Uma análise da relação entre code smells e dívida técnica auto-admitida

Felipe Gustavo de S. Gomes[1], Thiago Souto Mendes[2], Rodrigo O. Spínola[3], Manoel Mendonça[1], Mário Farias[4]

[1] Universidade Federal da Bahia, [2]Instituto Federal da Bahia, [3]Universidade Salvador, [4]Instituto Federal de Sergipe

{felipegustavo,mgmendonca}@dcc.ufba.br, thiagosouto@ifba.edu.br, rodrigo.spinola@unifacs.br, mario.andre@ifs.edu.br

Resumo. *Code smells indicam possíveis problemas na implementação de um sistema que podem levar à necessidade de refatoração do seu código. Eles podem ser detectados automaticamente e são considerados indicadores de presença de Dívida Técnica (DT). Contudo, estudos indicam que considerar apenas code smells na tarefa de detecção da presença de itens de dívida é insuficiente. É necessário utilizar estratégias de detecção complementares, como a utilização de informações extraídas a partir de comentários de código. Porém, ainda são poucos os estudos sobre a relação entre code smells e DT auto-admitida. Este trabalho analisa três projetos open source para investigar a relação existente, em termos de sobreposição e complementariedade de itens de DT identificados, utilizando detecção via code smells e DT auto-admitida. Os resultados indicam que as informações de comentários podem complementar as informações de code smells e apoiar os desenvolvedores a identificar DT.*

Abstract. *Code smells are an indication that there may be implementation problems in a system and may lead to the need to refactor its code. They can be detected automatically and are considered indicators of the presence of Technical Debt (TD). However, studies indicate that considering only code smells in the task of detecting the presence of debt items is insufficient. Complementary detection strategies, such as using information extracted from code comments, must be used. Although, there are still few studies about the relationship between code smells and self-admitted TD. This paper analyzes three open source projects to investigate the existing relationship, in terms of overlap and complementarity of identified TD items, using detection by code smells and self-admitted TD. Results indicate that comments information can complement code smells information and support developers to identify TD.*

1. Introdução

Code smells, surgem de escolhas de implementação de um sistema que não obedecem a princípios amplamente aceitos de um bom projeto de software [Zazworka et al. 2013]. Eles são uma indicação inicial de que pode haver sérios problemas na implementação do sistema [Fowler 1999]. Esses problemas podem dificultar a evolução do software e levar à necessidade refatoração do seu código [Fontana et al. 2012].

Por sua vez, o conceito de Dívida Técnica (DT) tem ajudado profissionais e pesquisadores a discutirem problemas associados à evolução do software [Seaman and Guo 2011]. Ele contextualiza o problema de tarefas de desenvolvimento de software pendentes como um tipo de dívida que traz um benefício de curto prazo para o projeto, mas que pode precisar ser paga com juros mais tarde no processo de desenvolvimento [McConnell 2008]. Por exemplo, a evolução de uma classe mal projetada tende a ser mais dispendiosa do que uma implementada considerando boas práticas de projeto.

A identificação de DT é uma etapa essencial no processo de seu gerenciamento. Algumas pesquisas têm focado na identificação automática de DT. Isto traz vantagens em termos produtividade e facilitaria a adoção do gerenciamento de DT. Todavia, a maioria destas técnicas é baseada na detecção de *code smells*. O problema desse tipo de abordagem é que ela é capaz de detectar apenas alguns tipos de dívidas, notadamente dívidas de código e de projeto [Alves et al. 2016].

Por outro lado, programadores explicitam a existência de soluções incompletas e temporárias que exigem retrabalho através de comentários em código. Neste caso, tem-se o conceito de DT Auto-Admitida (DTAA) [Potdar and Shihab 2014]. As DTAAs tratam de um espectro mais amplo de dívidas. Elas podem ser mais assertivas quanto à variedade de itens identificados uma vez que permitem obter informação de contexto explicitada por desenvolvedores [Farias et al. 2015]. Estratégias baseadas em análise de comentários podem ser utilizadas para a identificação automática de itens de DTAA. Um exemplo é o Contextualized Vocabulary Model for identifying TD on code comments (CVM-TD), que trata-se de um vocabulário contextualizado com expressões e termos proposto por Farias *et al.* [2015] para apoiar a identificação de DTAA através da análise de comentários de código fonte.

