

Bruno Santos de Lima

Uma Abordagem de Priorização para Apoiar o
Pagamento de Dívida Técnica Auto-Admitida em
Código-Fonte

São José do Rio Preto
2021

Bruno Santos de Lima

Uma Abordagem de Priorização para Apoiar o
Pagamento de Dívida Técnica Auto-Admitida em
Código-Fonte

Dissertação apresentada como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação, junto ao Programa de Pós-Graduação em Ciência da Computação, do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista “Júlio de Mesquita Filho”, Câmpus de São José do Rio Preto.

Financiadora: CAPES

Orientador: Prof. Dr. Rogério Eduardo Garcia

São José do Rio Preto
2021

L732a Lima, Bruno Santos de
Uma Abordagem de Priorização para Apoiar o Pagamento de
Dívida Técnica Auto-Admitida em Código-Fonte / Bruno Santos de
Lima. -- São José do Rio Preto, 2021
85 f. : il., tabs.

Dissertação (mestrado) - Universidade Estadual Paulista (Unesp),
Instituto de Biociências Letras e Ciências Exatas, São José do Rio
Preto
Orientador: Rogério Eduardo Garcia

1. Ciência da Computação. 2. Engenharia de Software. 3. Software
Manutenção. 4. Dívida Técnica. I. Título.

Sistema de geração automática de fichas catalográficas da Unesp. Biblioteca do Instituto de
Biociências Letras e Ciências Exatas, São José do Rio Preto. Dados fornecidos pelo autor(a).

Essa ficha não pode ser modificada.

Bruno Santos de Lima

Uma Abordagem de Priorização para Apoiar o Pagamento de Dívida Técnica Auto-Admitida em Código-Fonte

Dissertação apresentada como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação, junto ao Programa de Pós-Graduação em Ciência da Computação, do Instituto de Biociências, Letras e Ciências Exatas da Universidade Estadual Paulista “Júlio de Mesquita Filho”, Câmpus de São José do Rio Preto.

Financiadora: CAPES

Comissão Examinadora

Prof. Dr. Rogério Eduardo Garcia
UNESP – Câmpus de Presidente Prudente
Orientador

Prof. Dr. Manoel Gomes de Mendonça Neto
UFBA – Salvador – Bahia

Prof. Dr. Danilo Medeiros Eler
UNESP – Câmpus de Presidente Prudente

Presidente Prudente
27 de janeiro de 2021

Dedico este trabalho
aos meus pais, Lucimara e Rivaldo.

Agradecimentos

Primeiramente, agradeço a Deus, pelo direcionamento e pela oportunidade.

Agradeço, principalmente, aos meus pais, Lucimara e Rivaldo, vocês são minha base e meu maior exemplo. Obrigado pelo apoio incondicional em todos os momentos e desafios, sem vocês eu não conseguiria. Obrigado por tudo que me ensinaram, pelos valores transmitidos e por sempre colocarem a educação acima de qualquer coisa.

À minha irmã, Ju, por todo apoio e motivação durante minha caminhada, pela compreensão em muitos momentos que estive distante. Ju, agora finalmente podemos maratonar os episódios atrasados, você sabe quais são.

Ao professor Rogério, pela oportunidade, orientação, paciência e acompanhamento durante o desenvolvimento deste trabalho. Obrigado por todos os ensinamentos e auxílios desde a época de Iniciação Científica até hoje. Você é um exemplo de profissional em que eu espelho seguir, um amigo que a UNESP me apresentou desde o primeiro ano de graduação.

Aos professores Danilo e Ronaldo, por todo auxílio no meu crescimento acadêmico e na minha formação, tanto na graduação quanto pós-graduação, pelo estágio e contribuições, obrigado.

Aos meus amigos da graduação, obrigado por compartilhar risos, angústias e aprendizados, vocês fazem parte dessa trajetória. Aos meus companheiros de Laboratório, principalmente ao Leandro, Wilson, Ingrid e Rafael, obrigado pelos momentos de aprendizado, descontração e apoio, aguardo vocês para mais um café no LaPESA. Ao Jorge, pelo apoio e motivação durante a graduação e pós-graduação, obrigado, amigo.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001, à qual agradeço.

Agradeço a todos que, me apoiaram no decorrer deste trabalho. Obrigado a todos vocês.

“A educação é a arma mais poderosa que você pode usar para mudar o mundo” –
Mandela (2003)

Resumo

*D*ívida Técnica é uma metáfora que descreve a relação custo-benefício entre adiar atividades técnicas de desenvolvimento e suas consequências a longo prazo. A falta de gestão da Dívida Técnica diminui a qualidade interna do software e torna complexa e custosa sua manutenção. Instâncias de Dívida Técnica são denominadas Dívida Técnica Auto-Admitida (DTAA) quando são cometidas de modo intencional e documentadas pelo engenheiro de software por meio de comentários no código-fonte. Diversos estudos exploram a identificação de DTAA, porém faltam abordagens para apoiar a etapa de pagamento, em particular, abordagens que indiquem quais instâncias de DTAA devem ser pagas prioritariamente. Este trabalho apresenta uma abordagem para apoiar a atividade de priorização no pagamento de DTAA. A Abordagem de Priorização desenvolvida concentra-se na criação de associações entre instâncias de DTAA e problemas encontrados no código-fonte, identificados por Análise Estática Automática de código. Os resultados indicam que a utilização dos problemas presentes no código-fonte em conjunto com a descrição dos comentários DTAA possuem maior precisão para estabelecer prioridade entre instâncias de DTAA, se comparado com o uso apenas da descrição dos comentários DTAA. Além disso, a Abordagem de Priorização foi aplicada em diferentes projetos de software por meio de um Estudo Experimental que indicou que a abordagem pode ser utilizada como guia de priorização para a tomada de decisão por parte dos desenvolvedores.

Palavras-chave: Dívida Técnica Auto-Admitida. Priorização de Dívida Técnica. Gestão de Dívida Técnica. Manutenção de Software.

Abstract

*T*echnical Debt is a metaphor that describes the cost-benefit relationship between postponing technical development activities and the consequences of this postponement. The lack of Technical Debt Management impacts the internal quality of the software and makes its maintenance complex and costly. Technical Debt instances are called Self-Admitted Technical Debt (SATD) when they are intentionally committed and documented by the software engineer through comments in the source code. Several studies explore the SATD identification, there are very few approaches on the payment of SATD, in particular, approaches that indicate which SATD instances should be paid as a priority. This dissertation presents an approach to support the prioritization activity in the SATD payment. The Prioritization Approach developed focuses on creating associations between SATD and problems found in the source code, identified by Automatic Static Analysis (ASA). The results indicate that the use of the problems present in the source code, together with the description of the SATD instances have greater precision to establish the priority SATD when compared to the use of only the description of the SATD comments. Also, the Prioritization Approach was applied to different software projects through an Experimental Study, indicating that the Prioritization Approach can be used as a prioritization guide for decision making by developers.

Keywords: Self-Admitted Technical Debt. Technical Debt Prioritization. Technical Debt Management. Software Maintenance.

Lista de Figuras

1.1	Panorama das atividades de apoio a Gestão de Dívida Técnica.	16
2.1	Visão Geral do Modelo de Vocabulário Contextualizado para identificação de DTAA.	24
2.2	Visão Geral da Abordagem de Classificação de DTAA Baseada em Processamento de Linguagem Natural.	26
2.3	Comparação entre as abordagens de identificação de DTAA de Código (a) e de Requisitos (b).	28
2.4	Quantidade ideal de comentários classificados manualmente para treinamento do classificador de entropia máxima para DTAA de Código (a) e de Requisitos (b).	28
2.5	Estrutura geral da técnica de identificação de DTAA baseada em classificação Composta.	29
3.1	Evolução da Dívida Técnica Auto-Admitida em quatro projetos de software. . .	35
3.2	Distribuição dos tipos de Dívida Técnica Auto-Admitida.	37
3.3	Porcentagem de modificações indutoras de defeitos relacionadas com DTAA e não relacionadas com DTAA.	38
3.4	A distribuição dos tempos de vida das DTAA removidas.	40
3.5	Visão geral do esquema de priorização para DTAA.	42
3.6	Distribuição do conjunto de tarefas DTAA complexas entre os quatro projetos de softwares estudados.	47
3.7	Comparativo quantitativo de instâncias DTAA classificadas em tarefas esperadas e aceleradas.	47
4.1	Esquema da Abordagem de Priorização para DTAA.	51
4.2	Instantâneo da interface da ferramenta <i>SATD Detector</i> utilizada para identificar instâncias de DTAA.	52
4.3	Exemplo de <i>Issue</i> identificada no software Apache JMeter: Problema de alta complexidade cognitiva.	54
4.4	Exemplo do princípio de associação entre uma instância de DTAA a uma <i>Issue</i> . . .	55

4.5	Associação entre uma instância de DTAA e <i>Issues</i> por proximidade.	56
4.6	Associação entre uma instância de DTAA e <i>Issues</i> por contexto.	57
4.7	Instantâneo da interface da ferramenta <i>SATDPriority</i>	60
5.1	Distribuição das <i>Issues</i> associadas com instâncias de DTAA por grau de severidade em cada repositório de software.	64
5.2	Visão geral do Estudo Experimental: Verificação da eficácia da abordagem de priorização.	68
5.3	Processo de obtenção das instâncias de DTAA pagas prioritariamente pelos desenvolvedores ao longo do desenvolvimento do software.	69
5.4	Tabela demonstrativa para Verdadeiros positivos (VP), Verdadeiros negativos (VN), Falsos positivos (FP) e Falsos negativos (FN).	70
5.5	Comparação entre a Abordagem Priorização completa, Abordagem associando somente Termos de prioridade e Abordagem associando somente <i>Issues</i> , considerando o valor em média de cada uma das métricas: Precisão, <i>Revocação</i> e <i>F1-Score</i>	73

Lista de Tabelas

1.1	Exemplificação de comentários que indicam DTAA.	17
2.1	Padrões Textuais que incorrem em Dívida Técnica Auto-Admitida.	22
2.2	Relação dos tipos de Dívida Técnica com os substantivos da <i>SEOntological</i> . . .	25
2.3	Exemplos de DTAA de Código e de Requisitos classificadas manualmente. . .	27
2.4	Sumário dos projetos utilizados para extração de comentários.	30
2.5	Comparação entre as principais abordagens de Identificação de DTAA conside- rando a métrica <i>F1-score</i>	31
3.1	Quantidade de comentários DTAA considerando diferentes granularidades. . .	34
3.2	Exemplo de tipos de DTAA classificadas.	36
3.3	Tipo de mudanças no código-fonte ao remover instancias de DTAA.	41
3.4	Abordagem de Priorização de DTAA definida em 6 passos.	42
3.5	Identificadores textuais utilizados para classificar as instâncias de DTAA em tarefas priorizadas.	44
3.6	Exemplo de instâncias de DTAA classificadas em tarefas priorizada.	46
3.7	Comparativo quantitativo de instâncias DTAA classificadas em tarefas princi- pais e secundárias.	46
3.8	Causas plausíveis de DTAA identificadas ao analisar os comentários de código-fonte dos quatro software estudados.	48
4.1	Termos que indicam prioridade para um pagamento futuro de Dívida Técnica Auto-Admitida	55
5.1	Sumário dos repositórios utilizados no processo de Análise Exploratória. . . .	62
5.2	Quantidade de associações com instâncias de DTAA.	62
5.3	Termos de prioridade mais frequentes associadas com instâncias de DTAA. . .	63
5.4	<i>Issues</i> frequentemente associadas com instâncias de DTAA.	66
5.5	Issues de maior severidade associadas com instâncias de DTAA.	67
5.6	Eficácia da Abordagem de Priorização.	71
5.7	Eficácia da Abordagem de Priorização com tratamento de Falsos Positivos (FP). .	72

5.8	Eficácia da Abordagem de Priorização associando somente Termos de prioridade.	73
5.9	Eficácia da Abordagem de Priorização associando somente <i>Issues</i>	73

Sumário

1	Introdução	14
1.1	Contexto	14
1.1.1	Gestão de Dívida Técnica	15
1.1.2	Dívida Técnica Auto-Admitida	16
1.2	Problema e Motivação	17
1.3	Objetivos	18
1.4	Organização do Trabalho	18
2	Identificação de Dívida Técnica Auto-Admitida	20
2.1	Considerações Iniciais	20
2.2	Identificação baseada em Padrões Textuais	20
2.3	Identificação baseada em Processamento de Linguagem Natural	26
2.4	Identificação baseada em Classificação Composta	29
2.5	Considerações Finais	32
3	Análise e Priorização de Dívida Técnica Auto-Admitida	33
3.1	Considerações Iniciais	33
3.2	Compreensão e Evolução de DTAA	33
3.3	Tipos de DTAA	36
3.4	Impacto causado pela presença de DTAA	37
3.5	Remoção de DTAA	39
3.6	Juros incorridos por DTAA	41
3.7	Atividade de Priorização de DTAA	41
3.8	Considerações Finais	48
4	Abordagem de Priorização para Dívida Técnica Auto-Admitida	50
4.1	Considerações Iniciais	50
4.2	Abordagem de Priorização	50
4.2.1	Identificação de DTAA	51
4.2.2	Identificação de <i>Issues</i>	53

4.2.3	Associação de Termos de Prioridade	54
4.2.4	Associação de <i>Issues</i> por contexto	54
4.2.5	Equação e Lista de Prioridade	58
4.3	SATDPriority	59
4.4	Considerações Finais	59
5	Análises e Estudo Experimental	61
5.1	Considerações Iniciais	61
5.2	Análise Exploratória	61
5.2.1	Associações: Análise Quantitativa	62
5.2.2	Termos de Prioridades frequentemente associados com DTAA	63
5.2.3	Severidade: Análise Quantitativa	63
5.2.4	<i>Issues</i> frequentemente associadas com DTAA	64
5.2.5	<i>Issues</i> de maior severidade associadas com DTAA	65
5.3	Estudo Experimental: Eficácia da Abordagem de priorização	68
5.4	Considerações Finais	74
6	Considerações Finais	75
6.1	Contribuições	76
6.2	Limitações e Ameaças à Validade do Estudo	76
6.3	Disseminação do Trabalho	77
6.4	Trabalhos Futuros	77
	Referências	79

Introdução

1.1 Contexto

Dívida Técnica é uma metáfora que descreve a relação custo-benefício entre o adiamento de atividades técnicas de desenvolvimento de software para obter vantagens imediatas ("lucros") e as consequências desse adiamento a longo prazo (Brown et al., 2010, Kruchten et al., 2012). Avgeriou et al. (2016) definiram Dívida Técnica como uma coleção de construções de projeto ou implementações convenientes a curto prazo, mas que podem tornar as mudanças futuras custosas ou difíceis de serem realizadas. O impacto causado pela presença de Dívida Técnica afeta à qualidade interna do sistema, sua manutenibilidade e evolução.

