Universidade Federal do Piauí
Centro de Ciências da Natureza
Programa de Pós-Graduação em Ciência da Computação

Efeitos da visualização de indícios de dívidas técnicas em um projeto de manutenção de software

Ronivon Silva Dias

Número de Ordem PPGCC: M001

Teresina-PI, 13 de Setembro de 2019

Ronivon Silva Dias

Efeitos da visualização de indícios de dívidas técnicas em um projeto de manutenção de software

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da UFPI (área de concentração: Sistemas de Computação), como parte dos requisitos necessários para a obtenção do Título de Mestre em Ciência da Computação.

Universidade Federal do Piauí – UFPI

Centro de Ciências da Natureza

Programa de Pós-Graduação em Ciência da Computação

Orientador: Pedro de Alcântara dos Santos Neto

Teresina-PI

13 de Setembro de 2019

Ronivon Silva Dias

Efeitos da visualização de indícios de dívidas técnicas em um projeto de manutenção de software/ Ronivon Silva Dias. – Teresina-PI, 13 de Setembro de 2019-

72 p. : il. (algumas color.) ; 30 cm.

Orientador: Pedro de Alcântara dos Santos Neto

Dissertação (Mestrado) – Universidade Federal do Piauí – UFPI
Centro de Ciências da Natureza
Programa de Pós-Graduação em Ciência da Computação, 13 de Setembro de 2019.

1. Code Smell 2. Dívidas Técnicas. 3. Engenharia Reversa. 4. Identificação de Funcionalidades. 5. Refatoração. 6. Manutenção de Software. 7. Mineração de Repositórios de Software. I. Pedro de Alcântara dos Santos Neto. II. Universidade Federal do Piauí. III. Efeitos da visualização de indícios de dívidas técnicas em um projeto de manutenção de software.

CDU 02:141:005.7

Ao meu pai Raimundo Nonato Dias (in memoriam),
por sempre ter batalhado muito pela minha educação.

Agradecimentos

Gostaria de agradecer primeiramente a Deus pelas bençãos a mim concedidas durante toda minha vida, dando-me força, capacidade e condições para alcançar este objetivo.

À minha esposa, Irlane, sem a qual a caminhada seria muito mais árdua. Seu incentivo, companheirismo e compreensão nos momentos mais difíceis foram fundamentais. suportou meus momentos de mau humor predominantes e teve de conviver com minha falta de atenção. Obrigado pelo carinho, amor e por acreditar sempre que eu venceria todas as dificuldades.

À minha família e principalmente aos meus pais, que me proporcionaram uma educação sólida e por me transmitirem seus valores que moldaram meu caráter. Obrigado por sempre me incentivarem a progredir de forma honesta.

Ao meu orientador, Prof. Dr. Pedro Alcantara, ao qual dei muito trabalho na organização de ideias. Agradeço pelos conselhos, motivação nos momentos difíceis e pelas cobranças, que me ajudaram a crescer como profissional e apresentar um trabalho melhor. Obrigado, acima de tudo, por sua amizade.

Gostaria de agradecer a todos os colegas de trabalho que supriram minhas atividades em períodos de ausência para dedicação a este trabalho.

Por fim, agradeço a todos que de uma forma ou outra contribuíram para a realização deste trabalho, seja com atos concretos ou com bons pensamentos.

Muito obrigado a todos!

"...O pensamento é a força criadora. O amanha é ilusório, porque ainda não existe, o hoje é real. É a realidade que você pode interferir, as oportunidades de mudança, Tá no presente. Não espere o futuro mudar sua vida, porque o futuro será a consequência do presente. Parasita hoje, um coitado amanhã. Corrida hoje, vitória amanhã...".

(Racionais MCs)

Resumo

A metáfora da dívida técnica (DT) é amplamente usada para encapsular inúmeros problemas de qualidade de *software*. Ela descreve o compromisso entre o benefício a curto prazo de tomar um atalho durante a fase de concepção ou implementação de um produto de *software* (por exemplo, para cumprir um prazo) e as consequências a longo prazo de tomar esse atalho, o que pode afetar a qualidade do produto de *software*. Algumas DT podem nunca causar falhas, mas, ao invés disso, tornam um *software* menos eficiente, menos escalável, mais difícil de aprimorar ou mais facilmente violável, como por exemplo os *code smells*. A presença de *code smells* no projeto fazem com que seja mais difícil adicionar novas funções, facilitam o surgimento de defeitos, impactam na qualidade externa e reduzem a manutenibilidade do código. Os *code smells* devem ser gerenciados para garantir a qualidade do *software* e também reduzir seus custos de manutenção e evolução. No entanto, as ferramentas para a detecção de *code smells* geralmente fornecem resultados apenas considerando a perspectiva dos arquivos (classes e métodos), o que não é usual durante a gestão do projeto. Neste trabalho, uma técnica é proposta para identificar/visualizar *code smells* em uma nova perspectiva: funcionalidade de *software*. A técnica proposta adota ferramentas de Mineração de Repositório de Software (MRS) para identificar as funcionalidades do *software* e em seguida os *code smells* que afetam essas funcionalidades. Além disso, também propusemos uma abordagem para apoiar tarefas de manutenção guiadas pela visualização dos *code smells* em nível de funcionalidades com o objetivo de avaliar sua aplicabilidade em projetos de *software* real. Foi realizado um estudo de caso para avaliar tal abordagem, sendo obtidos resultados que indicam que a abordagem pode ser útil para diminuir os *code smells* existentes, bem como evitar a introdução de novos *code smells*.

Palavras-chaves: *code smells*. Dívidas Técnicas. Identificação de Funcionalidades. Refatoração. Manutenção de *software*. Mineração de Repositórios de *software*.

Abstract

The technical debt (TD) metaphor is widely used to encapsulate numerous software quality problems. It describes the trade-off between the short term benefit that taking a shortcut during the design or implementation phase of a software product (for example, in order to meet a deadline) and the long term consequences of taking said shortcut, which may affect the quality of the software product. Some DTs can never cause crashes, but instead make software less efficient, less scalable, harder to upgrade, or more easily breached, such as code smells. The presence of code smells in a project make it more difficult to add new features, facilitate the emergence of defects, impact on external quality and reduce the maintainability of the code. Code smells must be managed to guarantee the software quality and also reduce its maintenance and evolution costs. However, the tools for code smells detection usually provide results only considering the files perspective (class and methods), that is not usual during the project management. In this work, a technique is proposed to identify/visualize code smells from a new perspective: software features. The proposed technique adopts Mining Software Repository (MRS) tools to identify the software features and also the code smells that affect these features. Additionally, we also proposed an approach to support maintenance tasks guided by code smells visualization at the feature level aiming to evaluate its applicability on real software projects. The results indicate that the approach can be useful to decrease the existent code smells, as well as avoid the introduction of new ones.

Keywords: code smells. Functionality Identification. Mining Software Repositories. Refactoring. Software Maintenance. Technical Debt.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

API	*Application Programming Interface*
AST	*Abstract Syntax Tree*
VCS	*Version Control System*
DCBD	Descoberta de Conhecimento em Banco de Dados
DT	Dívida Técnica
ER	Engenharia Reversa
ES	Engenharia de textitSoftware
FL	*Feature Location*
JDT	*Java Development Tools*
LPS	Linha de Produto de Software
MRS	Mineração de Repositório de Software
MSR	*Mining Software Repositories*
OO	Orientação a Objetos
QP	Questão de Pesquisa
QS	Qualidade de Software
SCV	Sistema de Controle de Versão
SINAPSE	Sistema Integrado de Acompanhamento a Projetos e Serviços
SIGAA	Sistema Integrado de Gestão de Atividades Acadêmicas
SVN	*Subversion*
UFPI	Universidade Federal do Piauí

Sumário

Introdução

Contexto e Motivação

A demanda por qualidade é um fator a ser perseguido por todas as áreas, não apenas pela Engenharia de *Software*. Comumente, empreendedores querem oferecer um produto ou serviço de qualidade ao cliente. A qualidade na Engenharia de *Software* é obtida pela criação de *software* estável, adequado ao uso, seguro, íntegro e que atenda bem ao usuário final e aos seus objetivos. Ou seja, para considerar que um software possua qualidade, não basta funcionar corretamente, mas necessita ser bem estruturado e que possa ser facilmente mantido (ROCHA ET AL., 2001).

De acordo com Pressman (2015), em geral, nos projetos de *software*, quando os prazos ou recursos se tornam escassos, as organizações tendem a comprometer tarefas e práticas relacionadas à qualidade de *software*. Como consequência, problemas de qualidade podem ser observados no produto durante o projeto ou após sua implantação. Tais tarefas comprometidas precisam ser concluídas em algum momento ao longo do projeto, e se não concluídas, podem gerar uma dívida, ou deficit ao projeto.

As equipes de desenvolvimento de *software* enfrentam com frequência o desafio de entregar produtos de *software* em prazos apertados, enquanto tentam manter um padrão de qualidade. Como resultado, surgem artefatos de baixa qualidade, pois os desenvolvedores tendem a deixar de lado as boas práticas de engenharia e princípios de programação, para entregar o produto dentro do prazo estipulado, ainda que imaturo sob diversos aspectos (BROWN Y. CAI, 2010; ZAZWORKA R. O. SPíNOLA; SEAMAN, 2013). Esse processo pode muitas vezes resultar na introdução do que é conhecido como Dívida Técnica (DT) (CUNNINGHAM, 1992), que ocasiona problemas para o *software* em futuras manutenções e consequentemente, compromete a sua evolução (ZAZWORKA R. O. SPíNOLA; SEAMAN, 2013).

W. Cunningham (1992) iniciou a metáfora da DT, referindo-se às violações de boas práticas arquitetônicas e de codificação como "dívida". De acordo com Cunningham, "o código de primeira entrega é como entrar em dívida. Uma pequena dívida acelera o desenvolvimento desde que seja devolvida prontamente com uma reescrita. O perigo ocorre quando a dívida não é paga. Cada minuto gasto em um código não muito certo conta como juros dessa dívida. Organizações inteiras de engenharia podem ser colocadas em risco sob a carga de dívida de uma implementação não consolidada".

A DT consiste em um conjunto de pendências ocorridas durante o desenvolvimento de *software*, proporcionado por uma escolha que é interessante a curto prazo, levando-se

em consideração a produtividade, tempo de entrega, redução de esforço e custo, mas que aumenta a complexidade e é mais oneroso a longo prazo (KRUCHTEN; NORD; OZKAYA, 2012). Além disso, os efeitos negativos da DT na evolução do *software* podem comprometer a qualidade e reduzir a manutenibilidade do código, e ainda contribuir para um ambiente favorável ao surgimento de defeitos (SPíNOLA R.; GONÇALVES; GOMES, 1992), gerando problemas para o ecossistema do desenvolvimento (TOM; VIDGEN, 2013). Por conta disso, é crescente a atenção para o gerenciamento de DT, tanto na comunidade de desenvolvimento de *software*, quanto na científica (BROWN Y. CAI, 2010).

A DT deve ser diferenciada de defeitos ou falhas. Falhas durante o teste ou operação podem ser sintomas de DT, mas a maioria das falhas estruturais que criam DT não causam falhas de teste ou operacionais. Algumas DT podem nunca causar falhas, mas, ao invés disso, tornam um *software* menos eficiente, menos escalável, mais difícil de aprimorar ou mais facilmente violável, como por exemplo os *code smell*. A presença de *code smell* indica que há problemas com a qualidade do código, como a compreensibilidade, por exemplo, o que pode levar a uma variedade de problemas de manutenção, incluindo a introdução de falhas (E. ALLMAN, 2012). Em essência, DT emerge de má qualidade estrutural e afeta um negócio como custo de TI e risco de negócio. Uma solução simples para o problema seria reembolsar (através de manutenção/refatoração) os indícios de DT conhecidos antes dos problemas começarem a aparecer.

Uma boa prática para evitar que isso aconteça é realizar modificações no código, isto é, refatorações. A refatoração surgiu da necessidade dos desenvolvedores em adicionar ou modificar funcionalidades em código fonte que, na maioria dos casos, estavam desestruturados, difícil de serem compreendidos e com trechos duplicados, o que tornava a manutenção ainda mais custosa (MOREIRA, 2015). Apesar da refatoração visar a melhoria do código fonte, deve ser bem pensada e executada, para não trazer riscos como atrasos, inserção de falhas no sistema ou tornar o código de difícil entendimento para futuras manutenções, ou seja, a refatoração é um esforço para melhorar o software existente no nível de código, sem alterar o comportamento do sistema.

A gestão da DT compreende as ações de identificação, avaliação e pagamento da dívida que acompanha um sistema durante o seu desenvolvimento (GRIFFITH ET AL., 2014). Para (GUO ET AL., 2014), o principal objetivo de identificar e medir a DT é viabilizar e facilitar a tomada de decisão sobre a necessidade de eliminá-la e o momento mais oportuno de fazer isso. Estratégias de gerenciamento de DT têm sido propostas no intuito de minimizar possíveis impactos negativos do acúmulo da dívida e têm como objetivo principal avaliar o momento mais adequado para que os itens da dívida sejam eliminados do projeto (GUO ET AL., 2014). Baseado nisso, este trabalho busca oferecer um método para identificar e visualizar indícios de DT em nível de funcionalidade e uma abordagem que faz uso desse método, para auxiliar no pagamento desses indícios de DT

em um ambiente de desenvolvimento real.

Objetivos

Este trabalho tem como objetivo geral apoiar a gestão dos *code smells*, que são indícios de DT. Isso é feito por meio da exposição de tais *code smells* associadas a tarefas de manutenção e evolução de *software*, na tentativa de influenciar a eliminação desses *code smell*. A questão de pesquisa diretamente ligada a esse objetivo é elencada a seguir:

- A exposição e visualização de *code smell*, ligadas a uma tarefa de manutençãoevolução de software, influencia na redução de indícios de DT em um projeto?

Para alcançar o objetivo geral deste trabalho, foram definidos os seguintes objetivos específicos:

1. Definição de um método para identificar indícios de dívidas técnicas utilizando a perspectiva de funcionalidades. O método proposto utiliza a abordagem desenvolvida pelo grupo de pesquisa ligado a este trabalho, para identificar funcionalidades de software e os arquivos que as compõem. Utilizando essa identificação infere-se os *code smells* ligados a tais arquivos e assim computasse o total de *code smell* da funcionalidade.

2. Extensão da CoDiVision (LIRA, 2016) para implementar o método de visualização de *code smell* por funcionalidades. Esse novo módulo será responsável por computar o total de *code smell* das funcionalidades e gerar informações para que sejam visualizadas via interface *Web*, permitindo assim que os desenvolvedores ligados a tarefas de manutenção/evolução de software visualizem os *code smells* ligados as funcionalidades a serem alteradas.

3. Definição de uma abordagem para manutenção/evolução de *software* com foco na redução de indícios de DT. Com essa abordagem é possível influenciar a gestão de DT durante a realização de tarefas cotidianas de manutenção/evolução de software em um ambiente industrial, sendo pouco invasivo e alterando de forma mínima o processo de software utilizado.

4. Realização de um estudo de caso avaliando a abordagem proposta neste trabalho em um cenário industrial.

Justificativa

A presença de *code smell*, situação enfrentada em diversos sistemas de software, torna mais difícil a adição de novos requisitos, cria um ambiente favorável para o surgimento de defeitos, impacta na qualidade externa e reduz a manutenibilidade do código (N. ZAZWORKA;SEAMAN;SPíNOLA, 2013). Esse cenário tem estimulado a realização de esforços no sentido de desenvolver estratégias para identificar e gerir a DT (ALVES ET AL., 2016). A redução de defeitos e a gerência de riscos no desenvolvimento de *software* são desafios para os desenvolvedores e gerentes (BUSE; ZIMMERMANN, 2012).

Para lidar com os *code smells* existentes em um sistema de *software* ou para evitar que dívidas potenciais sejam introduzidas, Li et al. (LI; LIANG., 2015) identificaram e propuseram oito atividades de manejo de DT. Essas oito atividades são: a identificação, medição, priorização, prevenção, monitoramento, reembolso, documentação e comunicação da DT. Dentre essas atividades, a identificação (detecção de DT usando técnicas como análise de código estático), medição (quantificação de DT usando técnicas de estimativa) e reembolso (resolução de DT por técnicas como reengenharia ou refatoração) recebem maior atenção da comunidade técnica e científica, com apoio de ferramentas e abordagens apropriadas (LI; LIANG., 2015).

No entanto, as ferramentas para detecção de *code smell* geralmente fornecem resultados só considerando a perspectiva de arquivos de código fonte (classes e métodos) (S.A., 2019; ARCELLI C. TOSI; MAGGIONI., 2008; MARTINI; BOSCH, 2017a; MARTINI; BOSCH, 2017b; MENDES et al., 2015; HOLVITIE; LEPPäNEN., 2013), que por si só, tende a não ser apropriada considerando um contexto real. É comum que durante a manutenção de *software*, não se analise um arquivo de código fonte em específico, mas sim uma funcionalidade, que pode ser composta por diversos arquivos de código fonte, em diversas pastas diferentes. Ou seja, a perspectiva de funcionalidade é a mais comumente ligada à manutenção, especialmente no que se refere à distribuição de tarefas (KERSTEN; MURPHY, 2005).

Na prática, quando um desenvolvedor recebe uma tarefa associada à resolução de um problema ou ligada a uma melhoria, é comum que tal tarefa cite a funcionalidade associada, sem qualquer ligação com os arquivos de código fontes ligados ao caso. Por conta disso, faz-se necessário a existência de uma nova perspectiva de visão das DT, para facilitar sua gestão, e a partir disso, facilitar as decisões gerenciais associadas.

Por fim, Wohlin et al. (WOHLIN et al., 2012) afirmam que a única avaliação real de um processo/método é ter pessoas usando-o, já que o mesmo é apenas uma descrição até que as pessoas o utilizem. De fato, o caminho da subjetividade para a objetividade é pavimentado por testes ou comparação empírica com a realidade (JUZGADO N. J.; VEGAS, 2004). Por conta disso, foi proposto um método que visa identificar e visualizar

os indícios de DT em nível de funcionalidade e uma abordagem que faz uso desse método, para que assim pudéssemos avaliá-lo em um ambiente real de desenvolvimento.

Estrutura do Trabalho

Além da introdução aqui apresentada, este trabalho está dividido em cinco outros capítulos. O Capítulo 1 apresenta os conceitos fundamentais sobre técnicas e componentes utilizados neste trabalho, abordando, por exemplo, conceitos sobre dívidas técnicas, Mineração de Repositórios de Software (MRS), manutenção e evolução de software e funcionalidades de software.

O Capítulo 2 discute sobre os principais trabalhos relacionados ao tema, apresentando uma visão geral da área de gestão de dívidas técnicas. No Capítulo 3 é detalhado o método proposto neste trabalho para identificação e visualização de dívidas técnicas, considerando a perspectiva de funcionalidades de software. Nesse mesmo capítulo, na Seção 3.1.4 é apresentado o modulo TDVision que é uma extensão da CoDiVision para implementar a visualização proposta. Na Seção 3.2 do referido capítulo é apresentada a abordagem criada que faz uso da visualização de DT por funcionalidade durante a manutenção de software, com foco na redução de DT.

O Capítulo 4 apresenta um estudo de caso para avaliação da abordagem proposta neste trabalho. Por fim, o Capítulo 5 apresenta as conclusões deste trabalho, assim como desafios e limitações encontrados e as perspectivas para trabalhos futuros.

1 Fundamentação Teórica

Nesta seção serão descritos alguns conceitos importantes para o entendimento do trabalho aqui proposto.

1.1 Dívida Técnica

A metáfora da Dívida Técnica (DT) no desenvolvimento de *software* foi introduzida há duas décadas por W. Cunningham (1992), para explicar aos stakeholders não técnicos do produto, a necessidade de refatoração. A maioria dos autores concorda que a principal causa da dívida técnica é a pressão do cronograma. No entanto, quando a dívida está associada a questões de qualidade e manutenibilidade, outras causas se tornam prováveis, como descuido, falta de instrução, processos insatisfatórios, verificação não sistemática da qualidade ou incompetência básica. Alguns autores caracterizam as DTs como sendo qualquer parte do *software* atual que é considerado abaixo de um bom nível técnico (TOM; VIDGEN, 2013).

Entendendo-se que é difícil estabelecer um modelo que pondere fatores humanos, artefatos de projeto de *software* de alto nível produzidos durante o processo de desenvolvimento e processos organizacionais, a dívida técnica é mais frequentemente associada ao nível de design e a artefatos de código (BROWN; NORD; OZKAYA, 2010). Entre os diferentes tipos de dívida técnica, uma das muitas questões que envolvem o código-fonte é a presença de cheiros (*smells*) de código, ou seja, sintomas de design inadequado e escolhas de implementação (W. CUNNINGHAM, 1992). Uma classe complexa, isto é, uma classe que contém métodos complexos e é muito grande em termos de linhas de código ou uma Classe Deus (*God Class*), ou seja, uma classe que faz muito ou sabe muito sobre outras classes, são apenas alguns exemplos de um dos mais importantes *smells* identificados e caracterizados em catálogos bem conhecidos (E. Allman (2012)). Portanto, há evidências empíricas de que os *smells* têm um efeito negativo na evolução do *software* e, portanto, devem ser cuidadosamente monitorados e, possivelmente, removidos por meio de operações de refatoração. Assim, muito esforço tem sido dedicado para a definição de abordagens com o objetivo de detectar e remover os *code smell* (W. CUNNINGHAM, 1992).