Apesar de serem os principais indicadores para identificação de DT citados na literatura, os *code smells* podem não trazer todas as informações necessárias para identificar os diferentes tipos de DT que podem ocorrer no software, como dívida de requisitos e defeito [da Silva Maldonado et al. 2017]. Em teoria, uma abordagem combinada das duas técnicas, análise de *code smells* e análise de comentários, poderia melhorar a identificação de itens de DT e ampliar o espectro de itens de DT identificados automaticamente.

Este artigo investiga a relação entre a detecção automática de *code smells* e DTAA extraída utilizando CVM-TD, a fim de avaliar a sobreposição e complementariedade das duas técnicas. Para atingir o objetivo, foi realizada uma análise quantitativa e qualitativa de três projetos *open source*. Os resultados do estudo indicaram que as informações de comentários de código podem complementar as informações produzidas pelos *code smells* e apoiar equipes de desenvolvimento a identificar DT. A análise quantitativa mostrou que instâncias de DTAA tendem a se relacionar com os *code smells Duplicated Code*, *God Class*, *Long Method* e *Complex Method*. Por outro lado, análise qualitativa mostrou que comentários possuem informações com sugestões e percepções do desenvolvedor que poderá ajudá-lo a decidir se pagará aquela dívida naquele momento e como resolvê-la.

O restante do artigo está organizado da seguinte forma. A Seção 2 discute alguns trabalhos relacionados. A Seção 3 descreve o contexto e o planejamento do estudo. A Seção 4 apresenta e discute os resultados quantitativos obtidos. A Seção 5 apresenta alguns resultados da análise qualitativa que foi realizada. A Seção 6 discute as ameaças à

validade do estudo. Por fim, a Seção 7 apresenta as considerações finais deste trabalho e direcionamentos para trabalhos futuros.

2. Trabalhos Relacionados

Em seu trabalho, Wehabi *et al.* [2016] examinaram a relação entre DTAA e a qualidade do código através da investigação se: **(i)** arquivos com DTAA têm mais defeitos que arquivos sem dívida; **(ii)** modificações nos arquivos com DTAA introduzem defeitos; e **(iii)** se modificações relacionadas a DTAA tendem a ser mais difíceis. Nesse estudo foram analisados cinco projetos *open source*. Os pesquisadores constataram que **(i)** não existe uma clara tendência em relação a defeitos e DTAA; **(ii)** modificações relacionadas a arquivos com DTAA induzem menos defeitos futuros do que modificações em arquivos sem DT; e que **(iii)** arquivos com DTAA são mais difíceis de serem modificados. O estudo também indicou que apesar da DT ter impactos negativos, os seus efeitos não são relacionados com defeitos, mas sim com a redução da manutenibilidade do sistema.

Mensah *et al.* [2018] introduziram um esquema de priorização composto principalmente por identificação, exame e estimativa de esforço de retrabalho de atividades priorizadas a fim de auxiliar na tomada de decisão antes do lançamento de uma versão do software. Usando o esquema proposto, os pesquisadores realizaram uma análise exploratória em quatro softwares *open source* para investigar como a DTAA pode ser minimizada. Foram identificadas quatro causas principais de DTAA: *code smells*, tarefas complicadas e complexas, testes inadequados e performance de código inesperada. Os resultados mostraram que, entre todos os tipos de DT, as dívidas de projeto eram mais propensas a *bugs*.

3. Planejamento do Estudo

A fim de avaliar a relação entre *code smells* e DTAA, foram analisados três projetos *open source* desenvolvidos em Java. Os projetos considerados foram ArgoUML[1] (versão 0.34), JFreeChart[2] (versão 1.0.19) e Apache Ant[3] (versão 1.10.2). O ArgoUML e o JFreeChart foram escolhidos, pois são conhecidos na comunidade científica na realização de estudos sobre comentários de código fonte [Farias et al. 2015] [da Silva Maldonado et al. 2017]. Por outro lado, o Apache Ant foi escolhido por ter uma grande quantidade de comentários de código. Na Tabela 1, é apresentada a quantidade de arquivos analisados e a quantidade total de linhas de comentários e de código. Os dados utilizados nas análises encontram-se disponíveis no endereço `http://bit.ly/dtaastudy`.