O acúmulo de Dívida Técnica resulta em um aumento do custo de desenvolvimento e problemas severos de qualidade de software (Seaman et al., 2015). Por consequência, prejuízos no processo de manutenção, seja ela evolutiva ou corretiva. Em algumas situações, a manutenção torna-se custosa e complexa, inviabilizando o uso do sistema, demandando o desenvolvimento de uma nova aplicação, o que caracteriza um possível dispêndio para a organização.

Na perspectiva econômica, Nugroho et al. (2011) definem Dívida Técnica como o custo necessário para solucionar um conjunto de problemas intrínsecos ao software, no intuito de alcançar o nível de qualidade almejado pela organização para o produto de software. Assim como as dívidas financeiras, Dívida Técnica também resulta em *Juros* que precisam ser pagos futuramente no processo de desenvolvimento. *Juros* são definidos como a diferença entre o custo de manutenção esperado, ou ideal, e o custo extra de manutenção, ocasionado pela presença da Dívida Técnica (Nugroho et al., 2011, Kruchten et al., 2013).

Inicialmente, a metáfora de Dívida Técnica teve seu escopo limitado a problemas existentes no código-fonte (Cunningham, 1992). Posteriormente, a metáfora teve seu escopo expandido passando a englobar também os diferentes estágios de desenvolvimento e os diferentes artefatos de software produzidos – por exemplo: um Documento de Requisitos incompleto ou inconsistente caracteriza uma Dívida de Requisito podendo resultar também em uma Dívida

de Documentação, a falta de Plano de Teste ou Casos de Testes desatualizados caracteriza Dívida de Teste (Guo e Seaman, 2011, Alves et al., 2016). Outros aspectos como usabilidade, versionamento, pessoas e serviços, foram considerados e novos tipos de Dívida Técnica foram definidos (Rios et al., 2018).

Na perspectiva de código-fonte, a Dívida Técnica é descrita como código que infringe boas práticas ou regras de codificação. Códigos de baixa qualidade, duplicados, complexos e de frágil legibilidade, são exemplos de instâncias de Dívida Técnica presentes no código-fonte (Li et al., 2015). Além disso, a Dívida Técnica de projeto e de arquitetura são afetadas por problemas encontrados no código-fonte e podem ser encontradas na forma de *Code smells*, classes ou métodos complexos, violações de modularidade, bem como dos demais princípios de Orientação a Objetos (Rios et al., 2018).

Uma maneira de identificar os problemas que incorrem em Dívida de Código é por meio da Análise Estática Automática de código-fonte (Izureta et al., 2012). Esse tipo de análise procura por violações de práticas de programação recomendadas que causam falhas ou degradam em algum âmbito a qualidade do software (Jesus e Melo, 2017, Digkas et al., 2018, Martini et al., 2018). A partir da Análise Estática de Código são listados os problemas identificados no código-fonte, contudo, não é possível revelar se os problemas identificados foram causadas intencionalmente, ou seja, esse tipo de análise não é capaz de identificar quais Dívidas de Código foram cometidas estrategicamente, caracterizando uma Dívida Técnica intencional.

1.1.1 Gestão de Dívida Técnica

A gestão eficaz da Dívida Técnica é fundamental para alcançar e manter a qualidade do software, já a falta de gerenciamento e monitoramento resulta em problemas a longo prazo, como o aumento da complexidade na etapa de manutenção e consequentemente no custo do projeto (Brown et al., 2010).

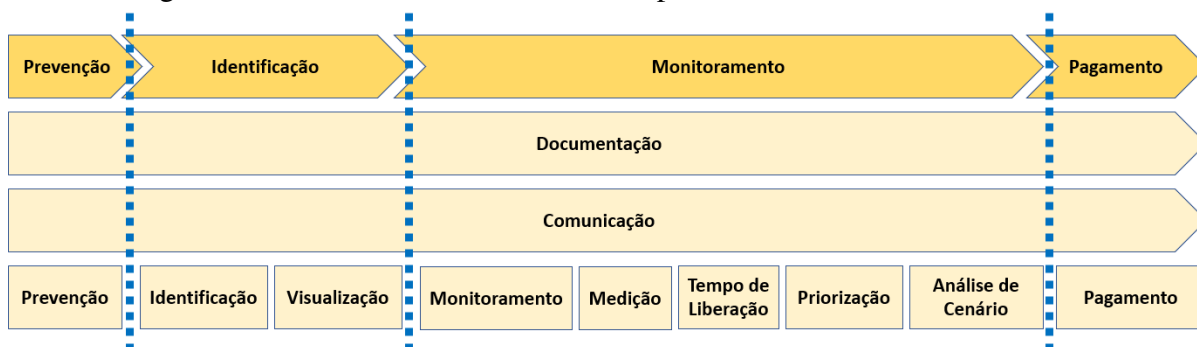
Desse modo, considerar informações acerca de Dívida Técnica para subsidiar a tomada de decisão sobre o pagamento de Dívida Técnica torna-se importante. Essas informações podem ser obtidas utilizando técnicas de identificação e ferramentas apropriadas (Seaman et al., 2015). Uma vez identificadas as instâncias de Dívida Técnica presentes no projeto, decisões precisam ser tomadas para indicar quais instâncias da Dívida Técnica devem ser pagas em um primeiro momento. Uma estratégia que facilita a tomada de decisão é a listagem de Dívidas Técnicas (Guo et al., 2011).

Guo et al. (2016) explicam que o principal objetivo da Gestão de Dívidas Técnica é possibilitar a tomada de decisões sobre a necessidade de eliminar uma instância de Dívida Técnica e o momento apropriado para fazê-lo. O Gerenciamento de Dívida Técnica concentra-se em reduzir o impacto negativo resultante da presença de instâncias da dívida e é um fator decisivo para o sucesso dos projetos de software (Guo e Seaman, 2011). A Gestão de Dívida Técnica é composta por um conjunto de atividades principais (Griffith et al., 2014, Rios et al., 2018), sendo elas: Prevenção, Identificação, Monitoramento e Pagamento.

Além das quatro atividades principais que compõem o Gerenciamento de Dívida Técnica, há 7 atividades de apoio à Gestão de Dívida Técnica, sendo elas: *Documentação*, *Comunicação*,

Medição, Priorização, Análise, Time-to-market e Visualização. As atividades que compõem a gestão de Dívidas Técnicas são organizadas como ilustrado na Figura 1.1. Rios et al. (2018) relatam que as atividades de *Identificação, Monitoramento e Medição* são as que possuem um maior conjunto de ferramentas de apoio e abordagens desenvolvidas. Em contrapartida as demais atividades de gestão, incluindo a de *Priorização*, não apresentam muitas ferramentas e abordagens para auxiliar sua execução e aplicação.

Figura 1.1: Panorama das atividades de apoio a Gestão de Dívida Técnica.



Fonte: Adaptado de Rios et al. (2018).

1.1.2 Dívida Técnica Auto-Admitida

Diferentes fatores e cenários que podem fazer com que desenvolvedores introduzam Dívida Técnica, como por exemplo: prazos apertados, existência de código de baixa qualidade, processo de software inapropriado ou realidade de negócios (Lim et al., 2012).

A Dívida Técnica pode ser introduzida consciente ou inconscientemente. Gerentes de projetos podem optar por não realizar uma atividade técnica causando a introdução consciente da dívida, ou mesmo, programadores inexperientes podem produzir código de baixa qualidade e acabar inconscientemente introduzindo Dívida Técnica. Brown et al. (2010) ressaltam que a Dívida Técnica tem como característica a sua visibilidade, que está relacionada com a introdução consciente ou inconsciente da dívida. No caso do programador inexperiente, produzir inconscientemente código de baixa qualidade, resulta em uma Dívida Técnica invisível perante os gerentes do projeto, os quais necessitam de mecanismos para identificá-la.

A situação em que atividades técnicas de desenvolvimento não são realizadas, ou mesmo correções rápidas ou temporárias são inseridas de modo intencional em um projeto de software (devido a corte de custos ou pressão do mercado), mas documentadas em comentários ao longo do código-fonte, caracteriza a formação da chamada Dívida Técnica Auto-Admitida (DTAA), do inglês *Self-Admitted Technical Debt (SATD)* (Sierra et al., 2019).

O conceito de Dívida Técnica Auto-Admitida (DTAA) foi idealizado por Potdar e Shihab (2014), sendo considerada uma linha de pesquisa recente com necessidade de investigação adicional (Sierra et al., 2019). O estudo em Dívida Técnica Auto-Admitidas torna-se importante, a medida que esse tipo de dívida aparece cada vez com mais frequência no desenvolvimento de software, além de comprovadamente prejudicar o desenvolvimento do mesmo, como revelam estudos presentes na literatura (Potdar e Shihab, 2014, Wehaibi et al., 2016), abordados em

maiores detalhes no Capítulo 3.

Os comentários presentes ao longo do código-fonte são o principal meio dos gestores e desenvolvedores documentar e indicar que cometeram uma Dívida Técnica intencionalmente. Assim, estudos recentes descrevem estratégias para identificar, entender e gerenciar DTAA (Potdar e Shihab, 2014, Bavota e Russo, 2016, Sierra et al., 2019). Na Tabela 1.1 são exemplificadas algumas instâncias de DTAA documentadas por meio de comentários no código-fonte. Em geral, essas instâncias de DTAA expressam a presença de defeitos ou necessidade de melhorias futuras.

Tabela 1.1: Exemplificação de comentários que indicam DTAA.

Exemplos de comentários DTAA
<i>//TODO this is such a hack it is silly</i>
<i>//Unsafe; should error</i>
<i>//FIXME: This is such a gross hack...</i>
<i>//Ugly, but what else?</i>
<i>//TODO: - This method is too complex, lets break it up</i>
<i>//TODO - need a lot more tests</i>
<i>//Copy function is not yet completely implemented</i>
<i>// Bug in above method</i>
<i>//Don't wait around; just abandon it</i>
<i>//Leave it for next release</i>
<i>//Strictly speaking, this is a design error</i>
<i>//DESIGN ERROR: a mix of repositories</i>
<i>//TODO: this isn't quite right but it's ok for now</i>

Fonte: Adaptado de Potdar e Shihab (2014), Maldonado e Shihab (2015) e Mensah et al. (2018).

1.2 Problema e Motivação

De maneira geral, o maior problema na presença da Dívida Técnica é o seu acúmulo. Tal acúmulo é prejudicial para manutenção, além de degradar a qualidade do software. Assim, tornam-se necessários meios que possibilitem o pagamento eficiente das instâncias da Dívida Técnica (Seaman et al., 2015). No contexto de Dívida Técnica Auto-Admitida, é comprovado que trechos de código relacionados com DTAA são mais difíceis de serem modificados se comparado com modificações em trechos de código que não possuem a presença de uma instância de DTAA (Wehaibi et al., 2016). Além disso, um agravante para o problema consiste na falta de estudos e estratégias que possam apoiar a identificação de quais instâncias devem ser pagas prioritariamente para evitar o acúmulo de DTAA.

Embora recentemente Mensah et al. (2018) tenham apresentado uma abordagem com o intuito de atender a essa necessidade, seu estudo é o primeiro encontrado para este fim e nele são consideradas apenas as informações extraídas das descrições dos comentários que indicam DTAA para, então, estabelecer um meio de priorizá-las. A abordagem de priorização existente não observa o código-fonte indicado por uma instância de DTAA e os possíveis problemas nele existentes. Observar as características dos problemas presentes no trecho do código-fonte na

qual foi cometida a DTAA pode ser um fator importante para mensurar a prioridade de uma instância de DTAA na etapa de pagamento de Dívida Técnica, uma vez que alguns problemas são mais prejudiciais do que outros. Além disso, estabelecer uma priorização somente com base na descrição da DTAA torna-se subjetivo ao ponto de que essa descrição pode não refletir o real impacto causado por essa Dívida Técnica e sua real necessidade de resolução.

Durante o desenvolvimento de software é inviável pagar todas as instâncias existentes da Dívida Técnica. Contudo, é necessário manter controlada sua quantidade pagando um conjunto de instâncias com o passar do tempo de desenvolvimento. Porém, algumas instâncias específicas quando pagas podem proporcionar um ganho de qualidade significativo no software ou uma alta melhora em sua manutenção. Saber quais são essas instâncias facilita que elas sejam pagas prioritariamente. Essas perspectivas motivaram a proposta deste trabalho.

1.3 Objetivos

Nos últimos anos, estudos apresentaram abordagens distintas para identificar, compreender e analisar os impactos causados pela DTAA, porém ainda são necessárias abordagens para ajudar gestores a priorizar o pagamento de DTAA.

Mensah et al. (2018) apresentaram a primeira abordagem visando a indicar quais instâncias de DTAA devem ser priorizadas em uma futura etapa de pagamento. Entretanto a abordagem considera apenas as informações extraídas das descrições dos comentários extraídos do código-fonte, que indicam DTAA para, então, por meio das descrições e dos termos presentes nela, estabelecer um esquema de priorização para DTAA. Assim, a abordagem existente não observa o código-fonte indicado por uma DTAA e os possíveis problemas nele existentes e que podem conter informações que influenciem uma possível ordem de priorização.

Nesse sentido, este trabalho teve como objetivo geral auxiliar a priorização de DTAA, de modo que possibilite o pagamento e por consequência favoreça a diminuição do acúmulo de Dívida Técnica. Para isso, é proposta e avaliada uma nova abordagem automatizada para apoiar a atividade de priorização de DTAA. A abordagem de priorização, proposta, concentra-se na criação de associações entre instâncias de DTAA indicadas em comentários e problemas encontrados no código-fonte ressaltados por Análise Estática Automática.

Para isso, tem-se como objetivos específicos: Associar instâncias de DTAA expressas em comentários e os problemas presentes no código-fonte; Comparar as associações estabelecidas em repositórios de softwares distintos identificando possíveis padrões; Avaliar se problemas presentes no código-fonte podem contribuir para a priorização de DTAA; Desenvolver a Abordagem de Priorização; Avaliar/Validar a Abordagem de Priorização desenvolvida.