Os *code smell* são inevitáveis em boa parte dos projetos reais, em que existe uma forte pressão por entregas. A questão não é eliminar dívidas, mas administrá-las. O problema dos *code smell* não é assumi-los, mas sim esquecê-los. Muitas empresas assumem os indícios de DT para atingir um objetivo e o esquecem, deixando-o fugir de seu controle. Deve-se aceitar a dívida técnica como inevitável, porém um "plano para mudança" deve existir, caso algo necessite ser remediado (E. ALLMAN, 2012). Quando um projeto

começa, a equipe quase nunca tem uma compreensão completa da totalidade do problema. A realidade é que os requisitos sempre mudam. Muitas vezes é melhor ter um protótipo funcional (mesmo que não seja completo ou perfeito) para que a equipe de desenvolvimento e os clientes possam começar a ganhar experiência com o sistema. Essa é a filosofia por trás da programação ágil, que aceita alguma dívida técnica como inevitável, mas também exige um processo de remediação. Por mais necessário que seja a dívida técnica, é importante que as partes estratégicas sejam prontamente reembolsadas. Igualmente importante é que algumas formas de dívida técnica são tão caras que devem ser evitadas inteiramente sempre que possível. A segurança é uma área onde os atalhos podem levar ao desastre. A DT é análoga à dívida financeira, pois algumas dívidas são benéficas, porque facilitam o crescimento, mas a dívida em demasia se torna um fardo por causa da necessidade de pagá-la à custa de recursos valiosos.

Para entender melhor a metáfora, pode-se dividi-la em duas partes. Na primeira, (W. CUNNINGHAM, 1992) enfatiza que se a qualidade do código é violada, ou seja, caso o membro da equipe utilize de práticas imaturas de codificação, a dívida é contraída. Na segunda parte, o autor ratifica a tendência das empresas em contrair a dívida para acelerar outros processos. Apesar de parecer interessante adquirir uma pequena dívida para adiantar alguma tarefa, isso é perigoso ao longo do projeto, pois durante o tempo em que essa dívida está no código, existem juros acrescidos e que deverão ser pagos por meio de refatoração (W. CUNNINGHAM, 1992).

Segundo McConnell (2013), a DT pode ser classificada em dois tipos básicos: intencional e não intencional. A primeira acontece normalmente quando uma organização toma uma decisão consciente de otimizar para o presente do que para o futuro. “Se não conseguirmos lançar à tempo, não haverá próximo lançamento” é um refrão comum – e normalmente bem convincente. Isso leva a decisões como “Não temos tempo para reconciliar esses dois bancos de dados, então vamos escrever alguma gambiarra que os mantenha sincronizados por agora e vamos reconciliar depois de lançar.”. A razão para a DT não intencional pode ser a falta de competência, a necessidade de atualizar as tecnologias existentes ou uma necessidade de mudança induzida pelo cliente ou pelo mercado. Um codificador pode não ter competência para desenvolver uma solução ideal. Uma equipe de desenvolvimento pode não conseguir fornecer instruções adequadas e padrões de codificação para desenvolvimento, o que reduz a qualidade da solução. No *software* legado, a tecnologia antiga que ainda está em uso também pode ser vista como DT não intencional. Nessas situações, uma empresa às vezes precisa começar a atualizar a tecnologia para uma versão mais nova. A dívida técnica mais perigosa que temos a priori é a sem intenção. Se a equipe não utilizar de métodos e ferramentas para identificá-la e, em seguida, medi-la e acompanhá-la, ela pode fazer um projeto ir à falência. A falência de um projeto significa que ele deverá ser reescrito ou até mesmo abandonado. Qualquer um dos casos pode gerar um grande volume de trabalho, uma perda de tempo e esforços ou até mesmo criar um

impacto financeiro muito grande.

Gerentes de *software* precisam equilibrar os benefícios da dívida técnica com os custos associados na tomada de decisão de adquirir ou pagar a dívida técnica e quando. A incerteza associada à dívida técnica torna a tomada de decisão mais complexa. Portanto, a identificação, medida e monitoramento da dívida técnica ajudariam os gerentes a terem informações para a tomada de decisão. Resultando, assim, em maior visibilidade, o que pode melhorar a qualidade e produtividade na manutenção do *software*. Gerenciar de onde vem a dívida dentro do ciclo de vida de *software* ajuda a identificar o pagamento da DT no ponto específico. Li et al. (2015) realizaram um mapeamento para classificar de onde podem vir as DT dentro do ciclo de vida de um *software*. As classificações obtidas foram:

- DT de Requisitos: refere-se à distância entre a especificação de requisitos ideal e a aplicação efetiva do sistema, com base em pressupostos de domínio e suas restrições.
- DT Arquitetural: é causada por decisões de arquitetura que fazem concessões em alguns aspectos internos de qualidade, tais como manutenção.
- DT de Projeto: refere-se a atalhos técnicos que são tomados no projeto.
- DT de Código: representa o código mal escrito que viola as melhores práticas de codificação ou regras de codificação. Exemplos incluem a duplicação de código e código excessivamente complexo.
- DT de Teste: refere-se a atalhos tomados em testes. Um exemplo é a falta de testes (por exemplo, testes unitários, testes de integração e testes de aceitação).
- DT de Construção: refere-se a falhas em um sistema de *software* no seu sistema de construção, ou em seu processo de construção que fazem a compilação excessivamente complexa e difícil.
- DT de Documentação: diz respeito à documentação insuficiente, incompleta ou desatualizada em qualquer aspecto do desenvolvimento de *software*. Exemplos incluem documentação da arquitetura desatualizada e a falta de comentários de código.
- DT de Infraestrutura: refere-se a uma configuração sub-ótima dos processos relacionados com o desenvolvimento, tecnologias, ferramentas de apoio, entre outros fatores. Tal configuração sub-ótima afeta negativamente a capacidade da equipe para criar um produto de qualidade.
- DT de Controle de Versão: refere-se aos problemas no versionamento do código fonte, como contribuições de código desnecessárias.
- DT de Defeito: representa defeitos, erros ou falhas encontradas em sistemas de *software*.

Segundo Li et al. (2015), foram identificadas oito atividades relacionadas à gestão de DT. São elas: Identificação, Medição, Priorização, Prevenção, Monitoramento, Reembolso, Representação/Documentação e Comunicação. Vale ressaltar que todas estas atividades são herdadas das DT's financeiras e ajudam na gestão de DT's em *software* (LI ET AL., 2015).

Li et al. (2015) ainda apresentam a relação entre estas 8 atividades, descrevendo que o primeiro passo está relacionado ao momento que se identifica uma DT, seja intencional ou não-intencional. O próximo passo é a medição desta DT, vendo qual o seu custo e benefício para saná-la. Então, faz-se uma priorização do quanto o pagamento da DT é importante para ser realizado ou se este pagamento pode ser efetuado mais tarde. Caso a DT esteja sendo paga, deve-se fazer um monitoramento que identifique se de fato ela está sendo diluída, ou, se não está sendo paga e está causando maiores custos, ou ainda, se a DT está sendo postergada. De forma contínua, é necessário monitorar a prevenção da DT para que ela não ocorra novamente em outros casos. Já o reembolso representa o ato de parar para pagar a DT.

O reembolso da DT é feito refatorando ou reescrevendo as soluções ruins (CODABUX; WILLIAMS, 2013). Refatorar ou reescrever pode ser visto como um processo para "mudar um sistema de *software* de tal maneira que ele não altere o comportamento externo do código, mas que melhore sua estrutura interna. É uma maneira disciplinada de limpar o código que minimiza as chances de introduzir erros. Em essência, quando você refatora, você está melhorando o design do código depois que ele foi escrito " (FOWLER ET AL., 1999). No entanto, a alteração de soluções antigas no código não é fácil, pois o aprimoramento da base de código requer um desenvolvedor significativamente competente e a empresa não pode usar todo o tempo de desenvolvimento para refatorar ou reescrever as soluções. Portanto, ter algumas abordagens de assistência para saber quando, o que é necessário refatorar pode ser útil para as equipes de desenvolvimento.

Algumas técnicas existentes e suas ações para efetuar o pagamento ou a correção da DT em um software, foram descritas por GONG; LYU. 2017, entre elas estão: (i) Refatoração: alterações no código, no design ou na arquitetura do software visando melhorar a qualidade interna sem afetar o comportamento do sistema, (ii) Reescrita: reescrever o código que contém *code smell*, (iii) Automação: automatizar trabalhos repetidos manualmente, como testes e compilações manuais, (iv) Reengenharia: evolução do software existente para exibir novos recursos e melhorar a sua qualidade operacional, (v) Reempacotamento: agrupar módulos coesos, para simplificar as dependências entre os mesmos, visando melhorar a capacidade de manutenção do sistema, (vi) Resolução de Bugs: resolver erros identificados no software e (vii) Tolerância a falhas: adicionar estrategicamente exceções de tempo de execução no local que contêm DT. Em geral, os problemas encontrados exigem uma refatoração para que possam ser corrigidos.

Existem três maneiras possíveis de tomada de decisão perante uma DT: pagar os juros, pagar a dívida ou converter a dívida (BUSCHMANN., 2011). Uma outra solução ainda pode ser acrescentada que é a estratégia de continuar com a dívida. Quando os juros são pagos, sofrem-se as consequências da dívida e para lidar com ela incorrem-se em custos adicionais (BUSCHMANN., 2011). Por exemplo, se existe um programador na equipe que sempre programa da mesma forma e isso implica na contração de dívida, toda vez que outro membro da equipe precisar dispender seu tempo para arrumar o código feito pelo programador causador da dívida se estará pagando juros dessa dívida. Se a opção for pagar a dívida, livra-se de vez dessa dívida ao custo de significativo esforço de refatoração extra (BUSCHMANN., 2011). No caso descrito anteriormente uma solução seria delegar alguém para treinar o programador a codificar no padrão desejado pela organização. Caso a escolha seja converter a dívida, substitui-se sua fonte com uma solução que implicará em uma nova dívida, porém, normalmente, menor. Por exemplo, quando se opta por pagar a dívida do design e a documentação é atrasada por conta disso. Por fim, pode-se ainda escolher continuar com a DT. De forma consciente, a dívida pode ser mantida no sistema, se bem monitorada. Como exemplo, pode-se imaginar um caso onde um método não foi codificado da forma que deveria, mas supre as necessidades. A organização pode monitorar esse trecho de código e optar por continuar com ele enquanto ele continuar cumprindo com suas obrigações.

1.2 Code Smells

Dentre as dívidas que podem surgir em um projeto, podemos destacar a Dívida de Código. Essas DT podem ser caracterizados por indicadores identificados por meio de cálculo de métricas de código, detecção de *code smells* e/ou análise de comentários. O conceito de *code smells* foi definido por Fowler et al. (1999) e descreve problemas relacionados com orientação a objetos e outros problemas comuns que envolvem conjuntos de linhas de código.

Fowler et al. (1999) definiram vinte e dois tipos de code smells que fornecem um conjunto de características utilizadas como indicadores de problemas de design de sistemas de software. Cada *code smell* examina um tipo específico de elemento do sistema (classe ou método, por exemplo). Por exemplo, eles definiram um *code smell* chamado de Data Class. Data Class é uma classe que tem atributos, métodos de acesso a esses atributos e pouca coisa a mais do que isso. Segundo Fowler et al. (1999), classes com essas características tendem a ser manipuladas em detalhes por outras classes. Muitas vezes, métodos que deveriam estar nessa classe estão em outras classes.

Como *code smells* envolve uma variedade de problemas é necessário que várias técnicas sejam adicionadas ao método para que se possa identificá-los, seja de forma

manual ou de forma automática. Seguem alguns problemas que são definidos:

- Código Duplicado;
- Métodos/funções longos;
- Classes/Módulos longos;
- Métodos/funções com uma grande lista de parâmetros;
- Mudanças divergentes;
- Aglomeração de dados;
- Mudanças de afirmação.

Neste trabalho, a caracterização de Dívidas de Código é baseado no conjunto de 18 métricas de código e 7 *code smells* (MENDES et al., 2017). Na Tabela 1 são apresentados os *code smells* considerados neste trabalho e suas respectivas descrições.

Tabela 1 – Tipos de code smells considerados neste trabalho.

Code Smell	Decrição
God Class	Caracterizado por classes que centralizam a inteligência de um sistema.
Brain Method	Caracterizado por métodos que centralizam as funcionalidades de uma classe.
Brain Class	Caracterizado por classes que acumulam uma excessiva quantidade de inteligência, normalmente possuindo muitos Brain Methods.
Data Class	Caracterizado por classes que somente armazenam dados e não possuem funcionalidades complexas. Elas oferecem mais dados do que serviços.
Conditional Complexity	Caracterizados por métodos que possuem muitas estruturas condicionais.
Long Method	Caracterizado por métodos que possuem muitas linhas de código.
Feature Envy	Caracterizado por métodos que acessam muitos dados de outras classes e poucos da sua própria classe.

1.3 Mineração em Repositório de *software*

Compreender o processo de evolução de um software é uma tarefa complexa. Sistemas de *software* grandes possuem um longo histórico de desenvolvimento com diversos desenvolvedores trabalhando em diferentes partes do sistema. É comum que nenhum desenvolvedor conheça o código do sistema por completo por conta da sua complexidade ou mesmo porque os integrantes que iniciaram o desenvolvimento do projeto já não fazem mais parte da equipe. Portanto, analisar os dados históricos do desenvolvimento de um *software* grande manualmente é inviável.

Assim, a Mineração de Repositórios de Software (MRS) analisa a evolução de *software* de forma automatizada aplicando técnicas da Mineração de Dados sobre o histórico do desenvolvimento de sistemas de software. A MRS tem como objetivo transformar os repositórios em ativos para apoiar nas atividades inerentes ao processo de desenvolvimento. Nesses repositórios são possíveis descobrir padrões e informações úteis para a compreensão do *software* e do ecossistema envolvido na sua construção (HASSAN, 2008a).

A mineração de dados na Engenharia de *Software* (ES) tem emergido como uma evidente área de pesquisa, oferecendo formas de interpretar a quantidade abundante de dados e, consequentemente, auxiliar na resolução de diversos problemas que envolvem o desenvolvimento de *software* (HASSAN; XIE, 2010). Um pouco à semelhança do que é feito na arqueologia, podemos aprender muito sobre o processo de desenvolvimento de *software* estudando a evolução histórica dos sistemas, a partir da informação guardada nos seus repositórios.

Os repositórios de sistemas de *software* são considerados depósitos de informação sobre as atividades de desenvolvimento e de manutenção de um *software*, ou seja, é formado a partir de dados das ferramentas que apoiam a execução do processo de desenvolvimento de *software* e que retenham informações sobre as atividades realizadas pelos agentes do processo. Neste contexto, a área de MRS se preocupa em extrair dados disponíveis desses repositórios de *softwares* para descobrir informações relevantes sobre sistemas de informações e projetos de *software* (HASSAN, 2008a). O objetivo da MRS é capturar essas informações que em geral estão sem uso e extrair delas coisas importantes que auxiliarão tanto o time de desenvolvimento quanto os gestores.

Uma grande quantidade de dados é produzida durante o processo de desenvolvimento de *software*. Desenvolvedores utilizam diversas ferramentas para coordenar suas tarefas, trabalhar cooperativamente sobre artefatos e comunicar informações sobre o projeto. Ambientes de gerência de tarefas, como o *Redmine, Trello*, e controladores de versão, como *Github*, são fontes de dados que podem ser mineradas e interpretadas (HASSAN, 2008a). Tais repositórios de *software*, na prática, são usados apenas para armazenar e reter informações, raramente são utilizados para dar suporte ao processo de decisão (HASSAN, 2008a), pois segundo Fayyad et al. (1996), o processo de Descoberta de Conhecimento em Banco de Dados (DCBD) não é um processo trivial.

A DCBD é um processo composto de diversas etapas que envolvem a preparação dos dados, procura por padrões, avaliação e refinamento. E uma das fases do processo de DCBD é a fase de mineração de dados. Esta etapa é responsável pela aplicação dos algoritmos que são capazes de identificar e extrair padrões relevantes presentes nos dados. Na engenharia de *software*, a mineração de dados tem emergido como uma evidente área de pesquisa, oferecendo formas de interpretar a quantidade abundante de dados e, consequentemente, auxiliar na resolução de diversos problemas que envolvem o desenvolvimento de *software*

(HASSAN; XIE, 2010).

No ano de 2004, surgiu o principal evento da área de MRS, denominado *Working Conference on Mining Software Repositories* (MSRConf)[1]. Atualmente, a MSRConf é a principal conferência na área de disseminação de resultados de trabalhos sobre novos processos, métodos, técnicas, ferramentas e idéias nesse campo (HEMMATI et al., 2013). A área de pesquisa em MRS vem ganhando cada vez mais notoriedade desde que surgiu, e um dos fatores que devem estar contribuindo cada vez mais para isso, além dos benefícios para área da ES, é o surgimento de novas tecnologias ou repositórios de *software*.

O termo "repositório de software" abrange todos os artefatos produzidos durante o desenvolvimento de um sistema de *software*, desde os arquivos com o código fonte do sistema que podem estar armazenados em um sistema de controle de versão (SCV), até mensagens em listas de emails trocadas entre os desenvolvedores. Tais repositórios contêm informações valiosas, que podem ser exploradas para compreender a evolução de software e contribuir com o desenvolvimento do projeto. A Tabela 2 apresenta exemplos de alguns tipos de repositórios.

Tabela 2 – Exemplos de repositórios de *software*.

Repositório	**Descrição**
Sistemas de controle de versão	Esses repositórios registram o histórico de desenvolvimento de um projeto. Eles rastreiam todas as alterações no código-fonte junto com meta-dados sobre cada alteração, por exemplo, o nome do desenvolvedor que realizou a alteração, o tempo a mudança foi realizada e uma mensagem curta descrevendo a alteração. Esses repositórios são mais comumente disponíveis e usados em projetos de software. *Git* e *Subversion* são exemplos de repositórios de controle de versão que são muito usados na prática.
Sistemas de rastreamento de *bugs*	Esses repositórios rastreiam o histórico de descoberta e resolução de bugs ou solicitações de recursos relatados por usuários e desenvolvedores de grandes projetos de software. *Bugzilla* e *Jira* são exemplos de repositórios de *bugs*.
Listas de discussão	Esses repositórios rastreiam as discussões sobre vários aspectos de um projeto de software ao longo de sua vida útil. As listas de endereços, e-mails, chats IRC e mensagens instantâneas são exemplos de comunicações arquivadas sobre um projeto.
Logs de *Deployment* (Implantação do software)	Esses repositórios registram informações sobre a execução de uma implantação de um aplicativo de software ou implementações diferentes dos mesmos aplicativos. Por exemplo, os *logs* de implantação podem registrar as mensagens de erro relatadas por um aplicativo em vários sites de implantação.
Repositórios de projetos	Esses repositórios arquivam o código-fonte, e diversas outra informações, para um grande número de projetos. *Sourceforge.net, Google Code* e *GITHUB* são exemplos de grandes repositórios de projeto.

Fonte – Adaptada de HASSAN (2008b)

Conforme apresentado na Tabela 2, um sistema de controle de versão é uma ferramenta utilizada no cotidiano de desenvolvimento de software. Por meio dessa ferramenta,

[1] http://msrconf.org/.

um desenvolvedor controla as alterações sobre código que está modificando. Durante o trabalho, o desenvolvedor agrupa as mudanças feitas no código em commits e o sistema de controle de versão armazena esses grupos de alterações. Essa é uma fonte de dados importante no contexto de Mineração de Repositórios.

O processo de Mineração de Dados pode ser conduzido de várias maneiras distintas, de acordo com os objetivos que se deseja alcançar com a aplicação da técnica. Tais objetivos podem ser organizados em dois grandes grupos (ZAKI MOHAMMED. WONG, 2003):

i. Atingir uma capacidade preditiva confiável, ou seja, buscar responder quais fenômenos podem vir a acontecer; e

ii. Alcançar uma descrição compreensível, ou seja, identificar a razão de fenômenos já conhecidos acontecerem da forma como acontecem;

As etapas típicas de um processo de MRS são: I) Extração, II) Modelagem de Dados, III) Síntese e IV) Análise (HEMMATI ET AL., 2013). Primeiramente é realizada a Extração de dados "brutos" de vários tipos de repositórios. Na etapa de Modelagem é realizada uma preparação de dados para serem usados. Na etapa de Síntese é realizado um processamento dos dados extraídos. Finalmente, é necessária a etapa de Análise e interpretação dos resultados para formulação de conclusões. Na Figura 1 são apresentadas as etapas citadas.

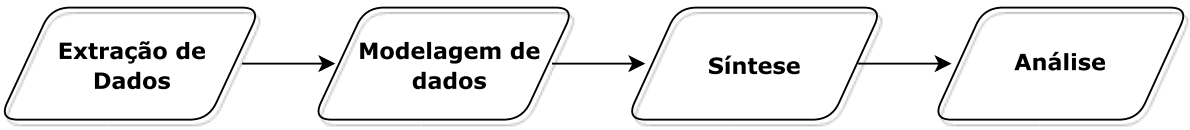


Figura 1 – Etapas típicas do processo de MRS.