Inicialmente, foram detectados os *code smells* utilizando a ferramenta Repository-Miner (RM) [Gomes et al. 2017]. O RM é uma ferramenta extensível para a mineração de repositórios de software para suportar a identificação automática de DT. Os *code smells* escolhidos para este estudo foram: *Brain Class*, *Brain Method*, *Complex Method*, *Data Class*, *Feature Envy*, *God Class*, *Long Method* e *Duplicated Code* [Fowler 1999] [Kerievsky 2005] [Lanza and Marinescu 2006]. Os *code smells* escolhidos estão entre os mais comuns e podem ser detectados pelo RM.

[1]https://github.com/stcarrez/argouml
[2]https://github.com/jfree/jfreechart
[3]https://github.com/apache/ant/

Logo após, foram extraídos os comentários de código fonte com algum indício de DT. A análise de comentários foi realizada utilizando o RM através do seu módulo de integração com o eXcomment [Farias et al. 2015]. O eXcomment é uma ferramenta capaz de identificar e classificar a DT através da análise de comentários de código. A ferramenta é baseada em mineração de texto e utiliza um vocabulário contextualizado, o CVM-TD, composto por padrões [Farias et al. 2015]. Esses padrões são um conjunto de palavras, códigos de DT e termos de engenharia de software que permitem a identificação daquele comentário como um item de DT. Em um comentário pode haver um ou mais padrões. Além disso, eXcomment também classifica os padrões entre os seguintes tipos de DT: código, projeto, arquitetura, construção, defeito, documentação, pessoas, requisitos, teste e desconhecido (ou seja, são considerados como uma DT, mas não foi possível determinar o tipo exato).

Apesar das análises realizadas através do RM nos projetos de software terem considerado diferentes tipos de granularidade em relação aos elementos do software, como classes e métodos, neste estudo essa granularidade não foi considerada na análise dos resultados. Para isso, as informações de *code smells* e comentários de código fonte referentes a classes e métodos foram agrupadas em função dos arquivos nos quais elas foram identificadas. Dessa forma, foi avaliada a existência de sobreposição entre arquivos reportados por ambas as estratégias como contendo itens de dívida e analisado o tipo de informação disponibilizado por cada técnica a respeito daquele arquivo.

Nas seções a seguir, são apresentados os resultados e discussões das análises quantitativa e qualitativa dos projetos selecionados. Na análise quantitativa, são apresentados os comparativos das informações dos arquivos dos três projetos analisados com as relações existentes entre *code smells* e comentários de código. Na análise qualitativa, devido a falta de espaço, são apresentados os resultados parciais da verificação dos comentários do projeto ArgoUML, que foi selecionado por ser o maior projeto entre os três.

4. Análise Quantitativa

Foi realizada uma análise quantitativa através do estudo das combinações entre *code smells* e comentários com o objetivo de verificar possíveis relações. Foi considerado como sendo uma relação se um arquivo apresenta simultaneamente comentários selecionados pelo eXcomment e *code smells* indicando a presença de dívida.

Na Tabela 1, é apresentada a quantidade de arquivos com *code smells*, arquivos com comentários de DT e de relações encontradas. Ao analisar os dados é possível verificar que existe uma alta taxa de relação nos projetos Apache Ant (76% (quantidade de relações / quantidade de arquivos com *code smells*)), ArgoUML (73%) e JFreeChart (59%), com uma média de 68% do total possível de relações, mostrando que *code smells* e DTAA tendem a caminhar juntos.

Com o objetivo de analisar quais tipos de DT têm relações mais fortes, foram gerados gráficos de bolha (ver Figura 1) com a combinação das informações sobre os tipos de *code smells* e os tipos de dívidas identificados pelos comentários. Analisando os gráficos, é possível notar um padrão entre os três projetos. Notou-se que os *code smells* tendem a se relacionar fortemente com quase todos os tipos de DT, exceto com dívidas de teste, pessoas e documentação. Nota-se também que os *code smells*: *Duplicated Code*, *God Class*, *Long Method* e *Complex Method* tendem a ser maiores indicadores de DT

Tabela 1: Dados dos Projetos Analisados.

Projeto	# Arquivos	LOC Comentários	LOC Código	# Arquivos Comentários	# Arquivos Code Smells	# Relações
Apache Ant	1220	103393	137732	622	276	209
ArgoUML	1870	148051	173855	995	451	327
JFreeChart	1017	144851	144342	475	364	215

que os demais, principalmente para dívida de projeto, defeito, código e desconhecido. Além disso, é possível observar que os comentários são capazes de complementar as informações dos *code smells*, mostrando relações com outros tipos de DT que vão além da qualidade do código fonte, como documentação, requisitos, construção e defeitos.