1.4 Organização do Trabalho

Além deste capítulo de introdução, esta dissertação está organizada nos seguintes capítulos:

- No Capítulo 2 são apresentadas e comparadas as diferentes técnicas de identificação de Dívida Técnica Auto-Admitidas presentes na literatura.

- No Capítulo 3 são descritos estudos que analisam Dívida Técnica Auto-Admitida visando à compreensão de aspectos relacionados a esse fenômeno, como os possíveis impactos, suas características e evolução. Além de expor o esquema de priorização para Dívida Técnica Auto-Admitida presente na literatura.
- No Capítulo 4 é descrita a Abordagem de Priorização embasada em associações para Dívida Técnica Auto-Admitida desenvolvida neste trabalho, além de apresentar a ferramenta *SATDPriority* que materializa a abordagem de priorização.
- No Capítulo 5 são apresentadas as análises dos conjuntos de dados e associações obtidas após a aplicação da Abordagem de Priorização, assim como, um Estudo Experimental que mostra a eficácia da Abordagem de Priorização aplicada em diferentes softwares.
- No Capítulo 6 são apresentadas as conclusões do trabalho desenvolvido, suas contribuições, limitações e trabalhos futuros.

Identificação de Dívida Técnica Auto-Admitida

2.1 Considerações Iniciais

Uma boa prática de programação é documentar o código-fonte, essa documentação pode facilitar possíveis modificações futuras. Em geral, um código-fonte possui comentários aplicados a diferentes contextos, por exemplo: um comentário pode simplesmente descrever o funcionamento de um método e seus parâmetros; pode também informar que um trecho do código-fonte precisa de manutenção corretiva, seja para melhorar sua eficiência, legibilidade ou sanar uma inconsistência; além disso, também pode indicar uma implementação futura.

Nas situações em que um comentário presente no código-fonte indica um possível problema ou mesmo a não realização de uma atividade técnica de desenvolvimento, esse comentário é então caracterizado como uma Dívida Técnica Auto-Admitida (DTAA) (Potdar e Shihab, 2014).

Considerando o conjunto total de comentários presentes no código-fonte de um software nem todos são caracterizados como DTAA. Na literatura são encontradas diferentes técnicas com o propósito de identificar DTAA. Essas técnicas são baseadas em Padrões Textuais, Processamento de Linguagem Natural (PLN), e Classificação Composta. Neste capítulo, são apresentadas as técnicas para identificação de DTAA.

2.2 Identificação baseada em Padrões Textuais

As técnicas de identificação de DTAA baseadas em padrões textuais buscam descobrir quais são os termos textuais e/ou palavras presentes em um comentário de código-fonte que são relacionados à presença de uma instância de DTAA. A partir de um conjunto previamente estabelecido de termos, é possível identificar quais comentários descrevem uma DTAA e quais comentários não indicam uma DTAA.

Potdar e Shihab (2014) apresentam uma das primeiras técnicas baseadas em padrões textuais para identificar DTAA. Visando possibilitar o uso de sua técnica, inicialmente foram extraídos comentários presentes nos seguintes softwares de código-fonte aberto: Eclipse, Chromium OS, ArgoUML e Apache httpd. A extração de comentários foi realizada com o auxílio da ferramenta srcML Toolkit (Collard et al., 2011).

Após a extração, foram lidos manualmente um total de 101.762 comentários de código-fonte para descobrir os padrões que indicam DTAA. Dentre os comentários que indicam DTAA, foram identificados 63 padrões textuais diferentes que, por sua vez, foram utilizados para classificar DTAA em comentários presentes no código-fonte. Os padrões textuais identificados podem indicar problemas, soluções temporárias, inconsistência, entre outros fatores. O estudo de Potdar e Shihab (2014) posteriormente realiza uma análise acerca das instâncias de DTAA identificadas, detalhada no Capítulo 3. Todos os padrões textuais identificados por Potdar e Shihab (2014) são apresentados na Tabela 2.1.

Na técnica apresentada por Potdar e Shihab (2014) é possível identificar se um comentário presente no código-fonte pode indicar uma DTAA, contudo não é identificado qual tipo de Dívida Técnica foi detectada. Ou seja, se é uma DTAA relacionada a teste, documentação, código-fonte, entre outros tipos. Para solucionar essa lacuna, uma melhoria desse processo de identificação foi apresentada por Maldonado e Shihab (2015), sendo possível não somente identificar uma DTAA em um comentário, mas também indicar em qual tipo ela se enquadra.

Em seu estudo, Maldonado e Shihab (2015) extraíram 166 mil comentários referente a cinco projetos de código-fonte aberto pertencentes a diferentes domínios de aplicação, implementados utilizando a linguagem Java, sendo eles: Apache Ant, Apache Jmeter, ArgoUml, Columba e JFreeChart. Para a etapa de extração de comentários foi utilizado a ferramenta JDeodorant (Tsantalis et al., 2008).

Com os comentários extraídos é então aplicado um processo de filtragem. Esse processo tem como objetivo eliminar do montante total, comentários que tem baixa probabilidade de serem classificados como DTAA (Maldonado e Shihab, 2015). O processo de filtragem segue quatro heurísticas idealizadas pelos autores, descritas a seguir:

- **Heurística 1:** Remoção dos comentários que descrevem licenças. Esses comentários são inseridos antes da declaração de uma classe e não indicam DTAA.
- **Heurística 2:** União dos comentários aninhados feitos em linha única, mas que juntos representam um único comentário. Algumas vezes os desenvolvedores fazem comentários longos, usando varias linhas de comentários únicos, ao invés de utilizar um comentário em bloco. Essa heurística elimina esse problema e agrupa os comentários aninhados em um único bloco de comentário.
- **Heurística 3:** Remoção de blocos de códigos comentados. Alguns blocos de códigos são comentados porque são apenas usados para fins de depuração ou porque não são utilizados. Expressões regulares ajudaram a identificar trechos de código Java comentados.

Tabela 2.1: Padrões Textuais que incorrem em Dívida Técnica Auto-Admitida.

Padrões Textuais	
<i>hack</i>	<i>this isn't very solid</i>
<i>retarded</i>	<i>this is temporary and will go away</i>
<i>at a loss</i>	<i>is this line really safe</i>
<i>stupid</i>	<i>there is a problem</i>
<i>remove this code</i>	<i>some fatal error</i>
<i>ugly</i>	<i>something serious is wrong</i>
<i>take care</i>	<i>don't use this</i>
<i>something's gone wrong</i>	<i>get rid of this</i>
<i>nuke</i>	<i>doubt that this would work</i>
<i>is problematic</i>	<i>this is bs</i>
<i>may cause problem</i>	<i>give up and go away</i>
<i>hacky</i>	<i>risk of this blowing up</i>
<i>unknown why we ever experience this</i>	<i>just abandon it</i>
<i>treat this as a soft error</i>	<i>prolly a bug</i>
<i>silly</i>	<i>probably a bug</i>
<i>workaround for bug</i>	<i>hope everything will work</i>
<i>kludge</i>	<i>toss it</i>
<i>fixme</i>	<i>barf</i>
<i>this isn't quite right</i>	<i>something bad happened</i>
<i>trial and error</i>	<i>fix this crap</i>
<i>give up</i>	<i>yuck</i>
<i>this is wrong</i>	<i>certainly buggy</i>
<i>hang our heads in shame</i>	<i>remove me before production</i>
<i>temporary solution</i>	<i>you can be unhappy now</i>
<i>causes issue</i>	<i>this is uncool</i>
<i>something bad is going on</i>	<i>bail out</i>
<i>cause for issue</i>	<i>it doesn't work yet</i>
<i>this doesn't look right</i>	<i>crap</i>
<i>is this next line safe</i>	<i>inconsistency</i>
<i>this indicates a more fundamental problem</i>	<i>abandon all hope</i>
<i>temporary crutch</i>	<i>kaboom</i>
<i>this can be a mess</i>	

Fonte: Adaptado de Potdar e Shihab (2014).

- **Heurística 4:** Remoção de comentários referentes ao Javadoc, uma vez que esses não indicam DTAA.

Maldonado e Shihab (2015) desenvolveram uma ferramenta, implementada em linguagem Java, que aplica as heurísticas propostas automaticamente e permite que o usuário classifique os comentários DTAA quanto ao seu tipo. Um comentário por vez é exibido juntamente com uma lista de possíveis classificações de Dívidas Técnicas que podem ser rotuladas manualmente ao comentário (Maldonado e Shihab, 2015). A lista de classificações possíveis é baseada no trabalho de Alves et al. (2014), sendo elas: *Dívida de Arquitetura*, *Dívida de Compilação*, *Dívida de Código*, *Dívida de Defeito*, *Dívida de Projeto*, *Dívida de Documentação*, *Dívida de Infraestrutura*, *Dívida de Pessoas*, *Dívidas de Processos*, *Dívidas de Requisitos*, *Dívidas de Serviços*, *Divida de Teste* e de *Testes Automatizados*.

Outra técnica baseada em padrões textuais para identificação de DTAA em comentários de código-fonte é apresentada por Farias et al. (2015). Em sua técnica, os autores desenvolveram um Modelo de Vocabulário Contextualizado (CVM-TD) para identificar DTAA. O CVM-TD é estruturado em classes de palavras e *tags* de código para possibilitar a identificação de DTAA. O CVM-TD tem como objetivo a identificação de DTAA, e também indicar o tipo da instância de DTAA identificada, assim com a técnica apresentada por Maldonado e Shihab (2015).

O CVM-TD considera termos que podem indicar má qualidade de código, atividades não executadas por desenvolvedores, problemas que podem relatar um trabalho futuro e uma ontologia para identificar substantivos em Engenharia de Software (Farias et al., 2015) (de Freitas Farias et al., 2020), além de fazer uso dos 62 padrões textuais indicados por (Potdar e Shihab, 2014).

Na técnica de Farias et al. (2015) são considerados termos compostos ao invés de palavras isoladas – isso é justificado pelos autores, uma vez que um termo composto possui informações textuais adicionais em comparação com palavras isoladas. Por exemplo: o termo composto *"TODO: This is a stupid test"* possui mais informações sobre a intenção do desenvolvedor, podendo indicar precisamente uma Dívida Técnica, enquanto a palavra isolada *"stupid"* não apresenta informações suficientes sobre a intenção do desenvolvedor (Farias et al., 2015) (de Freitas Farias et al., 2020). O CVM-TD possibilita, por meio de relações, a especificação de quais palavras isoladas podem aparecer em conjunto.

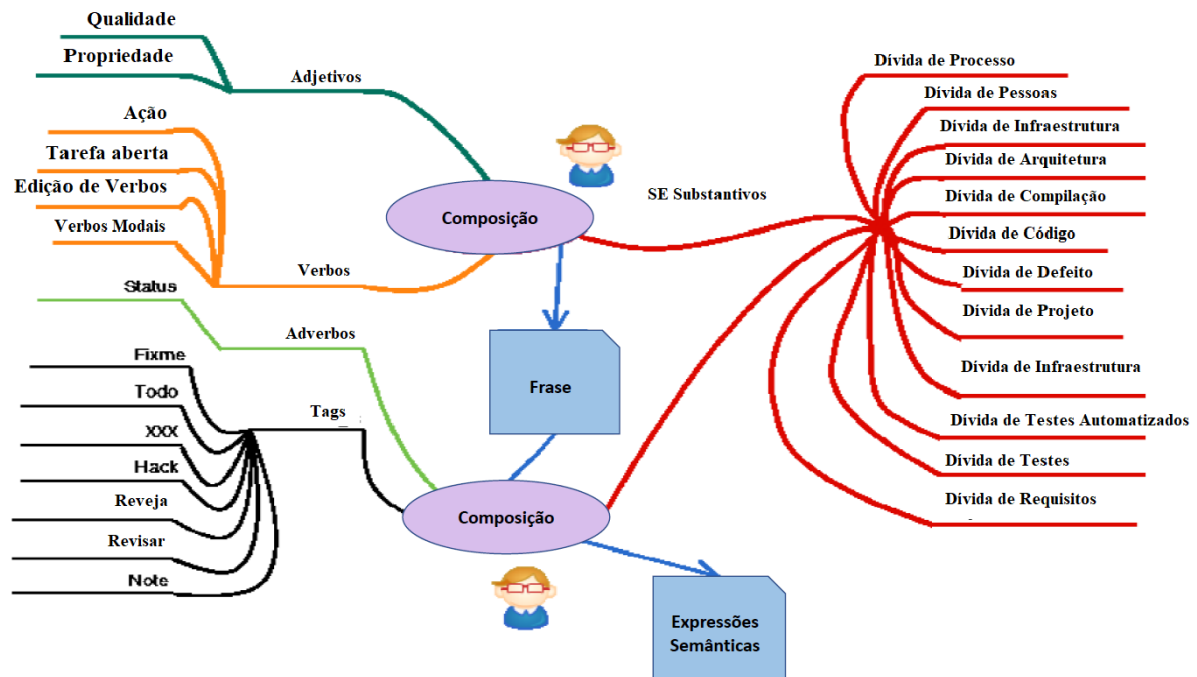
Na Figura 2.1 é ilustrado uma visão geral do CVM-TD, bem como as relações entre as dimensões dos termos, nas quais são compostas por:

- **Substantivos:** São termos relacionados à Engenharia de Software. Eles foram extraídos da ontologia apresentada por Wongthongtham et al. (2009), baseada em domínios de Engenharia de Software, incluindo linguagens de programação, algoritmos, estruturas de dados e decisões de design como padrões de design e arquitetura de software (Farias et al., 2015). Na Tabela 2.2 é apresentado a relação dos tipos de Dívida Técnica e os substantivos da ontologia de Wongthongtham et al. (2009), os substantivos são mapeados para um tipo específico de Dívida Técnica (Alves et al., 2014).
- **Adjetivos:** Um Adjetivo pode ser usado para expressar a qualidade e as propriedades de um substantivo. Exemplos de termos que configuram um adjetivo: *wrong, dangerous, erratic, nuclear, inconsistent, incorrect, obsolete*. Exemplo de termo composto com adjetivo: *"the encoding might be wrong"* (Farias et al., 2015).
- **Verbos:** São usados para indicar ação em uma expressão. Podem denotar atividades específicas de Engenharia de Software, realizadas ou não por desenvolvedores, necessidades ou possibilidades, trabalhos que precisam ser concluídos ou que devem ser feitos. Exemplos de termos que configuram verbos: *add, insert, duplicate, edit, delete, fix, remove, change, create, test, read, modify review, test, debug, specify, implement, must, shall, will, should, would, can, could, may, might, to do, need to, should be, must be,*

can/could/may/might be, would be, should remove, have to e missing. Exemplo de termo composto com verbo: *"We need to refactor the fixtures of tests"* (Farias et al., 2015).