Fonte – Hemmati et al. (2013).

Pesquisadores no campo da engenharia de *software* têm desenvolvido abordagens para extrair informações pertinentes e descobrir relações e tendências de repositórios no contexto da evolução do *software* (KAGDI H.; MALETIC, 2007). Hassan e Xie (HASSAN, 2008c) analisaram a evolução das pesquisas sobre a mineração de repositório de *software*, frisando a importância das informações que estão em repositórios de *software* que são extremante úteis para tomada de decisões em projetos de *software*. A mineração de repositório de *software* permite a utilização de indicadores mais precisos do que simplesmente a intuição e a experiência de desenvolvedores e gestores nas tomadas de decisão. Eles apontaram que ferramentas online e ou integradas ao ambiente de desenvolvimento de *software* estão surgindo. Ou seja, novas oportunidades estão sendo criadas para serem exploradas nesta área.

Diversos problemas podem ser solucionados por meio da utilização de técnicas baseadas em MRS, tais como: detecção de entidades de código semelhantes (KIM et al., 2005; SAGER et al., 2006), predição de *bugs* (GRAVES et al., 2000; HASSAN; HOLT, 2005; KIM et al., 2007; HASSAN, 2009; HATA; MIZUNO; KIKUNO, 2012), estimativa de esforço em desenvolvimento de software (YU, 2006; MOSER et al., 2008; ROBLES et al., 2014), análise da evolução de sistemas (GALL; LANZA, 2006; KOCH, 2007; YAN; MENARINI; GRISWOLD, 2014; CHAIKALIS et al., 2014) e reutilização de código (TANGSRIPAIROJ; SAMADZADEH, 2005; XIE; PEI, 2006). D'Ambros et al. (2008) destacam os seguintes tópicos estudados na análise de repositórios de software:

i. **Concentração do trabalho dos desenvolvedores e análise de redes sociais.** O objetivo nesse tópico é descobrir quanto e em quais pontos do software os desenvolvedores estão dedicando mais seus esforços e como eles se comunicam, para buscar soluções de problemas no processo de desenvolvimento e na estrutura da equipe.

ii. **Impacto e propagação de alterações.** Busca entender o efeito das mudanças feitas em certa parte do sistema sobre o resto do código do projeto. Compreendendo melhor o efeito dessas alterações, a equipe estima melhor os custos das tarefas. Além disso, um desenvolvedor pode ser informado de quais arquivos ele precisará checar após ter feito uma certa alteração.

iii. **Análise de tendências e hotspots.** Hotspots são pontos do software que sofrem alterações frequentemente. Encontrar tais pontos do sistema, ajuda a levantar deficiências na arquitetura do projeto e sugerir possíveis refatorações para aprimorar sua manutenibilidade.

iv. **Previsão de falhas e defeitos.** Os dados disponíveis em repositórios de software podem ser usados como entrada para algoritmos de aprendizagem de máquina, para criar modelos preditivos de possíveis falhas no sistema. Com esse modelo, a equipe toma ações preventivas para previnir falhas em versões futuras.

Com isso, é evidente a gama de benefícios que a aplicação de técnicas baseadas em MRS pode oferecer para a área de Engenharia de *Software*.

1.4 Manutenção e Evolução de *Software*

Em 1976, (LEHMAN ET AL., 1976) publicam um estudo quantitativo que traz as primeiras evidências sobre a degradação da qualidade e manutenibilidade de *software* durante seu ciclo de vida. Segundo os autores, essa degradação se intensifica após a fase de implantação, em que os usuários do sistema costumam achar erros no *software*, descobrir novos usos e solicitar novas funcionalidades. Segundo a norma (ISO/IEC-14764, 2006),

manutenção é um processo centrado na modificação de um sistema após sua entrega ao cliente. Ela ocorre porque problemas ou necessidades são identificados quando o *software* é colocado em operação.

Em um trabalho posterior Lehman define 5 leis para evolução de sistemas de *software* (LEHMAN, 1980) que mais tarde tornaram-se 8 leis (LEHMAN, 1996), comumente aceitas e referenciadas pela comunidade de engenharia de *software*, como pode ser visto em (SOMMERVILLE, 2007). Na Tabela 3 são listadas as 8 leis de Lehman.

Tabela 3 – Leis de LEHMAN

I	Mudança Contínua	Um *software* deve ser continuamente adaptado, caso contrário se torna progressivamente menos útil.
II	Complexidade crescente	À medida que um *software* é alterado, sua complexidade cresce, a menos que um trabalho seja feito para mantê-la ou diminuí-la.
III	Auto regulação	O processo de evolução de *software* é autorregulado próximo a distribuição normal com relação às medidas dos atributos de produtos e processos.
IV	Conservação da estabilidade organizacional	A não ser que mecanismos de retroalimentação tenham sido ajustados de maneira apropriada, a taxa média de atividade global efetiva num *software* em evolução tende a ser manter constante durante o tempo de vida do produto.
V	Conservação da Familiaridade	De maneira geral, a taxa de crescimento incremental e taxa crescimento a longo prazo tende a declinar.
VI	Crescimento contínuo	O conteúdo funcional de um *software* deve ser continuamente aumentado durante seu tempo de vida para manter a satisfação do usuário.
VII	Qualidade decrescente	A qualidade do *software* será entendida como declinante a menos que o software seja rigorosamente adaptado às mudanças no ambiente operacional.
VIII	Sistema de Retroalimentação	Processos de evolução de *software* são sistemas de retroalimentação em múltiplos níveis, em múltiplos laços (*loops*) e envolvendo múltiplos agentes.

Fonte – Adaptada de LEHMAN (1996)

A teoria é que todo *software* precisa ser considerado como um projeto inacabado, um sistema dinâmico e em contínua mudança e evolução. Esse conceito deve ser concebido desde o início do ciclo de vida do software, ou seja, o software precisa ser desenvolvido de forma tal que beneficie a sua manutenção e evolução (LEHMAN, 1996).

De acordo com (GRUBB ET AL., 2003), existem diversas definições de manutenção de *software* na literatura, algumas mais abrangentes e outras específicas. As encontradas com maior frequência estão associadas à correção de erros e à necessidade de adaptar sistemas quando o ambiente operacional ou requisitos originais são alterados.

As intervenções em manutenção de *software* são divididas em dois grandes grupos: Ações de Correção e Ações de Evolução do software ((LEHMAN, 1980); (SOMMERVILLE, 2007); (ISO/IEC-14764, 2006)) se relacionando com algumas características e subdivisões como pode ser visto na Figura 2.

É comum associar a execução da etapa de manutenção a uma mera atividade para reparo de erros em sistemas (APRIL, 2010). A falta de compreensão de todas as ações que envolvem esse processo leva, muitas vezes, a uma percepção de que organizações de

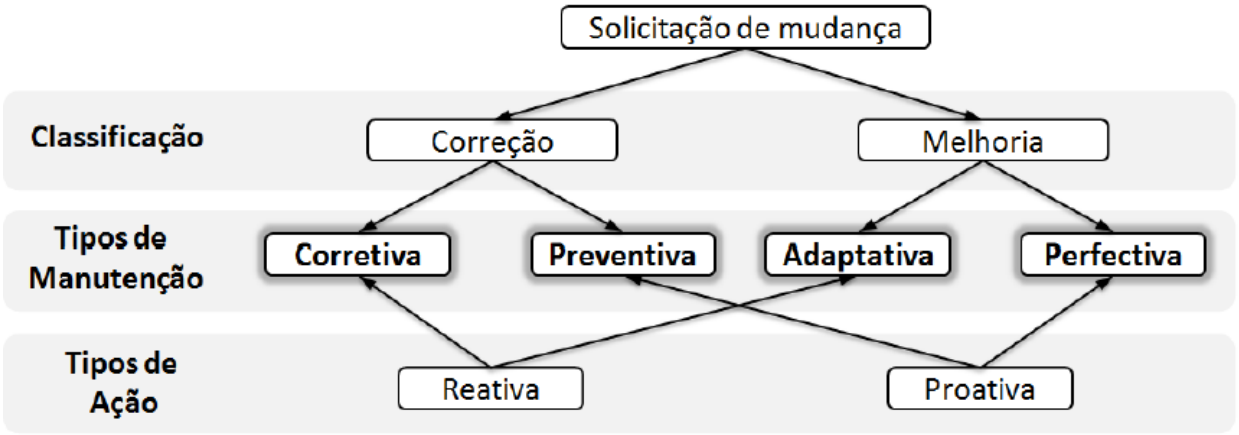


Figura 2 – Categorias de Manutenção de *Software*

Fonte – Adaptada de ISO/IEC-14764 (2006)

manutenção de *software* são caras e ineficientes. Porém, estudos mostram que grande parte do esforço despendido nesta etapa do ciclo de vida do *software* acrescenta valor para as instituições.

A manutenção de *software* pode ser classificada de diversas formas. Uma delas foi estabelecida por (TRIPATHY ET AL., 2008) e se baseia na intenção, isto é, no objetivo do desenvolvedor ao realizar tarefas específicas de manutenção no sistema. Inicialmente, ela foi desenvolvida por (SWANSON, 1976) que a dividiu em três tipos:

- Manutenção Corretiva: Tem como objetivo o reparo de defeitos nas aplicações, isto é, a correção de problemas que surgem durante a utilização do sistema. Essas falhas podem ser de processamento, comumente atribuídas a *bugs* de software, de performance ou até mesmo de implementação, como, por exemplo, violações em padrões de programação.

- Manutenção Adaptativa: É executada em resposta a modificações no ambiente externo, visando a adequação do *software* ao contexto no qual ele deve operar. Mudanças no hardware ou a necessidade de instalação de uma nova versão de um sistema operacional, por exemplo, podem requerer este tipo de manutenção.

- Manutenção Perfectiva: Visa à realização de modificações apenas no sentido de melhorar o software, como por exemplo a inclusão de novas funcionalidades, a eliminação de ineficiências no processamento, um aumento no desempenho, ou, até mesmo, um aprimoramento na própria manutenibilidade do sistema.

Posteriormente, essas três definições foram incorporadas pela norma (ISO/IEC-14764, 2006), tendo sido introduzida uma quarta categoria denominada Manutenção

Preventiva. Ela é realizada a partir de modificações após a liberação do *software* com o intuito de detectar e corrigir falhas latentes, antes que se tornem falhas operacionais.

Modelos que estabelecem o processo de manutenção de *software* apresentam diferentes focos, sendo que alguns se atentam mais a questões econômicas, outros ao produto e outros ao próprio processo. Todos possuem vantagens e desvantagens, não existindo um modelo único e aplicável às diversas situações. A melhor solução, muitas vezes, é utilizá-los em conjunto para obter o resultado que agregue mais benefícios para as organizações (YONGCHANG ET AL., 2011).

(LEHMAN, 1980) observou em estudos empíricos, no desenvolvimento de sistemas de grande porte, que os tipos de sistema podem ser sumarizados em 3 categorias :

1. **Sistemas do Tipo S**: é um sistema que possui uma especificação já definida e fixa, o mesmo não sofre modificações ao longo do tempo. Normalmente são sistemas matemáticos onde é indispensável saber todas as especificações e propriedades necessárias no início do projeto.

2. **Sistemas do Tipo P**: são aqueles cuja solução não é inicialmente aparente. Em geral estes são os sistemas que as partes interessadas sabem qual o resultado final desejado, mas não sabem como alcançá-lo. Sistemas desse tipo exigem uma abordagem interativa entre desenvolvedores e pessoas interessadas, para facilitar o entendimento e articular o que as partes interessadas precisam e o que não precisam e assim identificar o meio de desenvolvê-lo.

3. **Sistemas do Tipo E**: são sistemas muito utilizados e que estão embutidos em problemas do mundo real. Onde as especificações mudam ao longo do tempo e normalmente não tem como serem todas definidas no início do projeto. As mesmas evoluem ao longo do tempo impulsionadas pelo feedback do usuário e consequente necessidade de adaptação as mudanças do mundo real. São sistemas que uma vez que estejam em funcionamento, são avaliados pelos resultados que fornecem. Por serem mutáveis, por fatores de qualidades (ainda não definidos) e inclusão de novas funcionalidades, comumente são sistemas que geram grande preocupação. Este é o tipo de sistema alvo da aplicação da abordagem proposta nesse estudo.

Como os programas do tipo-E evoluem constantemente e novas mudanças são inclusas em todo momento, há a necessidade de muitas adequações ao mundo real e as exigências do usuário. Este modelo passa a auxiliar efetivamente no processo de desenvolvimento, assim as modificações podem ser dividas em etapas e o sistema passa a ter sucessivas releases. Um conjunto de releases, compõe as versões do sistema, estas por sua vez, representam melhorias implementadas no complexo processo de evolução em andamento. Assim o sistema passa a suportar a necessidade destas constantes mudanças,

mantendo seu processo de evolução, pois caso contrário, iria acabar em desuso pelo mundo real.

1.5 Funcionalidade de *Software*

Assim como a maioria dos desenvolvedores tem sua própria noção de funcionalidade, as definições fornecidas pelos pesquisadores também variam significativamente (APEL ET AL., 2013) (BERGER ET AL., 2015). Por exemplo, existem mais de dez definições do termo funcionalidade (CLASSEN ET AL., 2008), variando de algo abstrato - "aspecto, qualidade ou característica visível ao usuário" (KANG ET AL., 1990) - a definições muito técnicas - "um incremento do recurso do programa" (BATORY, DON., 2005). Como resultado, o conceito de funcionalidade pode ser difícil de entender, dada ausência de uma definição comum. Contamos com a definição de Apel (APEL ET AL., 2013), que consolidam as definições anteriores:

"Uma funcionalidade é um comportamento característico ou visível ao usuário final de um sistema de *software*. As funcionalidades são usadas na Linha de Produto de (LPS) para especificar e comunicar semelhanças e diferenças dos produtos entre as partes interessadas e para orientar a estrutura, a reutilização e a variação em todas as fases do ciclo de vida do *Software*. "

Nesta definição, as funcionalidades são principalmente meios para comunicar características visíveis ou comportamentos observáveis de um sistema. Outra definição sobre funcionalidade que destacamos é a proposta por EISENBARTH; KOSCHKE; SIMON (2003). Segundo eles uma funcionalidade pode ser entendida como um requisito funcional de um sistema, ou ainda um comportamento observável do sistema que pode ser disparado pelo usuário. Além disso, como apresentado em (COCKBURN, 2000), uma funcionalidade pode ser classificada em três níveis:

I. ***summary*** (agregador): As funcionalidades de nível *agregador* englobam várias tarefas ou *features* de nível *usuário*.

II. ***user*** (usuário): Representam uma única tarefa a ser executada pelo usuário (comportamento observável), de forma que este obtenha um resultado que o atenda.

III. ***subfunctions*** (subfunção): As funcionalidades de nível *subfunção* representam subtarefas executadas dentro de uma funcionalidade de nível *usuário*. É importante frisar que essas *features* podem representar um comportamento observável (tarefas executadas diretamente pelo usuário) do sistema, como por exemplo: **realizar *login***. Além disso, elas podem representar tarefas indiretamente executas pelo usuário, isto é, não representam um comportamento observável do sistema, como por exemplo: **validar *login***.

Para diferentes contextos, diferentes tipos de características das funcionalidades são relevantes, levando a diferentes características de como os seus componentes (como arquivos) se relacionam (KRüGER ET AL., 2018). No contexto da Linha de Produto de Software (LPS), o foco é principalmente no gerenciamento de recursos opcionais e menos em recursos obrigatórios (ou seja, comuns a todas as variantes). Os locais de tais recursos opcionais são tipicamente documentados usando anotações de preprocessador (APEL ET AL., 2013; MEDEIROS ET AL., 2015), que permite ativar e desativar recursos ao derivar variantes individuais da LPS. Em contraste ao que é relevante para outras comunidades, como a manutenção de *software*, em que são importantes qualquer característica visível ou comportamento observável de um sistema e mesmo que não seja necessariamente configurável.

A manutenção e evolução de *software* envolve o incremento de novas funcionalidades aos programas, melhorando as funcionalidades existentes e removendo *bugs*, que é análogo à remoção de funcionalidades indesejadas. Nenhuma atividade de manutenção pode ser concluída sem primeiro localizar o código que é relevante para a tarefa em questão, tornando a localização característica essencial para manutenção de *software*, uma vez que é realizada no contexto de mudança incremental (DIT et al., 2013). Ou seja é importante que desenvolvedores tenham a informação sobre componentes computacionais que compõem uma determinada funcionalidade para que eles possam realizar as tarefas de evolução de *software*, como depuração, compreensão, e reutilização (HILL et al., 2013). Esses componentes podem ter diferentes granularidades, tais como método/função, *statement* (bloco de instruções) ou arquivo, por exemplo. A atividade para identificação de partes do código fonte que implementam uma funcionalidade no sistema é denominada *Feature Location* (FL)(BIGGERSTAFF; MITBANDER; WEBSTER, 1993; RAJLICH; WILDE, 2002).

1.6 Considerações Finais

Neste capítulo foram apresentados os principais conceitos sobre técnicas e componentes utilizados no trabalho. Na Seção 1.1 foram apresentados alguns conceitos básicos sobre DT's. Nas seções 1.3, 1.4 e 1.5 foram apresentados os principais conceitos relacionados à área de pesquisa que envolve mineração, a definição e classificação de manutenção e a descrição do conceito de funcionalidade.

2 Trabalhos Relacionados

Um dos objetivos deste trabalho é propor um método para identificação e visualização de *code smells* sobre uma nova perspectiva: funcionalidade de *software*. Essa nova visão de *code smells* busca facilitar as decisões sobre seu gerenciamento. Nas pesquisas realizadas na literatura não foram identificados trabalhos que utilizam a visão de funcionalidade para atingir esse objetivo. Por conta disso, é apresentado a seguir, alguns trabalhos e ferramentas que foram consideradas na elaboração da nossa abordagem, mas que não possuem o mesmo objetivo do trabalho aqui proposto.

Tom e demais autores realizaram dois estudos. No primeiro (TOM ET AL., 2012) eles apresentaram uma revisão sistemática, com a intenção de fornecer uma compreensão consolidada do fenômeno da DT. Suas questões de pesquisa eram: Quais são os elementos da dívida técnica? Por que a dívida técnica surge? E quais são os benefícios e desvantagens de permitir que a dívida técnica acumule? Eles procuraram responder essas perguntas por meio de um arcabouço teórico e discutir os resultados positivos e negativos de DT. Segundo os autores, o arcabouço teórico resultante retrata uma visão holística da DT que incorpora um conjunto de precedentes e resultados, bem como o próprio fenômeno (comportamentos, metáforas e elementos). Em seu segundo estudo (TOM ET AL., 2013), eles ampliaram o trabalho anterior, realizando um estudo de caso exploratório que envolveu uma revisão na literatura, usando textos acessíveis, como blogs na internet, livros e artigos de revistas especializadas, complementados por entrevistas com profissionais de software e acadêmicos para estabelecer os limites do fenômeno da DT. O objetivo desta pesquisa foi consolidar a compreensão da natureza da dívida técnica e suas implicações para o desenvolvimento de software, estabelecendo assim as fronteiras do fenômeno e um arcabouço teórico mais completo para facilitar futuras pesquisas. Os resultados deste estudo incluíram a criação de um quadro teórico útil, consistindo em um conjunto de dimensões, atributos, precedentes e resultados de DT, bem como o fenômeno em si e uma taxonomia que descreve e engloba diferentes formas de DT. No entanto, não fornece uma taxonomia abrangente nem dos tipos de DT, nem dos indicadores que podem ser utilizados para apoiar a identificação de diferentes tipos, que é comumente aceito e amplamente utilizado pela comunidade de pesquisadores.

Palomba e demais autores (PALOMBA G. BAVOTA; LUCIA, 2014) investigaram quais *code smells* são considerados pelos desenvolvedores como os mais prejudiciais. Os desenvolvedores receberam trechos de código de três sistemas com doze tipos de *code smells* e foram solicitados a classificar a severidade dos *smells*. Os resultados sugerem que existem algumas categorias de *code smells* que: i) são altamente percebidos e identificados pelos desenvolvedores; ii) criam mais preocupações para os desenvolvedores com mais

experiência e conhecimento do sistema; e iii) são classificados com valores de severidade altos. Os *code smells* com tais propriedades são *complex class*, *god class*, *long method* e *spaghetti code*. Nesse mesmo estudo foi constatado que existe um grupo de *code smells* para os quais a percepção de severidade varia caso a caso. Esses *smells* são *feature envy*, *refused bequest* e *speculative generality*.

