Identificação e Remoção de Dívidas Técnicas Congênitas

Aleson Teruji Makino[1], Helio Toshio Kamakawa[1], Willian Nalepa Oizumi[1]

[1]Instituto Federal do Paraná (IFPR) – Campus Paranavaí

87703-506 – Paranavaí – PR – Brasil

alesonteruji@gmail.com, {helio.kamakawa, willian.oizumi}@ifpr.edu.br

Abstract. *This article presents tools that helps to identify code anomalies and vulnerabilities that can occur at the beginning of the development of a system. A first stable version of a law firm software was used the Open Source FindBugs, JDeodorant and Owasp Zap tools, with which tests were performed to analyze the possible occurrence of technical debt. It is concluded that the tools can raise important information in order to avoid the incidence of technical debt early in the life of the software. With the achievements of the tests, significant actions can be taken to increase the quality of software.*

Resumo. *Este artigo apresenta ferramentas que servem para auxiliar na identificação de anomalias de código e vulnerabilidades que podem ocorrer no início do desenvolvimento de um sistema. Foi utilizado uma primeira versão estável de um software para escritório de advocacia com as seguintes ferramentas Open Source* para análise sendo elas o *FindBugs, JDeodorant e Owasp Zap, com as quais foram realizados testes para verificar a possível ocorrência de dívida técnica. Conclui-se que as ferramentas podem levantar importantes informações a fim de evitar a incidência de dívida técnica já no início da vida do software. Com a realização de testes, significativas ações podem ser tomadas ao modo que aumente a qualidade do software.*

1. Introdução

Em 1970, Canning citou uma classificação onde a manutenção foi caracterizada apenas como uma operação para a correção de erros e a ideia de expandir e estender as funcionalidades do software era vagamente pensada [Chapin 2001]. O autor cita a manutenção como parte do processo de desenvolvimento evolucionário de software que retarda o envelhecimento do software tornando-o viável por um maior período de tempo.

O teste no processo de desenvolvimento de software gasta 40% de esforço [Pressman 2006], o custo do teste na fase de desenvolvimento pode chegar próximo a 50% do gasto total do desenvolvimento [Harrold 2000]. Neste contexto, há uma necessidade de investigar fatores e criar técnicas que possam auxiliar a manutenção - uma das etapas mais caras do ciclo de vida de um software [Glass 2001].

Com intuito de desenvolver software com melhor qualidade e menor custo,

Cunningham (1992) criou uma metáfora conhecida como *Dívida Técnica* (DT), que tem como um de seus principais objetivos, mensurar o custo de anomalias de código cuja a modificação é postergada - protelação de refatoração e não de funcionalidade.

Desta forma, considerando o exposto anteriormente, o presente trabalho tem como objetivo geral de verificar se as ferramentas de análise estática de código são capazes de revelar dívidas técnicas relevantes em versões iniciais de um sistema. O estudo de caso foi executado com base em um sistema web - desenvolvido por um dos autores - na linguagem *Java* v. 8 para a plataforma *Java Enterprise Edition* (JEE). O desenvolvimento do sistema foi planejado e conduzido seguindo boas práticas de engenharia de software [Pressman 2005][Sommerville 2010]. Após o desenvolvimento inicial do sistema, foi realizada uma análise qualitativa considerando três tipos de dívida técnica: *code smells*, vulnerabilidades de segurança e potenciais bugs. Para essa análise, foram utilizadas 3 ferramentas de análise estática para a identificação de diferentes tipos de dívida técnica (JDeodorant, OWASP Zap e FindBugs).

Portanto, este artigo apresenta um experimento com o objetivo de avaliar a eficácia de detecção de DT através de ferramentas de análise estática após o inicio do desenvolvimento do software. Com realização dos testes nota-se o porquê da importância de realizar a verificação da DT desde o início do desenvolvimento, pois podem o ocorrer alguns casos de dívida técnica já nesta fase de desenvolvimento e a verificação já no início auxilia na correção DT que futuramente poderiam comprometer a estrutura do software.

A estrutura deste artigo está organizada da seguinte forma: a Seção 2 apresenta uma contextualização sobre os fatores que podem levar a ocorrência de dívida técnica, na Seção 3 abordamos as ferramentas que serão utilizados para realização das análises junto em qual ambiente será desenvolvido o estudo. Em suma a Seção 4 apresenta os resultados derivados da execução das ferramentas, onde mostra a eficácia na detecção de DT no software. Por fim, apresentamos as principais ameaças à validade na Seção 6 e indicamos conclusões e direções para trabalhos futuros na Seção 7.