- **Advérbios:** Pode ser usado para indicar o *status* do trabalho. Exemplos de termos que configuram Advérbios: *not, now, nowadays, at the moment, never, instead, only, just, incorrectly, enough e still*. Exemplo de termo composto com uso de advérbios: *"This method incorrectly caches the method structure"* (Farias et al., 2015).
- **Tags:** É um conjunto específico de palavras-chave associadas a alguns tipos de comentários. As *tags* podem ser encontradas em IDE's ou mesmo serem personalizadas pelos desenvolvedores. Os autores do CVM-TD, investigaram na literatura e em um conjunto de software de código-fonte aberto e encontraram 10 *tags* diferentes utilizadas pelos desenvolvedores, sendo elas: *Fixme, Todo, XXX, Bug, Hack, Remind, Review, Revisit, Note e Remark*. Exemplo de termo composto com o uso de *tag*: *"Note: this part of code was not tested as expected"* (Farias et al., 2015).

Figura 2.1: Visão Geral do Modelo de Vocabulário Contextualizado para identificação de DTAA.



Fonte: Adaptado de Farias et al. (2015).

O CVM-TD fornece um vocabulário usado para filtrar comentários que podem indicar uma DTAA caracterizada como um tipo específico de Dívida Técnica (Farias et al., 2015). Assim como ilustrado na Figura 2.1, o CVM-TD utiliza os substantivos, adjetivos, verbos, advérbios e *tags* para realizar a composição de palavras, frases, que indicam uma instância de DTAA.

Para colocar em prática o CVM-TD, os autores desenvolveram uma ferramenta para extração de comentários em código-fonte Java denominada *eXcomment*¹. Dois projetos de software

¹goo.gl/9Mg19m

Tabela 2.2: Relação dos tipos de Dívida Técnica com os substantivos da *SEOntological*.

Tipos de Dívidas Técnicas	Substantivos da <i>SEOntological</i>
Dívida de Arquitetura	<i>Coupling, Aspect, Base, Component, Environment, Gui, Function, architecture, Software, System, Query, package, Procedure, Polymorphism, Library, Interface, Hierarchy, Connector, Layer, Link, Message, Module, Tier, ...</i>
Dívida de Compilação	<i>Artifact, Aspect, Const ruction, Deployment, Environment, Polymorphism, Hierarchy, Compilat ion, Risk, Tool, ...</i>
Dívida de Código	<i>Algorithm, Code, Component, Deployment, Gui, Query, class, method, file, clone, Met ric, Logic, Language, Compilat ion, Declarat ion, Variable, Except ion, Legibility, Loop, Performance, Rapidity, Release, Version, Speed, ...</i>
Divida de Defeito	<i>Bug, Deployment, defect, Error, Except ion, ...</i>
Dívida de Projeto	<i>Component, encapsulation, design, method, metric, code, coupling, Cohesion, Class, clone, Polymorphism, Object, ...</i>
Dívida de Documento	<i>Documentation, Diagram, Use case, Template, Model, Manual, Metadata, Remarks, Survey, ...</i>
Dívida de Infraestrutura	<i>HD, Backup, Channel, Data, Database, infrast ructure, St ructure, Software, System, Screen, Report, Program, Library, Communicat ion, Engineer, Firewall, Hardware, Link, Product, Protocol, Proxy, ...</i>
Dívida de Pessoas	<i>Developer, Programmer, User..</i>
Dívida de Processo	<i>Goal, Business, Analysis, process, Alterat ion, Allocat ion, Template, Result, Paradigm, Measurement, Homologat ion, Cleaning, Communicat ion, Effect iveness, Installat ion, Interact ion, Maintenance, Security, St rategy, ...</i>
Dívida de Requisitos	<i>Scope, Business, Art ifact, Concept ion, Requirements, Specificat ion, ...</i>
Dívida de Serviço	<i>Client, Server, Connect ion, Proxy, Service, ...</i>
Dívida de Teste	<i>Test, Reliability, Precision, Coverage, ...</i>

Fonte: Adaptado de Farias et al. (2015).

tiveram seus comentários extraídos, sendo eles: jEdit 4.3.2 e Apache Lucene 4.3.0. Após a extração, cada comentário passou por um processo automático de classificação, se um comentário possui pelo menos uma palavra-chave do vocabulário proposto então esse comentário era classificado como um indicador de DTAA.

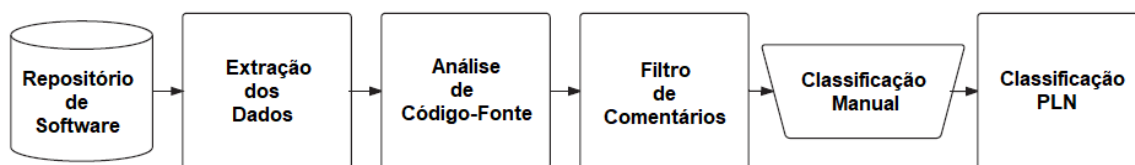
Para validar o modelo proposto, Farias et al. (2015) disponibilizaram o conjunto de comentários DTAA classificados para quatro Engenheiros de Software (todos alunos de mestrado e com experiência em desenvolvimento), solicitando que verificassem se os comentários classificados como DTAA são indicadores ou não de Dívida Técnica.

2.3 Identificação baseada em Processamento de Linguagem Natural

Nos últimos anos, novas técnicas para identificação de DTAA têm sido estabelecidas. Maldonado et al. (2017) propuseram uma técnica que utiliza Processamento de Linguagem Natural (NLP) para detecção automática de DTAA, além de classificar em quais tipos de Dívidas Técnicas elas são caracterizadas. Com o uso de Processamento de Linguagem Natural (NLP) foi possível detectar DTAA de modo mais eficiente que as técnicas descritas anteriormente, baseadas em padrões textuais (Maldonado et al., 2017).

Na Figura 2.2 é ilustrada uma visão geral da abordagem de classificação de DTAA baseada em Processamento de Linguagem Natural (NLP). São executadas quatro etapas: Extração de comentários; Filtragem de comentários seguindo as heurísticas de Maldonado e Shihab (2015); Classificação manual dos comentários em tipos específicos de Dívidas Técnicas; por fim, construção de um classificador treinado com os comentários classificados para identificar DTAA (Maldonado et al., 2017).

Figura 2.2: Visão Geral da Abordagem de Classificação de DTAA Baseada em Processamento de Linguagem Natural.



Fonte: Adaptado de Maldonado et al. (2017).

Foram extraídos comentários de 10 projetos de software de código-fonte aberto, sendo eles: Ant, ArgoUML, Columba, EMF, Hibernar, JEdit, JFreeChart, JMeter, JRuby e SQL Squirrel. Maldonado e Shihab (2015) justificam a escolha desses projetos principalmente por terem uma alta taxa de comentários, medida expressa pelo OpenHub ², pré-requisito para o estudo de DTAA. Para a extração de comentários foi utilizada a ferramenta JDeodorant (Tsantalis et al., 2008).

O conjunto de comentários extraídos passaram por um processo de filtragem para remoção de comentários que por natureza não representam DTAA. A etapa de filtragem segue as heurísticas propostas por Maldonado e Shihab (2015), discutidas na Seção 2.2.

Em seguida, o conjunto de comentários resultante da etapa de filtragem foi submetido ao processo de classificação manual. Cada comentário foi rotulado de acordo com um tipo de Dívida Técnica (Alves et al., 2014). Foram classificados manualmente 62.566 comentários em cinco tipos diferentes de DTAA: 1) Dívida de Projeto; 2) Dívida de Defeito; 3) Dívida de Documentação; 4) Dívida de Requisito; 5) Dívida de Teste. O processo de classificação manual levou aproximadamente 185 horas, e foi realizado pelos próprios autores da técnica (Maldonado et al., 2017). Na Tabela 2.3 são descritas instâncias de DTAA referentes aos dois principais tipos

²www.openhub.net/

identificados: Dívida de Código e de Requisito.

Tabela 2.3: Exemplos de DTAA de Código e de Requisitos classificadas manualmente.

Projeto	DTAA	Tipo de Dívida Técnica
ArgoUML	<i>TODO: - This method is too complex, lets break it up</i>	Dívida de Código
ArgoUML	<i>TODO: really should be a separate class</i>	Dívida de Código
Apache Ant	<i>Probably not the best choice, but it solves the problem of relative paths in CLASSPATH</i>	Dívida de Código
Apache Ant	<i>TODO não há métodos ainda para getClassname</i>	Dívida de Requisito
Apache Ant	<i>TODO nenhum método para newInstance usando um invereclssloader</i>	Dívida de Requisito

Fonte: Adaptado de Maldonado et al. (2017).

O processo de classificação manual é suscetível à erro humano, além de apresentar viés na classificação, o que caracterizam ameaças a validade do estudo. Para minimizar ameaças, os autores extraíram uma amostra estatisticamente significativa do conjunto de dados (nível de confiança de 99% e intervalo de confiança de 5%), que foi classificada manualmente por um estudante e comparada com a classificação manual original feita por Maldonado et al. (2017). O nível de concordância entre os revisores originais e o revisor da amostragem foi calculado utilizando o coeficiente Cohen's Kappa (Cohen, 1960). O nível de concordância ao aplicar o coeficiente de Cohen's Kappa foi de +0,81 (quanto mais próximo de +1 mais forte é a concordância), sendo que, valores superiores a +0,75 são considerados uma excelente concordância (Fleiss e Paik, 2004).

Ao fim da etapa de classificação manual, foi obtido o conjunto de dados utilizado para treinar um classificador (aprendizado supervisionado) visando identificar DTAA por meio de Processamento de Linguagem Natural (NLP) (Maldonado et al., 2017). O classificador é implementado utilizando a linguagem Java e a *Stanford Classifier Toolkit*³, possibilitando a construção de um modelo de entropia máxima (Manning e Klein, 2003).

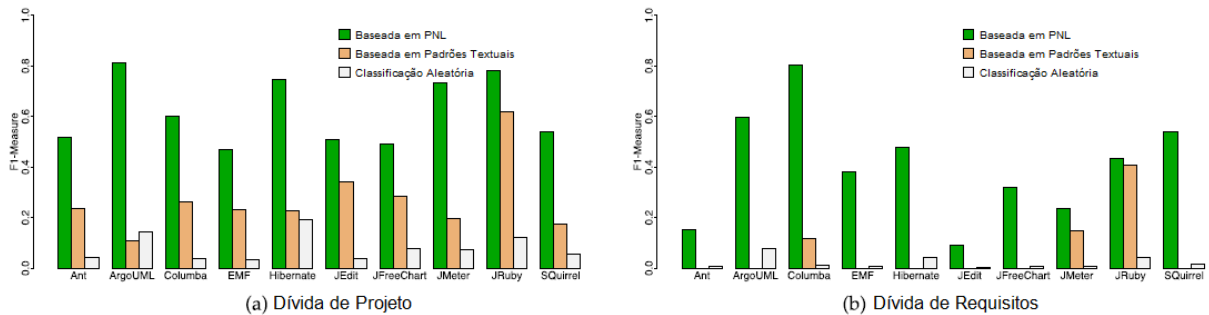
Uma fragilidade dessa técnica consiste no classificador a ser treinado para identificar apenas dois tipos específicos de DTAA, de código e de requisitos. Maldonado et al. (2017) justificaram que os outros tipos de Dívidas Técnicas representam juntas apenas 10% do conjunto de comentários classificados manualmente, o que seria insuficiente para treinar o classificador com o intuito de identificar os outros tipos de Dívidas Técnicas com uma alta precisão.

Apesar da técnica baseada em Processamento de Linguagem Natural implementada por Maldonado et al. (2017) ser capaz de classificar DTAA em apenas dois tipos (Código e Requisitos), a técnica demonstrou um desempenho superior ao das técnicas apresentadas anteriormente (Seção 2.2) para classificar esses dois tipos de DTAA. Na Figura 2.3 é ilustrado um comparativo da precisão para identificar DTAA entre as técnicas de Processamento de Linguagem Natural, Padrões Textuais e Abordagem Aleatória, sendo esta última não discutida, pois segundo Maldonado et al. (2017) consiste simplesmente em aleatoriedade, sem utilizar base científica, para classificar ou não um comentário como uma DTAA. Observando os gráficos ilustrados na Fi-

³nlp.stanford.edu/

gura 2.3 é possível observar o desempenho superior da técnica utilizando Processamento de Linguagem Natural em comparação com a baseada em Padrões Textuais e baseada em aleatoriedade para identificar DTAA de Projeto e de Requisitos, considerando a medida *F1-Score*.

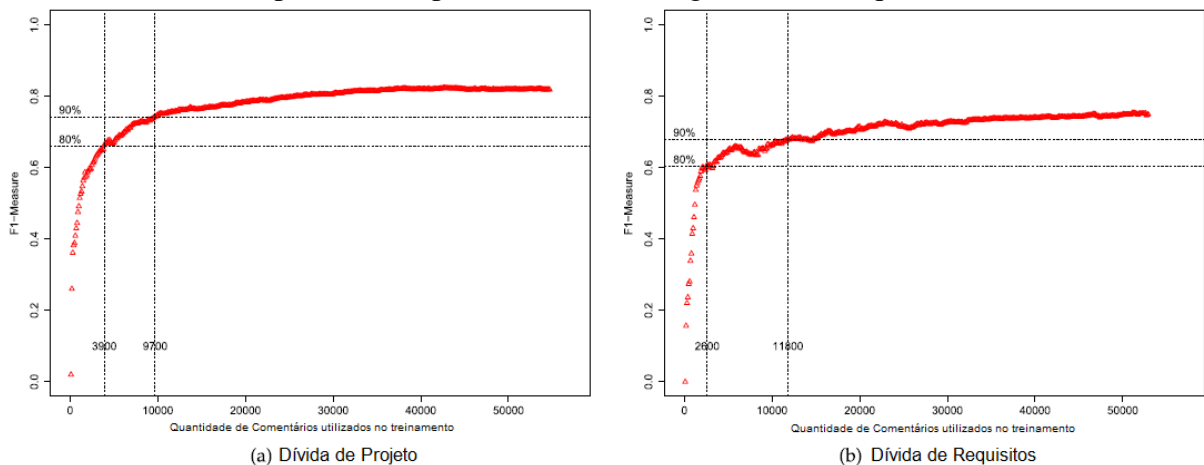
Figura 2.3: Comparação entre as abordagens de identificação de DTAA de Código (a) e de Requisitos (b).