É importante também destacar a elevada quantidade de relações de dívidas com o *code smell God Class* nos três projetos. O *God Class* refere-se a classes que tendem a centralizar a inteligência e a maioria do trabalho do sistema [Lanza and Marinescu 2006]. *God class* também é um dos principais indicadores para identificar DT de projeto [Alves et al. 2016]. Nesse cenário, a combinação da identificação do *code smell God Class* e de comentários com indicando DTAA pode aumentar as chances de identificação e priorização do pagamento das dívidas do projeto.

O projeto JFreeChart não apresenta relações de DTAA com *Data Class*, isso ocorreu pois existiam poucas ocorrências do *code smell* (apenas quatro em todo o projeto). Também, destaca-se o fato do projeto Apache Ant ter quantidade de relações próximas aos dos outros projetos mesmo tendo uma quantidade de linhas de comentários consideravelmente menor. Isso pode indicar que os desenvolvedores do projeto Apache Ant costumam reportar com mais frequência problemas através de comentários no código fonte.

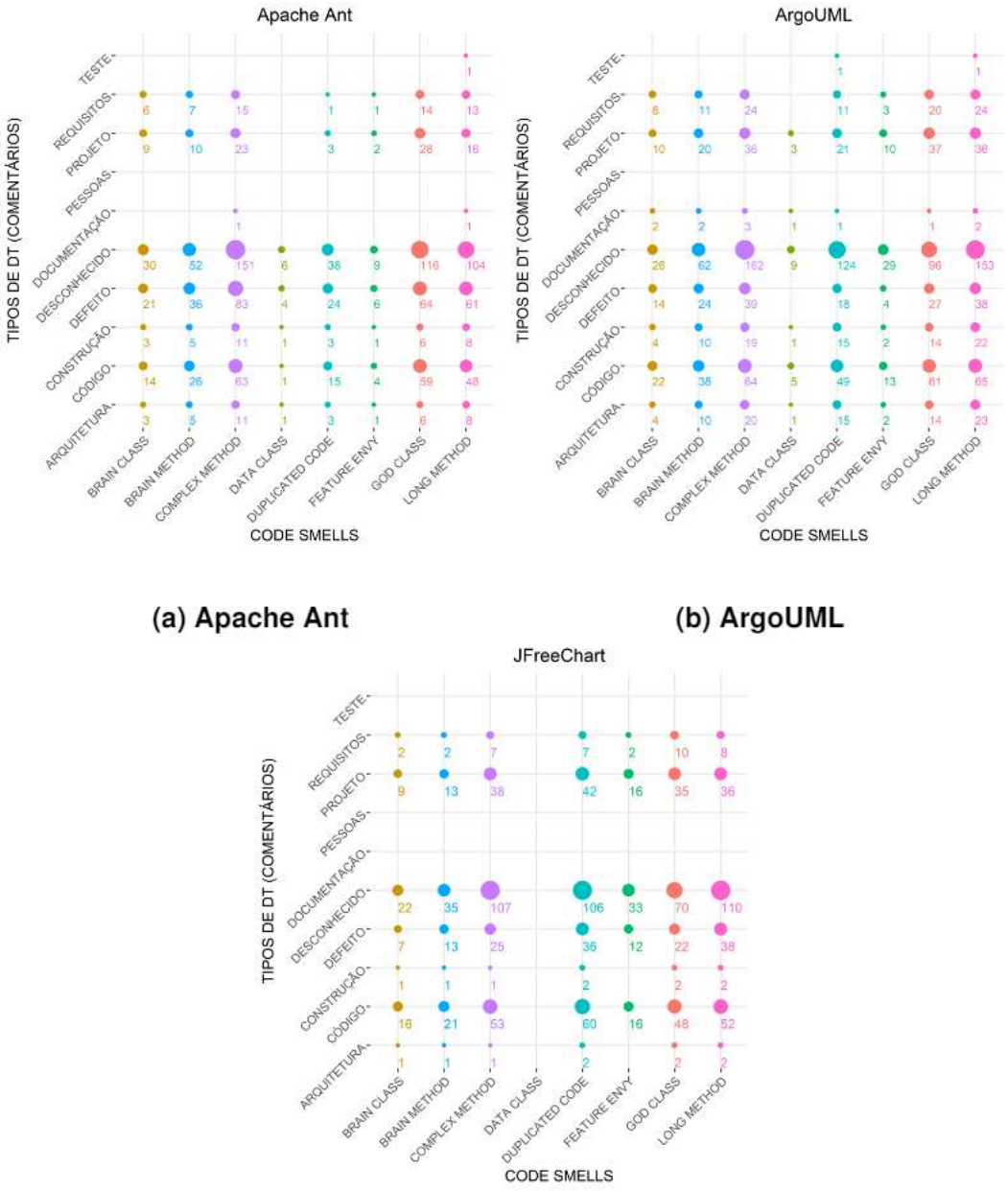
5. Análise Qualitativa

Foi realizada uma análise qualitativa através da leitura dos comentários de código fonte do projeto ArgoUML. Foram analisados todos os arquivos que apresentaram relações entre *code smells* e DTTA. Nas subseções a seguir será apresentada uma breve análise das relações dos comentários com os *code smells Duplicated Code* e *Long Method*.

5.1. Identificação de *Long Method*

O *Long Method* indica que um método, função ou procedimento está crescendo muito. A métrica utilizada para a detecção deste *code smell* é referente ao número de linhas de código [Fowler 1999]. Foram encontrados vários casos onde existem comentários apontando uma sugestão de melhorias em métodos classificados com *Long Method*, como exemplo, o **Comentário #01** que está localizado em um método que apresenta um *Long Method*. Neste caso, o desenvolvedor deixa registrada uma sugestão de melhoria na implementação do método, ele sugere a quebra de método grande e complexo em vários outros métodos menores e mais simples. Neste método, que contém 315 linhas de código, foram encontrados os seguintes *code smells*: *Brain Method*, *Complex Method*, *Long Method* e *Duplicated Code*, *code smells*, que se alinham com o problema apresentado no comentário.