(GUO AND SEAMAN., 2011) propõem um modelo para o cálculo de dívida técnica, com base na análise de dados históricos de defeitos em aplicações. A abordagem é denominada Gerenciamento de Portfólios. Usada na gestão financeira, consiste na analogia a uma carteira de investimentos (montante da dívida), onde o gestor decide onde aplicará fundos (correções). O investidor faz a aplicação analisando-se a variância (risco) em relação ao montante. Os autores procuraram traduzir o modelo em um *framework*, em que um dos componentes principais é denominado lista de dívida técnica. A lista contém alguns itens que podem refletir em dívida durante o desenvolvimento, tais quais itens que precisam ser modificados para atender o projeto. A descrição dessa lista contém o autor da violação, a data em que a mesma foi identificada, a estimativa de esforço para correção e à razão pela qual é considerada dívida. Segundo os autores, o interesse na dívida pode ser estimado através da relação entre esforços históricos registrados, dados de uso, registro de mudanças, e dados sobre defeitos. Como esta técnica é baseada na análise dos portfólios no mundo financeiro, acabou herdando os mesmos problemas e falhas da abordagem original, segundo os próprios autores as duas não representavam um ambiente real. Os autores ainda relatam que é difícil discutir resultados quanto à eficácia do modelo, por não ter sido feito nenhum estudo de caso sobre o mesmo.

Brown et al. (BROWN Y. CAI, 2010) realizaram um estudo a respeito de uma modelagem para pagamento da DT baseada na análise de impactos em longo prazo. Os resultados apontam que há divergência quanto às intenções de investimento de qualidade por parte de desenvolvedores, engenheiros e gerentes, que acabam não investindo em gerência da DT por não identificarem resultados em curto prazo.

Klinger e demais autores (KLINGER, 2011) realizaram um estudo sobre a premissa de que dívida técnica deve ser usada como uma ferramenta para conhecimento de benefícios não descobertos e que surge das más escolhas de desenvolvimento por parte de arquitetos e desenvolvedores. Foram feitas entrevistas com 4 arquitetos de *software* da IBM com diferentes níveis de experiência, com o objetivo de obter respostas sobre como os *stakeholders* influenciaram na dívida, e quais as estratégias para gerenciá-la e quantificá-la. Juntando-se as respostas dos entrevistados, conclui-se que: i) calcular a dívida, quando induzida por questões gerenciais é um desafio, havendo necessidade, em alguns casos, de se adquiri-la em razão de aceitação e sucesso; ii) as decisões são tomadas de maneira *ad-hoc*, ou seja, não há processos formais de notificar a dívida, tornando-a mais invisível ainda; iii) há falhas em comunicar a *stakeholders* a dívida, bem como faltam-lhes compreensão, pois os

mesmos priorizam funcionalidade à qualidade.

AnaConDebt é uma ferramenta que foi projetada para auxiliar na avaliação e na comunicação do juros da dívida técnica entre desenvolvedores, arquitetos e gerentes (MARTINI; BOSCH., 2017). Essa ferramenta possui uma interface web com a função de receber valores atribuídos pelos usuários em relação ao impacto negativo das dívidas técnicas, ou seja, estimativas de fatores vinculados ao juros das dívidas. A partir desses, a ferramenta calcula e atribui a cada item uma pontuação de gravidade entre 1 e 10, sendo que valores próximos a 10 representam as dívidas com um maior impacto negativo aos critérios analisados. Por fim, a AnaConDebt mostra as dívidas classificadas pelo seu valor de pontuação, auxiliando os desenvolvedores na tomada de decisão referente à priorização e à correção das mesmas.

A ferramenta *DebtFlag* foi criada com o objetivo de capturar, rastrear e resolver a dívida técnica em projetos de software. Essa ferramenta integra-se ao ambiente de desenvolvimento e fornece aos desenvolvedores vincular as dívidas às partes correspondentes na implementação através de árvores de dependências (HOLVITIE; LEPPäNEN., 2013). Primeiramente, uma lista é preenchida com itens de DT, que correspondem a ocorrências atômicas únicas de dívida no software (HOLVITIE; LEPPäNEN., 2013). Cada item mantém um conjunto de informações atreladas, como descrições contendo seu tipo, sua localização e o raciocínio vinculado a sua aquisição. Além disso, conta com estimativas do principal e da probabilidade e quantidade de juros esperados. No mecanismo *DebtFlag*, uma lista, com itens de DT, pode consistir em um único ou vários elementos *DebtFlag*. Um elemento *DebtFlag* é um link entre uma observação de dívida técnica e uma implementação correspondente, como por exemplo, um pacote, uma classe ou um método (HOLVITIE; LEPPäNEN., 2013).

A primeira implementação do mecanismo *DebtFlag* foi projetada para suportar a linguagem de programação Java. A ferramenta *DebtFlag* é um sistema de duas partições que consiste em um *plug-in* para o Ambiente de Desenvolvimento Integrado (IDE) do Eclipse e um aplicativo web. O *plug-in DebtFlag* é responsável por captar a DT através do ambiente de desenvolvimento, rastreando sua propagação e apoiando seu gerenciamento. O aplicativo web fornece uma apresentação dinâmica da lista de DT compilada a partir das informações produzidas com os *plug-ins DebtFlag* (HOLVITIE; LEPPäNEN., 2013).

Segundo seus autores, os principais benefícios do uso da ferramenta são:

- Captura observações feitas pelo desenvolvedor: a ferramenta permite fornecer aos desenvolvedores um raciocínio adicional a suas observações, garantindo que as informações sobre a DT sejam precisas e bem definidas.
- A ferramenta documenta a estrutura da DT: as implementações de software são estruturas complexas, hierárquicas e interconectadas. O mecanismo *DebtFlag* capta

a DT como itens. As listas com itens de DT são formadas como um conjunto de elementos *DebtFlag* para os quais o mecanismo *DebtFlag* resolve automaticamente os caminhos de propagação. Essa forma estruturada permite acompanhar a DT durante um desenvolvimento contínuo, além de propiciar a aplicação de várias abordagens de avaliação aos diferentes níveis da hierarquia adquirida.

- Apresenta a DT ao nível de implementação: ao projetar todas as observações sobre a DT ao nível de implementação, o mecanismo *DebtFlag* garante que o desenvolvimento seja conduzido enquanto estiver ciente da presença da dívida técnica. Isso permite que os desenvolvedores evitem aumentar involuntariamente o valor da DT por meio de dependências para as áreas afetadas ou reduzam seu valor, resolvendo-a em áreas onde o desenvolvimento é atualmente conduzido.

O *JDeodorant*[1], é um *plug-in* do Eclipse que emprega uma variedade de novos métodos e técnicas, afim de identificar *code smells* em programas Java e sugerir refatorações apropriadas que os resolvam (TSANTALIS, 2010). Além disso, a ferramenta faz uma pré avaliação do efeito na qualidade do projeto de todas as sugestões de refatoração, ajudando o usuário a determinar a sequência mais eficaz de aplicações refatoração.

Para controlar o número e a qualidade das oportunidades de refatoração relatadas, o *JDeodorant* fornece uma página de preferências onde o usuário pode definir vários valores de limite. Por exemplo, o número mínimo de declarações que um método deve possuir para ser examinado.

O *JDeodorant* emprega principalmente *APIs* que pertencem ao núcleo do Eclipse *Java Development Tools* (JDT). O *plug-in* usa a classe *ASTParser* para analisar os relacionamentos entre entidades do sistema e aplicar refatorações no código-fonte. Além disso, emprega a classe *ASTRewrite* para aplicar as refatorações e fornecer a funcionalidade desfazer. Como tal, *JDeodorant* depende fortemente da noção de uma árvore de sintaxe abstrata (*abstract syntax tree* - AST). As informações provenientes da AST podem ser reutilizadas em diversas ocasiões para fins diferentes. Conseqüentemente, a arquitetura do *JDeodorant* suporta a reutilização dessas informações sem armazená-las permanentemente na memória. Isso é obtido fornecendo uma representação intermediária dos elementos Java necessários e um mecanismo que permite a recuperação de nós AST de maneira rápida e eficiente.

Atualmente, a ferramenta pode detectar cinco tipos de cheiros de código, ou seja, *Duplicated Code, Feature Envy, God Class, Long Method* e *Type Checking,*. Além disso, ele determina imediatamente possíveis refatorações e as apresenta ao usuário, que pode decidir deixar o *JDeodorant* aplicá-las ou não. As *God Classes* são resolvidas através de refatorações apropriadas da *Extract Class*. *Feature Envy* são corrigidos sugerindo

[1] https://marketplace.eclipse.org/content/jdeodorant

refatorações apropriadas do *Move Method*. Os *Type Checking* podem ser corrigidos através da substituição de uma condicional por polimorfismo. Problemas de *Duplicated Code* são resolvidos pelas refatorações apropriadas do *Extract Clone* . Finalmente, os *Long Methods* são resolvidos sugerindo refatorações apropriadas do *Extract Method*.

Outra ferramenta que se destaca é a TEDMA, que é uma ferramenta de código aberto utilizada para auxiliar na gestão da dívida técnica por meio da execução de métricas de gerenciamento, levando em conta a evolução histórica do software (FERNáNDEZ-SáNCHEZ ET AL., 2017).

Segundo seus autores, a TEDMA facilita a integração de novas ferramentas e foi projetada para suportar diferentes implementações de modelos de gerenciamento de dívida técnica. As informações geradas são armazenadas em bancos de dados que podem ser explorados por ferramentas externas, incluindo o acesso à informações específicas usando linguagens de consulta. Os dados são organizados segundo a evolução do software, portanto, a TEDMA facilita a implementação de algoritmos eficientes para percorrer o histórico de arquivos e revisões.

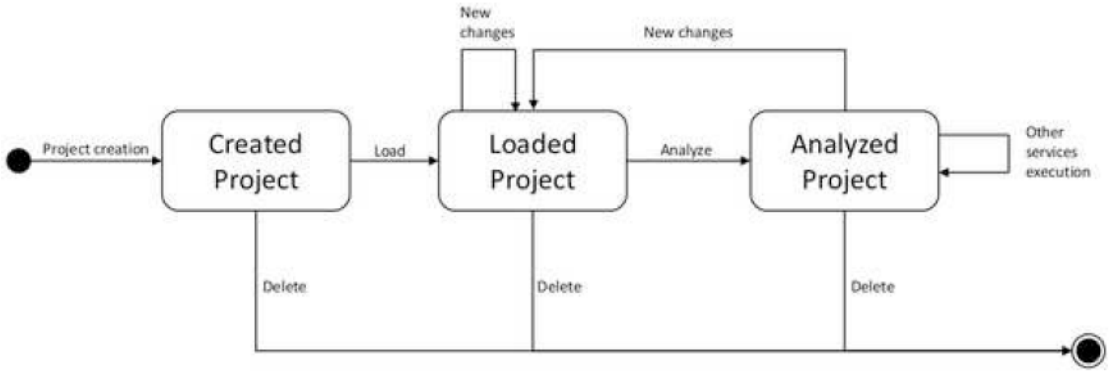


Figura 3 – Ciclo de vida de um projeto na ferramenta TEDMA

Fonte – (FERNáNDEZ-SáNCHEZ ET AL., 2017)

Conforme Figura 3, podemos ver como é o ciclo de vida de um projeto na TEDMA. Inicialmente um projeto deve ser adicionado à ferramenta informando um nome, uma descrição e a localização do repositório do código-fonte (local ou remoto). O próximo passo é carregar os dados do projeto no banco de dados da TEDMA. A partir dessas informações básicas, o software está apto a ser analisado, com isso, quando novas alterações são carregadas no repositório, as mesmas podem ser analisadas na ferramenta. Além disso, a qualquer momento um projeto pode ser retirado da TEDMA para liberar recursos (FERNáNDEZ-SáNCHEZ ET AL., 2017).

As informações são armazenadas na TEDMA para cada revisão do software e para cada arquivo várias métricas são guardadas, como por exemplo, métricas referentes ao tamanho do arquivo em bytes, número de linhas e, se o arquivo foi modificado, o tipo e o

tamanho da alteração (FERNáNDEZ-SáNCHEZ ET AL., 2017). Para Fernández-Sánchez et al. (2017) a TEDMA é uma ferramenta útil, pois permite que as organizações gerenciem a dívida técnica em seus projetos de software com uma perspectiva de evolução, permitindo a integração de ferramentas de terceiros e a seleção de métricas desejadas.

O *SonarQube*[2] é uma das ferramentas mais conhecidas e utilizadas na indústria de *software* para realizar inspeção de qualidade de código (CAMPBELL; PAPAPETROU, 2013). É uma plataforma aberta para gerenciamento da qualidade do código. A qualidade do *software* é avaliada de acordo com sete eixos: comentários, potencial erros, complexidade, testes de unidade, duplicações, regras de codificação, arquitetura e projeto (S.A., 2019).

A plataforma Sonar utiliza o *Technical Debt Plugin* para o cálculo da DT do projeto. Primeiramente, são calculados os valores de seis eixos básicos: duplicação, violação, complexidade, cobertura, documentação e projeto. Em seguida, essas métricas são somadas para fornecer uma métrica global. Os detalhes da composição dessa métrica são descritos abaixo:

- Proporção da dívida: fornece um percentual da DT atual em relação ao total da dívida possível para o projeto.
- O custo para pagar as DT's: determina o custo, em valor monetário, necessário para limpar todos os defeitos em cada eixo.
- O trabalho para pagar as DT's: fornece o esforço para pagar as DT's expresso utilizando a métrica homens/dia.
- A repartição: apresenta um gráfico com a distribuição do débito nos seis eixos de qualidade.

(MENDES et al., 2017) desenvolveram uma ferramenta denominada *RepositoryMiner*[3]. A mesma é um software livre e foi desenvolvida com a ideia de ser extensível. Por ser livre, qualquer um é autorizado a examinar, modificar e redistribuir o projeto. O projeto foi desenvolvido usando a linguagem de programação Java. E é composto por vários módulos, sendo um principal que funciona como base para todo o software. Por ter sido pensado com foco no reuso, extensão e simplicidade, a ferramenta pode ser a ampliada para suportar outros Sistemas de Controle de Versão (SCV), outras linguagens de programação e permitir a criação de novas funcionalidades. Esta ferramenta é capaz de extrair dados de um repositório de codigo fonte, calcular métricas de software e armazená-las em um sistema de banco de dados local.

[2] https://docs.sonarqube.org/
[3] https://github.com/visminer/repositoryminer

O principal objetivo do *RepositoryMiner* é realizar a análise de repositórios de software, coletando e combinando variados dados relativos à sua evolução. Ele permite a extração e processamento de informações relevantes para dar suporte a compreensão do software em diferentes perspectivas. Através do *RepositoryMiner* o usuário pode: (i) extrair e analisar métricas variadas de software; (ii) detectar e acompanhar a evolução de code smells; (iii) detectar código duplicado; (iv) detectar problemas de estilo no código; (v) detectar possíveis bugs; (vi) identificar comentários que contenham algum indício da existência de DT; e (vii) identificar itens de DT durante a evolução do software.

O *RepositoryMiner* não possui interface gráfica pela qual as suas funções possam ser acessadas. A ferramenta fornece uma API para que as suas funções possam ser acessadas através de código Java por outros projetos.

As funcionalidades disponibilizadas pela ferramenta são:

- Cálculo de 20 métricas de código: AMW, ATFD, CYCLO, LOC, LVAR, MAXNESTING,MLOC, NOA, NOAM, NOAV, NOM, NOPA, NProtM, PAR, TCC,WMC, WOC, BOvR, BUR e DIT;
- Detecção de code smells: Brain Class, Brain Method, Conditional Complexity, Data Class, God Class, Long Method, Refused Parent Bequest e Duplicated Code;
- Detecção de 2 tipos de Dívida Técnica: Code Debt e Design Debt;
- Detecção de possíveis falhas em programas Java;
- Detecção de problemas no estilo de codificação em programas Java;
- Detecção de possíveis itens de DT baseado em comentários em programas Java;
- Mineração de repositórios locais;
- Mineração de repositórios remotos;
- Exportação da análise da qualidade do código para XML;

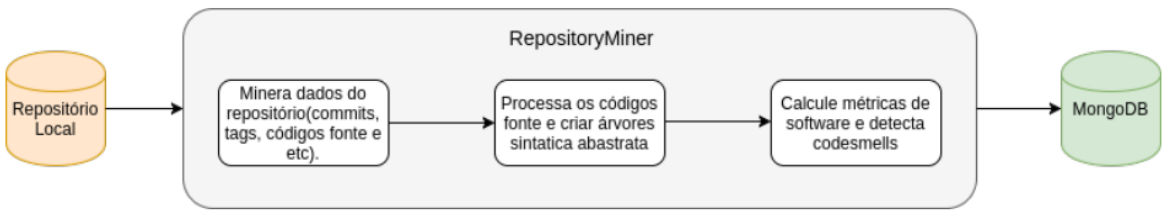


Figura 4 – Arquitetura do RepositoryMiner

Fonte – (MENDES et al., 2017)

A Figura 4 descreve os passos da ferramenta, onde no primeiro passo do ciclo de execução do *RepositoryMiner*, ele utiliza a *API JGit* para ler o código fonte do projeto do GIT local. Para isto ser possível, é necessário que seja efetuado, inicialmente, o clone do projeto para a máquina local. Esta abordagem permite uma análise das informações do repositório de maneira rápida e concisa, já que o repositório estará armazenado localmente. Os principais dados extraídos do repositório *GIT* são: código fonte, *commits, committers, branches e tags*.

A segunda etapa do processo recebe todo o código fonte como entrada e, usando a *API JDT (Java Development Tools)*, cria uma árvore sintática abstrata a partir do código fonte Java. A árvore sintática abstrata criada no *RepositoryMiner* possui uma estrutura flexível para que seja possível a adição de novos parsers para a mineração de projetos em outras linguagens de programação além da linguagem Java.

A terceira fase é onde está concentrado o maior processamento e tem a árvore sintática como sua entrada. Nesta fase, são calculadas as métricas de software, sendo elas a base para identificar os diferentes tipos de indicadores da DT, como os code smells, comparando-as com valores limites definidos no *RepositoryMiner*. Como cada projeto de software possui tamanho, complexidade e objetivos diferentes, o *RepositoryMiner* permite que os limiares sejam customizados para cada repositório. Os code smells e os valores limites padrões são definidos da seguinte maneira:

- **Brain Class**: (((A classe tem mais que 1 Brain Method) e LOC >= 197) ou ((A classe tem 1 Brain Method) e LOC >= 394 e WMC >= 94)) e (WMC >= 47 e TCC < 0.5);
- **Brain Method**: MLOC >= 20 e CYCLO >= 7 e MAXNESTING >= 5 e NOAV >= 5;
- **Conditional Complexity**: CYCLO > 7 ;
- **Data Class**: (WOC < 0.33) e (((NOPA + NOAM) > 8 e WMC < 47) ou ((NOPA + NOAM) > 16 e WMC < 94));
- **God Class**: ATFD > 5 e WMC > 46 e TCC < 0.33;
- **Long Method**: MLOC >= 20 ;
- **Depth of Inheritance**: DIT >= 6;
- **Refused Parent Bequest**: ((NProtM > 3 e BUR < 0.33) —— BOvR < 0.33) e ((AMW > 0.33 ou WMC > 47) e NOM > 5);
- **Duplicated Code**: Mais que 99 símbolos da linguagem duplicados

Ao final, as informações coletadas são armazenadas em um banco de dados não relacional.

Por ser um processo custoso e que pode levar tempo, a mineração apresenta o caráter incremental. Sendo assim, após o processamento inicial, somente aquilo que não tenha sido processado previamente será processado. Dessa forma, diminuindo bastante o tempo de execução de processamentos futuros e encorajando a análise periódica do repositório.

Assim como as ferramentas apresentadas acima, existem diversas outras que visam de forma automatizada auxiliar nas atividades de gestão da dívida técnica. Essas ferramentas exploram diferentes técnicas para detectar DT: algumas são baseadas em métricas (*CodeVizard*[4], *MARPLE* (ARCELLI C. TOSI; MAGGIONI., 2008), *VisminerTD* (MENDES et al., 2015)), enquanto outras são *plugins* integrados a uma ferramenta do ambiente de desenvolvimento (*FindBugs*[5]) outras usam análise estática do código para identificar oportunidades de refatoração (*CodeScene*[6]). Porém todas elas identificam e priorizam as DT's em nível de arquivo (classe) o que pode tornar difícil à negociação com os stakeholders sobre o pagamento das mesmas.

Nosso método se difere das demais ferramentas por propor a identificação de dívidas técnicas por funcionalidade e não por arquivo. Esse método tem como objetivo auxiliar na comunicação das dívidas aos envolvidos no processo.

2.1 Considerações Finais

Este capítulo apresentou os principais estudos identificados, assim como ferramentas, com o objetivo de identificar, analisar e sumarizar o estado da arte sobre estudos que apresentam técnicas de identificação e visualização de dívidas técnicas considerando diferentes perspectivas, e fornecer uma visão geral dessa linha de pesquisa, identificando os principais tópicos abordados por trabalhos anteriores e destacando lacunas passíveis de investigação.

Pôde-se perceber que os estudos visam gerenciar as dívidas técnicas considerando a identificação e visualização em nível de arquivo(classes e métodos). Dentre eles, foi apresentado em mais detalhes algumas ferramentas que fazem uso dessa perspectiva de arquivo.

[4] http://www.nicozazworka.com/tool-development/
[5] http://findbugs.sourceforge. net/
[6] https://codescene.io/

3 Um Método para Gestão de Code Smells em Projetos de Manutenção de Software

Dado um projeto que contém *code smells*, o método proposto neste trabalho visa identificar e visualizar os mesmos em nível de funcionalidade, facilitando assim a gestão dos indícios de DT durante essas atividades.

