suficiente

para determinar o sistema de tipos de cada declaração. Essa fase acontece antes do compilador tentar resolver quaisquer dependências externas, tornando possível analisar cada arquivo separadamente sem que seja necessário compilar o projeto.

Os seguintes tipos de declarações podem ser analisados

- Retorno de Métodos
- Parâmetros de Métodos
- Parâmetros de Construtores
- Campos
- Variáveis Locais

Para cada item listado acima, é possível obter ainda as seguintes informações

- A declaração é parte de um script ou de uma classe?
- A declaração é parte de uma classe de testes?
- Visibilidade (exceto para variáveis locais)

4. Resultados

A seguir são apresentados os resultados deste trabalho.

4.1. Resultado Geral

Cerca de 60% das declarações são estaticamente tipadas, enquanto apenas 40% destas são dinamicamente tipadas. Dado que grande parte dos programadores Groovy eram previamente programadores Java e, portanto, estavam acostumados com tipagem estática, este resultado sugere que a experiência prévia de um programador é um fator importante na escolha do sistema de tipos.

4.2. Resultados por Tipo de Declaração

A figura 2 mostra que tipagem dinâmica é utilizada em declarações de variáveis locais com muito mais frequência que em outros tipos de declaração. Dado que variáveis locais possuem menor escopo e menor ciclo de vida, programadores provavelmente sentem menor necessidade de documentá-las através da definição de tipos e acabam optando pela maneira mais direta de declará-las.

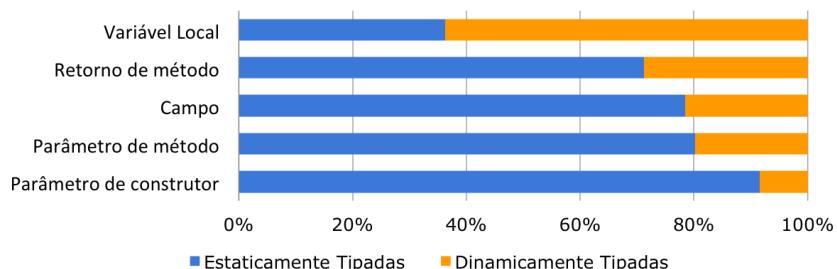


Figura 2. Sistemas de Tipo por Tipo de Declaração.

Por outro lado, parâmetros de construtores são o tipo de declaração mais frequentemente tipado. Pode se argumentar que há uma preocupação grande em tipar (e documentar) construtores uma vez que estes são importantes elementos da definição do contrato de um módulo.

4.3. Resultados por Visibilidade

De acordo com a figura 3, declarações com visibilidade pública ou protegida são as que, com mais frequência, utilizam tipagem estática. Essas são as declarações que definem a interface de um módulo e, ao tipá-las, programadores permitem que o compilador procure por erros de tipos na integração com outros módulos além de documentar o contrato deste módulo para que clientes saibam como utilizá-lo.

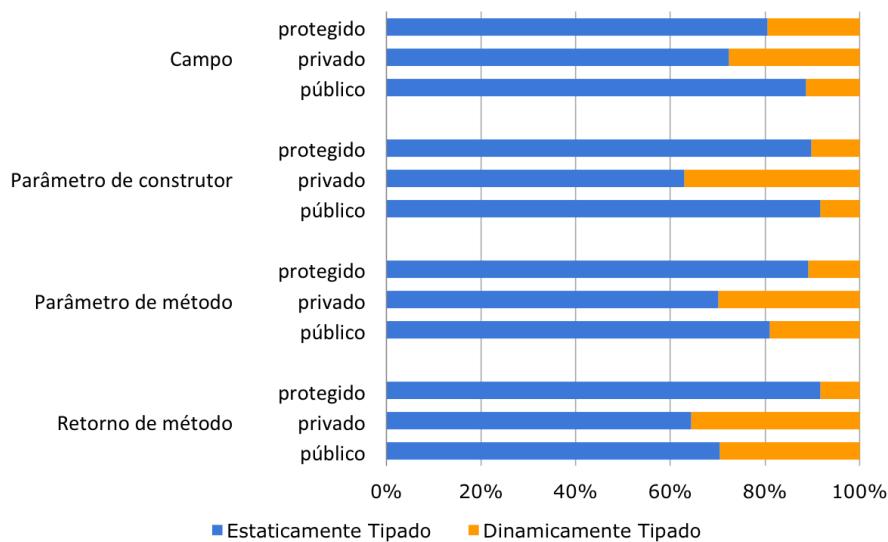


Figura 3. Sistemas de Tipo por Visibilidade da Declaração

No caso de retorno e parâmetros de métodos, o uso de tipos em declarações com visibilidade protegida chega a superar o de declarações com visibilidade pública. Pode-se argumentar que métodos e campos protegidos estabelecem um contrato delicado, já que expõem elementos internos de uma superclasse para uma subclasse. Aparentemente programadores enxergam a necessidade de documentar bem o código que define esse tipo de contrato através do uso tipagem estática.

4.4. Resultados por Tamanho de Projeto

A figura 4 mostra o uso de tipagem dinâmica em declarações públicas por tamanho de projeto. Cada barra desse gráfico representa um grupo de projetos agrupado por seu tamanho sendo que os limites de cada grupo são definidos sob cada barra. Por exemplo, um projeto com 1500 linhas de código se encontra na segunda barra, pois 1500 se encontra dentro do intervalo]400, 1600].

Este resultado mostra que o uso de tipagem dinâmica em declarações públicas diminui à medida que o tamanho do projeto aumenta. Projetos com mais de 6400 linhas de código usam tipagem dinâmica com praticamente metade de frequência que projetos menores. Intuitivamente, quanto maior o projeto, maior a dificuldade de integração e a necessidade de manutenção, o que pode levar programadores a preferirem o uso de tipagem estática em elementos públicos, os mais críticos para esse contexto. Nenhum padrão pode ser observado em outros tipos de declaração, reforçando a idéia de que esse padrão está relacionado ao papel de elementos públicos em projetos grandes.

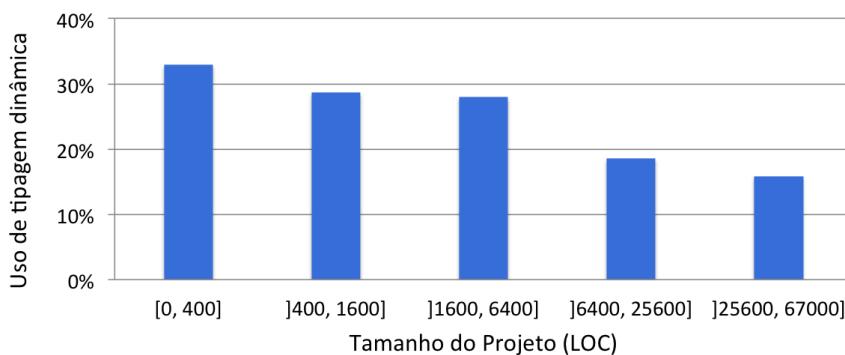


Figura 4. Sistemas de tipo de retornos e parâmetros de métodos públicos agrupados por tamanho de projeto.

4.5. Scripts e Testes

Scripts são, em geral, escritos para desempenhar tarefas simples e não se relacionam com muitos outros módulos. O mesmo pode ser dito a respeito de código de teste. Isso leva a crer que tipagem dinâmica seria utilizada com mais frequência nesses contextos uma vez que manutenibilidade e integração não são fatores críticos. O resultado da tabela 1 porem contradiz essa intuição mostrando que não há diferença significativa entre o perfil de uso dos sistemas de tipos nesses contextos.

Tabela 1. Todos os tipos de declarações agrupadas por classe/script e classes de teste/classes funcionais

	Tipagem Estática	Tipagem Dinâmica
Todas as Classes	61%	39%
Scripts	54%	46%
Classes Funcionais	62%	38%
Classes de Teste	57%	43%

5. Ameaças à validade

Como levantado na seção 4.1, programadores tendem a continuar usando o sistema de tipos com o qual já estão acostumados. Dado que grande parte dos programadores Groovy possuem experiência prévia com Java, uma linguagem estaticamente tipada, os resultados mostrados neste trabalho podem apresentar certa tendência ao uso de tipagem estática. Apesar disso, a análise por tipo de declaração apresentada na seção 4.2 mostra a predominância de tipagem dinâmica em variáveis locais, indicando que, apesar da experiência anterior com Java, programadores são capazes de aprender a usar tipagem dinâmica onde julgam necessário.

Alguns arcabouços impõem o uso de um dado sistema de tipos em certas situações. Spock, por exemplo, um arcabouço de testes automatizados, requer que o retorno de métodos que implementam testes seja dinamicamente tipado. Porem, graças à heterogeneidade e ao grande número de projetos analisados, acredita-se que não haja nenhum arcabouço com utilização tão extensa a ponto de influenciar os resultados gerais.

6. Trabalhos Relacionados

Há alguns trabalhos que realizam essa comparação através de estudos controlados. Em [Hanenberg 2010], o autor compara o desempenho de dois grupos de estudantes quando instruídos a desenvolver dois pequenos sistemas. Ambos os grupos utilizaram uma linguagem desenvolvida pelo autor, Purity, sendo que a única diferença entre eles é que um grupo utiliza uma versão desta linguagem com tipagem estática enquanto o outro utilizou uma versão com tipagem dinâmica. Resultados mostraram que o grupo utilizando a versão dinâmica foi significativamente mais produtivo. Assim como neste trabalho, o autor foi capaz de comparar dois sistemas de tipos diretamente, neste caso desenvolvendo sua própria linguagem. Porem, pode-se argumentar que esses resultados podem não representar bem situações do cotidiano da indústria de software, uma vez que esse foi um estudo de pequena duração onde estudantes são utilizados como exemplos de desenvolvedores e que, ainda, não possuem nenhum tipo de interação com outros programadores. Neste trabalho, tenta-se conseguir resultados mais relevantes ao analisar código fonte desenvolvido por programadores durante suas atividades cotidianas.

Em uma continuação do estudo acima [Kleinschmager et al. 2012], os autores chegaram a conclusões opostas. Eles compararam o desempenho de dois grupos de desenvolvedores em tarefas de manutenção, um utilizado Java, uma linguagem estaticamente tipada, e o outro, Groovy, usado de forma a simular uma versão de Java dinamicamente tipada. Nesse caso, o grupo utilizando Java, a linguagem estaticamente tipada, foi muito mais produtivo. Essa contradição reforça o argumento que os resultados de estudos controlados podem não ser confiáveis para analisar essa questão.

Em experimentos conduzidos em [Daly et al. 2009], os autores compararam o desempenho de dois grupos trabalhando em pequenas tarefas de desenvolvimento. Um grupo utilizou Ruby, uma linguagem dinamicamente tipada, enquanto o outro usou DRuby, uma versão estaticamente tipada de Ruby. Resultados mostraram que o compilador de DRuby raramente conseguiu capturar erros que já não eram evidentes para os programadores. A maior parte dos envolvidos no estudo tinha experiência prévia com Ruby, o que leva a crer que programadores se acostumam com a falta de tipagem estática em suas declarações.

7. Conclusão e Trabalhos Futuros

Este trabalho estuda quais são os fatores mais importantes para a escolha de um sistema de tipos estático ou dinâmico. Existem trabalhos na literatura que analisam as vantagens de cada um através de estudos controlados. Os resultados apresentados aqui, porem, mostram quais são os fatores que de fato influenciam essa decisão através da mineração de um amplo conjunto de repositórios de software e da visualização do uso destes sistemas de tipos.

Quando a necessidade de manutenção e a complexidade de integração entre módulos são questões importantes, tipagem estática aparentemente é preferida por programadores Groovy. Nessas situações a integração com ferramentas de desenvolvimento e a documentação do código oferecidas pelo sistema de tipos estático são vantagens importantes consideradas por programadores. Por outro lado, quando essas questões não são tão críticas, a simplicidade de tipagem dinâmica parece ser preferida, como visto com

declarações de variáveis locais. Outro fator importante é a experiência prévia de programadores com um dado sistema de tipos.

Em trabalhos futuros deseja-se analisar a influência dos sistemas de tipos estático e dinâmico sobre a robustez de sistemas de software. Em particular, deseja-se entender se o uso tipagem dinâmica, que limita a capacidade do compilador em descobrir problemas de tipo, possui alguma correlação com a ocorrência de defeitos no sistema e se o emprego de testes automatizados é capaz de diminuir essa correlação.

Agradecimentos

Este trabalho recebeu apoio financeiro da FAPEMIG, processos APQ-02376-11 e APQ-02532-12, e do CNPq processo 485235/2011-0.

Referências

- Tiobe programming community index. <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>. Acessado em 23/06/2013.
- Bruce, K. (2002). *Foundations of object-oriented languages: types and semantics*. MIT press.
- Bruch, M., Monperrus, M., and Mezini, M. (2009). Learning from examples to improve code completion systems. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 213–222. ACM.
- Cardelli, L. (1996). Type systems. *ACM Comput. Surv.*, 28(1):263–264.
- Chang, M., Mathiske, B., Smith, E., Chaudhuri, A., Gal, A., Bebenita, M., Wimmer, C., and Franz, M. (2011). The impact of optional type information on jit compilation of dynamically typed languages. *SIGPLAN Not.*, 47(2):13–24.
- Daly, M. T., Sazawal, V., and Foster, J. S. (2009). Work In Progress: an Empirical Study of Static Typing in Ruby. In *Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU)*, Orlando, Florida.
- Hanenberg, S. (2010). An experiment about static and dynamic type systems: doubts about the positive impact of static type systems on development time. *SIGPLAN Not.*, 45(10):22–35.
- Kleinschmager, S., Hanenberg, S., Robbes, R., and Stefik, A. (2012). Do static type systems improve the maintainability of software systems? An empirical study. *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, pages 153–162.
- Lamport, L. and Paulson, L. C. (1999). Should your specification language be typed. *ACM Trans. Program. Lang. Syst.*, 21(3):502–526.
- Mayer, C., Hanenberg, S., Robbes, R., Tanter, É., and Stefik, A. (2012). Static type systems (sometimes) have a positive impact on the usability of undocumented software: An empirical evaluation. *self*, 18:5.
- Pierce, B. (2002). *Types and programming languages*. MIT press.
- Tratt, L. (2009). Chapter 5 dynamically typed languages. volume 77 of *Advances in Computers*, pages 149 – 184. Elsevier.

Verificação de Conformidade Arquitetural com Testes de Design - Um Estudo de Caso

Izabela Melo¹, João Brunet¹, Dalton Serey¹, Jorge Figueiredo¹

¹Laboratório de Práticas de Software

Departamento de Sistemas e Computação

Universidade Federal de Campina Grande (UFCG)

Campina Grande – PB – Brasil

izabela@copin.ufcg.edu.br; (jarthur,dalton,abrantes)@dsc.ufcg.edu.br

Abstract. *Architectural conformance checking has been acknowledged as a key technique to assure that an implementation of a system conforms to its planned architecture. In this paper, we describe an experience of the application of design tests, an automatic approach to architectural conformance checking, in a real setting. The approach efficiently revealed divergences among documented, planned and implemented architectural rules. It also revealed that historically accumulated architectural violations, which could be easily dealt with at the moment they appeared, are difficult to address, due to the functional stability of the system.*

Resumo. *A verificação de conformidade arquitetural tem sido reconhecida como uma das principais técnicas para garantir que a implementação de um sistema de software atenda às regras de design e de arquitetura planejadas. Este artigo descreve a experiência de aplicar testes de design, uma proposta de abordagem automática para a verificação de conformidade arquitetural, em um projeto real. A abordagem revelou com eficiência as divergências entre as regras arquiteturais documentadas, planejadas e implementadas. E que as violações arquiteturais acumuladas historicamente, que seriam de fácil resolução no momento em que surgiram, são de difícil resolução face à estabilidade funcional do sistema.*

1. Introdução

À medida em que o software é desenvolvido, se faz necessário desempenhar atividades que monitorem e controlem essa evolução para que o produto final esteja de acordo com suas especificações. Em particular, um dos desafios enfrentados durante a evolução e manutenção de software é verificar se sua estrutura está de acordo com as regras arquiteturais previamente estabelecidas – verificação arquitetural de software [CLEMENTS et al. 2003]. Essa atividade é de extrema importância no contexto de desenvolvimento de software, pois é através dela que a equipe detecta violações arquiteturais, isto é, divergências entre a arquitetura planejada e a implementada.

A detecção prévia de violações arquiteturais pode capacitar a equipe a reduzir os efeitos da erosão arquitetural de software, fenômeno causado pelo acúmulo de violações arquiteturais [PERRY and WOLF 1992] ao longo do tempo. Entre outros efeitos nocivos provenientes da erosão arquitetural estão a perda de estrutura do software, que impacta consideravelmente na sua complexidade e dificuldade de manutenção [GURP and BOSCH 2002][BROOKS 1975].

Este trabalho relata a experiência da aplicação de verificação arquitetural em um sistema real intitulado e-Pol – Sistema de Gestão das Informações de Polícia Judiciária,

desenvolvido em parceria pela Polícia Federal do Brasil e a Universidade Federal de Campina Grande. Apesar de haver outras formas de realizar a verificação arquitetural, como a linguagem DCL (Dependency Constraint Language), optamos por utilizar Testes de Design [BRUNET et al. 2009][BRUNET et al. 2011], uma abordagem baseada em testes que permite a escrita e a verificação automática de conformidade entre regras arquiteturais e uma dada implementação. Essa escolha foi apoiada na praticidade da abordagem por combinar a prática de verificação arquitetural à de testes automáticos. Isso permite integrar os procedimentos de verificação arquitetural às suítes de testes funcionais automatizados, sem a adição de nenhum outro ferramental ou processo.

O processo consistiu em coletar as regras arquiteturais junto à equipe de desenvolvimento, implementá-las na forma de testes de design e executar os testes. Ao final, analisamos e discutimos as violações arquiteturais junto à líder da equipe. Como era esperado, um número significativo de violações arquiteturais foi encontrado — 12 das 22 regras arquiteturais planejadas não são respeitadas na implementação. Este resultado corrobora outros anteriormente publicados [BRUNET et al. 2011] e pode ser explicado pela cultura em relação às regras de design e de arquitetura. Decisões arquiteturais não são adequadamente documentadas nem compartilhadas entre os desenvolvedores. Ao longo do tempo, tendem a ser esquecidas o que aumenta a chance de serem desrespeitadas. E a inexistência de ferramentas de apoio à verificação piora ainda mais o cenário, por dificultar a imediata detecção de violações durante o desenvolvimento e evolução do sistema.

Este artigo está organizado da seguinte maneira. A Seção 2 apresenta os conceitos fundamentais de verificação arquitetural e testes de design. A Seção 3 apresenta o estudo realizado e as lições aprendidas. A Seção 4 discute trabalhos relacionados. Por fim, a Seção 5 apresenta nossas conclusões.

2. Fundamentação Teórica

2.1. Arquitetura como conjunto de regras arquiteturais

Devido à sua importância na indústria e academia, o conceito de arquitetura de software tornou-se um tópico de interesse em Engenharia de Software. Nos últimos 20 anos, particularmente, vários pesquisadores se esforçaram em produzir uma definição para esse conceito [CLEMENTS et al. 2003, BUDGEN 2003, GARLAN and PERRY 1995, BASS et al. 2003, JANSEN and BOSCH 2005]. Essas definições abordam, por exemplo, aspectos dinâmicos, estáticos e de fluxo de dados do software. Embora a literatura conteña muitos trabalhos nesse sentido, é possível encontrar um denominador comum nessas definições – o fato da arquitetura especificar os componentes de um software e como eles devem se comunicar [BASS et al. 2003].

Nesse contexto, uma arquitetura de software pode ser vista como um conjunto de decisões/regras arquiteturais que estabelecem relações entre os componentes de uma aplicação [JANSEN and BOSCH 2005]. Por exemplo, suponha um sistema em que o arquiteto utilizou camadas para separar a apresentação, da lógica de negócios e os objetos de acesso aos dados. Um exemplo de regra arquitetural nesse sistema é o fato da camada de apresentação não poder depender diretamente dos objetos de acesso aos dados.