Fonte: Adaptado de Maldonado et al. (2017).

Visto o desempenho promissor da utilização de Processamento de Linguagem Natural para identificar DTAA, Maldonado et al. (2017) investigaram qual seria a quantidade ideal de comentários classificados manualmente para treinar o classificador. Utilizar um conjunto pequeno de comentários classificados não garante uma alta precisão para a detecção, por outro lado utilizar uma quantidade extremamente elevada pode não representar um ganho real de precisão. Analisando os resultados dessa investigação, apresentados na Figura 2.4, foi constatado que após utilizar 9700 comentários para treinamento, a precisão para classificar DTAA de código não possui aumento considerável. Já para classificar DTAA de requisitos após utilizar 11800 comentários a precisão se estabiliza sem ganhos significativos ao aumentar a quantidade de comentários utilizados na etapa de treinamento (Maldonado et al., 2017).

Figura 2.4: Quantidade ideal de comentários classificados manualmente para treinamento do classificador de entropia máxima para DTAA de Código (a) e de Requisitos (b).



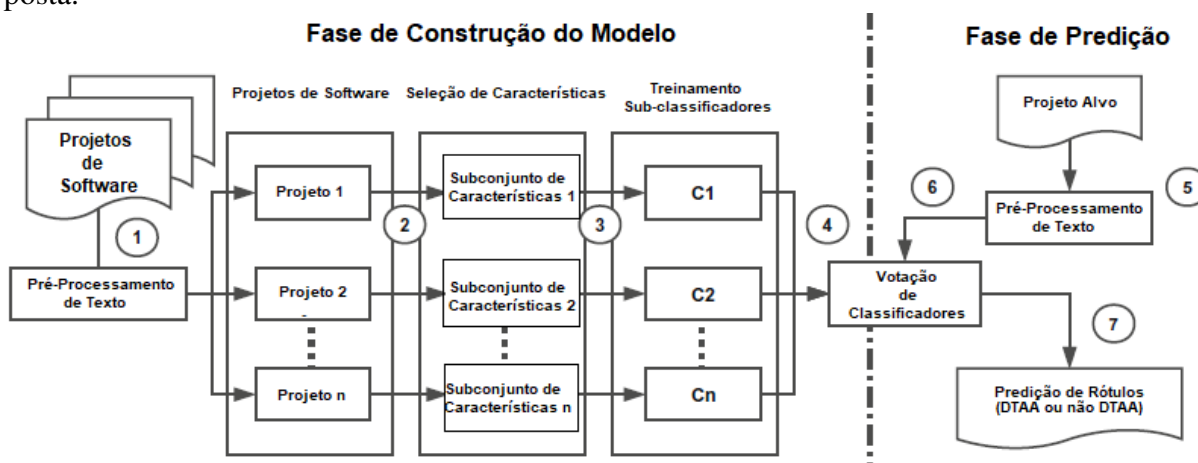
Fonte: Adaptado de Maldonado et al. (2017).

2.4 Identificação baseada em Classificação Composta

Entre as técnicas de identificação de DTAA existentes na literatura, a técnica descrita por Huang et al. (2018) obteve excelentes resultados, que foram superiores as demais técnicas apresentadas em situações nas quais não existe a preocupação em detectar o tipo de DTAA identificada. O propósito da técnica de Huang et al. (2018) é identificar DTAA com a maior precisão possível necessitando de um esforço mínimo para essa atividade. A técnica de Huang et al. (2018) obteve maior precisão e menor demanda de esforço para identificar DTAA se comparado com as outras técnicas de identificação, esse ponto será discutido ao final desta seção.

A técnica de Classificação Composta utiliza a seleção de características visando a separar características úteis para o treinamento de sub-classificadores. Cada sub-classificador é treinado com comentários de um projeto de software específico. A combinação dos sub-classificadores de projetos de origens distintas é utilizada na criação de um modelo de classificação composta, que por sua vez, é utilizado para identificar automaticamente comentários que indicam DTAA em um projeto de software arbitrário. Na Figura 2.5 é ilustrada a estrutura geral dessa técnica de classificação, que possui duas fases: *Construção* e *Previsão*.

Figura 2.5: Estrutura geral da técnica de identificação de DTAA baseada em classificação Composta.



Fonte: Adaptado de Huang et al. (2018).

A fase de *Construção* cria o modelo de classificação, para tornar isso possível, primeiramente, foi utilizado um conjunto de comentários, disponibilizados pelo estudo de Maldonado et al. (2017), classificados manualmente como indicadores de DTAA. Esses comentários foram extraídos de projetos de código-fonte abertos pertencentes a diferentes domínios e com taxas de comentários distintas, segundo o OpenHub⁴ – Tabela 2.4.

Com o conjunto de comentários previamente classificados, foi realizado um pré-processamento dos comentários para extrair características, ou seja, palavras (Figura 2.5 – etapa 1) que passam por uma seleção visando a obter as características úteis para a classificação de DTAA em softwares arbitrários (Figura 2.5 – etapa 2). A seleção de características foi aplicada para melhorar o desempenho da classificação (Hall, 1999).

⁴www.openhub.net/

Tabela 2.4: Sumário dos projetos utilizados para extração de comentários.

Projetos	Domínio	Desenvolvedores	LOC	Taxa de comentários
ArgoUML	UML Modeling Tool	87	926K	Alta
Columba	Email Client	10	155K	Alta
Hibernate	ORM Framework	456	745K	Baixa
JEdit	Text Editor	57	310K	Média
JFreeChart	Char Library	19	317K	Alta
JMeter	Performance Tester	41	354K	Média
JRuby	Ruby Interpreter	374	841K	Baixa
Squirrel	SQL Client	40	708K	Média

Fonte: Adaptado de Huang et al. (2018).

Foi usado o Modelo de Espaço Vetorial (VSM) (Salton et al., 1975) para representar cada comentário com um vetor de palavras. Então foi empregada a técnica de seleção de características para saber o ganho de informação ao selecionar características considerados úteis (Sebastiani, 2002, Yang e Pedersen, 1997).

As características selecionadas são utilizados para treinar os sub-classificadores (Figura 2.5 – etapa 3). Cada sub-classificador é treinado com um conjunto de comentários de um projeto específico, descritos anteriormente – Tabela 2.4. A combinação dos sub-classificadores (C1 até Cn) forma um classificador composto (Figura 2.5 – etapa 4) utilizado na fase de predição.

Para treinar cada sub-classificador é utilizada a técnica *Naive Bayes Multinomial* (NBM), que por sua vez é amplamente usada para mineração de texto (McCallum e Nigam, 1998), além de possuir como vantagem o curto tempo necessário para treinamento computacional (Huang et al., 2018).

Com o classificador treinado é possível utilizá-lo para identificar DTAA em outros projetos de software. Para tal, é realizada a fase de predição, na qual os comentários são extraídos do projeto em que se pretende identificar DTAA, são pré-processados (Figura 2.5 – etapa 5) e inseridos no classificador composto (Figura 2.5 – etapa 6). Como resultado da predição, é apresentado uma lista com os comentários rotulados como indicadores ou não indicadores de DTAA (Figura 2.5 – etapa 7).

Huang et al. (2018) explicam que o processo de classificação funciona como uma eleição, ideia similar à estratégia de Breiman (1996). Cada sub-classificador “vota” indicando se determinado comentário de entrada indica uma DTAA ou não. Após a soma dos votos, o classificador composto apresenta o rótulo final para o comentário de entrada. Por exemplo, considere um Classificador Composto por sete sub-classificadores e um comentário de entrada. Quatro sub-classificadores elegem o comentário de entrada como um indicador de DTAA, os outros três sub-classificadores não consideram o comentário como um indicador de DTAA. A soma dos *votos* é realizada, logo, o Classificador Composto rotula o comentário como um indicador de DTAA.

O desempenho dessa técnica baseada em Classificação Composta é apresentada como superior as demais abordagens para identificar DTAA, segundo os autores da técnica. Huang et al. (2018) compara a técnica de Classificação Composta com as demais técnicas de identificação:

Baseadas em Padrões Textuais e em Processamento de Linguagem Natural (NLP).

Para realizar a comparação entre as técnicas, Huang et al. (2018) utilizam como métrica o *F1-Score*, que é obtida por meio de duas outras métricas estatísticas: Precisão e Revocação. Huang et al. (2018) justificam o uso da métrica *F1-Score* por ser comumente utilizada em trabalhos na área de Engenharia de Software (Arisholm et al., 2007, Rahman et al., 2012, Jiang et al., 2013, Shihab et al., 2013).

Existem quatro possíveis resultados de classificação, são eles: *Verdadeiro Positivo* (TP), comentário classificado como DTAA no qual realmente é uma DTAA; *Verdadeiro Negativo* (TN), comentário classificado como não indicador de DTAA no qual realmente não é uma DTAA; *Falso Positivo* (FP), comentário que é classificado como DTAA quando não é uma DTAA; *Falso Negativo* (FN), comentário classificado como não indicador de DTAA quando na verdade é uma DTAA.

Na Tabela 2.5 são apresentados os resultados da comparação entre as três abordagens para identificação de DTAA, com base na métrica *F1-Score*. A abordagem baseada em Classificação Composta apresenta melhores resultados (*F1-Score* mais próximo de 1.0) para detectar DTAA em um conjunto de projetos alvos, seguida da técnica baseada em Processamento de Linguagem Natural (NLP). A abordagem baseada em Padrões Textuais obteve resultados inferiores (*F1-Score* mais próximo de zero) se contrastado com as outras duas técnicas. Huang et al. (2018) ainda relatam que o consumo de memória utilizando a abordagem de classificação composta é inferior que as demais abordagens para identificação de DTAA.

Tabela 2.5: Comparação entre as principais abordagens de Identificação de DTAA considerando a métrica *F1-score*.

Alvo	Classificador Composto	Padrão Textual	NLP
ArgoUML	0,828	0,054	0,802
Columba	0,801	0,129	0,682
Hibernate	0,788	0,133	0,587
JEdit	0,518	0,304	0,459
JFreeChart	0,687	0,093	0,500
JMeter	0,781	0,093	0,557
JRuby	0,841	0,102	0,584
Squirrel	0,651	0,073	0,435
Média	0,737	0,123	0,576

Fonte: Adaptado de Huang et al. (2018).

Huang et al. (2018) realizaram testes para verificar se a utilização de apenas um classificador treinado com todos os conjuntos de comentários obteria um desempenho superior, se comparado a sua abordagem de sub-classificadores com sistema de votação. Foi constatado que o desempenho para identificação de DTAA utilizando a técnica de Classificação Composta foi superior.

Apesar da técnica baseada em Classificação Composta demonstrar desempenho superior as demais, ela possui como fragilidade o fato de não indicar qual tipo de Dívida Técnica a DTAA identificada está caracterizada. Neste contexto, se for necessário apenas identificar DTAA, a técnica baseada em Classificação Composta situa-se na literatura como a mais apropriada, até

o momento. Contudo, se for necessário além disso saber qual tipo de Dívida Técnica é descrita pela DTAA, então as outras abordagens podem ser mais apropriadas.

O estudo apresentado por Huang et al. (2018) disponibilizou o conjunto de dados e a implementação da técnica de identificação de DTAA baseada em Classificação Composta por meio de um projeto Java ⁵. Entretanto, visando a consolidar e possibilitar a utilização dessa técnica de identificação de maneira prática, posteriormente, foi desenvolvida e disponibilizada uma ferramenta denominada SATD *Detector* ⁶ (Liu et al., 2018).

A SATD *Detector* possibilita a utilização prática da técnica de identificação baseada em Classificação Composta apresentada por Huang et al. (2018). O resultado da classificação é visualizado por meio de uma listagem simples de todos os comentários classificados como instâncias DTAA. Os autores poderiam explorar alternativas visuais para apresentar o resultado da classificação, visto que a visualização de Dívida Técnica é apontada como um campo que ainda necessita de exploração (Alves et al., 2016, Fernández-Sánchez et al., 2017, Rios et al., 2018).

2.5 Considerações Finais

Neste capítulo foram apresentadas as técnicas existentes na literatura voltadas para identificação de Dívida Técnica Auto-Admitida (DTAA). As abordagens para identificação de DTAA são baseadas em: Padrões Textuais (Potdar e Shihab, 2014, Farias et al., 2015, Maldonado e Shihab, 2015), Processamento de Linguagem Natural (NLP) (Maldonado et al., 2017) e por Classificação Composta (Huang et al., 2018).

A técnica baseada em Classificação Composta apresentada por Huang et al. (2018) resulta em excelentes resultados para identificar DTAA, descrita como superior as demais técnicas em precisão e tempo de identificação. Entretanto, sua limitação consiste em não informar qual o tipo de Dívida Técnica a DTAA identificada pode ser caracterizada.

Detectar DTAA pode lembrar os desenvolvedores e gerentes sobre a existência da Dívidas Técnica e assim ajudá-los a planejar seu pagamento de modo eficiente, resultando na melhoria da qualidade do software. Contudo, apesar de identificar de modo eficiente as instâncias de DTAA espalhadas pelo código-fonte, também é fundamental criar abordagens que facilitem a atividade de pagamento. Para isso, é necessário analisar o impacto que essas dívidas podem causar, bem como seu comportamento e característica de evolução. Esses pontos pertinentes à atividade de análise são discutidos em maiores detalhes no próximo capítulo.

⁵github.com/tkdsheep/TechnicalDebt

⁶github.com/Tbabm/SATDDetector-Core

Análise e Priorização de Dívida Técnica Auto-Admitida

3.1 Considerações Iniciais

*E*studos têm sido realizados para compreender DTAA, a compreensão desse fenômeno é obtida por meio da realização de análises que englobam diversos fatores pertinentes à metáfora de Dívida Técnica como, por exemplo, análise do impacto que a presença de DTAA pode causar durante o ciclo de vida do software, estudo da evolução da DTAA em projetos de software, suas características e por quanto tempo instâncias de podem permanecer no software até sua remoção.