Para isso ser possível, são duas contribuições deste trabalho, as seguintes: i) uma nova forma de visualizar *code smells*, utilizando a perspectiva de funcionalidade; e ii) uma abordagem para projetos de manutenção/evolução de software que utiliza essa visão para tentar influenciar na gestão de *code smells*.

Inicialmente, sera apresentado a metodologia utilizada para permitir a visualização de indícios de DT por funcionalidade. Em seguida, será apresentado a abordagem para projetos de manutenção de software que utiliza a visualização proposta.

3.1 Visualização de *code smells* por Funcionalidade

Para a visualização de *code smells* por funcionalidade é necessário realizar diversas atividades. Isso é realizado em 3 etapas, sendo inicialmente feita a extração de dados a partir de um sistema de controle de versão. Em seguida, são identificadas as funcionalidades que pertencem a um *software*. Com as funcionalidades e suas respectivas classes identificadas, é realizada a etapa de cálculo do total de *code smells* presentes nas funcionalidades. Por fim, os valores obtidos na etapa anterior são utilizados para gerar uma lista ordenada das funcionalidades para correção dos *code smells*. Na Figura 5 são apresentadas as etapas ligadas à construção dessa visualização.

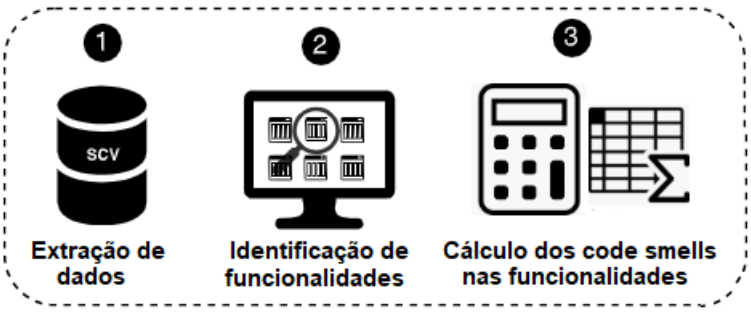


Figura 5 – Etapas para visualização de indícios de DT em funcionalidades de um software.

Fonte – Autor.

3.1.1 1ª Etapa - Extração de Dados

O objetivo desta etapa é realizar a extração de dados a partir de um Sistema de Controle de Versão (SCV). Os arquivos podem ser obtidos dos principais tipos de sistemas de versionamento: Git ou SVN. Os dados extraídos são: a) Histórico de *commits* realizados; b) *Diffs* (apresenta a diferença entre duas versões de um mesmo arquivo) de arquivos alterados e c) Arquivos de código fonte do sistema. Para a realização desta etapa de extração de dados, são utilizados os módulos já existentes na ferramenta *CoDiVision* (LIRA, 2016). A ferramenta utiliza-se das API's JGit[1] e SVNKit[2] para extração de repositórios Git e SVN, respectivamente.

3.1.2 2ª Etapa - Identificação de Funcionalidades

Nesta etapa serão identificadas as funcionalidades presentes no *software* a ser analisado. Será utilizada a abordagem proposta por Vanderson et al. (VANDERSON ET AL., 2018). A abordagem é composta por 4 etapas que são descritas brevemente a seguir.

1. **Extração de informações dos arquivos de código**: são extraídos informações a partir de arquivos de código fonte Java que foram extraidos na etapa 3.1.1. Isso é feito processando cada classe por um *parser* desenvolvido por meio da API Eclipse JDT. Esse *parser* mapeia o código fonte Java em uma *Abstract Syntax Tree* (AST). As informações são referentes a todos os dados contidos em um arquivo de código, como por exemplo, variáveis e métodos (incluindo seus parâmetros e tipo de retorno).

2. **Determinação do relacionamento entre arquivos**: utilizando as informações extraídas durante a subetapa anterior é identificado o relacionamento entre arquivos, isto é, determina-se para cada arquivo a, o conjunto de outros arquivos do sistema referenciados por a.

3. **Identificação de arquivos controladores**: identifica arquivos ou classes principais do sistema, denominados aqui de controladores, isto é, aqueles responsáveis por iniciar a execução de funcionalidades do sistema. Para isso, é criado um grafo de relacionamento entre arquivos para então identificar, com base em referências realizadas, quais são arquivos controladores.

4. **Identificação de métodos principais em arquivos controladores**: devem ser identificados os métodos ou funções principais de cada arquivo controlador. Os métodos ou funções principais de um arquivo controlador representam os pontos de início de execução das funcionalidades do sistema. Um método principal é identificado

[1] <http://www.eclipse.org/jgit/>.
[2] <https://svnkit.com/>.

por meio de sua acessibilidade (modificador de acesso), número de chamadas à outros métodos e chamadas (invocações) recebidas por outros métodos do mesmo arquivo.

Como resultado destas 4 subetapas para identificação de funcionalidades de *software*, é gerado um conjunto definido por $F = \{f_1, f_2, ..., f_n\}$, onde n é o número de funcionalidades identificadas e f_i representa a i-ésima funcionalidade. Para cada funcionalidade existe uma lista de arquivos (classes) que a integram.

3.1.3 3ª Etapa - cálculo do total de dívidas técnicas

Com as funcionalidades e os arquivos que as compõem devidamente identificadas, é necessário a identificação e contabilização dos *code smells* presentes nas mesmas. Para isso, deve-se usar uma ferramenta para identificação de *code smells* em cada classe relacionada a uma funcionalidade, para então exibir os *code smells* agrupados por funcionalidade. Neste trabalho foi utilizada a *API RepositoryMiner* (MENDES et al., 2017), que foi integrada a ferramenta *CoDiVision* (MOURA et al., 2016).

A *RepositoryMiner* detecta 7 tipos de *code smells* em projetos desenvolvidos na linguagem Java. Na Tabela 1 da Seção 1.2 pode-se ver quais os *code smells* e suas respectivas descrições. Esses *code smells* são calculadas a partir de métricas de *software* extraídos do código fonte do projeto e identificadas de forma estática, sem a necessidade de executar o código.

Na Tabela 4 coluna **A** podemos observar os arquivos da funcionalidade **Matricula Componente** que foram identificados na etapa 2, descrita na Seção 2, e na coluna **B** o total de *code smells* de cada arquivo da funcionalidade que foram encontrados usando a *RepositoryMiner*. Com base nesses dados, podemos realizar o somatório dos *code smells* de todos os arquivos e teremos o Total de code smells da funcionalidade, que nesse exemplo é 11 (ou seja, a funcionalidade **Matricula Componente** possui 11 *code smells*). Essa mesma operação é feita para todas as funcionalidades e seus arquivos.

Tabela 4 – Cálculo do total de *code smells* da Funcionalidade Matrícula Componente.

Classes	***code smells***
GenericDAO	0
ComponenteCurricular	3
ComponenteDetalhes	0
DiscenteAdapter	0
DocenteTurma	4
MatriculaComponenteMBean	0
SituacaoTurma	0
TipoComponenteCurricular	0
Turma	2
Curriculo	1
Discente	1

Com os valores dos code smells calculados para cada funcionalidade é feito a

ordenação de forma decrescente, de acordo com o número de *code smells* presentes nas mesmas.

3.1.4 Integração da Visualização na Ferramenta *CoDiVision*

A partir da definição de como obter o cálculo dos *code smells* por funcionalidade definido anteriormente, foi desenvolvido um módulo computacional, denominado TDVision[3] e que está incorporado à ferramenta CoDiVision (LIRA, 2016). Ele tem como objetivo identificar/visualizar *code smells* nas funcionalidades de um projeto de *software*, e assim direcionar os desenvolvedores no momento das correções dos mesmos.

A TDVision realiza o processo de extração dos dados de repositórios, bem como das métricas para cálculo dos *code smells* de forma automatizada. Conforme explicado na Seção 3.1.3, ela utiliza a *API* jGit[4] para extrair informações dos repositórios de *software*. Para obter os dados sobre as métricas de códigos, *code smells*, ela utiliza-se da *API* RepositoryMiner[5].

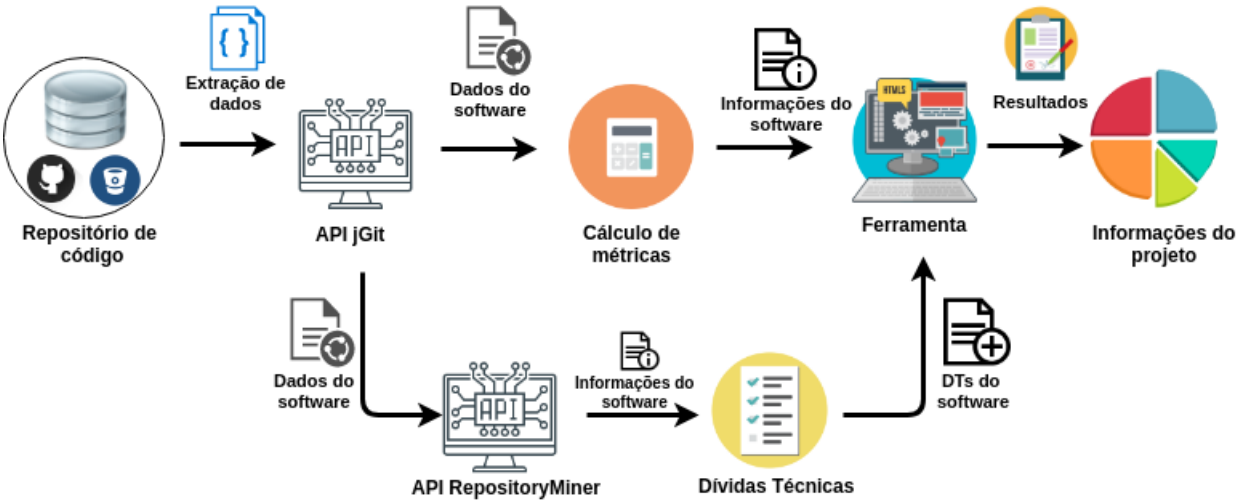


Figura 6 – Arquitetura simplificada do TDVision.

Na Figura 6 é apresentada a arquitetura simplificada do TDVision. Ela utiliza de serviços existentes da CoDiVision para obter informações do projeto. Após isso, a *API RepositoryMiner* realiza o processo de identificação de *code smells*. Atualmente, são extraídos *code smells* de códigos desenvolvidos na linguagem Java.

[3] http://easii.ufpi.br/codivision
[4] https://www.eclipse.org/jgit
[5] https://github.com/visminer/repositoryminer

Figura 7 – Funcionalidades exibidas na TDVision.

Fonte – Autor.

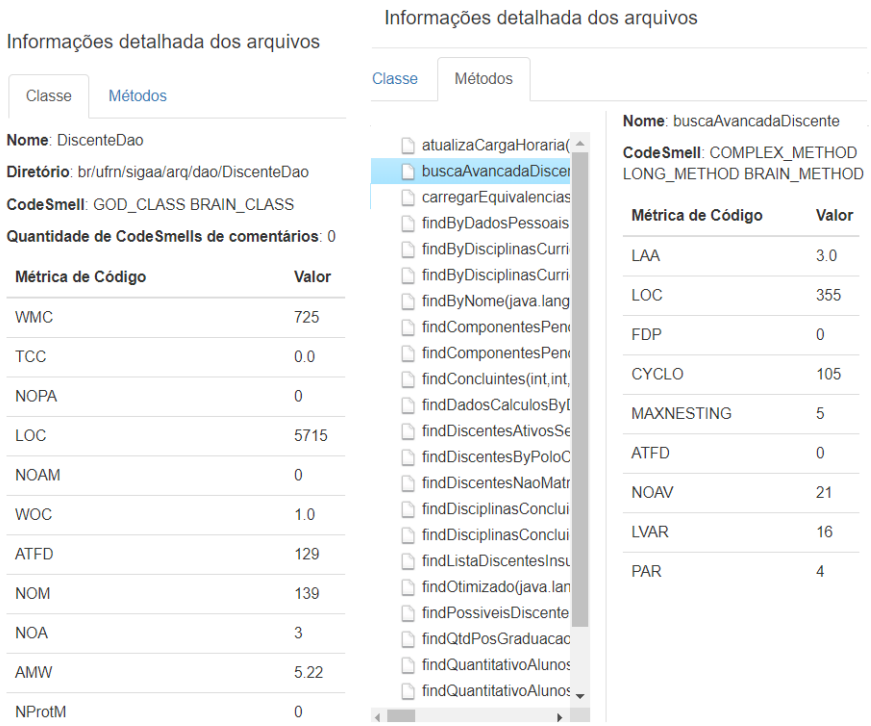
Na Figura 7 é apresentado um exemplo de tela da TDVision, na qual, é possível observar as funcionalidades ordenadas de acordo com a quantidades de *code smells*.

Figura 8 – Alguns arquivos que compõem a funcionalidade inscrição Seleção exibidos na TDVision.

Fonte – Autor.

Na Figura 8 é apresentado um outro exemplo de tela da TDVision, onde é possível observar informações sobre a funcionalidade, tais como: (i) os tipos de *code smells* presentes

na funcionalidade, (ii) a quantidade de cada tipo de *code smells*, (iii) os arquivos que compõem a funcionalidade e suas respectivas quantidades de *code smells*;



(a) Informações da Classe (b) Informações da Método

Figura 9 – Informações de uma classe e de um método que compõem a funcionalidade inscrição Seleção exibidas na TDVision.

Fonte – Autor.

Na Figura 9 é apresentado mais um exemplo de tela da TDVision, onde são exibidos os *code smells* e os valores obtidos nas métricas de *software*, tanto da classe quanto dos métodos. A Figura 9a, mostra informações calculadas para uma determinada classe da funcionalidade, e a Figura 9b, exibe informações calculadas de um determinado método de uma classe da funcionalidade.

3.2 Abordagem para Gestão de *Code Smells* em Projetos de Manutenção/Evolução de Software

Uma vez que o processo de manutenção é definido como uma parte fundamental do ciclo de vida do software e, a partir dele, é possível realizar correções, adaptações, aperfeiçoamentos e prevenções de erro em um software garantindo sua continuidade. E a partir da definição do método para identificação de *code smells* por funcionalidade e sua implementação (TDVision), foi definida uma abordagem para uso desse módulo em um ambiente de desenvolvimento real, cujo o principal objetivo é avaliar se a mesma pode estimular os desenvolvedores a corrigirem os indícios de DT durante a manutenção

do *software*. A Figura 10 mostra os passos da abordagem proposta adotando o modulo TDVision.

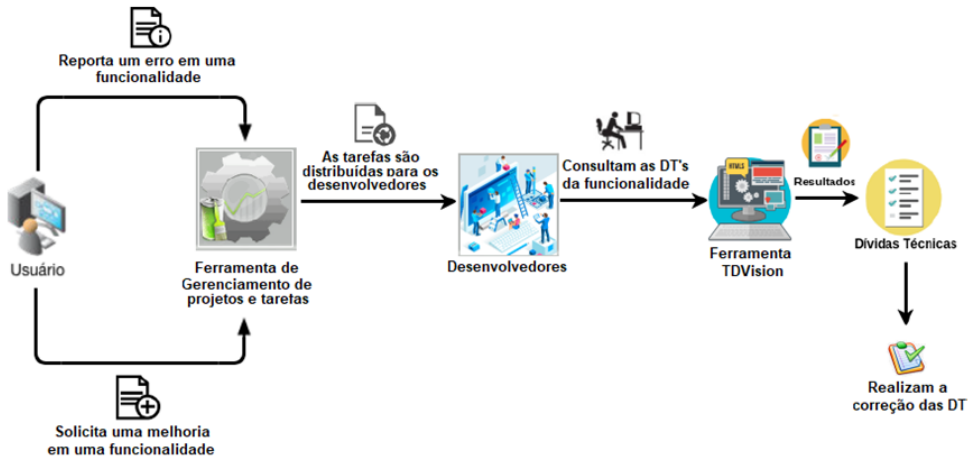


Figura 10 – Visão geral dos processos que compõem a abordagem proposta

Fonte – Autor.

A abordagem é projetada para ser flexível, de modo que possa ser adaptada para incorporar novas ideias e/ou abordagens. Imagine um processo simplificado de manutenção, em que um usuário abre um chamado, por meio de alguma ferramenta de registro de demandas. Tal demanda pode ser um erro em uma funcionalidade ou uma melhoria em uma funcionalidade existente. Esse chamado deve então ser distribuído ao time de desenvolvimento pelo gerente do projeto. Os desenvolvedores de posse das tarefas e seguindo processo de manutenção anterior existente, começariam a trabalhar nas mesmas.

A abordagem aqui proposta é bastante simples. O processo de software utilizado para gerir as atividades de manutenção/evolução de software devem incluir, antes da realização de qualquer tarefa de codificação, a visualização dos *code smells* relacionados à funcionalidade a ser mantida. Dessa forma, a partir do momento que os desenvolvedores recebem as tarefas de manutenção, os mesmos utilizam o modulo TDVision, para visualizar os *code smells* presentes na funcionalidade. Eles também devem ser estimulados a visualizar os *code smells* por funcionalidades durante a codificação.

Modelos que estabelecem o processo de manutenção de software apresentam diferentes focos, sendo que alguns se atentam mais a questões econômicas, outros ao produto e outros ao próprio processo. Todos possuem vantagens e desvantagens, não existindo um modelo único e aplicável às diversas situações. A melhor solução, muitas vezes, é utilizá-los em conjunto para obter o resultado que agregue mais benefícios para as organizações (YONGCHANG ET AL., 2011). A abordagem proposta nesse estudo, assim como outros modelos, possue algumas vantagens e desvantagens conforme listadas a seguir:

- **Vantagens**:
  - Introdução/Discussão do assunto DT no cotidiano dos desenvolvedores o que de certa forma minimiza a introdução de mais DT;
  - Possibilidade de criação de uma agenda positiva de refatoração do sistema para melhorar a qualidade do codigo;
- **Desvantagens**:
  - Possível overhead no tempo de atendimento das tarefas;
  - Uma etapa a mais no processo de desenvolvimento de software;

3.3 Considerações Finais

Este capítulo apresentou uma forma de gerar uma nova visualização de *code smells* em projetos de software, utilizando uma visão por funcionalidade. As principais etapas do método para essa visualização consistem em identificar funcionalidades de software e arquivos de código que fazem parte de cada funcionalidade. Assim é possível calcular os *code smells* sobre cada arquivo associado a uma funcionalidade, para então gerar a visão de *code smells* por funcionalidade, a partir da soma dos *code smells* dos itens associados. Esta forma de visualização foi implementada como um novo módulo na ferramenta *CoDiVision*, denominado TDVision. Com o TDVision é possível observar informações sobre a funcionalidade, tais como: (i) o total de *code smells* presentes nas funcionalidades, (ii) os tipos de *code smells* presentes na funcionalidade, (iii) a quantidade de cada tipo de *code smells*, (iv) os arquivos que compõem a funcionalidade e suas respectivas quantidades de *code smells*.

A partir da implementação da TDVision foi projetado uma abordagem que visa apoiar, de forma mais apropriada, gerentes de projetos e os desenvolvedores durante a etapa de manutenção de software. A informação de *code smells* por funcionalidades tende a ser mais apropriada em um contexto de desenvolvimento de software real, pois gerentes de projetos precisam comumente distribuir tarefas que normalmente são tratadas como funcionalidades.

No próximo capítulo é apresentada um estudo de caso aplicado em um projeto real para avaliar a abordagem proposta durante atividades de manutenção/evolução de software.

4 Estudo de Caso

4.1 Introdução

No campo da engenharia de *software*, a utilização de métodos experimentais pode trazer alguns benefícios, dentre os quais: acelerar o progresso através da rápida eliminação de abordagens não fundamentadas, suposições errôneas e modismos, bem como, ajudar a orientar a engenharia e a teoria para direções promissoras (WOHLIN et al., 2012). É possível encontrar na literatura três estratégias experimentais: *survey*, estudo de caso e experimento. Neste sentido, realizamos um estudo de caso para avaliar a abordagem proposta.

4.2 Definição de Estudo de Caso

Por definição, um estudo de caso é uma metodologia de pesquisa que consiste em uma investigação empírica de um fenômeno contemporâneo em um contexto real, especialmente quando os limites entre fenômeno e contexto não são claramente evidentes (YIN, R. K., 2014; RUNESON, P. ET AL., 2012). Os estudos de caso são realizados em contextos do mundo real favorecendo o alto grau de realismo e relevância da pesquisa ainda que isso custe a redução do seu nível de controle (RUNESON, P. ET AL., 2012).

Os estudos de caso não geram os mesmos resultados de relação de causalidade como experimentos controlados fazem, mas eles fornecem uma compreensão mais profunda dos fenômenos em estudo por conta de suas conclusões serem baseadas em análise de dados qualitativos e quantitativos providos por uma cadeia de evidências (RUNESON, P. ET AL., 2012).

Em resumo, as principais características do estudo de caso são:

- Lida com as características complexas e dinâmicas de fenômenos do mundo real;
- Suas conclusões são baseadas em uma clara cadeia de evidências, seja de natureza qualitativa ou quantitativa, coletadas de várias fontes de forma planejada e consistente.
- É menos formal quando comparado a experimentos controlados.
- Permitem entender a prática do uso de algo em seu contexto real de aplicação.