2. Contextualização

Ward Cunningham (1992), apresenta o termo *Technical Debt* ou Dívida Técnica (DT), onde define DT como consequências cumulativas no projeto por não desenvolver conjuntos de requisitos. Conforme autor, o fato de não cumprir análise de requisitos do sistemas auxilia ao desenvolvimento de uma futura DT.

Segundo Allman (2012), DT é resultado da lacuna entre as boas práticas de desenvolvimento e fatores como a falta de tempo ou o custo de ferramentas. Seus argumentos sugerem que a ocorrência da dívida técnica é devido a economia com ferramentas para análises e testes, bem como modo de agilizar os prazos de entrega, não atentando-se a qualidade da codificação do software.

Vários fatores podem influenciar na ocorrência de DT, como por exemplo, falta de domínio em uma tecnologia, seja por ela ser recentemente adotada ou também devido a mudanças nas regras de negócio do projeto. O desenvolvimento de uma classe no projeto com uma carga de responsabilidade exagerada, por ser mais propensa a

apresentar maiores falhas do que classes desenvolvidas de forma mais dividida. DT também pode ser ocasionada por pensamento de curto prazo de entrega de um projeto, pensando sempre em futuras correções com lançamento de novas versões.

Porém, segundo estudos de Kniberg (2008), um projeto desenvolvido em um baixo nível de codificação tende a continuar um projeto de mesmo nível ou até mesmo inferior. Sendo assim, DTs que ocorrem em projetos de software deve ser monitoradas e removidas sempre que possível. Todavia, essa é uma tarefa complexa que envolve a análise de diversas informações do projeto de software.

Martin Fowler (2010), define *Code Smells*, como um sintoma de que algo no código pode estar errado, segundo o autor geralmente indica a necessidade de um refactoring ou de alteração estrutural da aplicação.

O Bug é um erro do programa que causa a discrepância entre o resultado esperado e o resultado da execução [Maltempi 2000], o autor ainda alega que a origem do bug reside em falhas na lógica do algoritmo e/ou no uso de comandos da linguagem de programação, portanto um Bug trata-se de uma função programada que ao ser executada não apresenta o resultado esperado.

Existe número cada vez maior de ocorrências de falhas de segurança relativas a sistemas de informação que não contemplaram adequadamente os conceitos da segurança em sua formulação [Schneier 2000], portanto caso não seja aplicado correto desenvolvimento da arquitetura de segurança do software pode ocasionar falhas deixando-o vulnerável.

Para auxiliar os desenvolvedores nessas tarefas de verificação de existência de DT, existem diversas ferramentas de análise estática de código (Seção 3.1). Entretanto, ainda há pouca evidência na literatura sobre a relevância das informações providas por tais ferramentas. Sendo assim, neste trabalho pretendemos avaliar o uso de ferramentas de análise estática de código em versões iniciais de um sistema.

3. Metodologia

Neste trabalho, realizamos uma investigação sobre a identificação e resolução de dívidas técnicas congênitas, ou seja, dívidas técnicas que surgem juntamente com a implementação inicial do sistema. Esta investigação teve como objetivo responder à seguinte questão de pesquisa (QP):

QP1: *Ferramentas de análise estática de código são capazes de revelar dívidas técnicas relevantes em versões iniciais de um sistema?*

O nosso objetivo com essa questão de pesquisa é investigar, de forma qualitativa, quão relevantes são as dívidas técnicas reveladas por ferramentas de análise estática de código. Como o conceito de dívida técnica abrange diferentes tipos de artefato (código fonte, especificação, testes, etc), nós optamos por focar nossa investigação em três tipos específicos de dívida técnica, que são: *code smells*, vulnerabilidades de segurança e potenciais bugs. Uma descrição detalhada dos três tipos de dívida técnica é apresentada na Seção 2.

Para respondermos à questão de pesquisa QP1, utilizamos o método *Estudo de Caso Exploratório* (EASTERBROOK, 2008). Segundo Easterbrook (2008), estudos de caso exploratórios são ideias para investigações iniciais que servirão como base para derivar novas hipóteses e para construir teorias. Acreditamos que esse método seja o

mais adequado pois, na medida do nosso conhecimento, não há nenhum trabalho anterior que avalie de maneira qualitativa o uso de ferramentas de análise estática para identificar e remover dívidas técnicas em versões iniciais de um sistema.