Sierra et al. (2019) destacam que obter uma melhor compreensão acerca de DTAA permite que a comunidade científica e profissionais desenvolvam abordagens para gerenciá-las efetivamente, dentre elas, abordagens de priorização e pagamento de DTAA. A seguir são descritos os estudos direcionados à análise de DTAA e as suas principais descobertas, bem como, a primeira e única, até onde levantado na literatura, abordagem para priorização de DTAA em etapas futuras de pagamento.

3.2 Compreensão e Evolução de DTAA

Na literatura, um dos primeiros esforços para compreender a DTAA foi realizado por Potdar e Shihab (2014), autores do termo Dívida Técnica Auto-Admitida e idealizadores da primeira técnica de identificação baseada em Padrões Textuais. Potdar e Shihab (2014) realizaram um estudo exploratório para entender a ocorrência de DTAA, medir a quantidade de instâncias de DTAA existentes em um projeto de software, identificar os motivos para sua introdução e a quantidade de DTAA que é removida após sua inserção.

O conjunto de dados utilizado para essa investigação consiste em comentários classifica-

dos como instâncias de DTAA, identificados em quatro projetos de softwares de código-fonte abertos: Eclipse, Chromium OS, ArgoUML e Apache httpd. A primeira investigação revelou que 2,4% - 31,0% dos arquivos, 0,4% - 3,3% das classes e 0,3% - 2,6% dos métodos/funções contêm DTAA. Isso revela que, considerando diferentes granularidades (arquivo, classe, método/função), a quantidade de comentários DTAA é mínima se comparada com a quantidade total de comentários. Na Tabela 3.1 são sumarizados os resultados obtidos com essa investigação.

Tabela 3.1: Quantidade de comentários DTAA considerando diferentes granularidades.

Tipo	Projeto	Total de Comentários	Comentários DTAA	%
Arquivos	Eclipse	6.389	152	2.4%
	Chromium OS	14.328	574	4.0%
	ArgoUML	1.846	77	4.2%
	Apache httpd	255	79	31.0%
Classes	Eclipse	11.993	151	1.3%
	Chromium OS	18.249	82	0.4%
	ArgoUML	2.304	76	3.3%
	Apache httpd	-	-	-%
Métodos/Funções	Eclipse	48.382	145	0.3%
	Chromium OS	153.514	626	0.4%
	ArgoUML	12.845	71	0.6%
	Apache httpd	4.361	112	2.6%
Construtores	Eclipse	5.561	151	0.2%
	Chromium OS	18.988	82	0.1%
	ArgoUML	1.920	76	0.5%
	Apache httpd	-	-	-%

Fonte: Adaptado de Potdar e Shihab (2014).

Com relação à introdução de DTAA, Potdar e Shihab (2014) investigaram como a experiência dos desenvolvedores, o tempo de liberação do software para uso ou a complexidade das mudanças resultaram na adição de Dívidas Técnicas. Contrariando a premissa de que os desenvolvedores menos experientes geram mais instâncias de DTAA, os autores descobriram que desenvolvedores experientes são os que introduzem a maior parte da DTAA. Os desenvolvedores mais experientes inserem DTAA durante todo o desenvolvimento do projeto. Potdar e Shihab (2014) constataram também que os fatores complexidade ao realizar mudanças e prazos apertados para liberação do software não afetaram a introdução de DTAA.

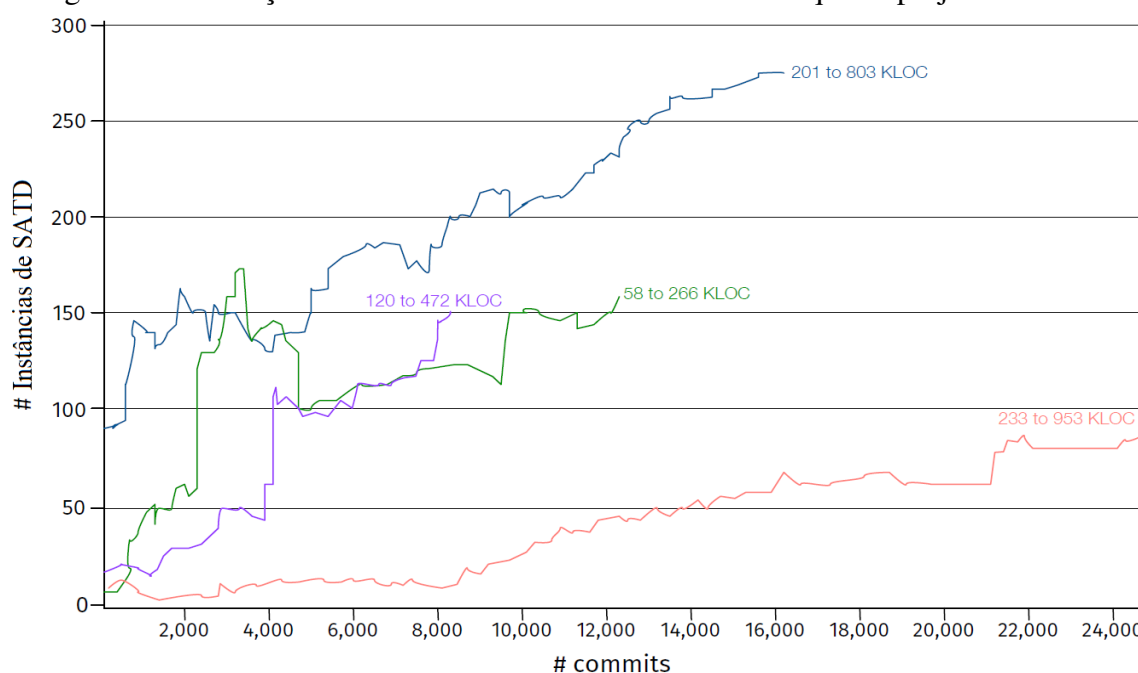
Em relação à remoção da DTAA, Potdar e Shihab (2014) descobriram que de 26,25% a 63,45% da DTAA é removida nas *releases* subsequentes. A maior parte da DTAA é removida imediatamente na próxima *release*. Contudo, a DTAA como um todo, tende a persistir nos projetos de software em várias *releases*.

Posteriormente, Bavota e Russo (2016) realizaram uma replicação do estudo realizado por Potdar e Shihab (2014), em uma perspectiva diferenciada, uma vez que, buscou-se investigar a dimensão e evolução da DTAA e sua relação com a qualidade de software. O estudo foi realizado em larga escala utilizando como base dois ecossistemas de software distintos: Apache (composto por 120 softwares) e Hibernate (composto por 39 softwares), o que resultou em um processo de mineração de mais de 600 mil *commits* e 2 bilhões de comentários.

Bavota e Russo (2016) reafirmaram os resultados descritos por Potdar e Shihab (2014), principalmente no que diz respeito ao pagamento de DTAA, nos quais essas costumam ser removidas pelos desenvolvedores em versões subsequentes do software. Um fato interessante é que na maioria dos casos (63%), o desenvolvedor que paga a Dívida Técnica é o mesmo que a introduziu no projeto de software em versões anteriores.

Além disso, observando como a DTAA evolui durante o histórico de mudanças dos projetos de software foi constatado que, apesar das instâncias de DTAA serem pagas entre uma mudança e outra, a quantidade total de instâncias de DTAA aumenta durante o histórico de modificações dos softwares. Na Figura 3.1 é ilustrada a evolução de quatro projetos de software (Apache CloudStack, Apache Ant, Eclipse PDT e Eclipse JDT) exemplificando essa situação.

Figura 3.1: Evolução da Dívida Técnica Auto-Admitida em quatro projetos de software.



Fonte: Adaptado de Bavota e Russo (2016).

Como último ponto de pesquisa, Bavota e Russo (2016) investigaram empiricamente se os desenvolvedores são mais propensos a introduzir DTAA em componentes de código-fonte que possuem baixa qualidade interna, considerando avaliação por métricas de qualidade. A qualidade interna do código foi considerada sob três aspectos: *Complexidade*, *Acoplamento* e *Legibilidade*, medidos respectivamente pelas métricas: Soma de complexidade ciclomática (McCabe, 1976); Soma de acoplamento entre objetos (Chidamber et al., 1998); Calculo de legibilidade de Buse e Weimer (2010).

Em suma, não foi encontrado nenhuma correlação entre a qualidade interna do código-fonte e o número de instâncias de DTAA. Ou seja, as evidências indicam que a qualidade interna do código não influencia a introdução de DTAA.

3.3 Tipos de DTAA

Visando a obter maior entendimento acerca das características de DTAA, Maldonado e Shihab (2015) investigaram quais tipos de Dívida Técnica são indicados por DTAA existentes em projetos de software. Para essa investigação, os autores utilizaram como base o estudo de Alves et al. (2014) que estabelece uma classificação de Dívida Técnica em 13 diferentes tipos.

Foram extraídos os comentários de cinco projetos de softwares de código-fonte abertos, pertencentes a diferentes domínios de aplicação, sendo eles: Apache Ant, Apache Jmeter, ArgoUML, Columba e JFreeChart. O conjunto de comentários extraído foi dividido em dois grupos: *Comentários DTAA* e *Comentários não DTAA*. Para identificar os comentários que indicam DTAA foi utilizada a técnica de Maldonado e Shihab (2015) baseada em Padrões Textuais, a qual utiliza um conjunto de heurísticas para desconsiderar comentários não indicadores de DTAA.

O conjunto de comentários DTAA é composto por 33.093 comentários, pertencente aos cinco softwares utilizados no estudo. Os comentários DTAA identificados foram classificados manualmente nos tipos de Dívida Técnica descrito por Alves et al. (2014). Os resultados da classificação indicaram a existência apenas 5 tipos de DTAA, sendo eles: 1) Dívida de Projeto; 2) Dívida de Defeito; 3) Dívida de Documentação; 4) Dívida de Requisito; 5) Dívida de Teste. Os 8 tipos restantes de Dívida Técnica que foram definidos por Alves et al. (2014) não foram encontrados, e provavelmente não apareceram em comentários de código-fonte, mas em outros artefatos (Maldonado e Shihab, 2015).

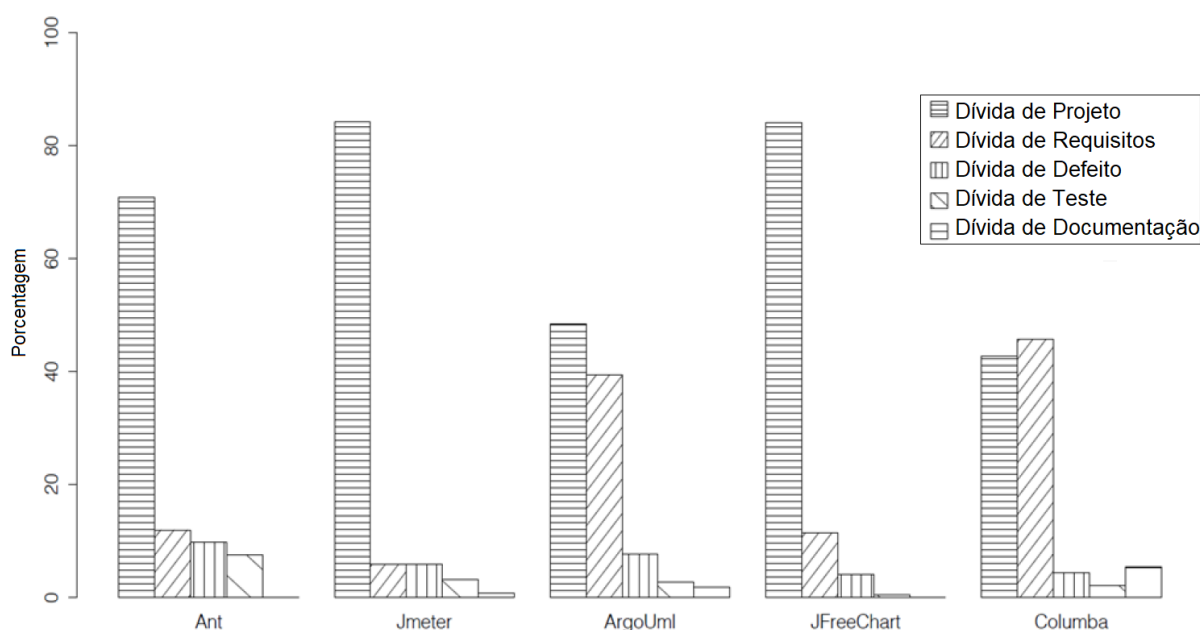
Maldonado e Shihab (2015) ainda efetuaram uma análise quantitativa das instâncias de DTAA identificadas. Foi identificado que a maioria dos comentários DTAA (42% a 84%) são do tipo Dívida de Projeto, seguidas de Dívida de Requisitos (5% a 45%). Dívidas de Defeito, Documentação e Teste representaram juntos menos de 10% do conjunto de comentários classificados como DTAA (Maldonado e Shihab, 2015). Na Tabela 3.2 são exemplificados comentários DTAA e seus respectivos tipos, e na Figura 3.2 é ilustrada a distribuição dos tipos de DTAA.

Tabela 3.2: Exemplo de tipos de DTAA classificadas.

Tipo	DTAA
Dívida de Projeto	//TODO: – This method is too complex, lets break it up
Dívida de Projeto	//TODO: really should be a separate class
Dívida de Defeito	//Bug in above method
Dívida de Defeito	//WARNING: the OutputStream version of this doesn't work!
Dívida de Documentação	//FIXME – This function needs documentation
Dívida de Documentação	//TODO Document the reason for this
Dívida de Requisito	//TODO no methods yet for getClassname
Dívida de Requisito	//TODO no method for newInstance using a reverse-classloader
Dívida de Teste	//TODO – need a lot more tests
Dívida de Teste	//TODO enable some proper tests!!

Fonte: Adaptado de Maldonado e Shihab (2015).

Figura 3.2: Distribuição dos tipos de Dívida Técnica Auto-Admitida.



Fonte: Adaptado de Maldonado e Shihab (2015).

3.4 Impacto causado pela presença de DTAA

Apesar de ser possível identificar DTAA e de existirem estudos visando o entendimento de seu comportamento e evolução, a literatura ainda necessita de estudos que objetivam a evidenciar o impacto direto na qualidade do software causado pela presença de DTAA. Neste contexto, Wehaibi et al. (2016) realizaram um estudo com o objetivo de investigar a relação entre a presença de DTAA e a qualidade do software produzido.