Estudo de caso é uma estratégia de pesquisa comumente usada em áreas como psicologia, sociologia, ciência política, serviço social, negócios e planejamento social. Nessas

áreas, estudos de caso são realizados com os objetivos não só de aquisição de conhecimento, mas também de trazer mudanças no fenômeno em estudo (por exemplo, melhorar educação ou de assistência social). A pesquisa em engenharia de software tem objetivos de alto nível similares de entender o porquê da engenharia de software deve ser realizada e, com esse conhecimento, tentar melhorar o processo engenharia de software, os produtos e os serviços de software gerados (RUNESON, P. ET AL., 2012). A área de engenharia de software envolve desenvolvimento, operação e manutenção de software e artefatos relacionados. A pesquisa na engenharia software tem, em grande medida, o objetivo de investigar como desenvolvimento, operação e manutenção são realizados por engenheiros de software e outras partes interessadas sob diferentes condições.

4.3 Planejamento

Esta seção apresenta o plano do estudo de caso, destacando seus objetivos, questões, contexto, participantes e procedimentos.

4.3.1 Objetivo e questão da pesquisa

O objetivo deste estudo de caso é investigar os benefícios da identificação/visualização de *code smells* em nível de funcionalidade para *softwares* em manutenção/evolução.

Assim, pretende-se avaliar a seguinte questão geral:

- A utilização da abordagem que contempla o método de visualização de *code smells* por funcionalidade pode estimular os desenvolvedores a corrigirem os indícios de dívidas técnicas durante a manutenção do *software*?

4.3.2 Proposições e **Hipóteses**

A sentença a seguir, baseada no método GQM (BASILI VICTOR R., 1999), define o estudo de caso.

"**Analisar** a aplicação da abordagem aqui proposta, **com o propósito de** avaliar seus benefícios, **com respeito ao** gerenciamento de *code smells*, **do ponto de vista** de desenvolvedores e gestores de projetos, **no contexto** de projetos de manutenção e evolução de *softwares*".

Conforme será esclarecido em maiores detalhes na próxima seção, a avaliação da abordagem de acordo com a definição acima será feita a partir da coleta e análise de dados quantitativos e qualitativos durante a operação desse estudo experimental.

Em um estudo de caso, é necessário estabelecer claramente o que se pretende avaliar (JUZGADO N. J.; VEGAS, 2004). Para isso hipóteses precisam ser definidas. Neste trabalho utilizamos as seguintes hipóteses:

- HN_0 : Não há benefício em relação ao gerenciamento de *code smells* em usar a abordagem proposta durante a manutenção do *software*.

$$HN_0 : (\Delta_1 = \Delta_2) \wedge (\Omega_1 = \Omega_2)$$

- HA_1 : A abordagem proposta produz benefícios no gerenciamento de *code smells* durante a manutenção do *software*.

$$HA_1 : (\Delta_2 \neq \Delta_1) \wedge (\Omega_2 \neq \Omega_1)$$

Onde:

- Δ_1: Número de *code smells* inseridos pré abordagem;
- Δ_2: Número de *code smells* inseridos pós abordagem;
- Ω_1: Número de *code smells* corrigidos pré abordagem;
- Ω_2: Número de *code smells* corrigidos pós abordagem.

4.3.3 Seleção de Variáveis

O experimento utiliza as seguintes variáveis:

- Variável independente: a abordagem utilizada para visualizar *code smells* em projetos de *software*.
- Variáveis dependentes: as quantidades de *code smells* pagos e inseridos durante o uso da abordagem, o processo utilizado, a experiência dos desenvolvedores.

4.3.4 Conjectura e unidade de análise

Este estudo experimental foi executado de forma on-line, uma vez que ele aconteceu em tempo real durante um projeto de manutenção. Os participantes do estudo são profissionais que atuam em um projeto real de manutenção e evolução de *software*. Houve um monitoramento contínuo sobre as atividades dos participantes.

A unidade de análise definida foi um órgão responsável por Tecnologia da Informação em uma Instituição de Ensino de Superior. O projeto utilizado durante o estudo é composto por mais de 764K linhas de código (contabilizado somente as linhas escritas em linguagem

de programação Java e que não estavam comentadas), 5.335 classes que compõem 982 funcionalidades distribuídas em 18 módulos e 14 portais. Nesse projeto, foram identificadas 4.216 *code smells*, contabilizado somente os 7 tipos identificadas pelo módulo TDVision: *complex method* = 1.947, *long method* = 1.436, *brain method* = 439, *god class* = 150, *brain class* = 139, *data class* = 69 e *feature envy* = *36*).

Os participantes deste estudo foram selecionados por conveniência (WOHLIN et al., 2012), ou seja, não foram escolhidos de forma aleatória. Este fato otimizou o tempo e o custo da realização do estudo. Dessa forma, os participantes selecionados são o gerente do projeto e os analistas de desenvolvimento que trabalham no projeto em que foi executado o estudo. O perfil dos participantes, em termos de sua experiência prática com desenvolvimento de *software* e tempo no projeto, foi levantando através da aplicação de um questionário, conforme Apêndice B. A Figura 11 indica a caracterização dos participantes em relação as características mencionadas.

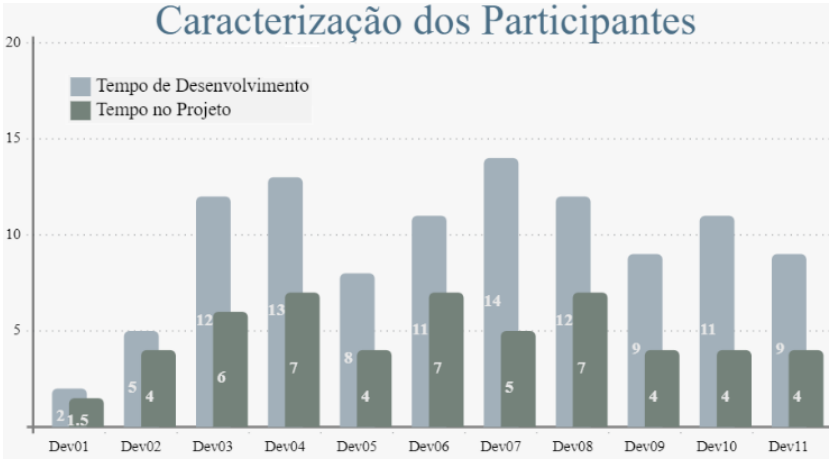


Figura 11 – Caracterização dos Participantes

Fonte – Autor.

4.3.5 Método de coleta dados

Foram utilizados diferentes fontes de dados para a realização da coleta de dados antes e depois da execução do estudo de caso, listados a seguir:

- **Registros de realização de tarefas:** a contabilização da quantidade de tarefas realizadas pela equipe de desenvolvimento, assim como o tempo gasto na execução dessas tarefas foram obtidas a partir da ferramenta de registro de tarefa existente no órgão, tendo sido obtidos dados referentes a períodos anteriores à realização do estudo de caso e os dados obtidos durante a aplicação da abordagem;
- **Repositórios de código de fonte dos *software* desenvolvidos:** a contabilização dos *code smells* inseridos e pagos foram obtidas minerando o repositório de código

fonte do órgão. Esses dados também estavam relacionados a períodos anteriores ao estudo de caso, bem como durante a realização do estudo.

- **Questionários aplicados aos participantes:** um questionário de caracterização dos participantes e outro de *feedback* foi aplicado, visando possibilitar a análise qualitativa dos elementos envolvidos no estudo.

4.4 Operação

Inicialmente, os participantes receberam um treinamento com duração de duas horas e trinta minutos, em que foi explicado a finalidade do estudo, os objetivos da abordagem, a descrição do cenário da atividade proposta. Em seguida foi solicitado aos participantes que preenchessem os formulários de consentimento, conforme Apêndice A, e caracterização, conforme Apêndice B. Então, uma primeira rodada foi utilizada para que os participantes se familiarizassem com o uso da ferramenta TDVision, bem como eventuais refatorações que poderiam ser feitas a partir dessa visualização. O estudo em si, foi realizado durante o período de 04/12/2018 a 25/04/2019, período em que os dados foram coletados, tais como duração de atividades de manutenção, gerenciamento dos *code smells*, total de *code smells* inseridos e pagos.

Para o registro e o acompanhamento das tarefas, sejam elas evolutivas (customizações) ou corretivas (sustentações), o órgão utiliza um *software* próprio que foi desenvolvido pelo time, denominado de SINAPSE (Sistema Integrado de Acompanhamento a Projetos e Serviços)[1]. Esse sistema é integrado ao *Redmine*, que é uma aplicação *web* flexível utilizada para gerenciamento de projetos. O processo de trabalho da equipe para customizações é elaborado de acordo com o *framework Scrum*. As *sprints* no órgão tem a duração de 10 dias úteis e possuem alguns passos bem definidos.

A ferramenta descrita na Seção 3.1.4 foi inserida no ambiente de desenvolvimento do órgão e operou neste ambiente durante vinte semanas. Durante esse período as tarefas que estavam relacionadas a melhorias em funcionalidades já existentes e as que reportavam erros em uma funcionalidade, tinham seus *code smells* checados na TDVision, pelo desenvolvedor, antes de começar a trabalhar na tarefa. Dessa forma os desenvolvedores obtinham conhecimento dos tipos de *code smells* encontrados na funcionalidade e em quais arquivos da funcionalidade eles estavam presentes, podendo assim realizar a correção dos mesmos. Foi incentivado pelo gerente de projeto que os desenvolvedores utilizassem parte do tempo da tarefa para pagamento dos indícios de DT associadas à tarefa em execução.

A cada 2 semanas, eram realizadas extrações no repositório de código, coletando o número de *code smells* inseridos e pagos, e na ferramenta de acompanhamento das tarefas

[1] Sistema SINAPSE: <https://sinapse.ufpi.br>.

eram coletadas informações sobre o total de tarefas e o tempo de execução das mesmas. No final do estudo, os participantes foram solicitados a preencher um questionário de acompanhamento, permitindo obter dados qualitativos.

4.5 Resultados e discussão

4.5.1 Análise quantitativa

A análise quantitativa do estudo tem por objetivo comparar os dados coletados no período de execução do experimento com o período anterior ao mesmo.

A Figura 12 mostra uma comparação do Período 1 (antes da aplicação da abordagem), com o Período 2, em que foi realizada o uso da TDVision, mostrando o número de novos *code smells* inseridos e o número de *code smells* corrigidos.

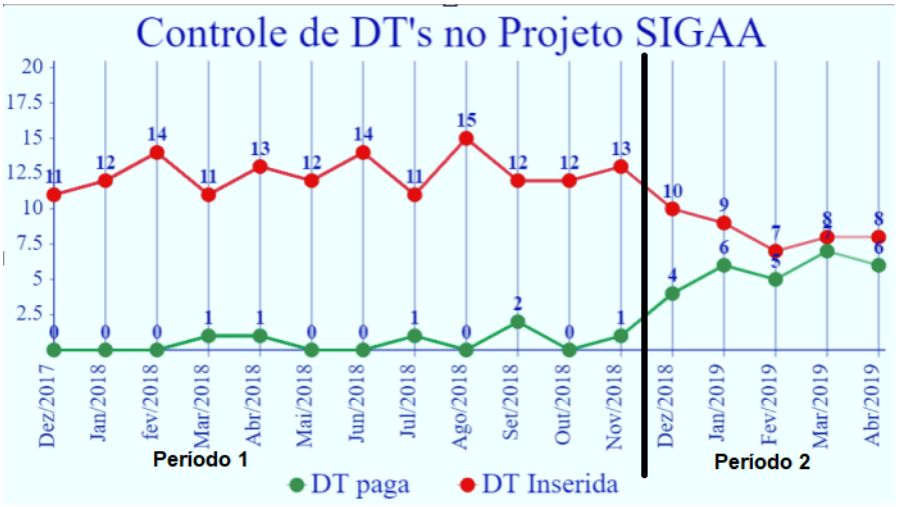


Figura 12 – Evolucão das DT's no projeto.

Fonte – Autor.

Como é possível observar na comparação do Período 1 com o Período 2, com a utilização do uso da abordagem proposta, que se baseia no uso da TDVision, houve uma melhora no tratamento dos *code smells*: (i) aumento do número de *code smells* corrigidos e (ii) diminuição do número de *code smells* inseridos.

Adicionalmente, foi realizado um levantamento do total de chamados abertos, e do tempo gasto pelo time para resolvê-los, para com isso podermos comparar com o período em que o estudo foi realizado. Na Figura 13 é possível observar que o número de chamados é maior no período de dezembro de 2018 a abril de 2019, e mesmo assim o número de *code smells* inseridos foi menor quando comparado ao período de dezembro de 2017 a abril de 2018, conforme Figura 12. Com relação a Figura 14, podemos observar que houve uma redução na resolução dos chamados em até 4 horas, e consequentemente houve um

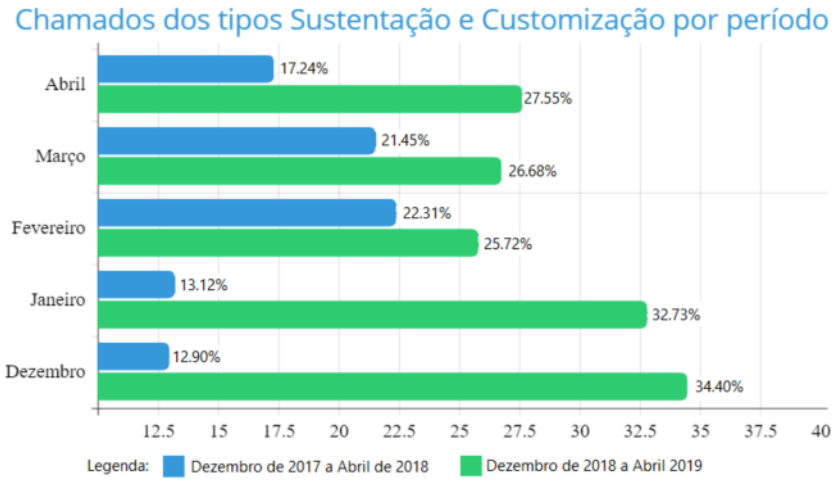


Figura 13 – Chamados dos tipos sustentação e customização por período.

Fonte – Autor.

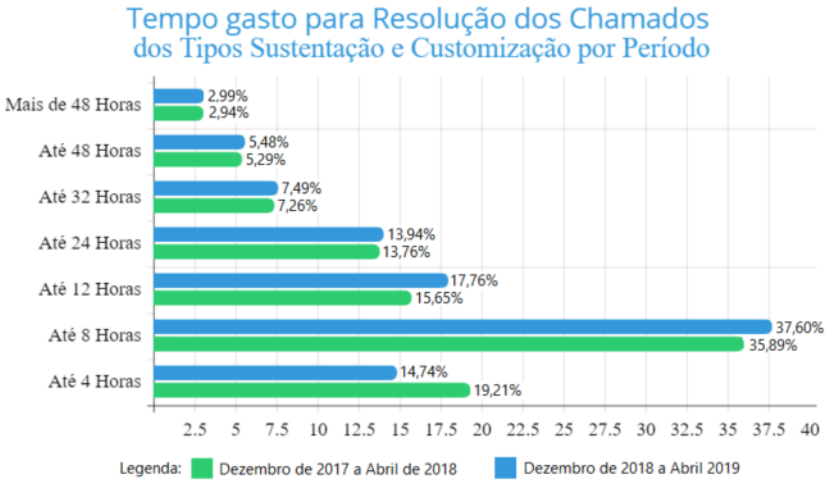


Figura 14 – Tempo gasto para resolução dos chamados.

Fonte – Autor.

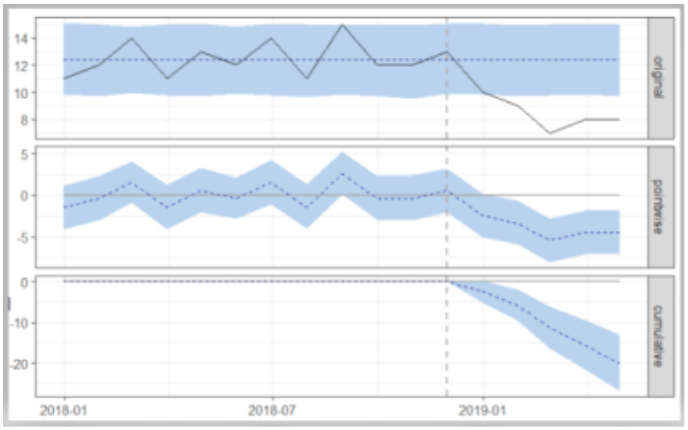
aumento significativo nos chamados resolvidos em até 8 horas e em até 12 horas, o que é perfeitamente normal tendo em vista que os desenvolvedores passaram gastar tempo refatorando para corrigirem os *code smells*.

Usando o *CausalImpact*[2,3], que é um algoritmo desenvolvido pelo Google para estimar o efeito causal de uma intervenção projetada em uma série temporal, realizamos uma projeção, baseada no histórico, de como seria caso o ambiente de desenvolvimento não tivesse sido alterado e comparamos estatísticamente essa projeção com os dados obtidos no estudo. O *CausalImpact* funciona da seguinte forma: dada uma série temporal de resposta

[2] https://github.com/google/CausalImpact
[3] https://google.github.io/CausalImpact/CausalImpact.html

(no nosso caso a inserção e correção dos *code smells* pós implantação da abordagem) e um conjunto de séries temporais de controle (inserção e correção dos *code smells* pré implantação da abordagem), o pacote constrói um modelo *bayesiano* de séries temporais estruturais. Este modelo é então usado para tentar prever o contrafactual, ou seja, como a métrica de resposta teria evoluído após a intervenção se a intervenção nunca tivesse ocorrido.

Por padrão, o gráfico gerado pelo *CausalImpact* contém três painéis, conforme é possível observar nas Figuras 15 e 16. O primeiro painel mostra os dados e uma previsão contrafactual para o período pós-aplicação da abordagem. O segundo painel mostra a diferença entre dados observados e previsões contrafactuais. Este é o efeito causal pontual, conforme estimado pelo modelo. O terceiro painel adiciona as contribuições pontuais do segundo painel, resultando em um gráfico do efeito cumulativo da aplicação da abordagem.



	Average	Cumulative
Actual	8.4	42.0
Prediction (s.d.)	12 (0.71)	62 (3.53)
95% CI	[11, 14]	[55, 69]
Absolute effect (s.d.)	-4 (0.71)	-20 (3.53)
95% CI	[-5.4, -2.6]	[-26.8, -13.0]
Relative effect (s.d.)	-32% (5.7%)	-32% (5.7%)
95% CI	[-43%, -21%]	[-43%, -21%]

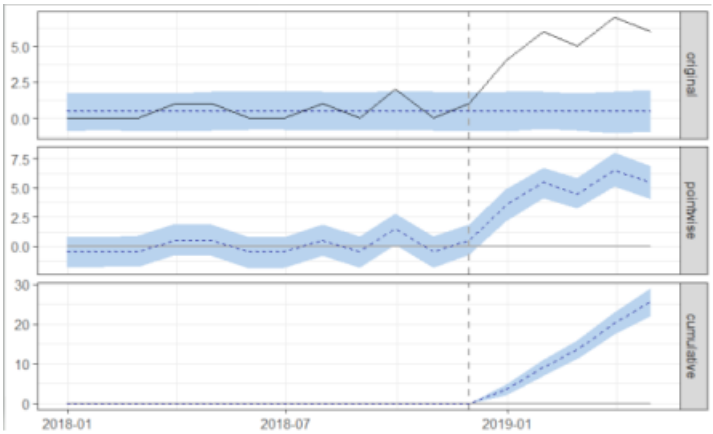
Posterior tail-area probability p: 0.001
Posterior prob. of a causal effect: 99.9%

Figura 15 – Gráfico dos *code smells* inseridos

Fonte – Autor.

Conforme Figura 15, durante o período após aplicação da abordagem, a variável resposta apresentou um valor médio de aproximadamente 8,40. Em contraste, na ausência de uma intervenção, era esperado uma resposta média de 12,41. O intervalo de 95% desta previsão contrafactual está entre 11,01 e 13,77. A variável de resposta teve um valor global de 42,00. Em contraste, se a aplicação da abordagem não tivesse ocorrido, teríamos esperado uma soma de 62,06. Ou seja era esperado um valor dentro do intervalo de 55,04 e 68.84, caso a abordagem não tivesse sido aplicada.

Os resultados anteriores são dados em termos de números absolutos. Em termos relativos, a variável resposta mostrou uma diminuição de 32%. A probabilidade de obter este efeito por acaso é muito pequena (probabilidade *bayesiana* de área de cauda unilateral p = 0,001). Isso significa que o efeito causal, provocado pela aplicação da abordagem, pode ser considerado estatisticamente significativo.



	Average	Cumulative
Actual	5.6	28.0
Prediction (s.d.)	0.48 (0.34)	2.38 (1.72)
95% CI	[-0.19, 1.2]	[-0.97, 5.9]
Absolute effect (s.d.)	5.1 (0.34)	25.6 (1.72)
95% CI	[4.4, 5.8]	[22.1, 29.0]
Relative effect (s.d.)	1077% (72%)	1077% (72%)
95% CI	[928%, 1218%]	[928%, 1218%]

Posterior tail-area probability p: 0.001
Posterior prob. of a causal effect: 99.8999%

Figura 16 – Gráfico dos *code smells* Corigidos

Fonte – Autor.