2.2. Testes de design

Teste de design é uma abordagem de verificação arquitetural. Através de testes de design, o desenvolvedor/arquiteto é capaz de especificar regras arquiteturais em formato de testes

automatizados e verificar se a implementação está de acordo com essas regras. Por exemplo, vamos novamente considerar a restrição de acesso entre o componente da camada de apresentação (e.g. pacote GUI) e o componente de acesso a dados (e.g. pacote DAO). O Algoritmo 1 ilustra o teste de design para essa regra.

Como podemos observar no Algoritmo 1, o teste consiste em obter informações a respeito dos módulos acessados por GUI e verificar se DAO não pertence a esse conjunto de módulos. Em tempo, um módulo é uma agregação de entidades, sejam elas classes, pacotes ou interfaces.

Algoritmo 1. Pseudocódigo da implementação de um teste de design

```

1 componenteGUI = getPackage("GUI")
2 componenteDAO = getPackage("DAO")
3 calledPackages = componenteGUI.getcalleePackages()
4 assertFalse(calleePackages.contains(componenteDAO))

```

Testes de design detectam violações arquiteturais. Por exemplo, o teste do Algoritmo 1, quando executado tendo como sistema a ser verificado o exemplo do Algoritmo 2 detectaria uma violação arquitetural na linha 4. Uma violação arquitetural pode ser representada como uma tripla (*caller, callee, type*), em que *caller* indica o módulo que viola uma regra arquitetural, *callee*, o módulo que é usado pelo caller, e *type* o tipo da violação (chamada a método, acesso a variável, generalização, etc). A violação apresentada no Algoritmo 2, por exemplo, pode ser representada por: (*GUI, DAO, "chamada de método"*)

Algoritmo 2. Pseudocódigo de uma violação arquitetural

```

1 package gui;
2 class MainWindow() {
3     show() {
4         DAO.getData();
5     }
6 }
7
8 package dao;
9 class DAO() {
10    getData() {...}
11 }

```

3. Estudo de Caso: Aplicação de testes de design

Neste estudo relatamos a experiência da aplicação de verificação arquitetural baseada em testes de design em um sistema real. O sistema em questão é o e-Pol – Sistema de Gestão das Informações de Polícia Judiciária, que vem sendo desenvolvido em parceria pela Polícia Federal do Brasil e a Universidade Federal de Campina Grande desde 2010. O projeto visa desenvolver uma plataforma integrada de investigação e inquérito para a Polícia Federal. No atual estágio, o sistema consiste em aproximadamente 57.000 linhas de código, 539 classes e 31 pacotes. Atualmente, conta com 10 desenvolvedores, embora um total de 41 tenham atuado no projeto, desde seu início. O processo de desenvolvimento do e-Pol é baseado em métodos ágeis. Nesse sentido, o processo adotado foca pouco em atividades de documentação e de verificação arquitetural.

O estudo realizado é de natureza exploratória em que procuramos identificar questões de pesquisa, mais que respondê-las.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1 GUI					x	x	x	x	x	x	x	x	x	*
2 Action	*		x	x	x				x	x	x	x	x	
3 Converter	x	x	x	x	x	x	x	x	x	x	x	x	x	x
4 Validator	x	x	x	x	x	x	x	x	x	x	x	x	x	x
5 Listagem	x	x	x	x	x	x			x	x	x	x	x	
6 Cenario	x	x	x	x	x	x			x	x	x	x	x	
7 DAO	x	x	x	x	x	x		x	x	x	x	x	x	*
8 Session	x	x	x	x	x	x		x	x	x	x	x	x	
9 Util	x	x	x	x	x	x	x	x	x	x	x	x	x	
10 Test	x													
11 Test.unit	x	x			*	x								
12 Test.integration	x													
13 Log	x	x	x	x	x	x	x	x	x	x	x	x	x	x
14 Model	x	x	x	x	x	x	x	x	*	x	x	x	x	x

Figura 1. DSM dos relacionamentos entre os módulos do e-Pol.

3.1. Metodologia

A metodologia utilizada para conduzir o estudo foi dividida em três etapas. Na primeira – Documentação arquitetural – coletamos as regras arquiteturais do projeto. Produzimos: 1) uma lista dos módulos em que consiste o projeto; 2) o mapeamento entre os módulos e os pacotes/classes da implementação; e 3) as regras arquiteturais planejadas, expressas na forma de relacionamentos não admissíveis entre módulos. Os dados foram obtidos através de entrevistas com a líder de desenvolvimento do projeto.

Na segunda – Verificação arquitetural – implementamos as regras arquiteturais na forma de testes de design e executamos os testes para verificar a conformidade arquitetural. A implementação dos testes foi feita por um dos autores deste artigo em concordância com a líder de projeto. Foi produzida uma lista de violações arquiteturais para cada uma das regras identificadas.

Por fim, na terceira etapa – Análise das violações – analisamos as violações arquiteturais do projeto junto à líder da equipe. Embora o método usado tenha sido em boa parte *ad-hoc*, de modo geral, o objetivo era o de confrontar a líder de desenvolvimento com as violações. Para cada violação identificada, pedimos à líder que avaliasse, confirmasse e explicasse.

3.2. Resultados

Documentação arquitetural O primeiro passo consistiu em identificar os módulos do sistema. Embora os 12 módulos que compõem o sistema tenham sido identificados facilmente, o mapeamento das entidades do código aos módulos não foi tarefa fácil. Em diversos momentos, percebemos que houve dúvidas por parte da líder em relação ao módulo a que pertencem várias das entidades. Em particular, a divisão em pacotes do sistema não reflete bem a decomposição em módulos identificada. Há vários casos em que as classes de um mesmo pacote foram mapeadas para diferentes módulos. A tarefa, contudo, claramente obrigou à reflexão e crítica da arquitetura do sistema, por parte da líder de desenvolvimento.

Uma vez coletados os módulos que compõem o sistema, coletamos as restrições arquiteturais planejadas. A DSM apresentada na Figura 1 sumariza as restrições arquiteturais identificadas. Na DSM, cada “x” representa um relacionamento proibido – o módulo da linha não pode depender do módulo da coluna. E cada “*” representa uma proibição parcial – existem exceções para os quais a dependência é permitida.

A atividade seguinte foi organizar a informação na DSM em regras arquiteturais. Identificamos um total de 22 regras, das quais 20 são do tipo *módulo A não pode depender*

do módulo B. As outras duas regras são semelhantes, mas de menor granularidade, e proibem dependências entre conjuntos de entidades específicas contidas pelos módulos (classes ou mesmo métodos). São, portanto, regras do tipo *módulo A não pode depender dos métodos M₁, M₂, ..., M_n*.

Verificação arquitetural Para cada uma das regras arquiteturais identificadas, implementamos os testes de design correspondentes, utilizando o DesignWizard [BRUNET et al. 2009] e o JUnit. A título de exemplo, o Algoritmo 3 corresponde ao pseudocódigo do teste de design para a regra *nenhum módulo pode ter acesso direto ao módulo Test*. Os testes de design foram executados na última versão estável do e-Pol e as violações arquiteturais detectadas foram armazenadas. No total, foram identificadas 298 violações arquiteturais referentes a 12 das regras. A implementação, portanto, está em conformidade com apenas 10 das 22 regras (45%) coletadas. A distribuição das violações foi bastante concentrada tanto por regra (160, 53, 31, 17, 12, 5, 5, 5, 4, 3, 2, 1), quanto por módulo em que ocorre (160, 54, 37, 17, 12, 7, 6, 5, 0, 0, 0, 0). A semelhança dos vetores de distribuição é consequência da lógica de formação das regras: cada regra trata, praticamente, das restrições de um módulo. A título de exemplo, o processo de verificação revelou que o módulo Model, cujas regras estabeleciam que *Model não pode depender de nenhum outro módulo*, acumula um total de 54 violações.

Algoritmo 3. Pseudocódigo de um teste de design no e-Pol

```

1  testaAcessoAoModuloTeste () {
2
3      moduloTeste = DesignWizard .getPackage ("Test");
4      classesModuloTeste = moduloTeste .getAllClasses ();
5
6      for (classNode in classesModuloTeste) {
7          for (caller in classNode .getCallerClasses ()) {
8              Assert .assertTrue ("Ha violacao",
9                  caller .getPackage () .equals (moduloTeste));
10         }
11     }
12 }
```

Análise das violações O relatório de violações identificadas foi analisado em conjunto com a líder de projeto. Como se havia antecipado, a líder já esperava que algumas violações fossem encontradas, dada a natureza ágil do processo de desenvolvimento e à indisponibilidade de ferramentas para apoiar o processo de verificação arquitetural. O número de violações encontrado, contudo, ainda foi surpreendente. Todas as violações mencionadas foram confirmadas¹ pela líder de projeto. Houve uma ressalva, contudo, em relação às violações dos módulos Model e GUI. Segundo a líder, um número significativo dessas violações são decorrentes de mudanças em decisões arquiteturais ao longo do tempo que implicaram no acúmulo de violações não corrigidas. Logo, embora não fossem violações às regras arquiteturais vigentes no momento do desenvolvimento, são violações à arquitetura atual. Há, portanto, um débito arquitetural, embora as razões pelas quais

¹De fato, o processo de coleta foi iterativo. Apresentamos à líder de desenvolvimento, relatórios preliminares de violações, a partir dos quais houve a oportunidade de refletir sobre as regras arquiteturais e reescrevê-las ou aperfeiçoá-las.

tenha sido criado sejam compreensíveis. A correção das violações, contudo, pode ter um alto custo para o projeto, dado que, do ponto de vista funcional, o sistema se encontra estável. Trata-se, portanto, da identificação da necessidade de um significativo processo de refatoramento.

Um aspecto que chamou a atenção é a distribuição concentrada das violações em determinados módulos e/ou regras – 71% das violações ocorrem em apenas dois módulos do sistema (GUI e Action), sendo que 53% delas estão em um único módulo (GUI). O módulo GUI é responsável pela apresentação do sistema para o usuário e, portanto, precisa ter acesso a várias entidades. Por vezes, a divisão entre entidades possíveis de serem acessadas e entidades proibidas pode não ficar tão clara para os desenvolvedores, acarretando em violações arquiteturais. O módulo Action, por sua vez, é responsável pelas regras de negócio e pela disponibilização de informações para a camada de apresentação. Logo, o módulo Action separa a camada de visão da camada de modelo. Claramente, ter testes de design incorporados às suítes de testes automáticos do sistema permitiria evitar a adição de novas violações durante a evolução do sistema. Para o caso de haver explicações plausíveis para aceitar a violação, os testes permitem manter sob controle as situações em que as regras serão ignoradas. Em particular, a simplicidade do uso de testes de design foi percebida pela própria líder de desenvolvimento como um motivo para adotar o método para o controle e a verificação arquiteturais.

3.3. Lições aprendidas

Duas lições foram aprendidas durante o estudo de caso: 1) conformidade arquitetural requer apoio ferramental, e 2) Decisões arquiteturais mudam ao longo do tempo.

Conformidade arquitetural requer apoio ferramental A verificação de conformidade arquitetural deve ter papel relevante nos processos de desenvolvimento para garantir a qualidade do software. Contudo, a falta de ferramentas que automatizem, ainda que parcialmente, o processo torna a atividade impraticável. Técnicas de inspeção e revisão de código, programação em pares, checklists de padrões ajudam a amenizar os problemas. Contudo, como são essencialmente manuais estão sujeitas aos problemas já conhecidos: diferenças nas interpretações, erros na execução e custos elevados. Nesse sentido, defendemos que testes de design são uma forma barata e simples de incorporar a verificação de conformidade arquitetural a qualquer processo de desenvolvimento. Em particular, permite que a verificação arquitetural seja integrada ao processo de testes automáticos e que, assim, se monitore continuamente o entendimento e a atuação da equipe em relação às regras de design e de arquitetura. Por elevar a documentação das regras à condição de executáveis, reduz significativamente as falhas de comunicação da equipe em termos de decisões de design e de arquitetura. Uma vez implementadas como testes, as regras podem ser integradas ao repositório e podem ser usadas para compor as suítes de testes pré-commit (testes que condicionam a aceitação de cada commit feito pelos desenvolvedores). Além disso, por serem testes, o feedback proporcionado ao desenvolvedor em caso de falha é concreto: indica uma violação em termos dos módulos envolvidos e da regra desrespeitada.

Decisões arquiteturais evoluem À medida que se evolui no desenvolvimento do sistema ou que se comprehende melhor os requisitos, é natural que as decisões arquiteturais

evoluam. O problema é que isso implica na situação revelada em nosso estudo de caso. Ao criar uma nova regra de design é necessário decidir o que fazer em relação aos trechos da implementação já concluídos que desrespeitam essa regra. Essa situação pode criar instantaneamente um débito arquitetural com o qual é necessário lidar. Se se optar por refatorar o sistema para deixá-lo conforme a nova regra, há que se considerar os possíveis impactos funcionais. Especialmente, quando se trata de porções estáveis do código (partes antigas do código com baixa frequência de alteração). Se se optar por não alterar o código antigo e fazer a regra valer apenas para as futuras modificações, há que se conviver com as violações arquiteturais já existentes, sem no entanto permitir que os desenvolvedores aumentem o número de violações em relação a essa regra². Em ambos os casos, tomar uma decisão com clareza e de forma racional parece ser melhor alternativa do que manter o problema escondido, como ocorre quando não há um processo efetivo de verificação de conformidade arquitetural.

Da mesma forma que ocorreu nos outros estudos de caso que realizamos com testes de design para a verificação arquitetural, o líder de equipe viu-se motivado a adotar o método, tendo declarado isso explicitamente. Em geral, o único empecilho para adotar o método de imediato é a dificuldade em lidar com o débito arquitetural identificado. Embora isso demande um outro estudo, provavelmente a forma mais apropriada seja iniciar por um significativo refatoramento, para colocar o sistema em conformidade arquitetural.

4. Trabalhos Relacionados

A literatura relacionada à verificação de conformidade arquitetural é prolífica [BRUNET et al. 2009, BRUNET et al. 2011]. O objetivo desta seção é discutir os trabalhos que relatam a experiência da aplicação dessas abordagens ao longo do desenvolvimento de sistemas de software. Neste contexto, Murphy e Notkin descrevem a aplicação da abordagem Modelos de Reflexão no gerenciador de planilhas Excel da Microsoft [MURPHY and NOTKIN 1997, MURPHY et al. 2001]. O experimento foi executado como um exercício de reengenharia. Os resultados indicaram que o Excel, mesmo com uma documentação arquitetural vasta e constantemente atualizada, possuía uma grande quantidade de violações. Além desta constatação, como resultado da atividade de reengenharia, os desenvolvedores passaram a dominar mais os conceitos arquiteturais do sistema, uma vez que as violações eram detectadas e discutidas.

Unphon e Dittrich [UNPHON and DITTRICH 2010] realizaram um importante estudo com o objetivo de levantar conhecimento a respeito de como os *stakeholders* lidam com preocupações arquiteturais na prática. Como resultado desse estudo, os autores identificaram que usualmente existe um arquiteto chefe ou desenvolvedor central, intitulado “*walking architecture*”, que detém grande parte do conhecimento a respeito da arquitetura e que esse conhecimento é passado para os membros da equipe através de discussões pontuais. Além disso, esse estudo também confirma o fato de que a documentação arquitetural tende a não ser atualizada e, por esse motivo, torna-se obsoleta.

5. Conclusão

Neste estudo reiteramos a importância da verificação de conformidade arquitetural e, em particular, para reforçar a necessidade de ferramentas de apoio à atividade. Apesar de

²Nesse caso, é aconselhável manter a regra e tratar, explicitamente, as violações antigas como exceções. É possível escrever testes de design com exceções. Dessa forma, novas violações seriam acusadas, sem a necessidade de conviver com a reiterada indicação das violações antigas.

todo esforço por parte de uma equipe de desenvolvimento, é improvável que seja capaz de efetivamente detectar violações sem o apoio de ferramentas. Prova disso é o acúmulo de violações do mesmo tipo ao longo do desenvolvimento e a dificuldade das equipes em lidar com o chamado débito arquitetural (violações arquiteturais acumuladas no histórico do software).

Concluímos ainda que, embora restritos a decisões de design estruturais, testes de design constituem uma forma efetiva e eficiente de verificação arquitetural que podem ser facilmente integrados ao processo de desenvolvimento. Os problemas encontrados no processo de análise das regras arquiteturais apenas refletem o fato de que, na prática, as decisões arquiteturais não são concretas e objetivamente documentadas. Com isso, os problemas encontrados não indicam que a adoção na prática será difícil.

Como trabalho futuro, pretendemos entender melhor como verificação de conformidade arquitetural com base em testes de design pode ser incorporada a um projeto em andamento e com débito arquitetural já identificado.

Referências

- BASS, L., CLEMENTS, P., and KAZMAN, R. (2003). Software architecture in practice. *Addison-Wesley Professional*.
- BROOKS, F. (1975). The mythical man-month. *Addison-Wesley Reading, Mass*, 79.
- BRUNET, J., GUERRERO, D., and FIGUEIREDO, J. (May 2009). Design Tests: An Approach to Programmatically Check your Code Against Design Rules. *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009), New Ideas and Emerging Results*.
- BRUNET, J., GUERRERO, D., and FIGUEIREDO, J. (September 2011). Structural Conformance Checking with Design Tests: An Evaluation of Usability and Scalability. *Proceedings of the 27th International Conference on Software Maintenance (ICSM 2011)*.
- BUDGEN, D. (2003). Software design. *Addison Wesley*.
- CLEMENTS, P., GARLAN, D., LITTLE, R., NORD, R., and STAFFORD, J. (2003). Documenting software architectures: views and beyond. *Proceedings of the 25th International Conference on Software Engineering - IEEE*, page 740–741.
- GARLAN, D. and PERRY, D. (1995). Introduction to the special issue on software architecture. *IEEE Trans. Softw. Eng.*
- GURP, J. and BOSCH, J. (2002). Design erosion: problems and causes. *Journal of systems and software*.
- JANSEN, A. and BOSCH, J. (2005). Software architecture as a set of architectural design decisions. *Proceedings of the 5th Working Conference on Software Architecture - IEEE*, page 109–120.
- MURPHY, G. and NOTKIN, D. (Aug 1997). Reengineering with reflexion models: a case study. *Computer*, 30(8):29,36.
- MURPHY, G., NOTKIN, D., and SULLIVAN, K. (Apr 2001). Software reflexion models: bridging the gap between design and implementation. *Software Engineering, IEEE Transactions*, 27(4):364,380.
- PERRY, D. and WOLF, A. (1992). Foundations for the study of software architecture. *ACM SIG-SOFT Software Engineering Notes*, 17:40–52.
- UNPHON, H. and DITTRICH, Y. (November 2010). Software architecture awareness in long-term software product evolution. *J. Syst. Softw.* 83, 11.