Nessa investigação, Wehaibi et al. (2016) utilizaram uma abordagem empírica, em que os dados foram extraídos de cinco projetos de software: *Chromium*, *Hadoop*, *Spark*, *Cassandra*, *Apache Tomcat*. Todos de código-fonte aberto e pertencentes a diferentes domínios, com uma alta taxa de comentários e implementados em diferentes linguagens de programação (Java, C, C++, Scala, Python e Javascript).

Os autores investigaram o impacto causado por DTAA na qualidade de software observando: 1) A quantidade de defeitos em arquivos com e sem a presença de DTAA; 2) A porcentagem de alterações relacionadas ao DTAA que introduzem defeitos; 3) Se as mudanças que envolvem arquivos que possuem instâncias de DTAA são mais difíceis de serem realizadas do que as que não envolvem.

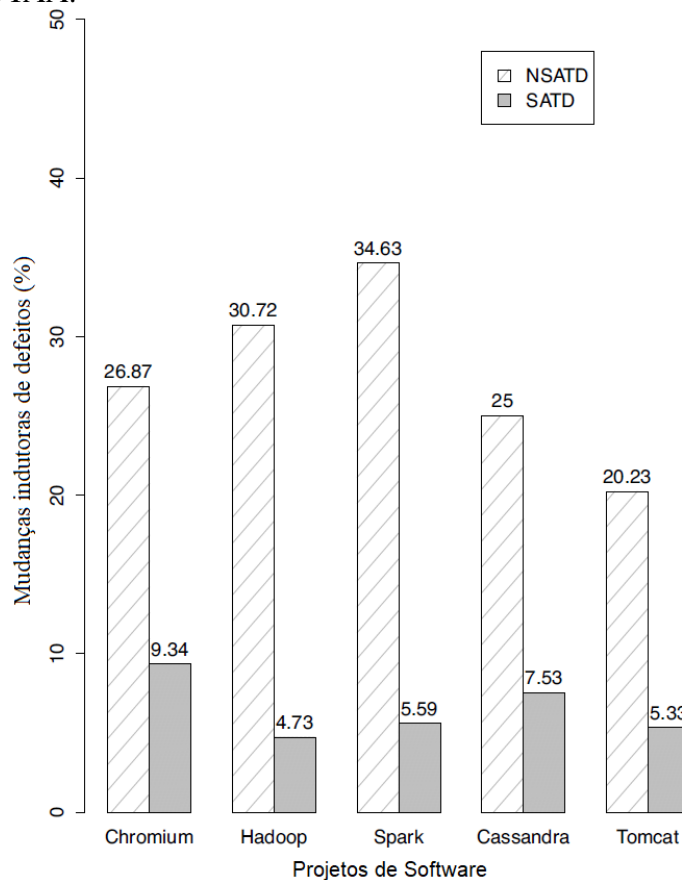
Os comentários DTAA foram identificados por meio da técnica idealizada por Potdar e Shihab (2014) baseada em padrões textuais. As mudanças realizadas para corrigir defeitos foram identificadas por meio de análise de *logs* do repositório *Git* de cada projeto de software e de suas respectivas ferramentas de acompanhamento de problemas: Bugzilla¹ ou JIRA² (Wehaibi

¹www.bugzilla.org

²www.atlassian.com/software/jira

et al., 2016).

Figura 3.3: Porcentagem de modificações indutoras de defeitos relacionadas com DTAA e não relacionadas com DTAA.



Fonte: Adaptado de Wehaibi et al. (2016).

Wehaibi et al. (2016) concluíram que não há uma tendência clara quando se trata de defeitos e DTAA, considerando a quantidade de defeitos em arquivos com e sem a presença de DTAA. Também foi observado que, alterações relacionadas à DTAA têm menor chance de introduzir defeitos futuros no software, se comparadas com alterações que não têm relação com DTAA. Na Figura 3.3 é ilustrada a ocorrência dessa situação em todos os projetos de software utilizados no estudo.

Apesar do estudo realizado por Wehaibi et al. (2016) revelar que as modificações relacionadas com DTAA não implicam em defeitos futuros, mas são as mais difíceis de serem realizadas se comparada com as modificações não relacionadas com DTAA (Wehaibi et al., 2016). Para mensurar a dificuldade de uma mudança foram utilizadas quatro medidas diferentes: 1) Número total de linhas modificadas; 2) Número de diretórios modificados; 3) Número de arquivos modificados; (essas três primeiras definidas por Eick et al. (2001)) 4) Entropia de alteração, essa última definida por Hassan (2009).

Wehaibi et al. (2016) concluem que a Dívida Técnica pode ter algumas implicações negativas no processo de desenvolvimento de software, em particular, tornando-o mais complexo. Assim, os profissionais precisam administrá-lo adequadamente para evitar quaisquer consequências (Wehaibi et al., 2016).

3.5 Remoção de DTAA

Outras pesquisas analisaram DTAA considerando o aspecto de remoção da Dívida Técnica. Maldonado et al. (2017) investigaram a quantidade de instâncias de DTAA que são removidas do código-fonte, quem realiza a remoção, por quanto tempo a DTAA permanece em um sistema e o que leva os desenvolvedores a realizarem atividades de remoção.

Maldonado et al. (2017) estudaram 5 projetos de software de código-fonte aberto, sendo eles: Camel, Gerrit, Hadoop, Log4j, e Apache Tomcat. Todos esses projetos de software são implementados utilizando a linguagem Java, possuem alta taxa de comentários, que variam em tamanho, domínio e número de contribuintes. Os comentários que representam instâncias de DTAA foram identificados utilizando a técnica baseada em Processamento de Linguagem Natural (Maldonado et al., 2017).

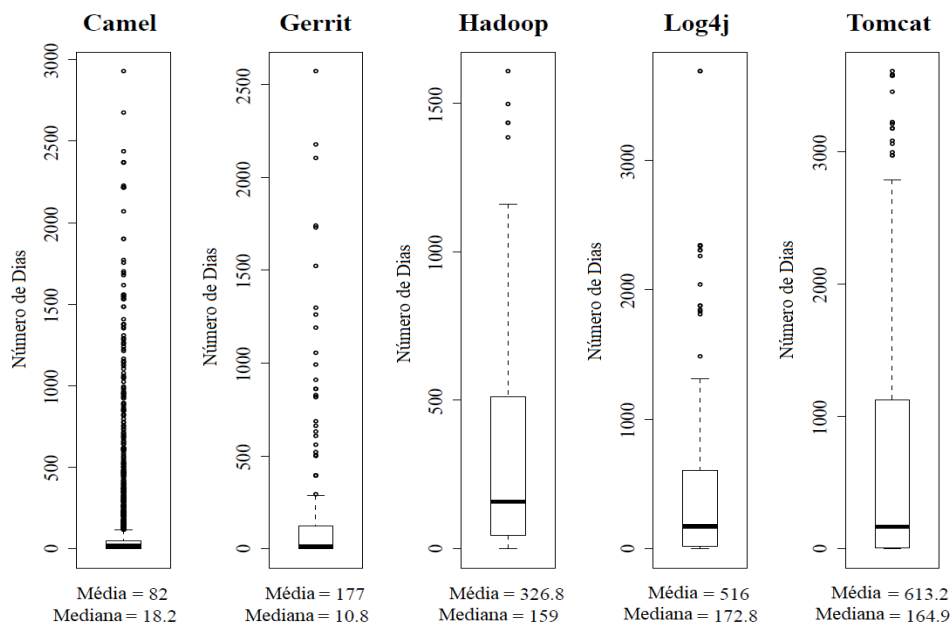
Inicialmente, visando a entender como os desenvolvedores enfrentam a DTAA, foi investigada a quantidade de instâncias de DTAA removidas. Foi constatado que a maioria das instâncias de DTAA são removidas ao longo do tempo. Considerando os cinco projetos de software analisados, 40.5% e 90.6% das instâncias de DTAA identificadas são removidas. Além disso, foi identificado que a maioria das instâncias de DTAA são removidas pelos próprios desenvolvedores que as inseriram no software. Tais resultados foram obtidos ao analisar o histórico de versões de cada um dos projetos de software considerados no estudo, assim, foi possível identificar o *commit* em que cada instância de DTAA é inserida ou removida e qual integrante da equipe de desenvolvimento realiza essa ação.

Posteriormente, ao investigar por quanto tempo uma DTAA persiste em um projeto de software, Maldonado et al. (2017) descobriram que o tempo que a DTAA permanece em um projeto antes de sua remoção varia de um projeto para outro, sendo essa variação temporal entre 82 a 613,2 dias, em média, com mediana entre 10,8 e 172,8. Na Figura 3.4 é ilustrado a distribuição do tempo de vida das instâncias DTAA removidas nos cinco projetos de software estudados.

Além do ponto de vista quantitativo descrito anteriormente, os autores exploram qualitativamente os motivos da remoção de DTAA. Para isso, foi submetido um conjunto de perguntas aos desenvolvedores dos projetos de software analisados. As perguntas tinham como objetivo investigar o papel e a experiência do desenvolvedor no projeto e os motivos para a remoção da DTAA. Maldonado et al. (2017) descobriram que os desenvolvedores adicionam DTAA para rastrear possíveis *bugs* futuros, códigos que precisam de melhorias ou áreas para implementar novos recursos. Em geral, os desenvolvedores removem uma DTAA quando consertam *bugs* ou adicionam novos recursos. Muito raramente os desenvolvedores removem a DTAA como parte de esforços de refatoração ou atividades de melhoria de código.

Posteriormente, motivados pelo estudo de Maldonado et al. (2017), Zampetti et al. (2018) investigaram o processo de remoção de DTAA. Contudo, explorando a vertente de quando essa remoção ocorre acidentalmente. O conjunto de dados utilizado foi o mesmo escolhido por Maldonado et al. (2017), a técnica baseada em Processamento de Linguagem Natural foi utilizada para identificação de DTAA. Por fim, para identificar a remoção, foi analisado o histórico de *commits*. Além de investigar até que ponto a DTAA é removida acidentalmente, os autores in-

Figura 3.4: A distribuição dos tempos de vida das DTAA removidas.



Fonte: Adaptado de: Maldonado et al. (2017).

vestigaram se existe alguma evidência documentada das remoções de DTAA realizadas e quais alterações ocorrem no código-fonte quando os desenvolvedores removem DTAA.

Como resultado do estudo de Zampetti et al. (2018), foi constatado que os comentários indicadores de DTAA são removidos acidentalmente durante a evolução do software. Em geral, a remoção acidental é identificada principalmente quando uma classe inteira é removida do projeto de software, e assim, consequentemente a DTAA também é removida. Além disso, a DTAA é removida acidentalmente quando um método é excluído ou quando não existe modificação no código de um método e a instância de DTAA é removido. Ao analisar, quantitativamente, essas três situações ocorrem na maioria dos projetos de software em menos da metade do total de remoções de instâncias de DTAA (Zampetti et al., 2018).

Considerando o processo de documentação das atividades de remoção de DTAA, 8% do total de remoções são documentadas por meio de mensagens de *commits*. Quando a documentação de remoção acontece, a mensagem menciona que a instância de DTAA foi solucionada ou explica por que não é mais o caso para resolvê-la (Zampetti et al., 2018).

Por fim, Zampetti et al. (2018) apresentam quais mudanças que ocorrem no código-fonte ao realizar a remoção de DTAA. Na Tabela 3.3 são sumarizadas as mudanças resultantes da remoção, assim como, a porcentagem de cada tipo de mudança em relação ao total de instâncias de DTAA removidas. O maior percentual de tipo de mudança é relacionado a modificações complexas no código-fonte, ou seja, modificações que envolvem mais da metade do código de um método, seguidos respectivamente dos seguintes tipos de mudanças: de Adição/Remoção de chamada de método, Adição/Remoção de condicionais, Modificação de assinatura de método, Modificação de retorno e Adição/Remoção de *Try-Catch*.

Tabela 3.3: Tipo de mudanças no código-fonte ao remover instancias de DTAA.

Projeto	Adicionar ou Remover Chamada de Método	Adicionar ou Remover Condicionais	Adicionar ou Remover Try-Catch	Modificar Assinatura do Método	Modificar Retorno	Modificações Complexas	Total
Camel	165 (45%)	61 (17%)	9 (3%)	36 (10%)	15 (4%)	145 (40%)	364
Gerrit	16 (36%)	8 (18%)	3 (7%)	3 (7%)	3 (7%)	23 (51%)	45
Hadoop	42 (34%)	13 (11%)	2 (2%)	6 (5%)	4 (3%)	67 (55%)	122
Log4j	8 (33%)	7 (29%)	0	0	0	10 (42%)	24
Tomcat	59 (26%)	59 (26%)	5 (2%)	17 (8%)	7 (3%)	94 (42%)	224
Total	290	148	19	62	29	339	779

Fonte: Adaptado de Zampetti et al. (2018).

3.6 Juros incorridos por DTAA

No contexto de Dívida Técnica, o *Juros* é o possível aumento de custo em alguma dimensão relacionada ao software devido à realização do pagamento da Dívida Técnica em uma fase futura do desenvolvimento (Nugroho et al., 2011, Kruchten et al., 2013). Kamei et al. (2016) apresentaram um trabalho para determinar maneiras de medir *Juros* incorridos por DTAA e usar essas medidas para observar quanto da Dívida Técnica incorrem nos chamados juros positivos, ou seja, dívidas que custam mais para serem pagas no futuro.

Com o auxílio da ferramenta JDeodorant³, foram extraídos comentários do código-fonte do software Apache JMeter 2.10, nos quais foram classificados manualmente em comentários indicadores ou não de DTAA. Para calcular os juros, foram consideradas métricas de qualidade de código. Inicialmente foram avaliadas 16 métricas, contudo apenas duas delas foram utilizadas no estudo, sendo elas: *LOC* e *Fan-In*. Kamei et al. (2016) justificam que a maioria das métricas estavam estritamente relacionadas com a métrica *LOC*, exceto a *Fan-In*, por esse motivo ambas foram utilizadas e as demais não foram utilizadas. Os autores consideram que a DTAA incorrem em juros quando o código com DTAA se tornar mais complexo ao longo do tempo em comparação ao momento que a instância de DTAA foi inserida.