O estudo de caso foi executado com base em um software desenvolvido por um dos autores deste artigo (Seção 3.2). Conforme apresentaremos nas subseções a seguir, foram utilizadas 3 ferramentas de análise estática para a identificação de diferentes tipos de dívida técnica.

Nas subseções a seguir apresentamos maiores detalhes sobre a metodologia. A Seção 3.1 apresenta informações sobre as ferramentas de análise estática utilizadas neste estudo. A Seção 3.2 apresenta o sistema de software analisado no estudo de caso. Por fim, a Seção 3.3 contém os procedimentos seguidos para a realização do estudo e para a coleta e análise de dados.

3.1. Ferramentas para Análise Estática de Código

Conforme supramencionado, neste trabalho foram consideradas dívidas técnicas dos seguintes tipos: *code smell*, vulnerabilidade de segurança e potencial bug. Para a identificação desses tipos de dívida técnica, nós selecionamos 3 ferramentas: JDeodorant para *code smells*, OWASP ZAP para vulnerabilidades de segurança e FindBugs para potenciais bugs. Abaixo apresentamos uma descrição detalhada de cada ferramenta.

JDeodorant trata-se de um plug-in do Eclipse de código aberto para Java, que auxilia a identificar sintomas de *code smell* em código fonte. Essa ferramenta auxilia na resolução dos *code smells* através da aplicação de refatorações, conforme site JDEODORANT. Por meio deste *plug-in*, é possível identificar Classes Grandes, Métodos Longos, Códigos Duplicados e *Feature Envies*. JDeodorant adiciona no Eclipse uma nova opção na barra de menu, com o nome de *Bad Smells*, contendo cinco opções, sendo uma para cada tipo de *smell*. Para cada seleção do tipo de smell, o plug-in apresenta uma nova visão. Após a análise ser realizada as anomalias encontradas serão adicionadas em uma tabela e marcadores serão adicionados no código, estes marcadores informam as partes do código que devem ser refatorados segundo a ferramenta.

O OWASP Zed Attack Proxy (ZAP) é uma das ferramentas de segurança gratuitas mais populares do mundo, segundo site OWASP. Trata-se de uma ferramenta de código aberto para a detecção de vulnerabilidades de segurança. A principal funcionalidade do OWASP Zap é realizar uma análise em aplicações web, a fim de localizar vulnerabilidade. Seu fácil modo de uso torna o ZAP uma ótima opção de uso para especialista em segurança, desenvolvedores ou testadores.

FindBugs é uma ferramenta de análise estática que examina sua classe ou arquivos JAR procurando por possíveis problemas, combinando seus bytecodes com uma lista de padrões de bugs, conforme site International Business Machines - IBM. A análise de um projeto é iniciada de forma manual e realizada por um verificador de erros que checa a existência de regras no código. Ao analisar um projeto a ferramenta insere marcadores nos locais das classes onde existam tais problemas. FindBugs possui duas visões Bug Explorer e Bug Info, sendo a primeira apresenta uma lista com todos os

problemas encontrados na análise feita, já a segunda lista uma descrição detalhada de um problema quando é selecionado.

3.2. Sistema Administrador de Processos - SAPROC

Neste trabalho, nós realizamos a análise do Sistema Administrador de Processos (SAPROC). O SAPROC possui a arquitetura computacional cliente/servidor seguindo o padrão *Model-View-Controller* (MVC). A gestão de dependências do projeto (bibliotecas e JARs) foi configurado para ser realizado por meio da ferramenta Maven.

Luckow e Melo (2015) esclarece que o padrão MVC é uma arquitetura que separa as responsabilidades em camadas. Maven é uma ferramenta de construção usada dentre outros, para criar artefatos e gerenciar dependências de projetos (O'REILLY, 2008).

O desenvolvimento foi realizado por meio da linguagem de programação *Java* v.1.8 para a plataforma *Java Enterprise Edition* (JEE). O ambiente de desenvolvimento integrado Eclipse Neon foi utilizado para auxiliar na edição do código-fonte. Gonçalves (2013) explica que o JEE *s*urgiu no final da década de 1990 e trouxe uma robusta plataforma de software para desenvolvimento empresarial.