ChangeMiner: um *plug-in* visual para auxiliar a manutenção de software

Francisco R. Santos¹, Methanias Colaço Júnior^{2,3}, Manoel Mendonça²

¹ GRUFE – Grupo de Pesquisa no Desenvolvimento de Ferramentas Computacionais

Educacionais

Instituto Federal de Sergipe (IFS) – Lagarto – Brasil

² LES – Laboratório de Engenharia de Software

Universidade Federal da Bahia (UFBA) – Salvador – Brasil

³ NUPIC – Núcleo de Pesquisa e Prática em Inteligência Competitiva

Universidade Federal da Sergipe (UFS) – Itabaiana – Brasil

frchico@gmail.com, mjrse@hotmail.com, manoel.g.mendonca@gmail.com

Abstract. *Software repository mining is an important approach to understand and improve software development and maintenance activities. A way to leverage this approach even further is to use IDEs, plugin-in mined results and models to these environments, to support decision making during software engineering activities. This article discusses the construction of an IDE plug-in to predict module change association on the fly, in order to help programmers to reduce omission errors during software maintenance activities.*

Resumo. *A mineração de repositórios de software tem se apresentado como uma importante alternativa para compreensão e melhoria das atividades de desenvolvimento e manutenção de software. Uma forma de aplicar esta abordagem na prática é explorar sua utilização em IDEs. Plug-ins podem ser construídos para acoplar resultados e modelos minerados a estes ambientes, disponibilizando apoio à decisão on-line. Este artigo discute a construção de um plug-in que prediz associações entre modificações de módulos à medida que estes são acessados, com o objetivo de reduzir erros de omissão durante atividades de manutenção de software.*

1. Introdução

A garantia da qualidade de produtos de software reside na execução de tarefas sistemáticas e de auditorias para verificar como o processo está sendo ou deve ser implementado. Estas tarefas devem ser apoiadas por métricas de software, tratadas e exploradas por Basili et al (1994), IEEE (1998) e Fenton et al (1997), enquadrando-se nos padrões e requisitos de projeto com eficiência e eficácia.

Em qualquer processo de garantia de qualidade, a qualidade do produto está diretamente ligada com o processo de desenvolvimento e produção. Na engenharia de software esta relação é mais próxima, motivando a criação de padrões de maturidade de software como o CMM (*Capability Maturity Model*) e o CMMI (*Capability Maturity Model Integration*) [SEI 2002].

Modelos de maturidade sugerem um repositório integrado de métricas, visando coletar dados consistentes de diferentes projetos [Plaza et. Al. 2003]. Devido à disponibilidade de tais repositórios, pesquisas em sistemas de informação vêm agora sendo intensificadas para melhor explorá-los. Entre as principais metas dessas pesquisas, podemos citar a construção de modelos preditivos para estimação de defeitos em produtos de software [Bergel et. Al 2008]. Dados históricos dos projetos de sistemas são utilizados para o reconhecimento de padrões de modificações, evolução, decadências e identificação de erros em módulos de software.

Sistemas gerenciadores de versão (SCV) estão entre os tipos de repositórios que são frequentemente explorados, pelo fato de armazenarem as modificações que são realizadas durante a construção/melhoria de um sistema de informação. Estas modificações evidenciam o acoplamento e as dependências que módulos de software possuem entre si, os quais são geralmente modificados em grupos. A história pregressa de modificações conjuntas de módulos de software permite a criação de modelos de análise de impacto. Nestes modelos, a taxa de associação entre modificações conjuntas pode ser usada para sugerir quais outros módulos de software podem ser afetados, dado que um deles está sendo modificado.

Estudos anteriores a este trabalho fizeram uso deste tipo de abordagem em bases de projetos softwares livres e com SCV abertos [Zimmermann e Weissgerber 2004], havendo poucas evidências que atestem a sua validade em outros contextos. Este fato justifica a experimentação da abordagem de descoberta de associações de modificações de módulos de software em novos contextos. Em particular, é desejável que seja explorada a descoberta de associações em bases de SCV de projetos industriais com processos de desenvolvimento mais rígido e controlado.

Este artigo explora a arquitetura do ChangeMiner, cuja finalidade é a extração de dados de SCVs, para inferir um modelo de predição e acoplá-lo a um IDE (MS Visual Studio). Na abordagem utilizada, as estimativas de associações podem ser solicitadas pelo desenvolvedor, em tempo real, durante a execução de uma manutenção de sistema em um IDE comercial.

Para descrição da ferramenta, o restante do artigo é organizado da seguinte forma: a próxima seção discute trabalhos relacionados. Na seção 3, são descritos os princípios usados para prover a infra-estrutura básica da solução. A seção 4 descreve o funcionamento e arquitetura geral da ferramenta. Finalmente, a seção 5 discute aprimoramentos e oportunidades de pesquisas futuras.

2. Trabalhos relacionados

Alguns trabalhos já exploraram a utilização de regras de associação para predição de mudanças de software. Ying et al. (2004) apresenta uma abordagem de mineração de associação em modificações de arquivos, sem oferecer a mineração em tempo real (*on-the-fly*). Zimmermann et al. (2005) desenvolveu uma abordagem para a mineração *on-the-fly* para predizer relações entre módulos ao nível de classes e de métodos, na IDE Eclipse. Estes dois trabalhos treinaram seus modelos e testaram suas acuracidades com softwares e repositórios de código aberto. Ambos apresentam as associações através de uma interface textual.

A execução deste projeto segue uma abordagem semelhante a que é proposta por Zimmermann et al na concepção da ferramenta ROSE [Zimmermann e Weissgerber

2004]. A ROSE minera as mudanças de um módulo de software, sugerindo novas alterações a serem feitas nas entidades ou alertando para mudanças que ainda precisam ser feitas.

No trabalho anteriormente mencionado, tanto o SCV quanto a base de dados minerada provieram de sistemas de código aberto. Além disso, não foi utilizado um repositório projetado para a análise de dados históricos [Colaço 2004].

Em detrimento a essa estrutura, nossa solução possui seguintes características: (a) ela utiliza um Data Warehouse (DW) para armazenar os dados do SCV [Colaço et al 2009B]; (b) realização de testes em ambiente industrial com softwares de código fechado e (c) a avaliação do modelo e da ferramenta construída foi feita em um estudo experimental que compara os resultados de precisão e cobertura [Rijsbergen 1979] das regras de associação produzidas.

3. Princípios utilizados

3.1. Extração de dados de repositórios de códigos

Repositórios de versões são amplamente utilizados nas equipes de desenvolvimento de software, pois oferecem uma estrutura de controle da evolução do sistema, além de serem requisitos em modelos de maturidade tais como o CMMI e o MPS-BR [Pronus 2008].

Modelos de maturidade sugerem armazenamento histórico, o qual é suprido em parte pelos SCVs, à medida que os mesmos guardam as alterações feitas nos módulos de um software, além de oferecerem dados tais como programador responsável, data e hora da alteração, número da revisão e comentário de modificação.

Extrair esses dados é um requisito básico para criação de modelos de associação de modificações de módulos de software. Neste trabalho, para extração, foi utilizado o algoritmo proposto por Zimmermann e Weissgerber (2004), em conjunto com um algoritmo de identificação dos sistemas envolvidos na alteração. São considerados associados aqueles módulos modificados em uma mesma transação, sendo definida por uma janela de tempo (neste caso cinco minutos), o nome do autor, o comentário e o caminho da alteração. Com a identificação dos módulos alterados juntos, torna-se possível fazer a busca de associações descrita a seguir.

3.2. Regras de associação

Identificados os módulos que participaram de uma transação, utiliza-se um algoritmo para descoberta de regras de associação do pacote de *Business Intelligence* da Microsoft, o qual disponibiliza um algoritmo pertencente à família APRIORI [Agrawal e Srikant 1994], eficiente para achar relações entre os itens de uma transação.

Como saída, o algoritmo produz regras com a associação entre os módulos de software, informando o nível de suporte e confiança para as mesmas. Neste contexto, o suporte é o número de transações em que o antecedente aparece e a confiança da regra é o percentual de transações do suporte nas quais as entidades antecedente e consequente aparecem. Um exemplo de alerta ao programador pode ser visto na Figura 3, traduzindo-se assim: *quem alterou ngVendasCC.cs, também alterou dbAcaoMkt.cs e fcAcaoMkt.cs, com confianças de 93,85 e 93,39, respectivamente.*

4. ChangeMiner

O *plug-in* que construímos chama-se ChangeMiner. Ele é um *plug-in* gratuito para o Visual Studio, capaz de minerar um *Data Mart* que pode ser alimentado pelos repositórios *Visual Source Safe* (VSS) ou *Subversion* (SVN). No momento da alteração de código, baseado no histórico pregresso de alteração conjunta de módulos, o *plug-in* reporta ao desenvolvedor os próximos pontos prováveis (módulos) de manutenção tanto de maneira textual quanto gráfica, objetivando aumentar a eficiência da manutenção e diminuir a inserção de erros por omissão de código.

4.1. Pré-Requisitos

São pré-requisitos do ChangeMiner: (a) Microsoft SQL Server 2005 com Analysis Services; (b) Microsoft .NET Framework 3.0 ou superior e (c) Uso do SCV VSS 6.0a (ou superior) ou SVN 1.4 (ou superior).

4.2. Arquitetura da ferramenta

A arquitetura proposta para a ferramenta (ver Figura 1) é composta de quatro camadas básicas e integradas definidas como: (a) *Data Source*, para representar os repositórios que fornecerem os dados a serem trabalhos; (b) Ferramentas de ETL, apresentada na seção 4.2.1, realiza a coleta, o tratamento e unificação os dados; (c) Ambiente de DW, anteriormente publicado em [Colaço Jr. et al, 2009A], com o objetivo de agrupar os dados corporativos extraídos pelas diversas ferramentas de ETL e (d) Ambiente de Análise e construção de softwares, seção 4.2.2, para representar as ferramentas que podem fazer uso dos dados agrupados e armazenados no DW, seja através de consultas, de ferramentas OLAP, painéis de gerenciamento (*dashboards*) ou, até mesmo, dentro de um ambiente de desenvolvimento.

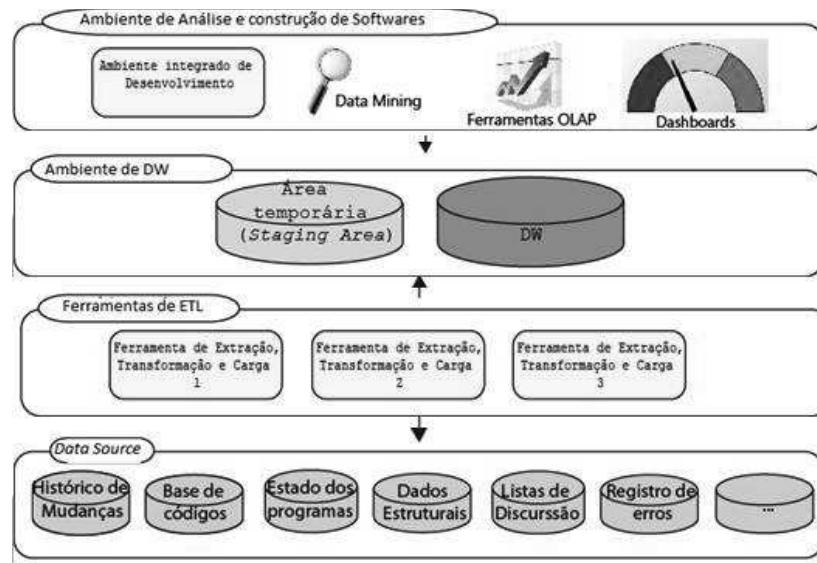


Figura 1. Arquitetura do ChangeMiner

4.2.1. Ferramenta de ETL

A ferramenta possui duas estruturas lógicas para extração e armazenamento dos dados: (1) Uma para camada de extração, transformação e carga dos dados (ETL) e (2) o *Data Mart* que recebe os dados já tratados, oriundos da camada ETL.

Para o modelo de ETL, foram definidas duas tabelas que recebem os dados importados dos SCVs (vide Figura 2). A tabela *VSSUser* armazena as informações dos usuários do repositório e a tabela *VssItem* guarda cada item de uma transação. A carga para o *Data Mart* pode ser realizada de duas maneiras: automática e manual, sendo a automática a configuração padrão, a qual é inicializada após a configuração de um agendamento (*job*) no Sistema Gerenciador de Banco de Dados (SGBD) para invocar um procedimento de carga. Já a manual é realizada para forçar uma carga dos SCVs.

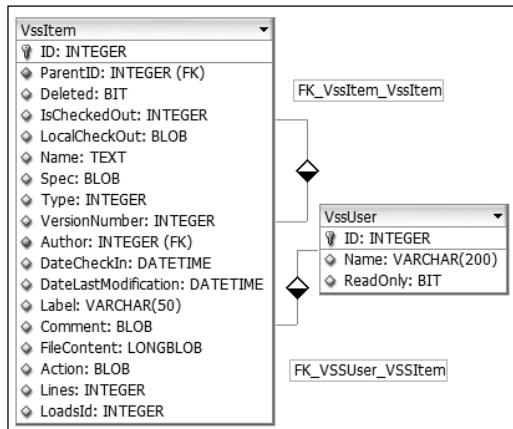


Figura 2. Tabelas básicas do ETL

O framework de extração dos dados foi construído utilizando a linguagem C# e permite a integração com outras bibliotecas e ferramentas de ETL através da implementação da interface *ExtractFromSourceControl*. A interface garante o funcionamento do módulo extrator com diversos SCVs, delegando as tarefas de obtenção dos dados a uma extensão especializada para um repositório SCV específico.

O presente trabalho customizou o módulo extrator para o VSS e SVN através das implementações *ExtractVSS* e *ExtractSVN*. Ou seja, o módulo realiza a conexão com o repositório e inicia uma pesquisa em profundidade, buscando e extraíndo os itens dos diretórios que atendem a um intervalo de data/hora previamente definidos (através de parâmetros). Os artefatos que atendem à condição são enviados para as tabelas do ETL.

Ao término do procedimento de importação dos dados dos repositórios, são executados as *store procedures* no banco de dados para seleção do que pode ou não pode ser enviado ao DW. Os procedimentos analisam as transações de modificações, utilizando o algoritmo proposto por Zimmermann e Weissgerber (2004), com uma janela de tempo de cinco minutos, em conjunto com um algoritmo de identificação dos responsáveis pela alteração, sistemas e classes modificadas.

Já a integração com outros tipos de programas, à exemplo de coleta métricas de engenharia de software, é dada através da implementação dos eventos (*delegates* em C#). Com isso, a API do ChangeMiner torna-se um framework para novas ferramentas de Engenharia de Software.

4.2.2. Módulo de Apresentação

Um botão no IDE permite a descoberta, em tempo real (*on the fly*), de associações existentes entre a entidade sendo editada e as outras entidades do sistema (Figura 3).

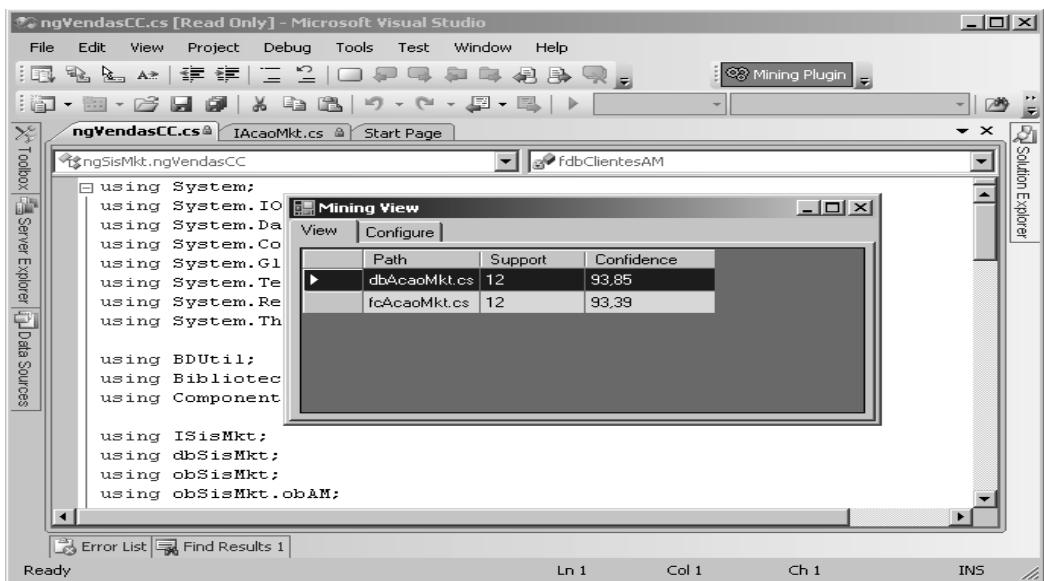


Figura 3. Execução do *Plug-in*

O resultado de consulta ao modelo é exibido de duas formas: grafo e tabular. Na forma tabular (vide Figura 3), são apresentadas ao programador as dez primeiras entidades que alcançam uma confiança de 75% de associação com a entidade sendo alterada, em ordem decrescente de suporte e confiança. Já na forma de grafo, conforme Figura 4, a raiz representa a entidade em alteração, ligada às suas possíveis dependências, respeitando-se a confiança e ordenação citadas anteriormente. Esta funcionalidade ainda está em evolução e na sua versão final será possível navegar na árvore exibindo as associações relacionadas a cada um dos itens.

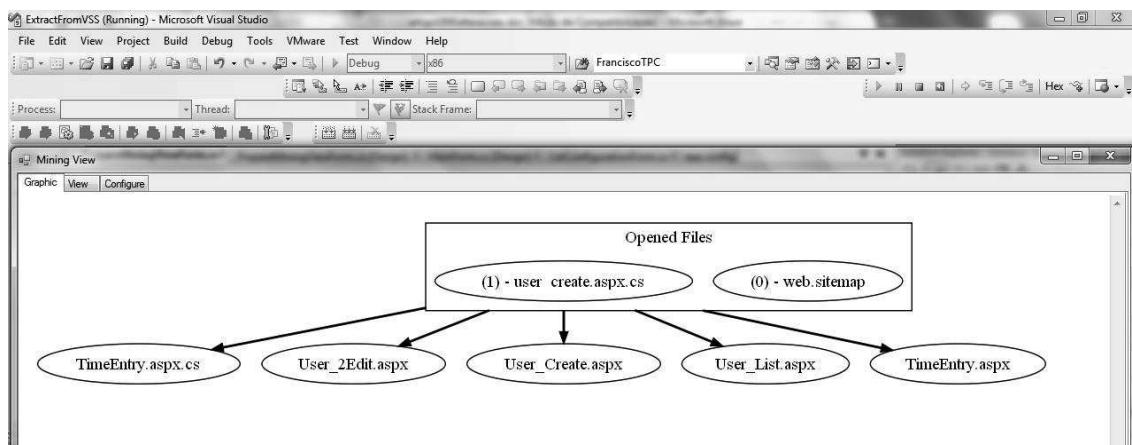


Figura 4. Resultados exibidos usando Grafo

4.3. Resultados Experimentais

Um experimento em ambiente industrial foi realizado para avaliar o plug-in e a acurácia do modelo [Colaço Jr. et al., 2009B]. O parceiro escolhido foi a CIAL, um fabricante e distribuidor de produtos Coca-Cola para os estados de Sergipe e Alagoas, que possui vários sistemas de grande porte desenvolvidos na plataforma Microsoft .NET.

Após oito anos de uso do VSS pela corporação, foram avaliadas as bases VSS de 18 grandes sistemas, perfazendo 256.804 transações em 4.096 arquivos. Destas

transações, após a limpeza e seleção dos dados, foram identificadas 153.288 transações caracterizadas como *labels* ou *tags*, 69.592 mudanças de diretórios e 33.924 transações realmente válidas para treinamento do modelo de mineração.

Os resultados médios de cobertura e precisão foram de 76% e 58%, respectivamente. Em outras palavras, dada uma mudança, a ferramenta acerta 58% das sugestões das próximas alterações, chegando a alcançar a precisão de mais de 70% em diversos sistemas, conforme pode ser observado na tabela 1. No quesito prevenção de erros, a ferramenta conseguiu um resultado médio de 90% de precisão [Colaço et al 2009B].