Analisando a Figura 16, é possível observar que durante o período pós-aplicação da abordagem, a variável resposta, que representa o número de *code smells* corrigidos, apresentou um valor médio de aproximadamente 5,60. Em contraste, na ausência de uma intervenção, esperávamos uma resposta média de 0,48. Ou seja era esperado um valor dentro do intervalo de -0,19 e 1,18. Subtrair essa previsão da resposta observada produz uma estimativa do efeito causal da aplicação da abordagem sobre a variável resposta. Este efeito é de 5,12. A variável resposta teve um valor global de 28,00. Em contraste, se a aplicação da abordagem não tivesse ocorrido, teríamos esperado uma soma de 2,38.

Os resultados acima são dados em termos de números absolutos. Em termos relativos, a variável resposta apresentou um aumento de 1077%. Isto significa que o efeito positivo observado durante o período de aplicação da abordagem é estatisticamente significativo e é improvável que seja devido a flutuações aleatórias. A probabilidade de obter este efeito por acaso é muito pequena (probabilidade *bayesiana* de área de cauda unilateral p = 0,001).

Esses resultados nos permite refutar a hipótese nula HN_0 e, ao mesmo tempo, validar a hipótese alternativa HA_1.

4.5.2 Análise qualitativa

Os dados dos sujeitos que usaram a abordagem foram analisados qualitativamente. A opinião foi coletada a partir do questionário de *feedback*, aplicado logo após a execução do estudo experimental. A escala utilizada para cada uma das questões é do tipo *Likert* (MCIVER J. P.AND CARMINES, 1991). No nosso estudo empregou-se os seguintes níveis de afirmação na escala de Likert: Discordo plenamente; Discordo parcialmente; Nem concordo nem discordo; Concordo parcialmente; Concordo plenamente. As seguintes questões foram feitas conforme Apêndice C: (i) O uso da abordagem ajudou a melhorar a qualidade do seu código produzido?, (ii) O esforço para realização das tarefas que tinham a presença de *code smells* verificados na CodVision foi diferente das que não tinham?, (iii) O fato de ter conhecimento sobre a presença de *code smells* afetou de alguma forma como você desenvolve *software*?, (iv) Você acha importante monitorar *code smells* dentro do processo de desenvolvimento de *software*? e (v) É possível fazer com que este gerenciamento faça parte do processo de desenvolvimento?

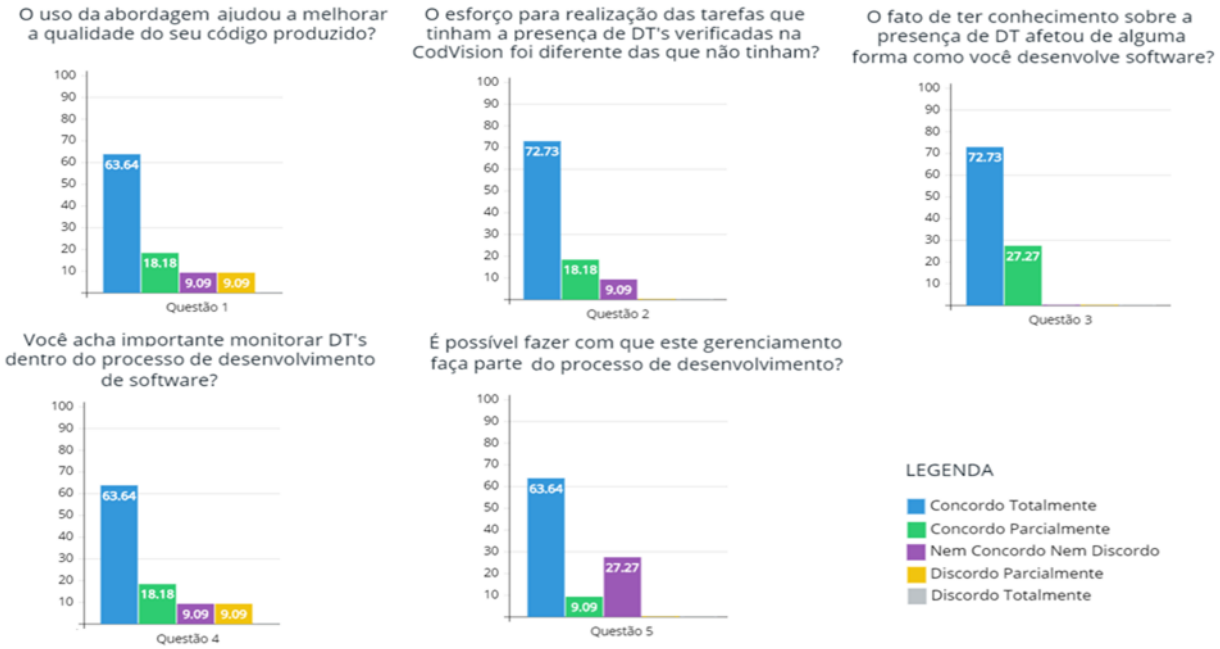


Figura 17 – Resultado do questionário de *feedback*.

Fonte – Autor.

As respostas foram analisadas e sumarizadas a seguir: 81.82% dos participantes notaram um aumento na qualidade do seu código produzido. No entanto, 90.91% notaram um aumento no esforço para realizar a tarefa, pois além de terem que corrigir um defeito, sem inserir novos *code smells*, eles tinham que refatorar para corrigigem os *code smells* já

existentes. Eles confirmaram de forma unânime que o conhecimento sobre a presença de *code smells* afetou o modo como eles desenvolviam. E 81.82% acham importante monitorar *code smells* dentro do processo de desenvolvimento de *software* e quanto a abordagem fazer parte do processo de desenvolvimento, 72.73% dos entrevistados consideraram que é possível e interessante.

4.6 Ameaças à Validade

Uma das questões fundamentais a respeito de uma avaliação refere-se ao grau de validade dos resultados obtidos. Assim, é importante analisar os riscos relacionados à avaliação e propor alternativas para tentar contornar os problemas que possam ameaçar à validade dos resultados (WOHLIN et al., 2012). Esta seção apresenta as ameaças ligadas à avaliação da abordagem proposta neste trabalho.

4.6.1 Validade Interna

Este aspecto está relacionado à validade dos resultados identificando a relação causa-efeito (WOHLIN et al., 2012), ou seja, uma ameaça relacionada ao controle do processo de avaliação para coletar os dados analisados.

Uma ameaça interna identificada é a interação entre os participantes no momento de responder o questionário, uma vez que o mesmo foi submetido aos participantes em um ambiente não controlado e em momentos distintos, por isso, não é possível afirmar que os participantes não interagiram entre si com o intuito de compartilhar informações sobre as suas respostas, oferecendo uma ameaça à autenticidade das respostas obtidas. Outra ameaça identificada é a classificação de experiência dos participantes, pois o fato de um participante estar com certo tempo trabalhando com uma linguagem ou em um projeto, não garante que eles tenham um grande conhecimento sobre os mesmos. Além disso, nas respostas do questionário eles podem relatar o que desejam ser e não o que de fato são. Isso poderia ser substituído pela execução de avaliações de habilidades dos participantes com custo de esforço e tempo maior para a realização do estudo. No entanto, o mesmo número de aspectos do histórico dos participantes poderiam permanecer incerto e ocorrer uma evasão no número de participantes, tendo em vista que os mesmos poderiam ter receio de represália por expor seu conhecimento quanto ao seu trabalho realizado.

Outra ameaça identificada, é o fato de o ambiente onde foi realizado o estudo não possuir nenhuma ferramenta para gerenciar indícios de DT (nem mesmo uma de visualização por arquivo, seja em classes ou métodos) anterior ao estudo, com isso não tivemos como fazer uma comparação entre o método proposto nesse trabalho, que mostra *code smells* por funcionalidade e outros métodos e ferramentas que os mostram por arquivos. Essa comparação poderia tornar os resultados ainda mais expressivos.

O fato dos pesquisadores terem estimulado os desenvolvedores a não inserir *code smells* e a realizarem pagamento dos mesmos pode configurar uma ameaça interna, pelo fato de eles estarem realizando o pagamento não pelo uso da abordagem mais sim pela provocação dada, no entanto só o estimulo sem o uso da ferramenta de identificação por funcionalidade talvez não gerasse um resultado tão bom, pois seria necessário um alto nível de conhecimento dos desenvolvedores sobre *code smells* para identificar e corrigir os mesmos. Além disso durante a realização do estudo diversos desenvolvedores tomavam a iniciativa em buscar informações de quais *code smells* a funcionalidade que eles iam trabalhar possuía e buscavam corrigir os mesmos, demonstrando preocupação com a qualidade do seu código produzido.

4.6.2 Validade Externa

A validade externa está relacionada ao risco de generalização dos resultados, pois alguns fatores podem ameaçar a validade e não permitir a generalização dos resultados obtidos durante as avaliações.

Apesar do estudo ter sido feito em um sistema de grande porte, por ser baseado em apenas um projeto, os resultados podem não refletir outros ambientes de desenvolvimento que utilizem outras tecnologias. Também estamos cientes de que limitamos a nossa atenção apenas aos códigos escritos em linguagem programação Java, devido às limitações da infraestrutura que usamos (por exemplo, a abordagem de identificação de funcionalidade e a ferramenta de detecção de *code Smell* que só funcionam em código Java). São desejáveis mais estudos que visem replicar o nosso trabalho sobre sistemas escritos em outras linguagens de programação.

4.7 Considerações Finais

Este capítulo apresentou uma avaliação da abordagem proposta neste trabalho. Por meio dos resultados obtidos pôde-se perceber que a abordagem proposta obteve um bom desempenho tanto no número de *code smells* pagos quanto na diminuição dos *code smells* inseridos, embora o tempo gasto nas atividades tenha elevado.

5 Conclusões e Trabalhos Futuros

A manutenção de software pode ser definida como a modificação do sistema após a entrega para corrigir falhas, incluir novas funcionalidades, melhorar o desempenho ou se adaptar a um novo ambiente (COMMITTEE, 1983). A etapa de manutenção é normalmente a atividade mais onerosa no ciclo de vida de um software, pois é executada durante toda sua vida útil.

Este trabalho apresentou uma forma de se visualizar *code smells* em projetos de software, baseado em técnicas de Engenharia Reversa (ER) (CHIKOFSKY; CROSS, 1990; MÜLLER et al., 1992) e Mineração de Repositórios de Software (MRS) (HASSAN, 2008b), visando calcular o total de *code smells* sob a perspectiva de funcionalidades de software. O objetivo principal dessa nova forma de se visualizar *code smells* é apoiar gerentes ou líderes de projetos durante o processo de desenvolvimento de software, principalmente durante as atividades de manutenção, sejam elas corretivas, adaptativas ou perfectivas.

A visualização proposta foi implementada com um novo módulo na ferramenta *CoDiVision* (LIRA, 2016), denominado TDVision. Com ele, as atividades de identificação e monitoramento de dívidas técnicas podem ser realizadas de forma simples e automatizada. Usando essa ferramenta os gerentes de projetos de software podem, por meio de visualização gráfica, obter a informação sobre os *code smells* presentes nas funcionalidades do sistema. Com essa informação é possível definir melhores estratégias durante o processo de manutenção de software para que seja realizado a correção dos *code smells* presentes.

Tambem apresentamos uma abordagem para manutenção/evolução de *software* que faz uso do modulo TDVision e tem como foco a redução de *code smells*. Essa abordagem de manutenção baseada na visualização dos *code smells* de uma funcionalidade foi objeto de um estudo de caso para verificar os benefícios da identificação/visualização dos *code smells* no nível de funcionalidade. Os resultados combinados com os relatos dos entrevistados perante as questões, mostram evidências de que a abordagem pode apoiar a gestão técnica da dívida, podendo-se assim incentivar o pagamento de *code smells* e com isso ampliar a qualidade do software mantido.

Por fim, para efeito de sumarização, a seguir são listadas as principais contribuições oriundas da realização deste trabalho:

- **Desenvolvimento de um método de identificação e visualização de *code smells* em funcionalidades de software**: foi desenvolvida uma forma para se visualizar *code smells* em projetos de software, baseada em técnicas de Engenharia Reversa (ER) e Mineração de Repositórios de Software (MRS).

- **Disponibilização da Ferramenta para auxiliar a gestão dos *code smells***: O método foi implementado como um novo módulo na ferramenta *CoDiVision* (LIRA, 2016), denominado TDVision[1]. A ferramenta passou a permitir a identificação/visualização dos *code smells* nas funcionalidades de um projeto de *software*, e assim direcionar os desenvolvedores no momento das correções dos mesmos.

- **Definição de uma abordagem para uso da TDVision durante o processo de manutenção e evolução de software**: foi definido uma abordagem simplificada para manutenção/evolução de *software* que faz uso do modulo TDVision e tem como foco a redução de *code smells*.

- **Avaliação do uso da visualização de *code smells* em funcionalidades durante manutenção de software**: foi realizada um estudo de caso em um projeto real para verificar os benefícios da identificação/visualização de *code smells* no nível de funcionalidade durante tarefas de manutenção. Os resultados mostram evidências de que a abordagem pode apoiar a gestão técnica de *code smells* e estimular a melhoria na qualidade do código mantido.

5.1 Desafios e Limitações

Como apresentado durante o estudo de caso (Capítulo 4) a abordagem proposta neste trabalho obteve resultados satisfatórios no auxilio ao gerenciamento dod *code smells*, contribuindo assim para uma redução no numero de *code smells* inseridos e um aumento no numero de *code smells* corrigidos. Contudo, o método que faz parte da abordagem apresenta algumas limitações. Uma delas está relacionada à linguagem de programação utilizada no desenvolvimento do sistema que será analisado. A visualização proporcionada pela TDVision hoje só funciona em códigos Java.

Em relação à avaliação apresentada no Capítulo 4, existe também outro desafio a ser superado. Como foi apresentado, a avaliação da abordagem para manutenção/evolução de *software* que faz uso do modulo TDVision, apesar de ter sido realizada em um ambiente real de desenvolvimento, o mesmo não possuía nenhuma ferramenta para gerenciar *code smells* (nem mesmo uma de visualização por arquivo, seja em classes ou métodos) anterior ao estudo, com isso não tivemos como fazer uma comparação entre o método proposto nesse trabalho, que mostra *code smells* por funcionalidade e outros métodos e ferramentas que os mostram por arquivos. Essa comparação poderia tornar os resultados ainda mais expressivos. Contudo, os resultados obtidos são promissores e sugerem que a abordagem proposta pode ser utilizada como um suporte a gerentes ou líderes de projetos, especialmente durante o processo de manutenção de software.

[1] http://easii.ufpi.br/codivision

Por fim, é notável que apesar do trabalho já realizado, existem algumas possibilidades de melhorias e trabalhos futuros. Assim, tais limitações aqui apresentadas oferecem novas oportunidades de pesquisa futuramente.

5.2 Trabalhos Futuros

Com base nos desafios e limitações apresentadas na Seção 5.1, foram definidas algumas direções para trabalhos futuros:

- **Extensão e melhorias na implementação do método:** O método proposto faz uso de uma abordagem de identificação de funcionalidade e uma ferramenta de detecção de *code smell* que só funcionam em código Java. Dessa forma o processo de identificação de funcionalidades, bem como a identificação de DT's depende da linguagem de programação utilizada na codificação do sistema analisado. Dessa forma, a visualização foi inicialmente implementada para análise de sistemas desenvolvidos em linguagem Java. Contudo, é importante também a implementação de uma extensão do método para análise de sistemas escritos em outras linguagens de programação. Além disso é importante estender a detecção de *code smells* considerando outros tipos.

- **Novas avaliações da abordagem:** Este trabalho apresenta uma avaliação em um ambiente real de desenvolvimento. Como nossa abordagem é projetada para ser flexível, de modo que possa ser adaptada para incorporar novas ideias e abordagens, é importante realizar um novo estudo aplicando o método de identificação por funcionalidade e um outro método que identifica *code smells* por arquivo, e assim comparar qual método tem melhores resultados (tanto no gerenciamento de *code smells* quanto na adaptação ao ambiente de desenvolvimento e facilidade de uso).

Referências

ALVES, N. S., MENDES, T. S., DE MENDONC¸A, M. G., SPíNOLA, R. O., SHULL, F., AND SEAMAN, C. *Identification and management of technical debt: A systematic mapping study.*: Information and software technology. [S.l.], 2016. 70:100–121 p. Citado na página 4.

APEL, SVEN, DON BATORY, CHRISTIAN KäSTNER, AND GUNTER SAAKE. *Feature-Oriented Software Product Lines.*: Springer. doi:10.1007/978-3-642-37521-7. [S.l.], 2013. Citado 2 vezes nas páginas 20 e 21.

APRIL. Studying supply and demand of software maintenance and evolution services. 2010. Citado na página 17.

ARCELLI C. TOSI, M. Z. F.; MAGGIONI., S. The marple project: A tool for design pattern detection and software architecture reconstruction. *In 1st International Workshop on Academic Software Development Tools and Techniques (WASDeTT-1)*, 2008. Citado 2 vezes nas páginas 4 e 31.

BASILI VICTOR R., F. S. F. L. Building knowledge through families of experiments. *IEEE Transactions on Software Engineering*, 1999. Citado na página 42.

BATORY, DON. *Feature Models, Grammars, and Propositional Gormulas.*: In international systems and software product line conference (splc), springer. doi:10.1007/11554844$_3$.[*S.l.*], 2005.7˘20 *p.Citadonapágina*20.

BERGER, THORSTEN, DANIELA LETTNER, JULIA RUBIN, PAUL GRüNBACHER, ADELINE SILVA, MARTIN BECKER, MARSHA CHECHIK, AND KRZYSZTOF CZARNECKI. *"What Is a Feature? A Qualitative Study of Features in Industrial Software Product Lines."*: In international conference on software product line (splc), acm. doi:10.1145/2791060.2791108. [S.l.], 2015. 16–25 p. Citado na página 20.

BIGGERSTAFF, T. J.; MITBANDER, B. G.; WEBSTER, D. The concept assignment problem in program understanding. In: IEEE COMPUTER SOCIETY PRESS. *Proceedings of the 15th international conference on Software Engineering*. [S.l.], 1993. p. 482–498. Citado na página 21.

BROWN, N.; NORD, R.; OZKAYA, I. *Enabling agility through architecture.*: Crosstalk,citeseer. [S.l.], 2010. 13 p. Citado na página 7.

BROWN Y. CAI, e. a. N. Managing technical debt in software-reliant systems. *Proceedings of the FSE/SDP workshop on Future of software engineering research*, 2010. Citado 3 vezes nas páginas 1, 2 e 24.

BUSCHMANN., F. To pay or not to pay technical debt. software. *IEEE, 28(6)*, p. 29–31, 2011. Citado na página 11.

BUSE, R. P. L.; ZIMMERMANN. *Information needs for software development analytics.*: 34th international conference on software engineering (icse).doi: 10.1109/icse.2012.6227122. [S.l.], 2012. 987–996 p. Citado na página 4.

C. FERNáNDEZ-SáNCHEZ, H. HUMANES, J. GARBAJOSA, AND J. DíAZ. *An open tool for assisting in technical debt management.*: In software engineering and advanced applications (seaa), 43rd euromicro conference on. [S.l.], 2017. 400–403 p. Citado 2 vezes nas páginas 27 e 28.

CAMPBELL, G.; PAPAPETROU, P. P. *SonarQube in action*. [S.l.]: Manning Publications Co., 2013. Citado na página 28.

CHAIKALIS, T. et al. Seagle: Effortless software evolution analysis. In: IEEE. *IEEE International Conference on Software Maintenance and Evolution (ICSME), 2014*. [S.l.], 2014. p. 581–584. Citado na página 16.

CHIKOFSKY, E. J.; CROSS, J. H. Reverse engineering and design recovery: A taxonomy. *IEEE software*, IEEE, v. 7, n. 1, p. 13–17, 1990. Citado na página 53.

CLASSEN, ANDREAS, PATRICK HEYMANS, AND PIERRE-YVES SCHOBBENS. *What's in a Feature: A Requirements Engineering Perspective.*: In international conference on fundamental approaches to software engineering, springer. doi:10.1007/978-3-540-78743-3$_2$.[*S.l.*], 2008.16˘30 *p.Citadonapágina*20.

COCKBURN, A. Writing effective use cases, the crystal collection for software professionals. *Addison-Wesley Professional Reading*, 2000. Citado na página 20.

CODABUX, Z., WILLIAMS, B. *Managing technical debt: an industrial case study.*: In: 2013 4th international workshop on managing technical debt (mtd). [S.l.], 2013. 8–15 p. Citado na página 10.

COMMITTEE, I. C. S. S. E. T. Ieee standard glossary of software engineering terminology. In: INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS. [S.l.], 1983. Citado na página 53.

CUNNINGHAM, W. The wycash portfolio management system. In: *Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications (Addendum)*. New York, NY, USA: ACM, 1992. (OOPSLA '92), p. 29–30. ISBN 0-89791-610-7. Disponível em: <http://doi.acm.org/10.1145/157709.157715>. Citado na página 1.

DIT, B. et al. Feature location in source code: a taxonomy and survey. *Journal of software: Evolution and Process*, Wiley Online Library, v. 25, n. 1, p. 53–95, 2013. Citado na página 21.