O projeto foi hospedado em um servidor *Java* local, o *Apache Tomca*t v. 8.5 - um *container* que permite o funcionamento do projeto em ambiente web (LUCKOW e MELO, 2015).

A interface foi desenvolvida com a tecnologia *JavaServer Faces*, com as *tags core: facelets, core, composite* e *primefaces*. *Tags core* são rótulos que facilitam a definição do conteúdo para os demais recursos do JSF e bibliotecas de componentes de terceiros (LUCKOW e MELO, 2015).

Para fornecer as informações exibidas nas páginas e as operações que serão executadas foi utilizando o *Backing Bean*, que segundo Luckow e Melo (2015), é uma classe java que suporta todo o funcionamento das páginas JSF.

A autenticação do usuário e a autorização dos recursos do sistema foi desenvolvida com auxílio do *framework Spring Security* - um conjunto de código-fonte aberto que visa facilitar o desenvolvimento JEE (LUCKOW e MELO, 2015).

Os dados estão sendo persistidos no sistema gerenciador de banco de dados MariaDB v. 5.1. O mapeamento das entidades do projeto para as tabelas relacionais da base de dados foi realizado com o *framework Hibernate* v.5.2, seguindo as especificações do Java Persistence API (JPA).

Hibernate é a solução de mapeamento objeto/relacional mais utilizada encontradas hoje no mercado para a linguagem *Java* e o JPA é um conjunto de especificações que padroniza os procedimentos de mapeamento objeto/relacional (LUCKOW e MELO, 2015).

O software é formado por 3 grupos principais de funcionalidades: (1) interface do gerenciador de processos, que auxilia no controle dos processos jurídicos administrados; (2) Agendamento de Compromisso, que fornecerá gestão da agenda de compromissos do usuário; e o (3) Andamento de Processos, onde manterá registros de reuniões e audiências do processo.

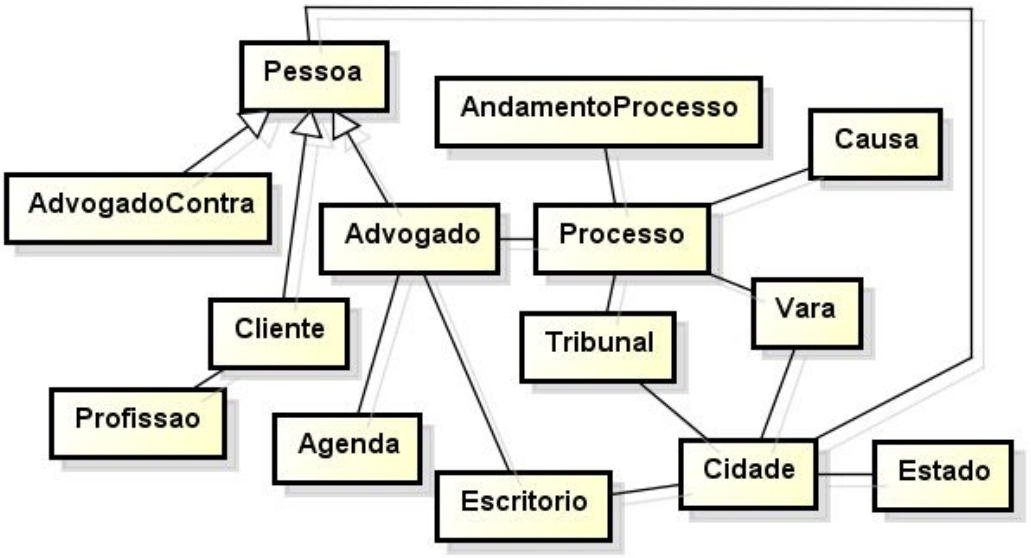

Figura 1. Diagrama de Classe do projeto SAPROC

A Figura 1 apresenta o diagrama de classe que representa as principais entidades e suas respectivas associações e heranças do sistema - os atributos foram omitidos com intuito de melhorar a visualização. O projeto desenvolvido é formado por 89 classes com aproximadamente 4.313 linhas de código.