Tabela 1. Dados minerados do ambiente industrial (C = Cobertura Média; P = Precisão Média)

Seq	Projeto: Descrição	C	P	Seq	Projeto: Descrição	C	P
1	CadAlmox: Utilitário para migração do sistema de almoxarifado	0.83	0.56	10	Logistica: Sistema de Logística	0.74	0.51
2	Cadastros: Sistema de Cadastros	0.79	0.61	11	SECV: Sistema de Administrativo	0.90	0.48
3	Cliente: Gerenciamento dos Clientes	0.87	0.73	12	SisAlmox: Sistema de almoxarifado	0.86	0.50
4	Contabil: Sistema contábil	0.65	0.53	13	SisAP: Sistema de pesquisa	0.57	0.46
5	DesenvolProjetos: Ferramentas de desenvolvimento	0.89	0.62	14	SisComo: Sistema de comodato	0.68	0.51
6	Fiscal: Sistema Fiscal	0.59	0.71	15	SisCusteio: Sistema de custos	0.67	0.50
7	FrameWork: Framework	0.73	0.61	16	SisManutencao: Utilitário para configuração do portal (WEB)	0.89	0.67
8	GUF: Sistema de Cosméticos	0.91	0.54	17	SisMkt: Sistema de Marketing	0.83	0.68
9	Legados: Legados	0.67	0.71	18	SisPortal: Portal WEB	0.65	0.56
				Média Geral:			
				0.76 0.58			

Uma amostra das regras geradas foi validada pelos programadores, os quais se beneficiaram da possibilidade de ver as dependências de forma visual (Figura 4) e, através de suas experiências, confirmaram a acuracidade das principais dependências apresentadas.

5. Conclusão e trabalhos futuros

Este artigo apresentou um *plug-in* que explora a mineração de regras de associação para melhoria do processo de manutenção de software. O desenvolvimento da ferramenta mostrou que é viável manter uma base de dados histórica do processo de manutenção de código, bem como explorar o acoplamento de técnicas de mineração de dados aos ambientes de desenvolvimento de software utilizados na indústria.

Assim, nós acreditamos que esse trabalho pode estimular a indústria a investir no uso de mineração de dados para auxiliar as tarefas de manutenção, uma vez que foi possível mostrar viabilidade técnica para alcançar altos valores de precisão e cobertura em ambientes industriais.

No que toca as dificuldades encontradas durante a execução deste trabalho, destacam-se: (a) a ausência de documentação que descrevesse como eram estruturadas as bases de dados utilizadas pelos repositórios de códigos, principalmente do VSS; e (b)

a descrição das mensagens de erros das APIs elaboradas pelos fabricantes dos produtos, isto é, dos VSS e SVN.

Para concluir, como trabalhos futuros, teremos que: (1) realizar avaliação utilizando níveis mais finos de granularidade e relacionamento entre sistemas; (2) analisar utilidade do plug-in através do uso controlado pelos programadores e do preenchimento de um *survey* pelos mesmos; (3) explorar diversas técnicas de mineração de dados para descoberta de conhecimento no DW construído; (4) integrar com sistemas de análise de códigos bem como sistemas de gerenciamento de erros e (5) realizar a integração com novos repositórios, principalmente os distribuídos como o GIT.

6. Referências

- Agrawal, R.; Srikant, R. Fast algorithms for mining association rules. In *Proceedings of the 20th Very Large Data Bases Conference*, p. 487–499, 1994.
- Carneiro, G. D.; Magnavita, R.; Spinola, E.; Spinola, F; Mendonça, M. G. An Eclipse-Based Visualization Tool for Software Comprehension, *XXII Brazilian Symposium on Software Engineering*, 2008.
- Colaço Jr., M.; Mendonça, M. G.; Rodrigues, F. Data Warehousing in an Industrial Software Development Environment. In: *33rd Annual IEEE/NASA Software Engineering Workshop*, 2009A.
- Colaço Jr., M.; Mendonça, M. G.; Rodrigues, F. Mining Software Change History in an Industrial Environment. *XXIII Brazilian Symposium on Software Engineering*, p. 54-61, 2009B.
- Alexandre Bergel and Stéphane Ducasse and Jannik Laval and Romain Peirs. Enhanced Dependency Structure Matrix for Moose. In FAMOOSr, 2nd Workshop on FAMIX and Moose in Reengineering, 2008
- Gall, H.; Jazayeri, M.; Krajewski, J. CVS release history data for detecting logical couplings. In *Proc. International Workshop on Principles of Software Evolution*, p. 13–23, 2003.
- Pronus. O que é Gerência de Configuração. Disponível em: <http://pronus.eng.br/artigos_tutoriais/gerencia_configuracao/gerencia_configuracao.php?pagNum=0>, 2008.
- Rijsbergen, C. J. *Information Retrieval*. 2 ed. Butterworths, London, 1979.
- Ying, A. T.; Murphy, G. C.; NG, R.; Chu-Carroll, M. C. Predicting source code changes by mining change history. *IEEE Transactions on software engineering*, v. 30, n. 9, p. 574-586, 2004.
- Zimmermann, T.; Weissgerber, P. Preprocessing CVS data for fine-grained analysis. In: *International Workshop on Mining Software Repositories*, 2004.
- Zimmermann, T.; Zeller, A.; Weissgerber, P.; Diehl, S. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, v. 31, n. 6, p. 429–445, 2005.

Visualização de Software como Suporte ao Desenvolvimento Centrado em Métricas Orientadas a Objetos

Elidiane Pereira dos Santos e Sandro Santos Andrade

Grupo de Sistemas Distribuídos, Otimização, Redes e Tempo-Real (GSORT)

Instituto Federal de Educação, Ciência e Tecnologia da Bahia (IFBa)

Av. Araújo Pinho, nº 39, Canela – 40.110 -150 – Salvador – BA – Brasil

{elidiane, sandroandrade}@ifba.edu.br

Abstract. *Software visualization approaches, mostly those ones directly integrated into development environments, have increasingly been considered major tools when designing and evolving complex and large-scale software systems. This paper reports on the experience of implementing the Lorenz & Kidd suite of object-oriented metrics, as well as three accompanying case-studies for visualizing such metrics. Furthermore, an empirical evaluation has been undertaken, which aimed at assessing the impact of the proposed tool on the quality of artifacts yielded from a code refactoring activity.*

Resumo. *Mecanismos para visualização de software, preferencialmente aqueles integrados em ambientes de desenvolvimento, representam ferramentas cada vez mais importantes para o desenvolvimento e evolução de sistemas computacionais modernos. Este artigo reporta a experiência de implementação da Suite de Métricas OO de Lorenz & Kidd no Ambiente Integrado de Desenvolvimento Qt Creator, a realização de três estudos de caso com a ferramenta desenvolvida e a condução de um experimento com o objetivo de investigar o impacto do uso da visualização de tais métricas na qualidade dos artefatos gerados por uma atividade de refatoração de código.*

1. Introdução

À medida em que os sistemas computacionais se tornam mais complexos, heterogêneos e de larga-escala, a utilização de mecanismos para visualização de *software* [Caserta and Zendra 2011] se torna fator importante no desenvolvimento, evolução e compreensão de tais aplicações. A Orientação a Objetos (OO), enquanto técnica de projeto e programação de sistemas, ainda representa uma das abordagens mais promissoras para gerenciamento de complexidade e suporte à evolução facilitada de aplicações com grandes *codebases*. Dessa forma, a prospecção e utilização de métricas baseadas nos constructos da OO [Lanza and Marinescu 2010] constitui um dos meios mais simples e diretos de suporte à visualização de sistemas desenvolvidos com o uso deste paradigma.

Técnicas de Visualização de Software têm como objetivo a geração de representações visuais que facilitam a compreensão de diferentes aspectos de um sistema de *software* em diferentes granularidades (níveis de abstração). Possíveis aspectos a serem visualizados incluem informações estruturais, comportamentais ou de evolução do *software*. Tais aspectos podem ser investigados em diferentes níveis de abstração, por exemplo com foco em linhas de código, classes ou no âmbito arquitetural [Caserta and Zendra 2011].

Desde a proposta da *Suite CK* [Chidamber and Kemerer 1994], em 1994, uma série de outros modelos para avaliação de qualidade de projeto OO foram apresentados [Sarkar et al. 2008, Purao and Vaishnavi 2003]. Adicionalmente, estudos como [Olague et al. 2007] tentam relacionar empiricamente o uso de tais métricas à predição de determinados atributos de qualidade de *software*, como por exemplo susceptibilidade a falhas.

Este artigo reporta uma experiência de implementação de uma variante das métricas propostas em [Lorenz and Kidd 1994] e sua visualização em um Ambiente Integrado de Desenvolvimento (IDE - *Integrated Development Environment*). Adicionalmente, foram também realizados três estudos de caso e um experimento de avaliação do impacto do uso deste recurso de visualização na qualidade dos produtos resultantes de uma atividade de refatoração de código.

O restante deste artigo está organizado como segue. A seção 2 apresenta as principais métricas presentes em [Lorenz and Kidd 1994]. Na seção 3, os aspectos técnicos e de projeto da implementação realizada são apresentados, enquanto a seção 4 apresenta os estudos de caso e o experimento de avaliação realizados. A seção 5 aponta os trabalhos correlatos e, finalmente, a seção 6 apresenta as conclusões e possíveis trabalhos futuros.

2. A Suite de Métricas OO de Lorenz & Kidd

Lorenz & Kidd, baseados nas métricas presentes na *Suite CK*, propuseram em 1994 um extenso conjunto de cerca de 30 métricas para avaliação de complexidade em sistemas OO. Tais métricas foram classificadas em grupos relacionados a cinco aspectos distintos:

1. Tamanho de Método: inclui métricas tais como quantidade de mensagens enviadas, quantidade de sentenças constituintes, quantidade de linhas de código e tamanho médio de método.
2. Aspectos Internos de Método: o objetivo é estimar a complexidade de manutenção de um método, por exemplo através do cálculo da Complexidade Ciclomática.
3. Tamanho de Classe: inclui métricas tais como quantidade de métodos públicos por classe, quantidade média de métodos por classe, quantidade de atributos por classe, quantidade de atributos/métodos estáticos por classe, etc.
4. Herança de Classe: dentre as métricas deste grupo destacam-se grau de aninhamento dentro de uma hierarquia, quantidade de classes abstratas e uso de herança múltipla.
5. Herança de Método: grupo com o maior número de métricas, incluindo quantidade de métodos sobrepostos por uma sub-classe, quantidade de métodos herdados por uma sub-classe, índice de especialização, coesão da classe, quantidade média de parâmetros por método, uso de funções *friend*, acoplamento entre classes e quantidade de vezes que uma classe é reutilizada.

As seguintes métricas foram implementadas na ferramenta descrita neste trabalho:

1. Relação Média entre Atributos e Métodos de uma Classe (MAC): é obtida pelo total de atributos dividido pelo total de métodos de uma determinada classe [Lorenz and Kidd 1994]. A presença de muitos atributos e poucos métodos é um indício de uma provável baixa coesão na classe em questão.

2. Quantidade de Métodos por Classe (QMC): considera métodos com qualquer visibilidade em uma determinada classe. Segundo [Lorenz and Kidd 1994], o valor recomendado para esta métrica é de 40 para classes de interface gráfica de usuário e 20 para as demais.
3. Tamanho Médio dos Métodos de uma Classe (TMC): considera o número de linhas físicas de código ativo presentes em um método [Lorenz and Kidd 1994]. Métodos de fácil manutenção geralmente possuem tamanho reduzido. O valor estimado é de cerca de 30 linhas para códigos escritos em C++ [Ambler 1998].
4. Quantidade de Atributos por Classe (QAC): é um possível indicador indireto da qualidade do projeto [Lorenz and Kidd 1994]. Uma classe com um alto número de atributos possivelmente sugere a presença de um alto número de relacionamentos com outros objetos do sistema. As classes com mais de três ou quatro atributos mascaram o problema de acoplamento da aplicação [Ambler 1998]. Em classes de interface gráfica de usuário o número de atributos pode chegar a nove, pois estas classes necessitam de mais recursos para lidar com aspectos de interação [Lorenz and Kidd 1994].
5. Quantidade de Métodos Públicos (QMP): visto que os métodos públicos representam aqueles serviços disponíveis para objetos de outras classes, esta métrica geralmente constitui um bom indicador do grau de coesão da aplicação.

3. A Ferramenta Proposta

O *Qt Creator Visualization Plugin*, ferramenta apresentada neste artigo, implementa a extração e visualização das métricas acima descritas, de forma integrada ao *Qt Creator* [Qt-Project 2013] - IDE padrão para desenvolvimento Qt. O Qt é atualmente um dos *toolkits* para desenvolvimento multi-plataforma mais completos e promissores. Desde o início do seu desenvolvimento, em 1991, o Qt vem se consolidando não só como um poderoso *toolkit* para construção de interfaces gráficas de usuário mas também como excelente biblioteca de propósito geral, com recursos por exemplo para *Inter-Process Communication*, acesso a banco de dados, programação concorrente e distribuída, manipulação de XML, renderização 3D, dentre outras funcionalidades.

O *Qt Creator* disponibiliza os principais recursos para construção facilitada de interfaces gráficas de usuário, depuração, *syntax highlighting*, funções básicas de refatoração, *profiling* e integração com sistemas de controle de versão. Entretanto, é ainda carente de recursos para visualização de *software*, apresentando apenas grafos simples gerados pela ferramenta de *profiling* Valgrind. Este trabalho projetou e implementou um *plugin* para visualização das métricas de *software* descritas, de forma integrada ao *Qt Creator*.

O principal requisito do *plugin* é calcular e visualizar as métricas por classe de um determinado projeto. Para isso, o *plugin* realiza a extração dos dados a partir da infraestrutura interna de representação de código-fonte do *Qt Creator* e calcula as métricas desejadas. A ferramenta exibe graficamente os valores das métricas, possibilitando a avaliação da qualidade do programa desenvolvido, e sinaliza valores limite que uma determinada métrica pode assumir. Isso possibilita a rápida identificação das classes e métodos mais problemáticos do projeto. A ferramenta foi desenvolvida em C++ no ambiente *Qt Creator* e requer somente a instalação do *plugin* correspondente para a sua utilização imediata.

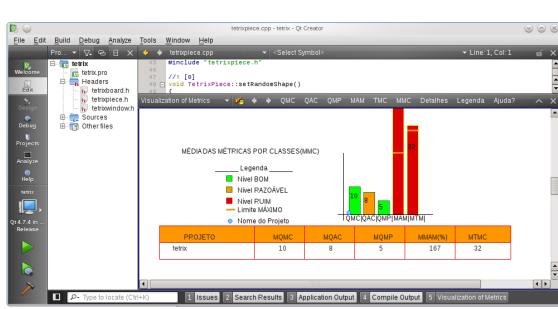


Figura 1. Projeto Tetrix - resumo das métricas

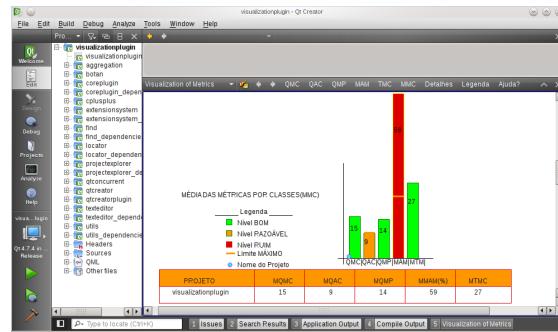


Figura 2. Visualização do próprio plugin - resumo das métricas

4. Avaliação

A ferramenta foi avaliada em duas dimensões: capacidade de extração das métricas em projetos de médio/grande porte (teste funcional) e análise do impacto de uso da ferramenta na qualidade dos artefatos gerados por operações de refatoração. As seções seguintes apresentam os estudos de caso utilizados no teste funcional e o experimento realizado para avaliação do impacto de uso da ferramenta.

4.1. Estudo de Caso 1: Tetrix

O primeiro estudo de caso analisou a utilização da ferramenta em um projeto de pequeno porte: o Tetrix. O jogo é formado por três classes: TetrixWindow, TetrixBoard e TetrixPiece. A classe TetrixBoard é a classe mais complexa pois lida com a lógica do jogo e a renderização. A Figura 1 exibe a visão geral da qualidade do sistema com o resumo das métricas.

A métrica QMC apresentou nível bom, pois obteve o valor de 10 métodos por classe, a métrica QAC nível razoável de 8 atributos por classe, enquanto a métrica QMP um nível bom de 5 métodos públicos por classe. A métrica MAC apresentou nível ruim (167%), ultrapassando o valor estimado de 22% de atributos por métodos. Solucionar este problema requer que a quantidade de atributos por método seja reduzida até que o *plugin* indique a adequação à faixa estimada. A métrica TMC apresentou nível ruim (32 linhas), pois o seu limite é de 30 linhas de código por métodos.

4.2. Estudo de Caso 2: *Qt Creator Visualization Plugin*

O segundo estudo de caso analisou o código da própria ferramenta implementada: um projeto de pequeno porte com apenas três classes, porém complexas por ter que extrair dados do código-fonte e ser uma extensão de uma IDE. O projeto é composto pelas classes: VisualizationMetricsPlugin, VisualizationMetricsPane e VisualizationMetricsCollector. A Figura 2 apresenta os resultados para este estudo de caso.

4.3. Estudo de Caso 3: *Qt Creator*

O terceiro estudo de caso analisou o *Qt Creator*: projeto de grande porte composto por mais de 3000 classes. A Figura 3 exibe parte da visualização das métricas. A Figura 4 apresenta o resumo das métricas. Para calcular as métricas deste projeto o *plugin* levou em torno de 10 segundos de execução.

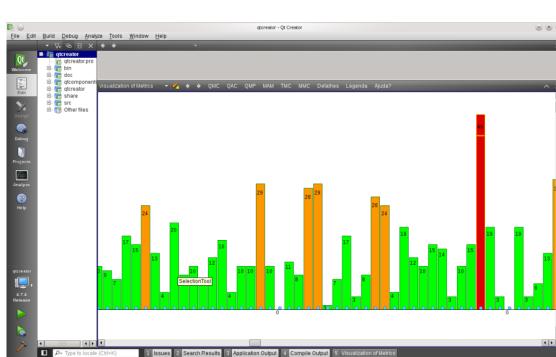


Figura 3. Projeto Qt Creator - métrica QMC

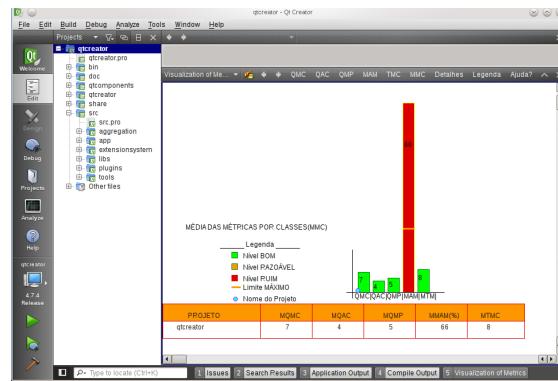


Figura 4. Projeto Qt Creator - resumo das métricas

4.4. Avaliação Experimental

Para estimar o impacto da utilização do *plugin* em atividades de refatoração de aplicações desenvolvidas em C++/Qt, um experimento com oito participantes foi conduzido com o objetivo de avaliar as seguintes hipóteses:

- Hipótese Nula (H_0): a utilização do *plugin* de visualização de *software* não implica em uma diferença estatisticamente considerável na diferença média das métricas antes e depois da refatoração.
- Hipótese Alternativa (H_1): a utilização do *plugin* de visualização de *software* implica em uma diferença estatisticamente considerável na diferença média das métricas antes e depois da refatoração.

Espera-se que, se rejeitada a hipótese nula, o valor das métricas após a refatoração utilizando o *plugin* indique uma melhoria na qualidade interna do sistema. Ou seja, se uma métrica menor indicar um sistema de melhor qualidade espera-se que as métricas após refatoração com o uso do *plugin* sejam menores que as métricas após refatoração sem o uso do *plugin*.

Para isso foi conduzido um experimento onde o jogo de Tetrix deveria ser refatorado. Neste experimento, o objetivo foi realizar operações de refatoração que melhorassem aspectos tais como manutenibilidade e legibilidade do código-fonte. Foi definido um grupo de controle, que não utilizou o *plugin*, e outro grupo que realizou as operações com o auxílio da ferramenta. Todas as métricas foram medidas antes e após o experimento para cada indivíduo de cada um dos dois grupos apresentados. O experimento possui um fator (uso do *plugin*), com dois níveis (sim ou não), sendo executado de forma emparelhada (os dois grupos foram sujeitos ao mesmo conjunto de atividades realizadas).