> **Comentário #01 - AbstractMessageNotationUml:** `TODO: This method is too complex, lets break it up.`



(a) Apache Ant **(b) ArgoUML**

(c) JFreeChart

Figura 1: Relação de tipos de DT e *code smells* dos projetos analisados.

5.2. Identificação de *Duplicated Code*

A ocorrência de *Duplicated Code* pode gerar um aumento no tamanho do código e dificultar o processo de sua manutenção [Lanza and Marinescu 2006]. Por exemplo, se for necessário alterar vários trechos de forma semelhante, fatalmente um trecho ou outro será esquecido. No estudo foram encontrados casos de comentários indicando a ocorrência de código duplicado tanto em partes que continham o *code smell Duplicated Code* como em partes que não possuíam. Em alguns desses casos, o desenvolvedor informa que existe uma duplicação do código e que ela deve ser corrigida, por exemplo: **Comentários #02** e **#03**. No **Comentário #02**, que referencia uma ocorrência do *code smell Duplicated*

Code, é descrita uma situação de um trecho de código que foi copiado de outra classe devido a um erro em uma dependência e que será removido quando o erro for resolvido. No segundo exemplo, o **Comentário #03**, que não tem relação com *code smell* algum, o desenvolvedor expressa a duplicação de código entre dois métodos. No caso do **Comentário #03**, não foi identificado o *Duplicated Code*, pois o trecho duplicado era muito pequeno, mas mesmo assim foi considerado um problema pelos desenvolvedores, mostrando que as informações de comentários podem complementar a de análise de smells.

> **Comentário #2 - FigComposite:** `TODO: All code below here is duplicated in FigBaseNode. The reason is the GEF defect - http://gef.tigris.org/issues/show_bug.cgi?id=358 Once we have taken a release of GEF with that fix we can remove this code.`
>
> **Comentário #03 - StateMachinesHelperMDRImpl:** `TODO: getAllPossibleSubvertices and getAllSubStates are duplicates - tfm?`

6. Ameaças à Validade

Foram identificadas algumas ameaças à validade, que são descritas seguindo a classificação usual de ameaças internas, externas, construção e conclusão:

- **Ameaças internas**: O estudo foi baseado em conjuntos de dados extraídos de três projetos consolidados. A principal ameaça é a ocorrência de falsos positivos na detecção de comentários e *code smells*, uma vez que a ocorrência excessiva deles pode comprometer os dados e distorcer as análises. Para melhorar a precisão na detecção dos *code smells* foram utilizados os valores de limiares apresentados por Lanza e Marinescu [Lanza and Marinescu 2006], amplamente adotados na comunidade científica. Portanto, as ameaças à validade interna foram reduzidas, apesar de não ser possível garantir a eliminação dos falsos positivos gerados via análises automáticas.
- **Ameaças externas**: Os resultados encontrados nas análises não podem ser generalizados para outros sistemas. É necessário realizar estudos adicionais com um maior número de projetos de software considerando diferentes contextos e dimensões para se obter maior confiança na validade externa dos resultados.
- **Ameaças de construção**: A análise foi realizada sem considerar os diferentes níveis de granularidade (classes e métodos), podendo resultar na consideração de relações inexistentes entre ocorrências de DTAA e *code smells* nas análises. Contudo, essa análise mais detalhada não é uma tarefa trivial e este é um estudo inicial.
- **Ameaças de conclusão**: Apesar do estudo ter considerado três projetos, o conjunto de dados é bastante amplo para análise devido ao número identificado de itens de DT. Além disso, os softwares ArgoUML e JFreeChart foram utilizados em vários trabalhos sobre identificação de DT [Farias et al. 2015] [da Silva Maldonado et al. 2017]. Todavia, a validade de conclusão será melhor sustentada após a replicação do estudo em outros projetos, para o qual os dados e informações coletadas permitirão um aprimoramento do estudo atual.

7. Considerações Finais

Neste trabalho, foi conduzido um estudo para avaliar a relação entre *code smells* e DTAA através da avaliação de três projetos *open source*. Os resultados do estudo corroboram

com a noção geral da literatura dos impactos dos *code smells* na qualidade do código fonte. Além disso, também são apresentadas relações ainda pouco exploradas na literatura, como os impactos dos *code smells* na construção, requisitos, defeito e documentação, indicando que *code smells* podem ser indicadores de problemas tanto na avaliação da qualidade interna como externa do projeto. Os resultados também mostram que em alguns casos a utilização de comentários de código fonte pode ajudar na complementação de informações que não poderiam ser obtidas apenas com o uso de *code smells*. Como trabalhos futuros, pretende-se estender o estudo e realizar análises em outros projetos e com a participação de engenheiros de software para triangulação de dados.

Referências

Alves, Nicolli, S., Mendes, T. S., Mendonça, M. G., Spínola, R. O., Shull, F., and Seaman, C. (2016). Identification and management of technical debt: A systematic mapping study. *Information and Software Technology*, 40:100–121.

da Silva Maldonado, E., Shihab, E., and Tsantalis, N. (2017). Using natural language processing to automatically detect self-admitted technical debt. *IEEE Transactions on Software Engineering*, 43(11):1044–1062.

Farias, M. A., de Mendonça Neto, M. G., da Silva, A. B., and Spínola, R. O. (2015). A contextualized vocabulary model for identifying technical debt on code comments. In *7th MTD*, pages 25–32. IEEE.

Fontana, F. A., Ferme, V., and Spinelli, S. (2012). Investigating the impact of code smells debt on quality code evaluation. In *Proceedings of the MTD*, MTD '12, pages 15–22, Piscataway, NJ, USA. IEEE Press.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, USA.

Gomes, F., Mendes, T., Carvalho, L., Spínola, R., Novais, R., and Mendonça, M. (2017). Repositoryminer – uma ferramenta extensível de mineração de repositórios de software para identificação automática de dívida técnica. *CBSoft - Salão de Ferramentas.*

Kerievsky, J. (2005). *Refactoring to patterns*. Pearson Deutschland GmbH.

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

McConnell, S. (2008). Productivity variations among software developers and teams: The origin of 10x. *10x Software Development*.

Potdar, A. and Shihab, E. (2014). An exploratory study on self-admitted technical debt. In *ICSME, 2014 IEEE International Conference on*, pages 91–100.

Seaman, C. and Guo, Y. (2011). Advances in computers. In Zelkowitz, M. V., editor, *Chapter 2 - Measuring and Monitoring Technical Debt*, volume 82 of *Advances in Computers*, pages 25 – 46. Elsevier.

Zazworka, N., Spínola, R. O., Vetro', A., Shull, F., and Seaman, C. (2013). A case study on effectively identifying technical debt. In *Proceedings of the 17th EASE*, EASE '13, pages 42–47, New York. ACM.