3.3. Procedimentos para a Coleta e Análise de Dados

Para a realização deste trabalho seguimos os seguintes procedimentos. Primeiramente, o sistema apresentado na Seção 3.2 foi implementado pelo primeiro autor deste artigo. Essa implementação foi planejada e conduzida seguindo boas práticas de engenharia de software, segundo autores como Pressman (2005) e Sommerville (2010). Os requisitos do sistema foram coletados por meio de entrevistas, observações e *brainstorms*. Os ciclos de desenvolvimento foram executados de forma iterativa e incremental, coletando *feedback* dos potenciais usuários a cada iteração. Foram realizados testes tanto para a verificação dos requisitos, como também para identificar possíveis defeitos no sistema. Após a implementação das funcionalidades necessárias para satisfazer os requisitos iniciais do sistema, nós capturamos a versão do código fonte utilizada neste estudo.

Com base nessa versão capturada, nós iniciamos as análises para a identificação de dívidas técnicas. Nós executamos as 3 ferramentas de análise estática sem realizar nenhum tipo de configuração ou customização especial. Ou seja, nós utilizamos as 3 ferramentas com as suas respectivas configurações básicas.

Os resultados apresentados pelas ferramentas foram classificados e sumarizados de acordo com três tipos (*code smell*, vulnerabilidade de segurança e potencial bug). Para cada tipo de dívida técnica, nós contabilizamos o número total de ocorrências.

Dívidas Técnicas Relevantes. Para responder à nossa questão de pesquisa QP1, nós realizamos uma análise qualitativa das ocorrências de dívida técnica apontadas pelas ferramentas de análise estática. Nesta análise, a relevância de cada ocorrência de

dívida técnica foi avaliada individualmente. A análise de relevância levou em consideração o impacto de cada dívida técnicas em atributos de qualidade como, manutenibilidade, legibilidade, robustez e segurança. Em sequência, as dívidas técnicas consideradas relevantes foram validadas por um especialista em qualidade de software. Com base nisso, nós obtivemos um conjunto de ocorrências consideradas relevantes. Todos os resultados e discussões apresentados neste artigo foram derivados da análise desse conjunto de dívidas técnicas relevantes.

4. Resultados

Na Tabela 1 apresentamos os resultados obtidos através da execução das ferramentas FindBugs, JDeodorant e OWASP Zap. Conforme apresentado na primeira coluna da tabela, as ferramentas de análise estática revelaram a ocorrência de 71 *code smells*, 4 vulnerabilidades de segurança e 2 potenciais bugs. Porém, nem todas as ocorrências foram consideradas relevantes conforme abordaremos a seguir.

Code smells. Proveniente da informação de situação de 71 casos de *Code Smells* ocorrentes, realizou-se uma análise, onde verificou-se a necessidade de refatoração do código fonte, a fim de diminuir a existência de dívida técnica. Com base nos dados analisados concluiu-se que 6 ocorrências necessitavam de alterações na codificação, pois o código fonte poderia ser refatorado e melhorado sem que as funções existentes sofressem algum tipo de alteração de funcionalidade. Foi constatado que algumas das recomendações de possíveis alerta não necessitavam de alteração, 65 ocorrências remanescentes, devido ao fato de estarem corretamente implementados e algumas modificações na implementação comprometeria o funcionamento do sistema.

Vulnerabilidades de segurança. A incidência de 4 vulnerabilidades de segurança no projeto, ocasionou um importante alerta, pois um software vulnerável pode comprometer a integridade dos dados, segurança informação e também a disponibilidade do software. Os resultados apresentados pela ferramenta OWASP Zap mostraram vulnerabilidades para ataques conhecidos como *ClickJacking* [OWASP 2018a], *Cross-Site Scripting (XSS)* [OWASP 2018b] *e Sniffing* [OWASP 2018c]. Após análise das evidências apresentadas, nós acatamos as correções propostas e resolvemos as 4 vulnerabilidades de segurança.

Tipo de Dívida Técnica	# Total de Ocorrências	# de Ocorrências Relevantes	# de Ocorrências Remanescentes
Code Smell	71	6	65
Vulnerabilidade de Segurança	4	4	0
Potencial Bug	2	2	0

Tabela 1. Quantidade de DTs Identificadas e Removidas.

Potenciais bugs. A ferramenta FindBugs apontou a existência de 2 potenciais bugs, revelando uma possível falha na implementação do sistema. As duas ocorrências

apresentadas foram referentes à implementação incorreta dos métodos *equals* e *hashcode*.