A Figura 5 apresenta o gráfico consolidado da métrica QMC para todo o projeto. Contém, para cada indivíduo, as médias das métricas obtidas de todas as classes do projeto em comparação com as métricas originais. Os indivíduos 1 e 2 retiraram a quantidade de métodos em excesso das classes, deixando-as com um valor aceitável para a métrica QMC. O indivíduo 3 manteve o *codebase* estável, não acrescentou nem retirou nenhum método das classes. O indivíduo 4 adicionou mais métodos às classes, porém a métrica continuou na faixa aceitável. No gráfico do "Grupo de Uso do Plugin", os indivíduos 1, 2 e 3 acrescentaram métodos às classes porém não ultrapassaram o limite, adicionando mais funcionalidades ao sistema. O indivíduo 4 manteve o *codebase* estável.

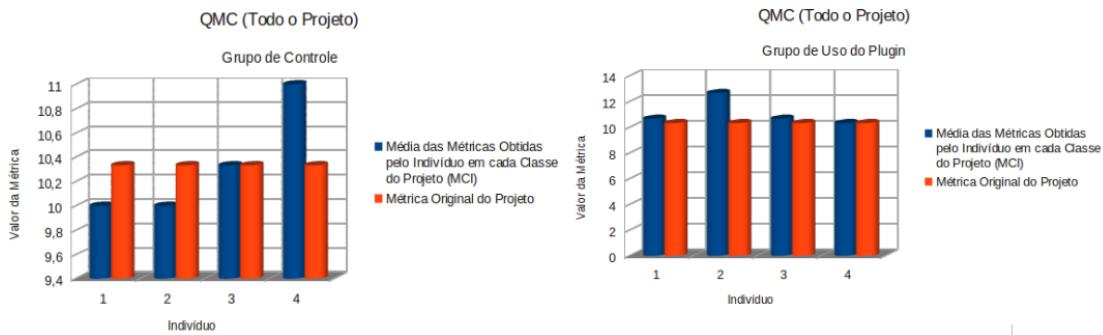


Figura 5. Gráficos consolidados para a métrica QMC em cada grupo

4.4.1. Análise dos Resultados

O teste estatístico *t-test* foi utilizado para avaliar se a diferença entre as médias das duas amostras é estatisticamente significante. Realizou-se o *t-test* para cada métrica sendo avaliada, assumindo que as duas amostras (resultados sem o uso do *plugin* e resultados com o uso do *plugin*) foram obtidas de duas populações com distribuição normal, médias desconhecidas e variâncias desconhecidas porém iguais.

Deseja-se conhecer o maior nível de confiança para o qual o *plugin* produz uma diferença estatisticamente significante na média das métricas:

- *t-test* para a métrica QMC: $p\text{-value} = 0,3618456438$; Nível máximo de confiança ($1-p\text{-value}$) = 0,6381543562;
- *t-test* para a métrica QAC: $p\text{-value} = 0,391002219$; Nível máximo de confiança ($1-p\text{-value}$) = 0,608997781;
- *t-test* para a métrica QMP: $p\text{-value} = 0,8542703292$; Nível máximo de confiança ($1-p\text{-value}$) = 0,1457296708;
- *t-test* para a metrica MAC: $p\text{-value} = 0,8354105574$; Nível máximo de confiança ($1-p\text{-value}$) = 0,1645894426;
- *t-test* para a métrica TMC: $p\text{-value} = 0,2622190236$; Nível máximo de confiança ($1-p\text{-value}$) = 0,7377809764;

Pode-se afirmar, com base nos resultados, que a métrica mais influenciada pelo uso do *plugin* foi a TMC, já que existe 73% de certeza que o *plugin* faz diferença na diferença desta métrica antes e depois das operações de refatoração. Outras métricas potencialmente afetadas pelo *plugin* foram a QMC (63%) e QAC (60%). As demais possuem nível de confiança extremamente baixo e nada pode-se afirmar.

5. Trabalhos Correlatos

Dentre as ferramentas relacionadas à proposta apresentada neste artigo destacam-se: *Visual Metrics* [Rosenberg 1998], ferramenta para cálculo de métricas de Seibt [da Silva Seibt 2001] e a visualização de métricas para acoplamento e coesão de [Dantas 2008].

O *Visual Metrics* permite o cálculo de algumas métricas de construção e de projeto, possibilitando a análise de códigos-fonte desenvolvidos em C# e Java. Com o *Visual*

Tabela 1. Comparação entre as ferramentas de métricas OO

Ferramentas	Métricas					Linguagens			Gráficos	IDE
	MAC %	QMC	TMC	QMP	QAC	Java	C++	Delphi		
<i>Qt Creator Visualization Plugin</i>	✓	✓	✓	✓	✓		✓		✓	✓
Visual Metrics						✓			✓	
Protótipo de Seibt	✓	✓	✓	✓	✓			✓		
Ferramenta de Marques						✓			✓	

Metrics o desenvolvedor pode selecionar os arquivos de projeto a serem analisados, selecionar elementos individuais para análise específica, escolher as métricas a serem calculadas, armazenar e recuperar métricas calculadas e definir limites mínimo e máximo que uma determinada métrica poderá atingir.

A ferramenta para cálculo de métricas de Seibt é um protótipo que fornece métricas pré-definidas a partir da análise de códigos-fonte de programas desenvolvidos em Delphi. As métricas selecionadas foram as de projeto e construção totalizando 19 métricas diferentes.

A visualização de Marques utiliza métricas para indicar o grau de coesão e acoplamento entre classes, exibindo pontos do projeto que pareçam menos estruturados ou pior implementados. A ferramenta analisa o código-fonte de um projeto Java, indicando o grau de coesão e acoplamento nestes sistemas. As métricas calculadas por esta ferramenta são as seguintes métricas da *Suite CK*: CBO (*Coupling Between Objects*), DIT (*Depth of Inheritance Tree*), NOC (*Number of Children*) e LCOM3 (*Lack of Cohesion of Methods*).

A ferramenta apresentada neste artigo se difere das demais nos seguintes aspectos:

- Disponibiliza representação gráfica dos valores calculados, diferente por exemplo da ferramenta de Seibt;
- É integrada a uma IDE, dispensando a utilização de outras ferramentas para compreensão e análise do projeto sendo desenvolvido;
- Implementa algumas métricas de construção, fundamentais e importantes, que o *Visual Metrics* e a ferramenta de Marques não implementam;
- Utiliza uma representação visual diferente para cada métrica, apresentando detalhes de cada uma. Nas demais soluções apenas um gráfico é utilizado para todas as métricas.

A Tabela 1 apresenta uma comparação entre as ferramentas apresentadas.

6. Conclusões e Trabalhos Futuros

Este artigo apresentou uma visão geral sobre o projeto e desenvolvimento de um *plugin* para a IDE *Qt Creator* que extrai e visualiza algumas das métricas descritas na *suite* de Lorenz & Kidd. A ferramenta calcula cinco métricas fundamentais para avaliar a qualidade de um *software*. Com a análise dos estudos de casos e experimento realizados, nota-se a importância da utilização de ferramentas para extração e visualização de métricas na melhoria da qualidade do produto final, desde sistemas simples com poucas classes até aplicações de grande porte.

Dentre as limitações da proposta apresentada pode-se destacar a forma incipiente e com escalabilidade limitada do paradigma de visualização utilizado (gráficos simples), se comparado a técnicas mais sofisticadas, tais como *tree maps* ou *hierarchical radial bundles*. Trabalhos futuros incluem o projeto de experimentos mais robustos e com melhor controle, implementação de métricas de âmbito arquitetural e recursos para integração/navegação entre visualização e edição de código-fonte.

Referências

- Ambler, S. W. (1998). *Building Object Applications That Work: Your Step-by-Step Handbook for Developing Robust Systems with Object Technology*. Cambridge University Press.
- Caserta, P. and Zendra, O. (2011). Visualization of the Static aspects of Software: a survey. *IEEE Transactions on Visualization and Computer Graphics*, 17(7):913–933.
- Chidamber, S. R. and Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493.
- da Silva Seibt, P. R. R. (2001). Ferramenta para cálculo de métricas em softwares orientados a objetos. In *Trabalho de Conclusão de Curso - Graduação em Ciência da Computação - Universidade Regional de Blumenau*.
- Dantas, M. M. M. (2008). Métricas para acoplamento e coesão em sistemas orientado a objetos em ambiente de visualização de software. In *Trabalho de Conclusão de Curso - Graduação em Ciência da Computação - Universidade Federal da Bahia (UFBa)*.
- Lanza, M. and Marinescu, R. (2010). *Object-Oriented Metrics in Practice. Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-oriented Systems*. Springer Verlag.
- Lorenz, M. and Kidd, J. (1994). *Object-oriented software metrics: a practical guide*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.
- Olague, H. M., Etzkorn, L. H., Gholston, S., and Quattlebaum, S. (2007). Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes. *IEEE Trans. Softw. Eng.*, 33(6):402–419.
- Purao, S. and Vaishnavi, V. (2003). Product metrics for object-oriented systems. *ACM Comput. Surv.*, 35(2):191–221.
- Qt-Project (2013). Qt creator 2.8 online documentation. <http://qt-project.org/doc/qtcreator-2.8>. Acesso: Agosto de 2013.
- Rosenberg, L. H. (1998). Applying and Interpreting Object Oriented Metrics. In *Software Technology Conference*. NASA Software Assurance Technology Center (SACT).
- Sarkar, S., Kak, A. C., and Rama, G. M. (2008). Metrics for measuring the quality of modularization of large-scale object-oriented software. *IEEE Trans. Software Eng.*, 34(5):700–720.

ReuseDashboard: Apoiando Stakeholders na Monitoração de Programas de Reutilização de Software

Marcelo Palmieri, Marcelo Schots, Cláudia Werner

Programa de Engenharia de Sistemas e Computação (PESC) – COPPE/UFRJ
Caixa Postal 68.511 – 21945-970 – Rio de Janeiro, RJ – Brasil

{palmieri, schots, werner}@cos.ufrj.br

Abstract. *Software reuse programs provide several benefits, such as increase in productivity, decrease of time-to-market, amongst others. However, reuse programs can be hard to be implemented in organizations due to the large amount of information, which is usually not targeted to the stakeholder involved in a reuse task. This leads to cognitive overloading, despite other problems. This work presents ReuseDashboard, a mechanism that aims to support reuse stakeholders by providing relevant information through visual analytics, based on reuse-related metrics. The mechanism supports software reuse programs in both development level (development for and with reuse) and management level.*

Resumo. *Programas de reúso de software proveem diversos benefícios, como o aumento da produtividade, a diminuição do time-to-market, dentre outros. No entanto, a implementação de programas de reúso nas organizações pode ser difícil devido à grande quantidade de informação, que usualmente não é direcionada ao stakeholder envolvido em uma tarefa de reúso. Isto resulta em sobrecarga cognitiva, além de outros problemas. Este trabalho apresenta o ReuseDashboard, um mecanismo que visa apoiar stakeholders de reúso provendo informações relevantes por meio de visual analytics, baseado em métricas relacionadas a reúso. O mecanismo apoia programas de reúso tanto em nível de desenvolvimento (com e para reúso) quanto em nível gerencial.*

1. Introdução e Motivação

A reutilização de software é a prática de desenvolvimento na qual artefatos de software preexistentes ou conhecimento de projetos anteriores são utilizados para a criação de novos produtos de software [Frakes *et al.* 2005]. Tal prática propicia inúmeros benefícios ao longo do processo de desenvolvimento, tais como: (i) a diminuição do esforço de implementação, devido à utilização de um mesmo produto de trabalho por várias vezes, culminando no aumento da produtividade, (ii) a amortização de custos de inspeção e teste, favorecendo o aumento da qualidade, (iii) a redução do tempo de entrega do produto, (iv) a facilitação na definição de padrões (*patterns*) e normas, e (v) a promoção da interoperabilidade e compatibilidade (Frakes *et al.*, 1994).

Entretanto, a reutilização pode ser uma tarefa custosa para o desenvolvimento para ou com reutilização. No desenvolvimento com reutilização, o desenvolvedor precisa conhecer o software reutilizável e suas características, bem como compreender sua execução, de forma a poder reutilizá-lo devidamente. Já no desenvolvimento para

reutilização, é preciso que o software possua algumas características importantes para reutilização, tais como generalidade e flexibilidade, além de estar enquadrado na política de reutilização da organização [Caldiera e Basili, 1991].

Além disso, a implementação de programas de reutilização nas organizações pode ser difícil devido a problemas de gestão, falta de compreensão, falta de incentivos financeiros e sobrecarga cognitiva, dentre outros [Kim e Stohr, 1998]. Muitas vezes isto se deve à grande quantidade de informação a ser analisada e à falta de ferramentas e técnicas de apoio à reutilização de software, que não consideram os diferentes *stakeholders* envolvidos no processo de reutilização e seus interesses em particular.

Diehl (2007) afirma que a compreensão de software é uma atividade complexa, que requer recursos específicos para facilitar o seu desenvolvimento. É neste contexto que técnicas de visualização, combinadas com a utilização de métricas apropriadas, podem facilitar o entendimento do software e seu desenvolvimento, promovendo representações intuitivas e relevantes ao contexto de cada *stakeholder* envolvido no desenvolvimento do software [Lanza e Marinescu, 2006]. Faz-se necessário, no entanto, identificar mecanismos adequados, abstrações apropriadas, e obter evidências sobre o estímulo da percepção humana e habilidades cognitivas [Schots *et al.*, 2012].

O apoio visual na tomada de decisão tem recebido atenção especial da indústria e da academia. Nas últimas décadas, o rápido crescimento de dispositivos de armazenamento de dados junto aos meios de criar e coletar dados influenciaram nossa forma de lidar com a informação [Keim *et al.*, 2008; Schots *et al.*, 2012]. Neste cenário, uma subárea que tem se sobressaído é a de *visual analytics*, considerada “a ciência do raciocínio analítico facilitado por interfaces visuais interativas” [Thomas & Cook, 2006]. Seu objetivo é tornar transparente a forma de o ser humano processar dados e informações, visando o raciocínio analítico [Keim *et al.*, 2008].

A utilização de informações de forma analítica pode ser transportada para o cenário de reutilização de software visando apoiar a atividade de compreensão do software. Por exemplo, a combinação de métricas de software com técnicas de visualização pode fornecer sumarizações dos dados extraídos, exibindo-os de forma agregada através de abstrações visuais intuitivas para cada *stakeholder* do programa de reutilização de software, permitindo que cada *stakeholder* customize as informações a serem apresentadas e suas representações visuais conforme sua necessidade.

Neste sentido, este trabalho apresenta o ReuseDashboard, um mecanismo interativo voltado para programas de reúso, que faz uso de métricas e *visual analytics* visando estimular o envolvimento dos *stakeholders*, provendo informações relevantes a cada papel envolvido em tarefas de reúso. O artigo está organizado da seguinte forma: a Seção 2 exibe trabalhos relacionados, a Seção 3 apresenta o ReuseDashboard, a Seção 4 descreve um cenário ilustrativo, e a Seção 5 contém as considerações finais.

2. Trabalhos Relacionados

Kuipers *et al.* (2007) apresentam um método baseado em ferramentas para a monitoração de software, visando a qualidade do mesmo. Assim como o ReuseDashboard, o trabalho dá ênfase nas tarefas dos *stakeholders*, exibição de visualizações e utilização de métricas. Entretanto, existem algumas limitações, a saber:

(i) a utilização de um modelo fixo para a avaliação de qualidade do software, baseado em poucas métricas, não se adequando às necessidades de cada *stakeholder*, dificultando assim a tomada de decisão; (ii) poucos tipos de visualizações, com informações categorizadas apenas em “muito alto”, “alto”, “moderado” e “baixo”; e (iii) a impossibilidade de investigar os resultados da análise em um ambiente de desenvolvimento indicando a que trecho do código corresponde um dado, o que facilitaria o desenvolvedor na localização de possíveis problemas.

Plösch *et al.* (2008) apresentam uma abordagem que tem por objetivo apoiar a avaliação de qualidade do software pelos *stakeholders* envolvidos no processo de desenvolvimento. O trabalho foi desenvolvido como um *plugin* do Eclipse, e utiliza um modelo de qualidade baseado no padrão ISO, oferecendo uma grande quantidade de métricas e atributos de qualidade para avaliação, apoiando diferentes *stakeholders*. Para realizar a avaliação de um software, é necessário preencher uma grande quantidade de informações, e vários relatórios são gerados. Além disso, não é mencionado nenhum apoio visual para a representação das informações geradas, dificultando a tomada de decisão. Há pouca flexibilidade na construção de modelos de avaliação (que deve ser feita via código), e todas as informações são apresentadas apenas na IDE, fazendo com que todos os *stakeholders* precisem utilizá-la para obter tais informações, o que pode ser incômodo e não muito familiar para *stakeholders* de nível mais gerencial.

3. ReuseDashboard

O mecanismo proposto neste trabalho, nomeado de ReuseDashboard, visa auxiliar os diversos *stakeholders* do processo de desenvolvimento no acompanhamento de um programa de reutilização de software, provendo informações visuais analíticas. Para isto, a abordagem permite a criação de um plano de avaliação a partir de um conjunto de métricas relacionadas à reutilização, que pode ser utilizado para apoiar atividades de reúso, tais como a identificação de componentes, a avaliação de qualidade do software, a identificação de anomalias etc. As medidas são, então, apresentadas por meio de visualizações configuráveis, a fim de facilitar a análise e tomada de decisão. Além disso, visando fornecer um ambiente heterogêneo que permita que cada *stakeholder* acompanhe o programa de reúso sem sair de suas atividades habituais, o ReuseDashboard é compatível com plataformas *desktop* e dispositivos móveis.

Os passos executados pelo mecanismo são: (i) identificação e extração de métricas relacionadas a reúso a partir de projetos de software, (ii) avaliação das métricas extraídas a partir de um plano de avaliação configurável, e (iii) visualização multidispositivo das métricas extraídas e dos resultados da análise realizada por meio do plano de avaliação. Este mecanismo é uma evolução da abordagem proposta em [Palmieri e Werner 2012], contemplando múltiplos *stakeholders*.

Por se tratar de um mecanismo multidispositivo, o ReuseDashboard está sendo desenvolvido em um ambiente web, visando alcançar tanto plataformas móveis quanto plataformas *desktop*, estando ainda integrado com o ambiente de desenvolvimento Eclipse. O mecanismo também possui a característica de ser independente de sistema operacional. Tais requisitos são importantes para se adequar à realidade de trabalho de cada um dos *stakeholders* envolvidos. Indivíduos com perfil mais técnico (tais como desenvolvedores) podem usufruir da integração do mecanismo com a IDE Eclipse,

enquanto indivíduos com perfil mais gerencial podem utilizar o ReuseDashboard a partir de seus computadores pessoais ou de dispositivos móveis (e.g., *tablets* e *smartphones*) que possuam um navegador (*browser*) disponível.

A arquitetura do ReuseDashboard está dividida em dois módulos principais. O primeiro, chamado MEEP (*Metrics Extraction and Evaluation Plan*), trata-se de um *plugin* da IDE Eclipse, responsável por fazer a extração e a análise das métricas a partir do código fonte de projetos de software. O segundo, denominado Viz (*Visualization*), é um projeto para web, com o objetivo de fornecer, por meio de um *dashboard*, informações analíticas – isto é, informações que já foram pré-analisadas (como por exemplo, média e somatório) – dos dados extraídos e analisados no primeiro módulo.

3.1. Extração de Métricas e Plano de Avaliação

O MEEP fornece uma interface para que os *stakeholders* possam utilizar planos de configuração previamente elaborados (que podem ser customizados) ou criar seus próprios planos. Tais planos são utilizados para analisar projetos em Java contidos na IDE Eclipse. O plano de avaliação se baseia nas estratégias de detecção presentes em [Lanza e Marinescu, 2006] e consiste em um conjunto de associações comparativas entre medidas e limiares (*thresholds*) que funcionam como uma regra que caracteriza uma determinada situação. Caso o resultado de todas as comparações medida-limiar seja verdadeiro, isto indica a ocorrência da situação especificada.