D'AMBROS, M., GALL, H., LANZA, M., AND PINZGER, M. *Analysing software repositories to understand software evolution*. [S.l.], 2008. Citado na página 16.

E. ALLMAN. *Managing technical debt.*: Communications of the acm. [S.l.], 2012. 50–55 p. Citado 2 vezes nas páginas 2 e 7.

EISENBARTH, T.; KOSCHKE, R.; SIMON, D. Locating features in source code. *IEEE Transactions on software engineering*, IEEE, v. 29, n. 3, p. 210–224, 2003. Citado na página 20.

F. VANDERSON M. A., PEDRO A. S. NETO, WERNEY A. L. LIRA AND IRVAYNE M. S. IBIAPINA. *Analysis of Code Familiarity in Module and Functionality Perspectives*: Sbqs 2018: Xvii simpósio brasileiro de qualidade de software. [S.l.], 2018. Citado na página 34.

FAYYAD, U.; PIATETSKY-SHAPIRO G.; SMYTH P. *The KDD process for extracting useful knowledge from volumes of data.*: Communications of the acm, new york, v. 39, n. 11, nov. [S.l.], 1996. Citado na página 13.

FOWLER, M. , BECK, K. , BRANT, J. , OPDYKE, W. , ROBERTS, D. *Refactoring: Improving the Design of Existing Code*: first ed. addison-wesley professional, reading, ma . [S.l.], 1999. Citado 2 vezes nas páginas 10 e 11.

GALL, H. C.; LANZA, M. Software evolution: analysis and visualization. In: ACM. *Proceedings of the 28th international conference on Software engineering*. [S.l.], 2006. p. 1055–1056. Citado na página 16.

GONG, Z.; LYU., F. Technical debt management in a large-scale distributed project: An ericsson case study. 2017. Citado na página 10.

GRAVES, T. L. et al. Predicting fault incidence using software change history. *IEEE Transactions on software engineering*, IEEE, v. 26, n. 7, p. 653–661, 2000. Citado na página 16.

GRIFFITH, I., IZURIETA, C., TAFFAHI, H., CLAUDIO, D. *A simulation study of practical methods for technical debt management in agile software development*: in proceedings of the 2014 winter simulation conference, piscataway, nj, usa. [S.l.], 2014. Citado na página 2.

GRUBB, P., TAKANG, A. *Software Maintenance: Concepts and Practice.World Scientific*. [S.l.], 2003. Citado na página 17.

GUO, Y.; SEAMAN. *A portfolio approach to technical debt management*: In: Acm.proceedings of the 2nd workshop on managing technical debt.[s.l.]. [S.l.], 2011. 31–34 p. Citado na página 24.

GUO, Y., SPíNOLA, R.O., SEAMAN, C. *Exploring the costs of technical debt management – a case study,*: Empirical software engineering, v. 1,. [S.l.], 2014. 1-24 p. Citado na página 2.

HASSAN, A. E. *The road ahead for mining software repositories.*: In frontiers of software maintenance. [S.l.], 2008. 48–57 p. Citado na página 13.

HASSAN, A. E. The road ahead for mining software repositories. In: IEEE. *Frontiers of Software Maintenance, 2008. FoSM 2008.* [S.l.], 2008. p. 48–57. Citado 2 vezes nas páginas 14 e 53.

HASSAN, A. E. The road ahead for mining software repositories. *In Frontiers of Software Maintenance*, p. 48–57, 2008. Citado na página 15.

HASSAN, A. E. Predicting faults using the complexity of code changes. In: IEEE COMPUTER SOCIETY. *Proceedings of the 31st International Conference on Software Engineering*. [S.l.], 2009. p. 78–88. Citado na página 16.

HASSAN, A. E.; HOLT, R. C. The top ten list: Dynamic fault prediction. In: IEEE. *Proceedings of the 21st IEEE International Conference on Software Maintenance, 2005. ICSM'05.* [S.l.], 2005. p. 263–272. Citado na página 16.

HASSAN, A. E.; XIE, T. *Software Intelligence: The Future of Mining Software Engineering Data.*: Proceedings of the fse/sdp workshop on future of software engineering research. foser '10.new york, ny, usa:acm. doi: 10.1145/1882362.1882397. [S.l.], 2010. 161–166 p. Citado 2 vezes nas páginas 13 e 14.

HATA, H.; MIZUNO, O.; KIKUNO, T. Bug prediction based on fine-grained module histories. In: IEEE PRESS. *Proceedings of the 34th International Conference on Software Engineering.* [S.l.], 2012. p. 200–210. Citado na página 16.

HEMMATI, H. et al. The msr cookbook: Mining a decade of research. In: IEEE. *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on.* [S.l.], 2013. p. 343–352. Citado na página 14.

HEMMATI, H., NADI, S., BAYSAL, O., KONONENKO, O., WANG, W., HOLMES, R., AND GODFREY,M. W. *he msr cookbook: Mining a decade of research. In Mining Software Repositories (MSR)*: Ieee working conference on. [S.l.], 2013. 343–352 p. Citado na página 15.

HILL, E. et al. Which feature location technique is better? In: IEEE. *Software Maintenance (ICSM), 2013 29th IEEE International Conference on.* [S.l.], 2013. p. 408–411. Citado na página 21.

HOLVITIE, J.; LEPPäNEN., V. Debtflag: Technical debt management with a development environment integrated tool. *In Managing Technical Debt (MTD), 2013 4th International Workshop on, pages 20–27. IEEE, 2013.*, 2013. Citado 2 vezes nas páginas 4 e 25.

ISO/IEC-14764. Software engineering - software life cycle processes - maintenance. *International Standard ISO/IEC/IEEE 14764*, 2006. Citado 3 vezes nas páginas 16, 17 e 18.

JUZGADO N. J., M. A. M.; VEGAS, S. Reviewing 25 years of testing technique experiments. *Empirical Software Engineering,9(1-2)*, p. 7–44, 2004. Citado 2 vezes nas páginas 4 e 43.

KAGDI H.; MALETIC, J. I. S. B. Mining software repositories for traceability links. *15th IEEE International Conference on Program Comprehension,ICPC '07. doi: 10.1109/ICPC.2007.28*, p. 145—-154, 2007. Citado na página 15.

KANG, KYO CHUL, SHOLOM G. COHEN, JAMES A. HESS, WILLIAM E. NOVAK, AND A. SPENCER PETERSON. *Feature-Oriented Domain Analysis (FODA) Feasibility Study.*: Pittsburgh. [S.l.], 1990. Citado na página 20.

KERSTEN, M.; MURPHY, G. C. Mylar: a degree-of-interest model for IDEs. *4th international conference on Aspectoriented software development (AOSD)*, p. 159–168, 2005. Citado na página 4.

KIM, M. et al. An empirical study of code clone genealogies. In: ACM. *ACM SIGSOFT Software Engineering Notes.* [S.l.], 2005. v. 30, n. 5, p. 187–196. Citado na página 16.

KIM, S. et al. Predicting faults from cached history. In: IEEE COMPUTER SOCIETY. *Proceedings of the 29th international conference on Software Engineering*. [S.l.], 2007. p. 489–498. Citado na página 16.

KLINGER, T. e. a. An enterprise perspective on technical debt. *In: ACM. Proceedings of the 2nd Workshop on Managing Technical Debt.*, 2011. Citado na página 24.

KOCH, S. Software evolution in open source projects?a large-scale investigation. *Journal of Software Maintenance and Evolution: Research and Practice*, Wiley Online Library, v. 19, n. 6, p. 361–382, 2007. Citado na página 16.

KRUCHTEN, P.; NORD, R. L.; OZKAYA, I. Technical debt: From metaphor to theory and practice. *Ieee software*, IEEE, v. 29, n. 6, p. 18–21, 2012. Citado na página 2.

KRüGER, JACOB, JENS WIEMANN, WOLFRAM FENSKE, GUNTER SAAKE, AND THOMAS LEICH. *Do You Remember This Source Code?*: In international conference on software engineering (icse). acm. doi:10.1145/3180155.3180215. [S.l.], 2018. 16–25 p. Citado na página 21.

LEHMAN, M. M. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE,vol. 68, no. 9*, p. 1060—-1076, 1980. Citado 2 vezes nas páginas 17 e 19.

LEHMAN, M. M. Laws of software evolution revisited. *Software process technology.[S.l.]:*, p. 108—124, 1996. Citado na página 17.

LI, P. A. Z.; LIANG., P. Software aging. In: *A systematic mapping study on technical debt and its management.* Journal of Systems and Software 101,Supplement C (2015), 2015. (ICSE '94), p. 193—220. Disponível em: <https://doi.org/10.1016/j.jss.2014.12.027>. Citado na página 4.

LI, Z. , LIANG, P. , AVGERIOU, P. *Architectural technical debt identification based on architecture decisions and change scenarios.*: In: Proc. 12th work. ieeeifip conf. softw. archit. wicsa. [S.l.], 2015. Citado 2 vezes nas páginas 9 e 10.

LIRA, W. A. L. *Um método para inferência da familiaridade de código em projetos de software*. Dissertação (Mestrado) — Universidade Federal do Piauí, Teresina, 9 2016. Citado 5 vezes nas páginas 3, 34, 36, 53 e 54.

M. M. LEHMAN. *Program evolution and its impact on software engineering*: Ieee computer society press. [S.l.], 1976. 350–357 p. Citado na página 16.

MARTINI, A.; BOSCH, J. The magnificent seven: towards a systematic estimation of technical debt interest. *In Proceedings of the XP2017 Scientific Workshops, page 7. ACM,*, 2017. Citado na página 4.

MARTINI, A.; BOSCH, J. The magnificent seven: towards a systematic estimation of technical debt interest. *In Proceedings of the XP2017 Scientific Workshops, page 7. ACM,*, 2017. Citado na página 4.

MARTINI, A.; BOSCH., J. The magnificent seven: towards a systematic estimation of technical debt interest. *In Proceedings of the XP2017 Scientific Workshops, page 7. ACM,*, 2017. Citado na página 25.

MCCONNELL, STEVE. *"Technical Debt"*: Capturado em: http://www.construx.com/10x$_s oftware_d evelopment/technical_d ebt/$.[*S.l.*], 2013.*Citadonapágina*8.

MCIVER J. P.AND CARMINES, E. G. Unidimensional scaling. *Sage Publications.*, 1991. Citado na página 50.

MEDEIROS, FLáVIO, CHRISTIAN KäSTNER, MáRCIO RIBEIRO, SARAH NADI, AND ROHIT GHEYI. *The Love/Hate Relationship with the C Preprocessor: An Interview Study.*: In european conference on object-oriented programming (ecoop), edited by john tang boyland, 37:495–518. schloss dagstuhl - leibniz-zentrum fuer informatik. doi:10.4230/lipics.ecoop.2015.495. [S.l.], 2015. Citado na página 21.

MENDES, T. et al. Repositoryminer - uma ferramenta extensível de mineração de repositórios de software para identificação automática de dívida técnica. In: *CBSoft 2017 - Sessao de Ferramentas ()*. [s.n.], 2017. Disponível em: <http://XXXXX/170925.pdf>. Citado 4 vezes nas páginas 12, 28, 29 e 35.

MENDES, T. S. et al. Visminertd: Uma ferramenta para identificaç ao automática e monitoramento interativo de dıvida técnica. 2015. Citado 2 vezes nas páginas 4 e 31.

MOREIRA, LUCAS NETO. *Avaliação de qualidade de software baseada em métricas estáticas: um estudo de caso no Tribunal de Contas da União.*: Unb. [S.l.], 2015. Citado na página 2.

MOSER, R. et al. A model to identify refactoring effort during maintenance by mining source code repositories. In: SPRINGER. *International Conference on Product Focused Software Process Improvement*. [S.l.], 2008. p. 360–370. Citado na página 16.

MOURA, F. et al. Codivision: Uma ferramenta para mapear a divisão do conhecimento entre os desenvolvedores a partir da análise de repositório de código. *Congresso Brasileiro de Software - Cbsoft*, 2016. Citado na página 35.

MÜLLER, H. A. et al. A reverse engineering environment based on spatial and visual software interconnection models. In: ACM. *ACM SIGSOFT Software Engineering Notes*. [S.l.], 1992. v. 17, n. 5, p. 88–98. Citado na página 53.

N. ZAZWORKA, R. O. SPíNOLA, A. V. F. S. AND SEAMAN. *A case study on effectively identifying technical debt*: Ease '13: Proceedings of the 17th international conference on evaluation and assessment in software engineering. [S.l.], 2013. Citado na página 4.

PALOMBA G. BAVOTA, M. D. P. R. O. F.; LUCIA, A. D. Do they really smell bad? a study on developers' perception of bad code smells. *in 2014 IEEE International Conference on Software Maintenance and Evolution*, 2014. Citado na página 23.

R. S. PRESSMAN. *Software engineering: a practitioner's approach. 7. ed.*: [s.l.]: Mcgraw-hill. [S.l.], 2015. Citado na página 1.

RAJLICH, V.; WILDE, N. The role of concepts in program comprehension. In: IEEE. *Program Comprehension, 2002. Proceedings. 10th International Workshop on*. [S.l.], 2002. p. 271–278. Citado na página 21.

ROBLES, G. et al. Estimating development effort in free/open source software projects by mining software repositories: a case study of openstack. In: ACM. *Proceedings of the 11th Working Conference on Mining Software Repositories*. [S.l.], 2014. p. 222–231. Citado na página 16.

ROCHA, ANA REGINA CAVALCANTE DA; MALDONADO, JOSé CARLOS; WEBER, KIVAL CHAVES. *Qualidade de software.* [S.l.], 2001. Citado na página 1.

RUNESON, P. ET AL. *Case study research in software engineering: Guidelines and examples.*: John wiley sons. [S.l.], 2012. Citado 2 vezes nas páginas 41 e 42.

S.A., S. Sonarqube. *Capturado em: http://www.sonarqube.org/, abril 2019.*, 2019. Citado 2 vezes nas páginas 4 e 28.

SAGER, T. et al. Detecting similar java classes using tree algorithms. In: ACM. *Proceedings of the 2006 international workshop on Mining software repositories*. [S.l.], 2006. p. 65–71. Citado na página 16.

SOMMERVILLE. Engenharia de software. *Pearson.*, 2007. Citado na página 17.

SPíNOLA R., e. a. M. T. S.; GONÇALVES, D. P.; GOMES. The wycash portfolio management system. *ACM SIGPLAN OOPS Messenger (Vol. 4, No. 2). ACM. pp. 29-30*, 1992. Citado na página 2.

SWANSON, B. The dimension of maintenance. 1976. Citado na página 18.

TANGSRIPAIROJ, S.; SAMADZADEH, M. H. Organizing and visualizing software repositories using the growing hierarchical self-organizing map. In: ACM. *Proceedings of the 2005 ACM symposium on Applied computing*. [S.l.], 2005. p. 1539–1545. Citado na página 16.

TOM, A. A. E.; VIDGEN, R. An exploration of technical debt. *Journal of Systems and Software*, 2013. Citado 2 vezes nas páginas 2 e 7.

TOM, E.; AURUM, A. VIDGEN, R. B. *A Consolidated Understanding of Technical debt*: in 20th european conference on information systems. [S.l.], 2012. Citado na página 23.

TOM, E.; AURUM, A. VIDGEN, R. B. *An exploration of technical debt*: Journal of systems and software 86(6). [S.l.], 2013. 1498-1516 p. Citado na página 23.

TRIPATHY, P. E NAIK, K. *Software Evolution and Maintenance. A Practitioner's Approach.* [S.l.], 2008. Citado na página 18.

TSANTALIS. Evaluation and improvement of software architecture: Identification of design problems in object-oriented systems and resolution through refactorings. *PhD thesis, University of Macedonia, Thessaloniki*, 2010. Citado na página 26.

W. CUNNINGHAM. *Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications (Addendum)*: W. the wycash portfolio management system. in. New York, NY, USA, 1992. 29–30 p. Citado 3 vezes nas páginas 1, 7 e 8.

WOHLIN, C. et al. *Experimentation in software engineering*. [S.l.]: Springer Science & Business Media, 2012. Citado 4 vezes nas páginas 4, 41, 44 e 51.

XIE, T.; PEI, J. Mapo: Mining api usages from open source repositories. In: ACM. *Proceedings of the 2006 International Workshop on Mining Software Repositories*. [S.l.], 2006. p. 54–57. Citado na página 16.

YAN, Y.; MENARINI, M.; GRISWOLD, W. Mining software contracts for software evolution. In: IEEE. *IEEE International Conference on Software Maintenance and Evolution (ICSME), 2014*. [S.l.], 2014. p. 471–475. Citado na página 16.

YIN, R. K. *Case study research: Design and methods.*: Sage publications. [S.l.], 2014. Citado na página 41.

YONGCHANG, R., TAO, X., ZHONGJING, L. XIAOJI, C. *Software Maintenance Process Model and Contrastive Analysis.* [S.l.], 2011. Citado 2 vezes nas páginas 19 e 39.

YU, L. Indirectly predicting the maintenance effort of open-source software. *Journal of Software Maintenance and Evolution: Research and Practice*, Wiley Online Library, v. 18, n. 5, p. 311–332, 2006. Citado na página 16.

ZAKI MOHAMMED. WONG, L. Data mining techniques. *WSPC/Lecture Notes Series.*, 2003. Citado na página 15.

ZAZWORKA R. O. SPíNOLA, A. V. F. S. N.; SEAMAN, C. A case study on effectively identifying technical debt. *EASE '13: Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*, 2013. Citado na página 1.

Apêndices

APÊNDICE A – Termo de Consetimento de Participação

Consentimento de Participação

Eu declaro ter mais de 18 anos de idade e que concordo em participar de um estudo conduzido pelo pesquisador Ronivon Silva Dias e pelo Prof. Dr. Pedro de Alcântara dos Santos Neto. Este estudo visa investigar os benefícios da identificação/visualização de dívidas técnicas em nível de funcionalidade para softwares em manutenção/evolução.

PROCEDIMENTO

Os pesquisadores conduzirão o estudo consistindo da coleta, análise e relato dos dados do exercício. Eu entendo que não tenho obrigação alguma em contribuir com informação sobre meu desempenho no exercício, e que posso solicitar a retirada de meus resultados do experimento a qualquer momento. Eu entendo também que quando os dados forem coletados e analisados, meu nome será removido dos dados e que este não será utilizado em nenhum momento durante a análise ou quando os resultados forem apresentados.

CONFIDENCIALIDADE

Toda informação coletada neste estudo é confidencial, e meu nome não será identificado em momento algum. Da mesma forma, me comprometo a manter sigilo das tarefas solicitadas e dos documentos apresentados e que fazem parte do experimento.

BENEFÍCIOS, LIBERDADE DE DESISTÊNCIA

Eu entendo que sou livre para realizar perguntas a qualquer momento ou solicitar que qualquer informação relacionada a minha pessoa não seja incluída no estudo. Eu entendo que participo do estudo experimental de livre e espontânea vontade com o único intuito de contribuir para o avanço e desenvolvimento de técnicas, ferramentas e processos para a Engenharia de Software.

Nome :_____

Assinatura:_____

Data _____

APÊNDICE B – Questionario de caracterização dos participantes

QUESTIONÁRIO DE CARACTERIZAÇÃO DOS PARTICIPANTES

1. Tempo de experiência com desenvolvimento em linguagem Java.

   _____

2. Tempo de experiência com desenvolvimento no projeto SIGAA.

   _____

APÊNDICE C – Questionario Pós Experimento

QUESTIONÁRIO PÓS EXPERIMENTO

1. O uso da abordagem ajudou a melhorar a qualidade do seu código produzido.
   ( ) Discordo Parcialmente
   ( ) Discordo Totalmente
   ( ) Não Concordo Nem Discordo
   ( ) Concordo Totalmente
   ( ) Concordo Parcialmente

2. O esforço para realização das tarefas que tinham a presença de dívidas técnicas verificadas na CodVision foi diferente das que não tinham.
   ( ) Discordo Parcialmente
   ( ) Discordo Totalmente
   ( ) Não Concordo Nem Discordo
   ( ) Concordo Totalmente
   ( ) Concordo Parcialmente

3. O fato de estar consciente sobre Dívida Técnica afetou de alguma forma como vocês desenvolvem software.
   ( ) Discordo Parcialmente
   ( ) Discordo Totalmente
   ( ) Não Concordo Nem Discordo
   ( ) Concordo Totalmente
   ( ) Concordo Parcialmente

4. Você acha importante monitorar dívida técnica dentro do processo de desenvolvimento de software.
   ( ) Discordo Parcialmente
   ( ) Discordo Totalmente
   ( ) Não Concordo Nem Discordo
   ( ) Concordo Totalmente
   ( ) Concordo Parcialmente

5. É possível fazer com que este gerenciamento faça parte do processo de desenvolvimento.

( ) Discordo Parcialmente
( ) Discordo Totalmente
( ) Não Concordo Nem Discordo
( ) Concordo Totalmente
( ) Concordo Parcialmente

6. Avaliando de modo geral, indique uma classificação para a importância da contribuição da proposta desenvolvida neste trabalho.
( ) Nem um Pouco Importante
( ) Ligeiramente Importante
( ) Moderadamente Importante
( ) Muito Importante
( ) Extremamente Importante