Após análise, seguindo os procedimentos apresentados na Seção 3.3, confirmou-se a existência de erros de programação em duas classes do sistema que poderiam causar falhas no funcionamento do software. Os erros de programação ocorreram porque a implementação dos métodos *equals* e *hashcode* não estava correta. Em sistemas desenvolvidos com a linguagem de programação Java, é muito importante a correta implementação dos métodos *equals* e *hashcode*, pois através destes pode-se evitar a duplicidade de dados, prevenindo assim comportamentos inesperados durante a execução do sistema.

Ferramentas revelam DTs relevantes. Após análise de DTs reveladas por ferramentas de análise estática, observamos que mesmo a partir de uma versão inicial as ferramentas podem revelar importantes informações a fim de evitar a incidência de DT já no início da vida do software. Através desta informações importantes e significativas ações podem ser tomadas ao modo que aumente a qualidade do software. Portanto, a resposta para a nossa questão de pesquisa QP1 é que sim, **ferramentas de análise estática são capaz de revelar dívidas técnicas relevantes**.

Entretanto, conforme pode-se observar nos resultados, determinados tipos de DT como, por exemplo, os *code smells*, podem apresentar uma quantidade excessiva de ocorrências irrelevantes. Isso pode fazer com que desenvolvedores de software sintam-se desencorajados em utilizar ferramentas de análise estática para avaliar a ocorrência de DTs. Esse resultado condiz com outros estudos da literatura sobre code smells, que têm apontado para a necessidade de técnicas mais efetivas para a detecção de *code smells* relevantes [Oizumi et al. 2016].

Reincidência de DT após correções. Após realizar as correções necessárias para a remoção das DTs consideradas relevantes, nós executamos novamente as 3 ferramentas de análise estática, a fim de verificar se as correções ocasionaram o surgimento de novas DT.

Durante essa análise de reincidência, constatou-se a ocorrência de 65 casos de *code smells*. Entretanto, esses 65 casos de code smell são os mesmos que foram considerados irrelevantes em nossa primeira análise. Conforme mencionado anteriormente, não seria possível remover esses *code smells* sem afetar o comportamento do sistema. Além disso, observamos que o tratamento dos 65 code smells remanescentes não traria benefícios para os atributos de qualidade do sistema.

5. Ameaças à Validade

A primeira ameaça à validade deste trabalho refere-se às ferramentas utilizadas, pois caso fossem selecionadas outras ferramentas os resultados poderiam ser diferentes. Para mitigar essa ameaça, nós testamos um conjunto de 3 ferramentas diferentes. Muitas das ferramentas testadas apresentaram erros, devido a incompatibilidade de versão, configuração de hardware, assim impedindo que fossem utilizadas neste estudo. Além disso, nós optamos por não utilizar ferramentas proprietárias devido ao alto custo para aquisição das licenças.

Por fim, temos uma outra ameaça à validade que diz respeito ao número de

sistemas utilizados no estudo de caso. Nós temos ciência de que a análise de um único sistema não é o suficiente para a obtenção de achados consistentes. Porém, este é um estudo inicial que será complementado em estudos futuros com análises envolvendo múltiplos sistemas.

6. Conclusão

Em todos os dados analisados pelas ferramentas, notou-se que algumas informações sugeridas como *code smells* não necessitavam de tratamento devido, pois alguns dos apontamentos referem-se aos métodos *Get* e *Set*, que constatado a correta implementação destes auxilia o acesso aos atributos das classes.

Entretanto as ferramentas apontaram importantes dados para prevenção de DTs, uma vez que por meio desses resultados foram resolvidas inconsistências no software. Diante disso, verificou-se que na fase inicial a manutenção do software fica mais facilitada do que após o término, pois as implementações não estão vinculadas a várias funcionalidades do sistema reduzindo assim o risco de novas DTs e não comprometendo alterações em funcionalidades já existentes.

Conforme abordado a questão de pesquisa QP1, após análise fornecida pelas ferramentas, mesmo a partir de uma versão inicial conclui-se que as ferramentas podem levantar importantes informações a fim de evitar a incidência de dívida técnica já no início da vida do software. Através destas informações importantes significativas ações podem ser tomadas ao modo que aumente a qualidade do software e futuramente possa ser mensurado o ganho de Tempo X Produção no desenvolvimento software.