A Figura 1 ilustra um possível plano de avaliação para a identificação de componentes candidatos à reutilização, com base em métricas extraídas de [Cho *et al.*, 2001], com valores customizados ao cenário de uma determinada empresa. Cabe ressaltar que tais valores são fortemente dependentes do tipo e tamanho do projeto. A Figura 2, por sua vez, exibe a tela de seleção e criação de um plano de avaliação.



Figura 1 – Estratégia de detecção (adaptada de [Lanza e Marinescu 2006]) para a identificação de candidatos à reutilização

Metrics	Suggested Value
Number of Interfaces	20
McCabe Cyclomatic Complexity	1
Total Lines of Code	22
Instability	6
Number of Parameters	2
Lack of Cohesion of Methods	13
Efferent Coupling	5
Number of Static Methods	17
Normalized Distance	8
Abstractness	7

Plan's Name: Plan A		
Metrics	Comparison	Value
McCabe Cyclomatic Complexity	<	3
Number of Parameters	<	5
Number of Classes	<	15

Name	Description	Selection
Plan A	Plan A	Selected
Plan B	Plan B	Available

Figura 2 – Exemplo de Plano de Avaliação

Todos os valores extraídos e o resultado da avaliação são repassados para o Viz, que contém um conjunto de gráficos e métricas que são selecionados e combinados de forma a atender aos *stakeholders* em suas diferentes necessidades.

O MEEP foi baseado no *plugin* Metrics2 (<http://metrics2.sourceforge.net/>), do qual foi reutilizada a parte da arquitetura responsável pela leitura das métricas a serem utilizadas e a extração das métricas propriamente ditas. Este *plugin* foi utilizado por ser de fácil reutilização, ser de código aberto e permitir a inclusão de novas métricas, além de disponibilizar um grande número de métricas voltadas para reutilização (e.g., falta de coesão, profundidade da árvore de herança e número de filhos).

Tanto as informações referentes às extrações quanto os planos de avaliação são armazenados em um banco de dados, que serve como ponte entre a MEEP e a Viz, compartilhando os dados necessários para as visualizações do ReuseDashboard.

3.2. Visualização

Visando atender aos diferentes *stakeholders* de reutilização, o ReuseDashboard possui visões customizáveis, que permitem selecionar quais pares métricas-gráficos devem ser exibidos para cada *stakeholder*. Além disso, para tornar o ambiente apropriado para cada *stakeholder*, o mecanismo pode ser utilizado tanto em plataformas *desktop* quanto em dispositivos móveis, como por exemplo *tablets*. Para isto, o Viz utiliza duas tecnologias: Java EE, para a parte servidor, e o framework Ext JS (<http://www.sencha.com/products/extjs/>), para a parte cliente. Este framework foi escolhido por se tratar da tecnologia HTML 5 que possibilita uma boa interatividade com objetos visuais, além de possuir vários tipos de gráficos, que são utilizados na parte visual do ReuseDashboard. A Figura 3 ilustra as telas de visualização do ReuseDashboard. A análise realizada por meio do plano de avaliação é exibida na parte inferior da tela, apontando itens nos quais foram detectados problemas.

4. Cenário de Exemplo

Nesta seção, são abordados dois cenários de utilização do ReuseDashboard para exemplificar um dos possíveis benefícios deste mecanismo.

No primeiro cenário, o desenvolvedor de um sistema online precisa melhorar o tempo de processamento. Para isto, este desenvolvedor decidiu investigar a complexidade ciclomática das classes e métodos do componente reutilizado, a fim de encontrar possíveis casos de alta complexidade. Desta forma, o desenvolvedor utiliza o plano de avaliação *Plan A*, exibido na Figura 2, em conjunto com as visualizações fornecidas pelo ReuseDashboard. Assim, ele pode constatar alguns casos cuja complexidade ciclomática era muito alta, além de ter a informação dos itens que ficaram fora dos padrões definidos no plano de avaliação. Estes fatos são evidenciados na Figura 3 (parte superior).

No segundo cenário, um gerente está buscando algum indicativo sobre o esforço de efetuar uma extensão das funcionalidades de um componente. Para isso, este gerente analisa métricas que informem a flexibilidade do código deste componente. A partir das métricas disponíveis no plano de avaliação, ele observa que o nível de abstração (*abstractness*) e o nível de instabilidade (*instability*) podem ser indicativos relevantes. A Figura 3 (parte inferior) ilustra as visualizações destas métricas, que podem ajudar o

gerente nesta tarefa. É possível notar que os valores estão próximos de zero, podendo indicar (dependendo do cenário do projeto) que o código do componente possui a flexibilidade desejada, isto é, é possível de alterações com mais facilidade.

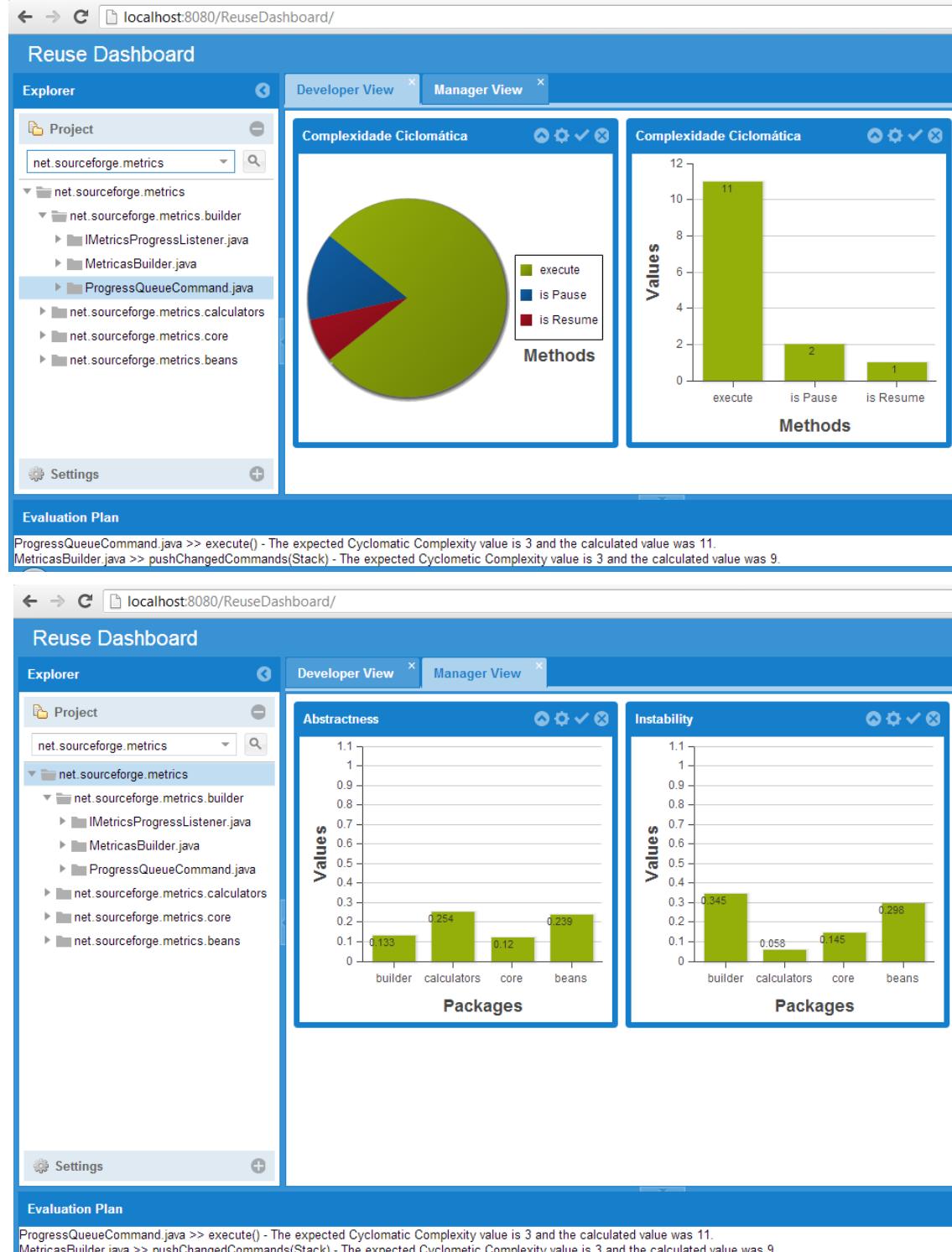


Figura 3 – Visão do Desenvolvedor (parte superior) e do Gerente (parte inferior)

5. Considerações Finais

Este trabalho apresentou o mecanismo ReuseDashboard, que visa apoiar os diversos *stakeholders* presentes no processo de desenvolvimento do software no acompanhamento de um programa de reutilização. O mecanismo disponibiliza informações analíticas em um *dashboard*, que pode ser consultado em plataformas móveis e *desktop*, sendo que as visualizações podem ser parametrizadas para cada *stakeholder* conforme suas necessidades. As informações visualizadas são baseadas em métricas relacionadas à reutilização, extraídas de projetos Java.

Embora a apresentação da visualização seja compatível com plataformas móveis e *desktop*, a interatividade atualmente fica comprometida em dispositivos móveis, em função do framework atualmente utilizado. Para tratar esta limitação, a camada de visualização incorporará os recursos providos pelo framework Sencha Touch (<http://www.sencha.com/products/touch>), que é compatível com as demais camadas do módulo Vis e permite portar aplicações web escritas em Ext JS para plataformas móveis. Outra limitação relacionada à visualização é a variedade de gráficos, que atualmente é limitada às que são providas pelo framework.

Como próximos passos, pretende-se efetuar a inclusão de novas métricas relacionadas ao reúso (sendo adquiridas a partir de uma revisão sistemática) e efetuar a integração do mecanismo com o ambiente APPRAiSER, em desenvolvimento no contexto de uma tese de doutorado da COPPE/UFRJ, que visa apoiar a reutilização através da percepção por meio de técnicas e recursos de visualização de software [Schots *et al.*, 2012].

Pretende-se, ainda, executar uma avaliação do ReuseDashboard por meio de um experimento, com o objetivo de avaliar a efetividade no acompanhamento do programa de reutilização de software. Serão considerados itens como a facilidade na interpretação das informações visuais, o tempo gasto na análise, e o índice de acertos (baseado num gabarito pré-determinado). Para isto, deverá ser realizada a definição de um plano de avaliação e, a partir do mesmo, a análise e a monitoração, envolvendo tarefas de identificação de componentes, avaliação da qualidade do software, ou identificação de problemas de implementação relacionados à reutilização.

Agradecimentos

Ao Zaedy Sayão, pelo suporte à implementação, e ao CNPq, pelo apoio financeiro.

Referências

- Caldiera, G., e Basili, V. R., (1991), “Identifying and qualifying reusable software components”. Computer, vol. 24, no. 2, pp. 61-70, Fevereiro.
- Cho, E. S., Kim, M. S., Kim, S. D., (2001), “Component metrics to measure component quality”. In APSEC, IEEE Computer Society, pp. 419-426, Dezembro.
- Diehl, S, (2007). “Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software”, 1 ed. Springer
- Frakes, W. B., & Isoda, S., (1994), “Success factors of systematic reuse”, IEEE Software, vol 11, no. 5, pp. 15-19, Setembro.

- Frakes, W.B., Kang, K., (2005), “Software reuse research: Status and future”, IEEE Transactions on Software Engineering, v. 31, n. 2, pp. 529-536, Julho.
- Keim, D. A., Mansmann, F., Schneidewind, J., Thomas, J., & Ziegler, H. (2008). *Visual analytics: Scope and challenges*, pp. 76-90. Springer Berlin Heidelberg.
- Kim, Y. and Stohr, E. A. (1998), “Software Reuse: Survey and Research Directions”. Journal of Management Information Systems, Vol. 14, Issue 4, pp. 113-147, March.
- Kuipers, T., Visser, J., Vries, G., (2007), "Monitoring the Quality of Outsourced Software", Workshop on Tools for Managing Globally Distributed Software Development (TOMAG), Agosto.
- Lanza, M., Marinescu, R. (2006) “Object-Oriented Metrics in Practice”. Springer-Verlag Berlin Heidelberg New York, 1st edition.
- Palmieri, M., Werner, C., (2012), “ReuseInvestigator: Um Mecanismo de Identificação de Componentes Reutilizáveis com o Apoio de Visualizações”, II Workshop de Teses e Dissertações do CBSOFT (WTDSOFT), Natal, Setembro, pp. 1-5.
- Plösch, R., Gruber, H., Pomberger, G., Saft, M., Schiffer, S., (2008), “Tool Support for Expert-Centred Code Assessments”, 1st International Conference on Software Testing, Verification, and Validation, pp. 258-267, Lillehammer, Abril.
- Schots, M., Werner, C., Mendonça, M., (2012). “Awareness and Comprehension in Software/Systems Engineering Practice and Education: Trends and Research Directions”, 26th Brazilian Symposium on Software Engineering (SBES), Natal, Setembro.
- SourceMiner, (2013), Disponível em <http://www.sourceminor.org/index.html>.
- Thomas, J. J., and Cook, K. A. (2006). “A visual analytics agenda”. *IEEE Computer Graphics and Applications*, v. 26, n. 1, pp. 10-13.

Uma Abordagem Baseada em Eventos para Adaptação Automática de Aplicações para a Nuvem*

Michel Araújo Vasconcelos¹, Davi Monteiro Barbosa²
Paulo Henrique M. Maia³, Nabor C. Mendonça¹

¹Programa de Pós-Graduação em Informática Aplicada (PPGIA)
Universidade de Fortaleza (UNIFOR)
Av. Washington Soares, 1321, Edson Queiroz, CEP 60811-905 Fortaleza – CE

²Centro Universitário Estácio do Ceará
Rua Vicente Linhares, 308, CEP 60135-270, Fortaleza – CE

³Centro de Ciências e Tecnologia (CCT)
Universidade Estadual do Ceará (UECE)
Campus do Itaperi, Av. Paranjana, 1700, CEP 60740-903 Fortaleza – CE

{michel.vasconcelos, davimonteiro.ce}@gmail.com
pauloh.maia@uece.br, nabor@unifor.br

Abstract. *Cloud Computing is a new computing paradigm whose benefits (pay-per-use business model, low infrastructure overhead, high availability and scalability) are attracting the interest of many organizations. However, those companies can face several obstacles to adopt this new paradigm. One major difficulty concerns the migration of their legacy applications since they may require many code changes in order to be deployed in the cloud. These modifications often demand a great effort from the developers and are error prone. This work proposes a novel event-based approach to the automatic adaptation of legacy applications to the cloud that brings as main benefits a greater reusability level and integration with other existing software adaptation solutions.*

Resumo. *A computação em nuvem é um novo paradigma computacional cujos benefícios (como pagamento sob demanda, baixo investimento em infraestrutura física e alta escalabilidade de recursos) têm cada vez mais atraído o interesse do mundo corporativo. Apesar disso, muitas organizações encontram dificuldades em adotar esse novo paradigma. Uma dessas dificuldades é a necessidade de migrar suas aplicações legadas, que podem exigir muitas mudanças em seu código fonte para que possam ser implantadas no ambiente de nuvem. Tais mudanças geralmente demandam um esforço considerável por parte dos desenvolvedores e podem introduzir novos erros de implementação. Este trabalho propõe uma nova abordagem baseada em eventos para a adaptação automática de aplicações para a nuvem, que traz como benefícios principais um alto nível de reusabilidade e integração com outras tecnologias de adaptação de software existentes.*

Palavras-chave: Computação em nuvem, adaptação automática de aplicações, migração, evolução.

*Este trabalho é parcialmente financiado pela Microsoft Research, através de um SEIF Award 2013.

1. Introdução

A Computação em Nuvem é um novo paradigma computacional que está se disseminando por toda indústria de Tecnologia da Informação rapidamente [Leavitt 2009]. Apesar de seus benefícios, como pagamento sob demanda, baixo investimento em infraestrutura física e alta escalabilidade de recursos, muitas organizações estão tendo dificuldades para migrar seus sistemas de software existentes para provedores de nuvem, como Windows Azure e Amazon EC2. Tal migração é inherentemente complexa porque as decisões de migração devem levar em conta (e podem ser influenciadas por) vários fatores, muitas vezes conflitantes, tanto técnicos (por exemplo, esforço de adaptação, escalabilidade, segurança) quanto não técnicos (por exemplo, perda da governança sobre dados e aplicações, reputação do provedor, restrições legais) [Beserra et al. 2012, Khajeh-Hosseini et al. 2012, Tran et al. 2011].

Este trabalho foca nos desafios técnicos envolvidos no processo de adaptação de aplicações existentes para que elas possam ser efetivamente migradas para a nuvem. Há dois desafios-chave para realizar essa migração [Andrikopoulos et al. 2012, Frey e Hasselbring 2011b]. O primeiro consiste em garantir que a aplicação a ser migrada não viole nenhuma restrição de ambiente imposta pela plataforma de nuvem de destino [Frey e Hasselbring 2011b]. Por exemplo, alguns provedores de serviços de plataforma de desenvolvimento na nuvem (*Platform-as-a-Service* – PaaS), como Windows Azure e Google App Engine, limitam o comportamento dos serviços neles implantados ao executá-los em um *sandbox* que não permite ou restringe o uso de funções básicas do sistema operacional, como abrir conexões de sockets ou acessar o sistema de arquivos local. Portanto, migrar aplicações que usam esse tipo de função para esses provedores pode não ser possível a menos que a aplicação seja alterada para evitar essas restrições. Analogamente, em algumas nuvens que fornecem infraestrutura como serviço (*Infrastructure-as-a-Service* – IaaS), como o Amazon EC2, discos virtuais montados localmente em uma máquina virtual são transientes, o que significa que as aplicações que utilizam o sistema de arquivos local para armazenamento teriam que ser mudados para usar um outro mecanismo de persistência a fim de manter seu correto comportamento na nuvem.

O segundo desafio envolve adaptar a aplicação de forma que ela tire amplo proveito do ambiente de nuvem [Andrikopoulos et al. 2012]. Por exemplo, para reduzir a sobrecarga de comunicação, o mecanismo de interação original da aplicação poderia ser substituída por um serviço de mensagem mais robusto, fornecido nativamente pela plataforma de nuvem. Em outro exemplo, para alcançar uma maior escalabilidade, o banco de dados relacional da própria aplicação poderia ser substituída por um serviço de armazenamento do tipo NoSQL baseado na nuvem.

Para realizar a migração e abordar os dois aspectos supra mencionados, a aplicação em questão pode necessitar mudanças consideráveis em sua arquitetura e código-fonte. Isso requer a realização extensiva de testes na nuvem, para garantir que o comportamento original do sistema foi preservado e que nenhum erro foi introduzido durante o processo de migração. Embora existam estudos recentes sobre detecção automática de potenciais violações de restrição no código-fonte da aplicação a ser migrada para a nuvem [Frey e Hasselbring 2011b], as transformações de código necessárias para que a aplicação fique de acordo com e se beneficie do ambiente de nuvem de destino

não têm sido abordadas por pesquisas recentes ou apoiadas por ferramentas. Além disso, a falta de tais ferramentas torna a tarefa de migração consideravelmente mais complexa para o desenvolvedor e suscetível a erros, pois, para realizar as alterações necessárias no código fonte manualmente, o desenvolvedor precisaria de um profundo conhecimento tanto sobre a estrutura interna do sistema quanto sobre as APIs e restrições técnicas específicas da plataforma de nuvem de destino.