Como trabalhos futuros sugerimos: (i) aplicação das mesmas ferramentas em outros sistemas a partir da primeira versão estável, a fim de prevenir e localizar possíveis anomalias e vulnerabilidades no código; (ii) realização de um comparativo de um sistema que desde início foi utilizado softwares para acompanhamentos de vulnerabilidades contra outro que não foi realizado nenhum teste.

Referências

Cunningham, W. (1992), The WyCash Potfolio Management System.

Kruchten, Philippe. Nord, Robert L. e Ozkaya, Ipek. (2012), Technical debt: From metaphor to theory and practice.

E. Allman.(2012), Managing technical debt. Communications of the ACM.

Zazworka, N. (2013), A case study on effectively identifying technical debt. In:Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering. (New York, NY, USA: ACM), Disponível em: <http://doi.acm.org/10.1145/2460999.2461005>. Acessado em 08 junho de 2018.

Oizumi W, Garcia A, Souza, A, S, Cafeo B, e Zao Y. (2016), “Code anomalies flock together: exploring code anomaly agglomerations for locating design problems”. In Proceedings of the 38th International Conference on Software Enginer (ICSE´16) (New York, New York, USA). 2016.

Tsantalis, N. (2018), Jdeodorant. Disponível em: <https://users.encs.concordia.ca/~nikolaos/jdeodorant/>. Acessado em 16 de junho de 2018.

IBM. (2004), Findbugs part 1: Improve the quality of your code. Disponivel em: <http://www.ibm.com/developerworks/java/library/j-findbug1/>. Acessado em 12 de junho de 2018.

Maltempi, Marcus Vinicius(2000), Construção de páginas WEB: Depuração e especificação de um ambiente de aprendizagem. Disponível em: <http://www.rc.unesp.br/igce/demac/maltempi/tese.pdf> Acessado em: 20/07/2018.

Owasp Zap. (2018), OWASP Zed Attack Proxy Project. Disponível em: <https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project>. Acessado em 4 de junho de 2018.

Costa, Gustavo. (2017), Automatizando Testes de Vulnerabilidades em Aplicações Web com o Owasp Zap e Python. Disponível em: <https://medium.com/@gustavoh/automatizando-testes-de-vulnerabilidade-em-aplica%C3%A7%C3%B5es-web-com-o-owasp-zap-e-python-fdcdcf78b587>. Acessado em 8 de junho de 2018.

Bennetts, S. (2018), Owasp Zed Attack Proxy Project. Disponível em: <https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project>. Acessado em 8 de junho de 2018.

Meucci, M. (2008), Owasp testing guide version 3.0. Owasp Foundation.

Fowler, Martin (2000), Refactoring: Improving the Design of Existing Code.

Schneier, Bruce (2000). Secrets and lies: digital security in a networked world

Koscianski A, Soares M. (2006), Qualidade de Software, Novatec.

Glass, R. L. (2001), Frequently Forgotten Fundamentals Facts about Software Engineering. IEEE Software.

Easterbrook, Steve et al. (2008), “Selecting empirical methods for software engineering research”. In: Guide to advanced empirical software engineering. Springer London.

Pressman, Roger S. (2005), Software engineering: a practitioner's approach.

Pressman, R. S. (2006), “Engenharia de software”. São Paulo: McGraw-Hill.

Sommerville, Ian. (2010), Software engineering. New York: Addison-Wesley.

Harrold, M. J. (2000), Testing: A roadmap. In The Future of Software Engineering.

Goncalves, Antonio. (2013), Beginning Java EE 7.

Sonatype. (2008), Maven: the definitive guide.

Schmid, K. (2013), “On the limits of the technical debt metaphor some guidance on going beyond”. In:4th International Workshop on Managing Technical Debt, MTD 2013–Proceedings.

Kniberg, Henrik. (2008), Scrum e XP direto das Trincheiras.

OWASP (2018a), Clickjacking. Disponível em: <https://www.owasp.org/index.php/Clickjacking>. Acessado em 17 de junho de 2018.

OWASP (2018b), Cross-site Scripting (XSS). Disponível em: <https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)>. Acessado em 17 de junho de 2018.

OWASP (2018c), Disponível em: <https://www.owasp.org/index.php/Sniffing_application_traffic_attack>. Acessado em 17 de junho de 2018.