Este trabalho propõe uma nova abordagem para a adaptação automática de aplicações a fim de reduzir a complexidade de migração para a nuvem. Ela se baseia na identificação e captura de trechos de código da aplicação que devem ser alterados para funcionarem na nuvem (de acordo com os desafios mencionados anteriormente) em forma de eventos. Através de uma arquitetura *publish/subscribe* [Eugster et al. 2003], os eventos são publicados e consumidos por adaptadores desenvolvidos para plataformas de nuvem específicas. Por utilizar uma arquitetura desacoplada e extensível, a abordagem proposta traz como benefícios a independência de técnica de interceptação do código e de plataforma de nuvem de destino, o que implica em um alto nível de reusabilidade e integração com outras tecnologias de adaptação de código existentes.

O restante do artigo está organizado da seguinte forma: a seção 2 discute os principais trabalhos relacionados ao tema em questão, enquanto a seção 3 apresenta o modelo de arquitetura *publish/subscribe*, base da nossa abordagem. A seção 4 descreve um exemplo motivador mostrando problemas para migração de uma aplicação para a nuvem. A seção 5 detalha o funcionamento da abordagem proposta. Por fim, as considerações finais e trabalhos futuros são mostrados na seção 6.

2. Trabalhos Relacionados

A maioria dos estudos feitos na migração de aplicações legadas para nuvem propõe metodologias e processos que auxiliem na decisão de migrar ou não uma dada aplicação para uma dada plataforma de nuvem (e.g. [Beserra et al. 2012, Khajeh-Hosseini et al. 2012]). Este trabalho complementa esses processos já que foca em auxiliar o responsável pela migração a tratar as mudanças necessárias nas aplicações a serem migradas para a nuvem.

Recentemente, houve uma mudança no sentido de prover soluções técnicas para os desafios existentes na implantação de aplicações legadas em uma nuvem. Em [Andrikopoulos et al. 2012], os autores elencam desafios e possíveis soluções para migrar uma solução de software para a nova plataforma. Eles identificam categorias para se distribuir um sistema na nuvem que abrangem desde uma implantação simples, em uma nuvem do tipo IaaS, a uma migração completa fazendo uso de vários recursos da plataforma destino. Os autores descrevem uma aplicação através de camadas e as dispõem segundo os modelos possíveis de distribuição em uma nuvem. Para cada camada, eles definem questões básicas a serem resolvidas antes de realizar a migração. Por fim, eles propõem uma comparação entre as camadas lógicas da aplicação, as categorias possíveis de migração para uma nuvem, seus benefícios e desvantagens. Desse trabalho [Andrikopoulos et al. 2012] usamos os questionamentos e desafios listados como insumo para construir as adaptações adequadas, que podem ser usadas para atender a uma categoria específica de migração.

Em [Frey e Hasselbring 2011a], é proposta uma abordagem de migração para nu-

vem chamada CloudMIG. A abordagem define um modelo conceitual e extensível, no qual é possível especificar um ambiente de nuvem, suas restrições¹ e violações². Esse trabalho também propõe um modo de avaliação do sistema candidato à migração e de sua conformidade em relação a uma nuvem específica. Enquanto o CloudMIG foca no processo para identificar restrições e migrar a aplicação, as modificações no sistema base devem ser feitas manualmente. As violações formam elementos de primeira classe responsáveis por delimitar que pontos devem ser ajustados na aplicação base. Já neste trabalho, as violações são substituídas por interceptadores e eventos. O interceptador atua na captura de eventos pré-definidos e os propaga para tratamento (ver seção 5). Usamos os modelos conceituais definidos em [Frey e Hasselbring 2011a] como guias na construção dos primeiros interceptadores (e.g., interceptar evento de escrita em disco em nuvens do tipo PaaS).

Em [Maia et al. 2007], é apresentado um modo de desenvolvimento que permite adaptar *threads* de uma aplicação através de aspectos e aplicá-la em um ambiente de GRID computacional. Esse modo se assemelha com as adaptações aqui propostas já que ambos usam técnicas de transformação de código não-intrusivas. A nossa proposta é mais abrangente pois concentra-se em outros serviços da nuvem além do paralelismo (e.g., armazenamento em disco, persistência de dados, troca de mensagens, etc.).

3. O Modelo de Arquitetura *Publish/Subscribe*

Publish/subscribe é um modelo de arquitetura utilizado em aplicações distribuídas de larga escala que provê uma comunicação desacoplada e flexível entre seus componentes [Eugster et al. 2003]. Ele se baseia em duas entidades principais, produtores e consumidores, que se comunicam de forma assíncrona através de mensagens que carregam dados, chamadas de eventos.

Os produtores (*publishers*) publicam eventos no sistema, enquanto os consumidores (*subscribers*) expressam seu interesse em receber certos eventos através de um mecanismo de subscrição. No momento em que um produtor publica um evento, ele é recebido por um gerenciador de eventos, que é um *middleware* orientado a mensagens denominado de *broker*, o qual sabe quais consumidores devem ser notificados do evento. Por ser uma comunicação assíncrona, tanto produtor quanto consumidor não precisam estar ativos ao mesmo tempo quando uma mensagem é enviada, pois ela é armazenada pelo *broker* durante o tempo necessário até que o consumidor possa recebê-la.

O modelo *publish/subscribe* pode ser classificado em três categorias, de acordo com a forma como os consumidores registram seu interesse por eventos[Eugster et al. 2003]. São elas:

- *baseada em tópico*: é o esquema mais antigo e estende a noção de canais, utilizados em comunicação em grupo, com métodos para classificar e caracterizar o conteúdo dos eventos. Participantes podem publicar eventos e subscrever para tópicos específicos identificados por palavras-chaves.
- *baseado em conteúdo*: essa categoria oferece maior expressividade à inscrição dos consumidores. Ao invés de se registrar em um tópico, um consumidor especi-

¹Limitações impostas pelo ambiente de nuvem destino

²Possíveis ações do sistema que não podem ser executadas devido a alguma restrição da nuvem

fica filtros, baseados no conteúdo dos eventos gerados, que serão aplicados para identificar se um evento é de seu interesse.

- *baseado em tipo*: é uma extensão do modelo baseado em conteúdo onde o tipo dos eventos desejados pode ser especificado em tempo de compilação por meio da parametrização da interface do evento de interesse do consumidor.

4. Exemplo Motivador

Para exemplificar a problemática abordada neste artigo, considere uma aplicação que faz uso intensivo de operações de leitura e escrita em disco. Deseja-se que essa aplicação passe a salvar os arquivos na nuvem ao invés de no disco local. Uma possível solução seria fazer a migração completa da aplicação para a nuvem de destino. Porém, como discutido na seção 1, isso pode envolver uma série de restrições técnicas relativas ao provedor de nuvem escolhido. Além disso, o esforço para fazer essa migração pode ser muito grande e desnecessário, uma vez que não foi solicitado que a aplicação execute na nuvem, mas apenas que salve os arquivos nela. Dessa forma, uma nova solução seria adaptar trechos da aplicação para que, quando solicitado que um arquivo seja salvo, este seja salvo na nuvem e não localmente.

Apesar de eficiente, essa solução pode trazer alguns problemas: primeiramente, o código da aplicação deverá ser alterado, o que pode introduzir novos erros; segundo, caso existam vários pontos onde esse trecho de código se repita, o esforço para adaptação pode ser grande e, consequentemente, mais propício a erros; por fim, a modularidade da aplicação pode ficar comprometida, uma vez que código que não é de interesse funcional da aplicação (código de acesso à nuvem) está sendo introduzido e espalhado entre seus módulos.

Uma outra alternativa seria usar técnicas de interceptação de código, como aspectos [Kiczales et al. 1997], que permitem alterar o fluxo de execução em pontos específicos do código. Neste caso, o aspecto conteria o código de envio de arquivo para a nuvem de destino. Com isso, a aplicação continua executando normalmente, porém é adaptada dinamicamente ao atingir um dos pontos de interceptação (uma chamada de método, por exemplo). As vantagens dessa abordagem são a possibilidade de reuso do aspecto entre aplicações e a separação de interesses. Apesar de melhorar a modularidade, essa solução ainda obriga que código de migração seja específico para uma nuvem de destino, o que pode ser limitante caso seja necessário trabalhar com outro provedor de nuvem em algum momento.

A seguir, mostramos uma solução que permite um maior decoplamento entre as técnicas de interceptação do código da aplicação e as chamadas aos serviços de nuvem oferecidos pelo provedor.

5. Abordagem Proposta

A abordagem proposta tem como principal objetivo possibilitar que uma aplicação legada possa ser adaptada de forma não-intrusiva, ou seja, sem intervenção direta no seu código-fonte, para que essa possa executar em uma determinada plataforma de nuvem. Para tanto, utilizou-se uma arquitetura *publish/subscribe*, conforme mostrada na Figura 1, cujos principais componentes serão discutidos em mais detalhes a seguir.

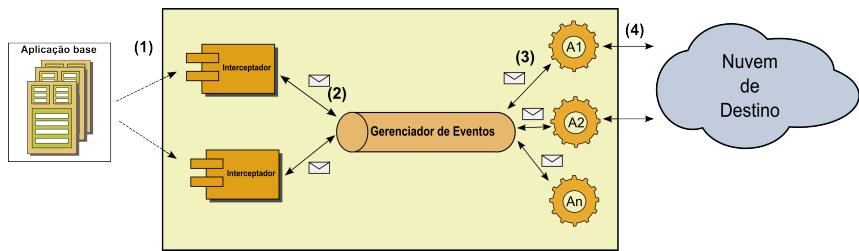


Figura 1. Arquitetura *publish/subscribe* utilizadas na abordagem proposta.

Os interceptadores de código (1) são responsáveis por identificar e capturar, na aplicação legada, os trechos de código que precisam ser adaptados para a plataforma de nuvem. A ideia consiste em interceptar a aplicação em pontos específicos para que seu fluxo de execução seja desviado para a nuvem sem modificar o seu código-fonte. As motivações para realizar uma interceptação podem ser a adição de uma funcionalidade, modificação dos resultados retornados de determinadas funções ou quando o código-fonte da aplicação utiliza uma biblioteca ou arcabouço de terceiros que não pode sofrer modificação.

O interceptador consegue atuar em níveis diferentes, conforme a tecnologia ou necessidade de adaptação do sistema base. Por exemplo, é possível criar um interceptador para atuar em nível de software em uma nuvem PaaS usando aspectos [Kiczales et al. 1997]. Já no caso de uma nuvem IaaS, poder-se-ia construir um interceptador que atue em nível de ambiente (sistema operacional) através de interposição funcional. As informações disponíveis para o interceptador são restritas conforme o seu respectivo nível de atuação. Enquanto no primeiro cenário o interceptador conhece o módulo do sistema que originou a interceptação, o último possui apenas as informações que são passadas para as funções nativas do sistema operacional. Para os nossos testes, usamos o AspectJ³ para interceptação em nível de software e o EasyHook⁴ para interceptação em nível de ambiente.

A criação de um interceptador em nível de software consiste em definir adequadamente quais serão os pontos de interceptação utilizados. Para tratarmos eventos de leitura e escrita em disco, por exemplo, o ponto de interceptação atuará em classes do tipo *java.io.File*, *FileInputStream* e *FileOutputStream*. Já para um interceptador em nível de ambiente no sistema operacional Windows, por exemplo, é necessária a criação de uma DLL (*Dynamic-link library*) que irá conter as funções que serão interceptadas. Essa biblioteca deverá ser registrada, junto com todas suas dependências, dentro do *Global Assembly Cache* (GAC) do Windows, um repositório de código de máquina onde são armazenadas funções que são compartilhadas por vários processos. O registro da biblioteca será feito por um programa executável. Após realizar o registro, a biblioteca será injetada no processo alvo, que é identificado pelo seu PID (*Process ID*). Assim, caso não ocorram erros durante o registro e a injeção da biblioteca, o interceptador funcionará enquanto o processo alvo existir. Quando o processo alvo for encerrado, o interceptador será removido da biblioteca do GAC. A biblioteca que será injetada contém as funções que serão executadas no lugar das funções originais interceptadas. Para tanto, essas funções

³<http://eclipse.org/aspectj>

⁴<http://easyhook.codeplex.com>

devem ter a mesma assinatura.

Note que mesmo sendo necessário utilizar uma tecnologia de interceptação específica, os códigos de interceptação são independentes de aplicação e podem ser reutilizados. Por exemplo, várias aplicações que fazem uso de escrita em arquivo em disco podem se beneficiar de interceptações desse tipo de função. Os dados da interceptação são encapsulados em forma de evento e publicados no Gerenciador de Eventos (2).

O Gerenciador de Eventos é o elemento principal da abordagem. Ele é um *middleware* responsável por fazer a comunicação entre interceptores de código (produtores de eventos) e adaptadores de nuvem (consumidores), e é implementado utilizando um servidor de mensagens, como o Apache ActiveMQ⁵. O Gerenciador de Eventos permite que tanto os interceptores de código quanto os adaptadores de nuvem sejam implementados em diferentes tecnologias, linguagens e protocolos de comunicação. Com isso, um desenvolvedor pode utilizar as técnicas de interceptação e de adaptação da nuvem que lhes forem mais convenientes para uma aplicação, e outras diferentes para uma outra aplicação. Os adaptadores de nuvem declararam seu interesse em eventos específicos ao Gerenciador de Eventos. O Gerenciador de Eventos mantém uma estrutura do tipo *dicionário*, que permite identificar todos os adaptadores interessados em um evento. Ao receber uma mensagem, o Gerenciador de Eventos verifica a lista de adaptadores interessados e a entrega àqueles adaptadores (3).

Os adaptadores de nuvem processam os eventos gerados pelos interceptadores de código. Eles são específicos para a nuvem de destino e implementam o código que transforma um evento na ação que deve acontecer na nuvem (4). Por exemplo, considerando a situação de interceptação de escrita em arquivo em disco, um adaptador de nuvem que está interessado nesse tipo de evento implementa como o arquivo deve ser gravado em uma nuvem determinada. São exemplos de adaptadores de nuvem o JCloud⁶ e LibCloud⁷.

Note que a comunicação entre os componentes da arquitetura está ilustrada através de setas de duas pontas. Isso significa que, muitas vezes, o interceptador precisa receber um retorno do adaptador de nuvem sobre o *status* da operação. Isso pode ser feito através de *callback queues*. Por exemplo, a interceptação em nível de ambiente da função do sistema operacional de escrita em arquivo tem como retorno um dado do tipo *boolean*. Portanto, o adaptador de nuvem deve informar ao interceptador se o arquivo foi gravado com sucesso, para que a aplicação legada não tenha seu comportamento modificado.

6. Considerações Finais

Este trabalho apresentou uma abordagem para adaptação automática de aplicações legadas para a nuvem. Ela baseia-se em uma arquitetura *publish/subscribe* que permite que o desenvolvedor responsável pela migração possa escolher uma tecnologia de interceptação de código e uma plataforma de nuvem de destino específicas. Ela traz como benefícios a independência de tecnologia de interceptação e de nuvem e o aumento do grau de reusabilidade e integração com soluções já existentes.

Atualmente, um estudo de caso está em desenvolvimento para avaliar a abordagem

⁵<http://activemq.apache.org>

⁶<http://jclouds.incubator.apache.org>

⁷<http://libcloud.apache.org>

proposta. Nele, uma aplicação que permite o *download* dos arquivos de um site *web* é adaptado para que, ao invés de salvar os arquivos no disco local, envie e salve-os na nuvem. As interceptações estão sendo feitas tanto em nível de software (com AspectJ) quanto de sistema operacional (com EasyHook). Além disso, estão sendo utilizados dois adaptadores para nuvem: JCloud e libCloud.

Como trabalhos futuros, pretendemos avaliar a abordagem proposta através de mais estudos de caso e utilizando métricas que indiquem o custo-benefício da sua utilização, como tempo de resposta e desempenho. Também planejamos evoluir a abordagem na forma de um *framework*, através do qual o desenvolvedor poderá implementar suas próprias interceptações e seus próprios adaptadores de nuvem.

Referências

- Andrikopoulos, V., Binz, T., Leymann, F., e Strauch, S. (2012). How to adapt applications for the cloud environment. *Computing*, pages 1–43.
- Beserra, P., Camara, A., Ximenes, R., Albuquerque, A., e Mendonca, N. (2012). Cloud-step: A step-by-step decision process to support legacy application migration to the cloud. In *Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA), 2012 IEEE 6th International Workshop on the*, pages 7–16.
- Eugster, P. T., Felber, P. A., Guerraoui, R., e Kermarrec, A.-M. (2003). The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131.
- Frey, S. e Hasselbring, W. (2011a). The cloudmig approach: Model-based migration of software systems to cloud-optimized applications. *International Journal on Advances in Software*, 4(3 and 4):342–353.
- Frey, S. e Hasselbring, W. (2011b). An extensible architecture for detecting violations of a cloud environment’s constraints during legacy software system migration. In *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*, pages 269–278.
- Khajeh-Hosseini, A., Greenwood, D., Smith, J. W., e Sommerville, I. (2012). The cloud adoption toolkit: supporting cloud adoption decisions in the enterprise. *Software Practice & Experience*, 42(4):447–465.
- Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., e Irwin, J. (1997). Aspect-oriented programming. In *Object-Oriented Programming (ECOOP), 1997. 11th European Conference on*, volume 1241, pages 220–242.
- Leavitt, N. (2009). Is cloud computing really ready for prime time? *IEEE Computer*, 42:15–20.
- Maia, M., Maia, P., Mendonca, N., e Andrade, R. (2007). An aspect-oriented programming model for bag-of-tasks grid applications. In *Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on*, pages 789–794.
- Tran, V., Keung, J., Liu, A., e Fekete, A. (2011). Application migration to cloud: a taxonomy of critical factors. In *Proceedings of the 2nd International Workshop on Software Engineering for Cloud Computing, SECLOUD ’11*, pages 22–28, New York, NY, USA. ACM.

Uma abordagem de reengenharia de software orientada por métricas de qualidade

Giuliana Silva Bezerra¹

¹Departamento de Informática e Matemática Aplicada
Universidade Federal do Rio Grande do Norte (UFRN)
Caixa Postal 1524 – 59078-970 – Natal – RN – Brasil

giu.drawer@gmail.com

Abstract. This article describes an approach to reengineering considering the analysis of software metrics. These metrics are indicative of quality and its analysis guides refactoring operations that involve the use of design patterns. The proposed approach is to define the index of maintainability of software through quality metrics and after that analyze design patterns that can be used to satisfy this requirement. Finally, it was made an analysis to assess whether the requirement was met after applying the refactoring operations. Also, it was made a case study to evaluate the proposed approach.

Resumo. Esse artigo descreve uma abordagem de reengenharia considerando a análise de métricas de software. Essas métricas são um indicativo de qualidade e a sua análise orienta operações de refatoração que envolvem o uso de padrões de projeto. A abordagem proposta consiste em definir o índice de manutenibilidade do software através de métricas de qualidade e após isso analisar os padrões de projeto que podem ser utilizados para satisfazer esse requisito. Por fim é feita uma análise final para avaliar se o mesmo foi atendido após aplicar as operações de refatoração. Também é feito um estudo de caso para avaliar a abordagem de reengenharia proposta.

1. Introdução

No ciclo de vida de um software ocorrem diversas mudanças em sua codificação que podem gerar degradação em sua estrutura. Para que um sistema se mantenha escalável e reutilizável é fundamental que hajam requisitos não funcionais que orientem o processo de desenvolvimento. Uma das formas de avaliar esses requisitos é através do uso de métricas de qualidade. Essas métricas estabelecem limites impostos pelo projeto para manter a qualidade do software. Quando o software não atinge os limites impostos pode ser necessário um processo de reengenharia para modificar a estrutura dos componentes sem modificar as funcionalidades do software.

A reengenharia de software consiste na reestruturação um sistema com base em requisitos funcionais e não funcionais. A maior dificuldade está associada aos requisitos não funcionais. É difícil mensurar um requisito não-funcional devido a sua natureza abstrata. Como forma de descrevê-los quantitativamente podem ser utilizadas métricas de qualidade, mas integrá-las num processo de reengenharia não é algo simples. É necessário selecionar um conjunto de métricas que estejam associadas ao requisito analisado. Essa tarefa é complexa devido ao grande número de métricas proposto na literatura [Hitz and Montazeri 1996] [Henderson-Sellers 1995] [Lorenz and Kidd 1994].

O presente trabalho propõe uma abordagem de reengenharia que utiliza métricas de qualidades como ferramentas auxiliares para o processo. Para isso é selecionado um requisito não funcional e a partir dele são analisadas as métricas de qualidade associadas. Após definir os limiares desejados são propostas operações de refatoração de código para alcançá-los. Para orientar essas operações são utilizados padrões de projeto [Gamma et al. 1995]. Para cada padrão aplicado são exibidas as principais métricas afetadas. Por fim o software final é analisado em função dos limiares pré-estabelecidos durante a análise das métricas selecionadas.

2. Reengenharia

Existem diversas aplicações de reengenharia de software. No que diz respeito a manutenção, estão envolvidas atividades de predição de erros, correções, melhorias e evolução [Sommerville 2007]. Na abordagem proposta neste trabalho, a reengenharia terá como objetivo melhorar o índice de manutenabilidade do software. O ciclo de vida do processo de reengenharia proposto pode ser descrito como ilustrado na Figura 1. As etapas propostas são sequenciais, podendo haver ciclos caso um dos objetivos da etapa corrente não seja atendido.

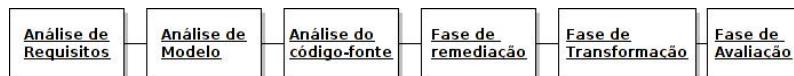


Figura 1. Etapas do processo de reengenharia

Nas etapas de análise serão definidas as métricas e os seus limiares, que devem ser satisfeitos ao final do processo. Esses limiares são uma representação numérica do requisito analisado, para que a avaliação do mesmo seja concreta. São identificados os componentes candidatos a refatoração, com base nas métricas obtidas, e a sua estrutura é analisada. Pode ser necessário um procedimento de engenharia reversa para facilitar essa compreensão. A elaboração de diagramas e documentação das funcionalidades também será útil para permitir uma visão mais alto-nível da estrutura atual do sistema, que será necessária para a etapa de transformação. Nessa etapa é proposta uma solução para atingir as metas estabelecidas. São selecionados os padrões de projeto capazes de obter melhores índices nas métricas medidas e a forma como serão aplicados nos componentes alvo. Por fim o software refatorado é analisado em função de suas métricas para avaliar se o requisito estabelecido foi atendido.

2.1. Etapa 1: Análise de Requisitos e Métricas de qualidade de software

O estudo feito neste trabalho analisou o requisito Manutenibilidade, que pode ser definido como a facilidade em manter um software. Esse requisito mede o custo de correções de erro, mudanças para atender novos requisitos e identificação de deficiências de um software [Center 1994]. As medidas de complexidade, modularidade e coesão tem uma alta correlação com os índices de manutenibilidade de um requisito [Coleman et al. 1995] [Lowther 1993] e por isso são utilizados neste trabalhos como os principais componentes da manutenibilidade.

Assumiremos então que a manutenibilidade pode ser descrita por métricas de modularidade, complexidade e coesão [Basili et al. 1995] [Tahvildari et al. 2012]

[Deraman and Layzell 1992] [Sauer 2005] para orientar a etapa de análise do requisito. Para as métricas de tamanho foram obtidos os valores referentes a média e desvio padrão para obter uma medida de distribuição e não apenas de tamanho. Por exemplo, um software com várias linhas de código é grande mas não necessariamente difícil de manter, pois depende de como essas linhas de código estão distribuídas. Com base na definição de manutenabilidade utilizada nesse trabalho, as seguintes métricas de software foram utilizadas para análise desse requisito:

1. **MLOC (Method Lines of Code)**: Distribuição de linhas de código por método no escopo analisado.
2. **NOC (Number of Classes)**: Distribuição de arquivos-fonte no escopo analisado.
3. **NOM (Number of Methods)**: Distribuição de métodos no escopo analisado.
4. **NOF (Number of Fields)**: Distribuição de campos no escopo analisado.
5. **LCOM (Lack of Cohesion of Methods)**: É uma métrica de coesão que representa a diferença entre o número de pares de métodos que acessam elementos não comuns e o número de pares que acessam elementos comuns num arquivo-fonte [Hitz and Montazeri 1996].
6. **MCC (McCabe Cyclomatic Complexity)**: Essa métrica calcula o número de fluxos alternativos em um método. Para cada fluxo alternativo a métrica é incrementada em 1 [Sauer 2005]. O seu valor foi definido empiricamente como no máximo 10 [Henderson-Sellers 1995] para o MCC máximo.
7. **NBD (Nested Block Depth)**: Enquanto a MCC mede o número de *branches* a NBD contabiliza a profundidade dos mesmos. Foi definido empiricamente o limiar máximo igual a 5 [Sauer 2005].
8. **AC (Afferent Coupling)**: O número de classes de fora do módulo que dependem de classes dentro do módulo.
9. **EC (Efferent Coupling)**: O número de classes dentro do módulo que dependem de classes fora do módulo.

A definição dos limiares das métricas analisadas também faz parte da etapa de análise. Para algumas métricas esses limiares dependem do domínio do software analisado. Dependendo da exigência do requisito de manutenibilidade os limiares podem ser ajustados empiricamente, pois para algumas métricas (tamanho, coesão) não existe uma definição global de um limiar ótimo. Uma vez estabelecidos esses limiares e quantificadas as métricas, é possível dizer se o requisito de manutenibilidade foi atendido quando os limiares estabelecidos são atingidos. Se o requisito não foi atendido pelo software, o processo de reengenharia segue com as operações de refatoração até atingí-lo.

2.2. Etapa 2: Refatoração

A aplicação de métricas e padrões de projeto tem o propósito comum de eliminar práticas inadequadas de programação. Com base nessa relação entre as métricas e os padrões de projeto, a etapa de refatoração proposta irá analisar os padrões de projeto em função das métricas obtidas na etapa anterior.

Os padrões revisados neste trabalho são os propostos pelo Gamma [Gamma et al. 1995]. Com base na solução proposta por cada padrão e em estudos feitos em outros trabalhos [Huston 2001] [Basili et al. 1995] [Tahvildari et al. 2012] [Deraman and Layzell 1992] [Sauer 2005], foram selecionados alguns dos padrões de

projeto para orientar a etapa de refatoração. Além disso os resultados obtidos na etapa de análise também orientaram a escolha desses padrões, pois o objetivo do processo de reengenharia é atingir os limiares das métricas analisadas. Por fim, foram selecionados os seguintes padrões:

1. **Visitor:** Pela facilidade em adicionar novas operações e manter as que já existem esse padrão oferece melhorias no índice de manutenabilidade de um sistema.
2. **Strategy:** A aplicação desse padrão permite diminuir a complexidade de um método abstraindo a implementação do algoritmo utilizado. Além disso, o padrão *Strategy* provoca uma melhoria nas métricas de modularidade e coesão pois os *branches* condicionais são agrupados em algoritmos específicos distribuindo melhor os componentes e tornando-os mais modulares.
3. **Fachada:** Esse padrão oferece uma forma de minimizar a comunicação e dependência de dados entre os subsistemas, diminuindo assim os índices das métricas associadas ao acoplamento. Como o número de objetos que o cliente lida é menor com a aplicação desse padrão, os subsistemas se tornam mais fáceis de manter.

Após selecionar os padrões de projeto é necessário localizar os componentes que serão alvos de refatoração. Para isso basta identificar os componentes que obtiveram os piores resultados na etapa de análise.

2.3. Etapa 3: Avaliação dos resultados

Após as etapas descritas nas seções 2.1 e 2.2 é necessário avaliar se o requisito não funcional proposto foi atendido. Para isso é necessário obter os novos valores das métricas de software propostas e analisar se os limiares pré-estabelecidos foram cumpridos. Com a aplicação de operações de refatoração adequadas haverá pelo menos alguma melhoria nas medições que pode ou não produzir um bom índice de manutenabilidade. Assumindo que as operações sejam conduzidas por desenvolvedores experientes e os padrões sejam corretamente aplicados os resultados obtidos nessa etapa serão satisfatórios, como já foi constatado em trabalhos semelhantes [Tahvildari et al. 2002] [Anquetil and Laval 2011] [Huston 2001] [Tahvildari et al. 2012].

3. Estudo de caso

Para avaliar a abordagem proposta neste trabalho, foi selecionado o software *J-Syncker* [Bezerra 2011], elaborado para o uso em oficinas na Escola de Música da Universidade Federal do Rio Grande do Norte. Como há novos membros na equipe de desenvolvimento do sistema e várias funcionalidades novas são solicitadas pelos clientes, é necessário que o software seja simples de manter devido a falta de familiaridade dos novos desenvolvedores com o sistema.

3.1. Análise

Considerando o requisito de manutenabilidade, o estudo feito considerou as métricas propostas que foram mensuradas durante a etapa de análise com o plugin *Metrics for Eclipse* [Sauer 2005]. A Tabela 1 contém os resultados obtidos antes de aplicar o processo de reengenharia. Para esse estudo foram utilizados os limiares ilustrados na Tabela 2. Para as métricas de tamanho foi utilizado o valor da média por pacote analisado como indicativo

Métricas	Total	Média	Desvio Padrão	Máximo
NOF	20	0,741	1,505	6
NOC	27	3	2,357	8
MLOC	4040	10,28	20,427	225
NOM	236	8,741	15,95	51
MCC	-	2,384	3,157	36
NBD	-	1,458	0,889	6
LCOM	-	0,063	0,184	0,725
EC	-	2	1,7	6
AC	-	5	3,682	11

Tabela 1. J-Syncker - Métricas

NOF	NOC	MLOC	NOM	MCC	NBD	LCOM	EC	AC
7	17	10	7	5	10	0,6	5	5

Tabela 2. Limiares das métricas

de distribuição dos componentes e não apenas do tamanho. Para as métricas restantes os limiares foram estabelecidos com base no valor máximo. Os limiares para as métricas de complexidade e coesão foram definidos com base na literatura revisada [Sauer 2005] [Henderson-Sellers 1995] [Hitz and Montazeri 1996]. Para as outras métricas os limiares foram definidos empiricamente.

3.2. Refatoração

Nessa etapa foi necessário identificar os componentes que obtiveram os piores índices para propor soluções utilizando padrões de projeto. Uma vez identificados os mau cheiros [Fowler 2000], foram elaborados alguns diagramas para permitir a visualização da organização dos componentes no módulo a ser refatorado. Após isso foram elaborados novos diagramas com a aplicação da solução proposta. Para as métricas de acoplamento foi necessário analisar os relacionamento entre as classes para propor uma solução que melhorasse as métricas analisadas utilizando o padrão Fachada. Para diminuir o acoplamento aferente e eferente foram criadas diversas fachadas, uma para cada pacote, para que as chamadas a métodos de fora e aos métodos de dentro do pacote sejam feitas pela fachada e não por diversas classes. Dessa forma apenas uma classe do pacote é vista pelos clientes. Devido a concentração desses métodos na fachada a manutenção é mais simples, pois as operações são mais facilmente identificadas. O principal efeito do uso desse padrão foi nos valores das métricas de acoplamento AC e EC.

Para obter melhores índices de modularidade foi aplicado o padrão *Visitor*. Foram identificadas as operações comuns a alguns componentes do sistema. Essas operações foram agrupadas em *visitors* específicos e as chamadas a elas foram substituídas por apenas uma chamada ao método *accept*. As operações associadas aos *visitors* foram aquelas comuns aos componentes *swing* [Oracle 1993]. O agrupamento dessas operações permitiu melhorias nas métricas de coesão e de distribuição de linhas de código. Embora sejam criados mais métodos e classes, as linhas de código são mais distribuídas entre eles, o que é um bom indicativo de modularidade.

Para melhorar os índices de complexidade foi aplicado o padrão *Strategy*. Fo-

ram identificados os métodos que possuíam maior complexidade e para cada um deles foi elaborada uma estratégia para eliminar os *branches* condicionais. Os contextos das estratégias utilizadas foram carregados em momentos adequados para evitar novos testes condicionais. Com a aplicação do padrão *Strategy* foi possível melhorar os valores obtidos para as métricas de complexidade.

Os resultados obtidos ao final do processo foram comparados com os limiares estabelecidos empiricamente. A Tabela 3 mostra os valores obtidos para as métricas analisadas. Para as métricas associadas a tamanho houve um aumento nos valores máximos

Métricas	Total	Média	Desvio Padrão	Máximo
NOF	23	0,397	1,016	5
NOC	58	5,8	8,518	31
MLOC	4369	5,537	15,3	268
NOM	248	4,276	10,282	52
MCC	-	1,655	1,69	10
NBD	-	1,221	0,63	5
LCOM	-	0,027	0,118	0,67
EC	-	1,8	1,166	4
AC	-	2,9	1,513	6

Tabela 3. Resultados

e totais. Isso ocorreu pois o uso de padrões de projeto tende a aumentar o número de classes no sistema, e consequentemente o de linhas de código. Mas os valores associados as médias e desvio padrão obtidos foram menores, indicando uma melhor distribuição dos componentes. Pelos resultados obtidos é possível constatar que os limiares estabelecidos foram atendidos, e consequentemente o requisito associado.

4. Trabalhos relacionados

Diversos trabalhos abordam a análise de requisitos não funcionais num processo de reengenharia ou manutenção de software. Anquetil e Laval em [Anquetil and Laval 2011] analisam a reengenharia de um software para torná-lo mais modular analisando as métricas de coesão e acoplamento. Esse trabalho constata através de um estudo de caso que apenas a avaliação dessas métricas não é suficiente para ajudar os desenvolvedores a atingir o requisito de modularidade, mas não é proposto um conjunto de métricas que seja descritivo o suficiente neste caso.

Emanuel em [Emanuel et al. 2011] propõe um índice quantitativo para modularidade de um software com base na análise de métricas de tamanho, complexidade, coesão e acoplamento. Esse índice serve apenas como uma ferramenta de comparação de softwares para identificar o mais modular, mas não pode ser utilizado para analisar se o requisito de modularidade foi atendido por um software ou para definí-lo quantitativamente. Para isso é necessário estabelecer limiares de referência para esse índice.

Tahvildari em [Tahvildari et al. 2002] propõe um framework de reengenharia de software orientada por requisitos não funcionais, como desempenho e manutenibilidade. Esses requisitos são modelados como grafos de interdependência e associados a alguns padrões de projeto. As métricas que são utilizadas na análise feita são as de Halstead

[Halstead 1977], LOC e MCC. Por fim é feito um estudo de caso para avaliar o framework proposta e analisar os efeitos das aplicações dos padrões de projeto nos requisitos não funcionais. Esse framework é bastante semelhante a abordagem aqui proposta, com a vantagem de incluir a análise das métricas de Halstead, que fornecem uma medida numérica para a complexidade de um sistema, mas não foi identificado neste trabalho um padrão de projeto utilizado para reduzir a MCC, embora tenha sido proposta a relação de diversos padrões de projeto com algumas métricas de software.

Huston em [Huston 2001] faz uma análise das relações entre métricas de software e padrões de projeto. São apresentadas algumas métricas de software e logo em seguida os padrões de projeto que permitem melhorar os seus índices. É afirmado que ambas as abordagens andam juntas pois enquanto as métricas indicam onde estão os problemas a aplicação dos padrões de projeto fornece uma maneira de melhorar esses índices. O estudo feito não analisou métricas de complexidade e os possíveis padrões associados aos seus índices.

O presente trabalho fez uso de diversos estudos específicos num contexto mais robusto, o de uma abordagem de reengenharia. A maior contribuição é que a abordagem está embasada nestes estudos que já mostraram resultados positivos mas nenhum deles foi apresentado num contexto mais global até o momento. As deficiências de cada trabalho foram exploradas para propor uma solução robusta e eficaz para o processo de reengenharia de software orientada a métricas de qualidade.

Embora seja uma abordagem robusta não é possível afirmar que ela é eficaz para todos os casos. Por exemplo, sistemas legados que precisam ser reprojetados para plataformas mais modernas deveriam utilizar um framework de reengenharia automatizada, para agilizar o processo. Para atividades corretivas ou de adição de novas funcionalidades envolvendo um processo de reestruturação do software é necessário ter uma visão mais ampla do módulo envolvido para analisar os componentes afetados, o que não é possível obter apenas com a análise de métricas de software.

5. Conclusão e trabalhos futuros

Esse trabalho apresentou uma abordagem de reengenharia orientada por métricas de software. Foram propostas etapas de análise, refatoração e avaliação para o processo. Durante a análise, com base num requisito não funcional definido, são selecionadas as métricas de qualidade associadas e os seus índices. Com base nos limiares definidos, são propostas operações de refatoração apoiadas por padrões de projeto adequados. Por fim, o software reestruturado é avaliado no que diz respeito ao requisito não funcional analisado.

Em trabalhos futuros serão analisadas as métricas de Halstead [Halstead 1977], que tem sido alvo de diversas pesquisas associadas a análise da manutenibilidade de um software. A dificuldade em utilizar essas métricas é a ausência de ferramentas que permitam obter seus valores de forma automatizada. Serão estudados também outros requisitos não funcionais para a obtenção das métricas de qualidade associadas. Por fim, serão analisados outros padrões de projetos que podem contribuir com a obtenção de melhores índices para essas métricas.

Referências

- Anquetil, N. and Laval, J. (2011). Legacy software restructuring: Analyzing a concrete case. In *Proceedings of the 15th European Conference on Software Maintenance and Reengineering*. CSMR.
- Basili, V. R., Briand, L., and Melo, W. L. (1995). A validation of object-oriented design metrics as quality indicators. Technical Report MD-20742, Dep. of Computer Science, Univ. of Maryland.
- Bezerra, G. S. (2011). J-syncker. <http://j-syncker.weebly.com/>.
- Center, J. S. (1994). Software design for maintainability. Technical Report DFE-6, NASA.
- Coleman, D., Lowther, B., and Oman, P. (1995). The application of software maintainability models in industrial software systems. In *Journal of Systems and Software*.
- Deraman, A. and Layzell, P. J. (1992). Software design criteria for maintainability. *Pertanika journal of science and technology*.
- Emanuel, A. W. R., Wardoyo, R., Istiyanto, J. E., and Mustofa, K. (2011). Modularity index metrics for java-based open source software projects. In *International Journal of Advanced Computer Science and Applications (IJACSA)*, volume 2.
- Fowler, M. (2000). *Refactoring: improving the design of existing code*. Addison-Wesley.
- Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Patterns: Elements of Reusable Design*. Addison-Wesley.
- Halstead, M. (1977). *Elements of Software Science*. Elsevier Science Ltd.
- Henderson-Sellers, B. (1995). *Object-Oriented Metrics: Measures of Complexity*. Prentice Hall, 1th edition.
- Hitz, M. and Montazeri, B. (1996). Chidamber and kemerer's metrics suite: A measurement theory perspective.
- Huston, B. (2001). The effects of design pattern application on metric scores. In *Journal of Systems and Software*, volume 58.
- Lorenz and Kidd (1994). *Object-oriented software metrics: a practical guide*. Prentice Hall.
- Lowther, B. (1993). The application of software maintainability metric models to industrial software systems. Master's thesis, Department of Computer Science, University of Idaho.
- Oracle (1993). Swing. <http://docs.oracle.com/javase/6/docs/technotes/guides/swing/>.
- Sauer, F. (2005). Eclipse metrics plugin. <http://metrics.sourceforge.net/>.
- Sommerville (2007). *Software Engineering*. Addison-Wesley, 8th edition.
- Tahvildari, L., Kontogiannis, K., and Mylopoulos, J. (2002). Quality-driven software re-engineering. *Journal of Systems and Software*.
- Tahvildari, L., Kontogiannis, K., and Mylopoulos, J. (2012). Comparison of software quality metrics for object-oriented system. *International Journal of Computer Science*, 12.