3.7 Atividade de Priorização de DTAA

Diversos trabalhos são direcionados para obtenção de uma compreensão acerca de DTAA, analisando suas características e comportamentos. A compreensão da DTAA subsidia o estudo de abordagem para aplicar atividades de gestão em DTAA. Na literatura faltam abordagens direcionadas às atividades de gestão de DTAA, a maioria dos estudos concentram-se na identificação de DTAA e sua compreensão.

Considerando o contexto deste trabalho, até onde se identifica na literatura, uma única abordagem foi proposta visando priorizar instâncias de DTAA para uma possível etapa de pagamento de Dívida Técnica. A abordagem de priorização situada na literatura é apresentada por Mensah et al. (2018), que por sua vez, em seu estudo estruturam um meio de priorizar instâncias de DTAA utilizando mineração de texto com base em padrões textuais (Potdar e Shihab, 2014). Mensah et al. (2018) descrevem que o esquema de priorização de DTAA ajuda

³github.com/tsantalís/JDeodorant

na tomada de decisões de quais instâncias de DTAA podem ser pagas prioritariamente antes da liberação do software, na tentativa de minimizar as despesas de manutenção.

A abordagem de priorização para DTAA definida por Mensah et al. (2018) é expressa em seis passos. Ao final da execução da abordagem, tem-se as instâncias de DTAA que indicam tarefas vitais que precisam ser pagas com prioridade antes da liberação do software para uso, bem como a estimativa de esforço de retrabalho necessário para pagá-las. Os seis passos para priorização de DTAA são descritos na Tabela 3.4 e esquematizados conforme mostrado na Figura 3.5.

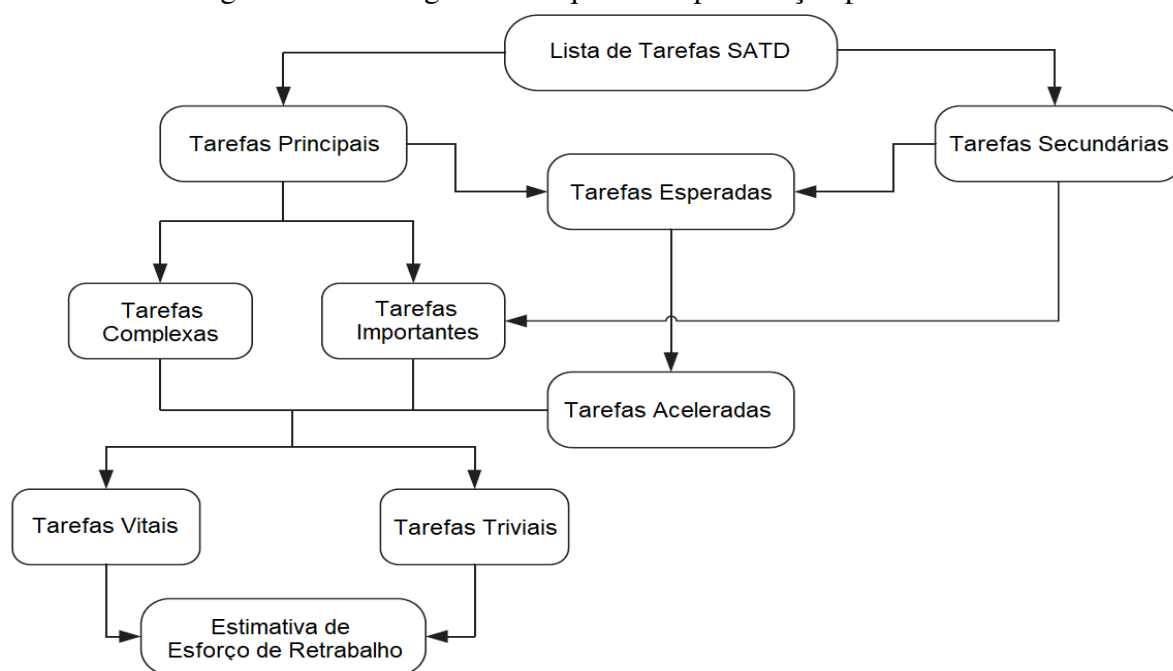
Tabela 3.4: Abordagem de Priorização de DTAA definida em 6 passos.

Seis passos para priorização de DTAA

- 1° Passo: Identifique a lista de DTAA do projeto e as tarefas associadas a elas
- 2° Passo: Análise a complexidade e a importância da tarefa indicada pela DTAA
- 3° Passo: Distinguir entre as tarefas de DTAA esperadas e aceleradas
- 4° Passo: Separação entre as tarefas de DTAA vitais das triviais
- 5° Passo: Identifique as causas plausíveis das principais tarefas de DTAA
- 6° Passo: Ponto de interrupção para decisão de liberação

Fonte: Adaptado de Mensah et al. (2018).

Figura 3.5: Visão geral do esquema de priorização para DTAA.



Fonte: Adaptado de Mensah et al. (2018).

O **primeiro passo** consiste em detectar comentários de código-fonte são instâncias de DTAA e a partir dessas instâncias identificar quais tarefas são descritas por esses comentários – essas tarefas podem ter uma alta ou uma baixa prioridade. Para o processo de identificação de DTAA é utilizada a técnica baseada em padrões textuais descrita por Potdar e Shihab (2014).

Ao final do primeiro passo, as instâncias de DTAA são organizadas em dois conjuntos: as instâncias que indicam tarefas *principais* e instâncias que indicam tarefas *secundárias* que

precisam ser pagas. Para separar as instâncias de DTAA nesses dois conjuntos são utilizados um subconjunto de padrões textuais. O subconjunto de padrões textuais que demonstram a relação à urgência, gravidade e significância é usado para identificar as principais tarefas de DTAA, enquanto as tarefas restantes formam o conjunto de tarefas *secundárias*. Exemplos de padrões textuais usados para identificação de tarefas *principais* são: *design error, remove me before production, fix me before production, system malfunctioning, still bugs, kludge, test failure, untested, problematic, add it now e hack for testing*. Exemplos de padrões textuais usados para identificação de tarefas *secundárias* são: *skip me, ignore, leave it, in future release, workaround, don't worry, check runtime, rename, poorly defined, does not match, eliminate, implement this, e remove me*.

O **segundo passo** é identificar a *complexidade* e a *importância* das tarefas indicadas pelas instâncias de DTAA identificadas. A complexidade da tarefa refere-se à natureza de uma determinada tarefa na qual a mesma é considerada difícil de ser realizada (Nakatsu et al., 2014), enquanto a Importância da tarefa se refere à relevância de realizar uma determinada tarefa para obter um funcionamento eficaz do sistema (Mensah et al., 2018). Os autores constataram que o conjunto de tarefas *complexas* são um subconjunto das tarefas *principais*, já o conjunto de tarefas *importantes* pode ser um subconjunto tanto de tarefas *principais* quanto de tarefas *secundárias*, observe a esquematização ilustrada na Figura 3.5. Os padrões textuais usados para identificação de tarefas *complexas* são: *sophisticated, bail out, give up, this is killing, serious hack, workaround bug, difficult, cannot fix e nasty hack*. Enquanto os padrões textuais usados para identificação de tarefas *importantes* são: *fixme before production, finishme before production, system malfunctioning, test failure, hack for testing e rework needed before production*.

No **terceiro passo** as tarefas são fragmentadas entre *esperadas* e *aceleradas*. As tarefas *esperadas* são tarefas previstas necessárias e que devem ser realizadas pelos desenvolvedores. Já as tarefas *aceleradas* são caracterizadas por uma série de atalhos realizados pelos desenvolvedores para acelerar o desenvolvimento em alguma perspectiva, apesar de que os trechos de códigos que caracterizam atalhos normalmente não aparentarem conter erros, nem sempre garantem que a lógica de implementação esteja correta. Exemplos de padrões textuais usados para identificação de tarefas *esperadas* são: *workaround, nuke it, temporary hack, really ugly, silly, yuck e nasty error*. Exemplos padrões textuais usados para identificação de tarefas *aceleradas* são: *fixme, todo, finishme, must fix it, we ought to fix, add it, hack for, detect, generate, rework e rewrite*.

O **quarto passo** evidencia a separação entre as tarefas *vitalis* e *triviais*. As tarefas *vitalis* tendem a erros no software e resultam em um sistema defeituoso se não forem resolvidas antes da liberação da aplicação para o uso (Wehaibi et al., 2016). Em contrapartida, as tarefas DTAA que não resultam em *bugs* são consideradas como tarefas *triviais*. Exemplos de padrões textuais usados para identificação de tarefas *vitalis* são: *remove me before production, bail out cannot do this, we hang our heads in shame, still bugs in the system, give up looking for the bug, cannot introspect, design error, nasty hack, too much bugs, really dirty hack, large hack here, complex to hack, complex to debug, this bug is killing e cannot introspect*. Exemplos padrões textuais

usados para identificação de tarefas *triviais* são: *remove when bug is resolved, test commented out until bug is fixed, temporary workaround for bug, temporary hack, fixme fails here, fixme this seems wrong, find a better way to fix bug e ugly trick for bug*. Na Tabela 3.7 são apresentados todos os identificadores textuais utilizados para detectar cada uma das tarefas DTAA visando a priorização.

Tabela 3.5: Identificadores textuais utilizados para classificar as instâncias de DTAA em tarefas priorizadas.

Tarefa DTAA	Identificadores Textuais
Principal	<i>design error, remove me before production, fix me before production, system malfunctioning, still bugs, kludge, test failure, untested, problematic, add it now e hack for testing.</i>
Secundária	<i>skip me, ignore, leave it, in future release, workaround, don't worry, check runtime, rename, poorly defined, does not match, eliminate, implement this, e remove me.</i>
Complexa	<i>sophisticated, bail out, give up, this is killing, serious hack, workaround bug, difficult, cannot fix e nasty hack.</i>
Importante	<i>fixme before production, finishme before production, system malfunctioning, test failure, hack for testing e rework needed before production.</i>
Esperada	<i>workaround, nuke it, temporary hack, really ugly, silly, yuck e nasty error.</i>
Acelerada	<i>fixme, todo, finishme, must fix it, we ought to fix, add it, hack for, detect, generate, rework e rewrite.</i>
Trivial	<i>remove when bug is resolved, test commented out until bug is fixed, temporary workaround for bug, temporary hack, fixme fails here, fixme this seems wrong, find a better way to fix bug e ugly trick for bug.</i>
Vital	<i>remove me before production, bail out cannot do this, we hang our heads in shame, still bugs in the system, give up looking for the bug, cannot introspect, design error, nasty hack, too much bugs, really dirty hack, large hack here, complex to hack, complex to debug, this bug is killing e cannot introspect.</i>

Fonte: Adaptado de Mensah et al. (2018).

Após a separação entre instâncias de DTAA que estão relacionadas a tarefas vitais e triviais, o **quinto passo** é o momento de identificar as possíveis causas das principais tarefas indicadas por DTAA. Uma vez que um conjunto de instâncias de DTAA devem ser pagas na próxima versão do software, identificar as causas que resultaram nas principais tarefas indicadas por DTAA torna-se importante para a futura etapa de pagamento. Essa etapa é realizada por meio de uma exploração manual.

Com a identificação das possíveis causas das ocorrências das principais tarefas indicadas por instâncias de DTAA, é realizado o **sexto passo** da abordagem de priorização. Nesse passo

é calculada a estimativa de esforço de retrabalho para as tarefas DTAA propensas a *bugs* e não solucionadas. As estimativas apoiam decisões gerenciais que podem ser tomadas para atingir o ponto de interrupção para pagamento das Dívidas Técnicas ou o lançamento do software desenvolvido para o mercado. A métrica para estimativa de esforço de retrabalho é baseada em um trabalho anterior de Mensah et al. (2016) e está sintetizada na Equação 3.1.

$$RW_{VT} = \frac{\sum_{i=1}^n (\sum_{j=1}^k LOC(f_j))_i}{\sum_{i=1}^m f_i} \quad (3.1)$$

onde:

- n = Número total de arquivos de origem de um software.
- m = Número total de arquivos de origem que contém instâncias de tarefas vitais ou triviais.
- k = Número total de LOCs formando instâncias de tarefas vitais ou triviais por arquivo de origem.
- f_i = Um dos arquivo de origem.
- $LOC(f_j)$ = Número de LOCs formando instâncias de tarefas vitais ou triviais em um arquivo fonte f_j .

A métrica de estimativa de esforço de retrabalho foi aplicada nas tarefas *vitalis*, consideradas tarefas de DTAA que devem ser priorizadas. Além disso, Mensah et al. (2018) também aplicaram a métrica nas tarefas *triviais*. Em suma, a métrica fornece uma estimativa em linhas de código por arquivo de origem DTAA que requer retrabalho com relação as tarefas *vitalis* ou *triviais*.

Para materializar a abordagem de priorização de DTAA possibilitando sua validação e utilização na prática, Mensah et al. (2018) elaboraram um algoritmo de Mineração de Texto com o propósito de automatizar a identificação de DTAA e posteriormente, seguindo a abordagem estabelecida, indicar quais delas devem ter prioridade em uma possível fase de pagamento de Dívida Técnica.

Um processo exploratório foi realizado por (Mensah et al., 2018) para possibilitar a construção do algoritmo de Mineração de Texto. No processo exploratório foram analisados, manualmente, comentários indicadores de DTAA de projetos de software com o intuito de extrair identificadores textuais que pudessem ser utilizados para agrupar as instâncias de DTAA em tarefas principais, secundárias, esperadas, aceleradas, complexas, importantes, vitais e triviais (conforme a abordagem de priorização descrita anteriormente).

Os comentários que fizeram parte do estudo exploratório pertencem a quatro projetos de software de código-fonte abertos, sendo eles: ArgoUML, Chromium OS, Apache HTTP e Eclipse Platform. Para identificar as instâncias de DTAA foi utilizada a técnica baseada em padrões textuais apresentado por Potdar e Shihab (2014). O conjunto de dados utilizado no estudo é composto por cerca de 600.000 comentários.

Após a análise manual de cada um dos comentários que representam instâncias de DTAA foi obtido como resultado inicial uma lista de identificadores textuais – Tabela 3.7, utilizados para agrupar as instâncias de DTAA em cada tipo de tarefas priorizadas. Com a lista total de indicadores textuais, foi possível agrupar manualmente as instâncias de DTAA nas diferentes tarefas de prioridades. Na Tabela 3.6 são apresentados exemplos de tarefas DTAA classificadas pelos autores.

Tabela 3.6: Exemplo de instâncias de DTAA classificadas em tarefas priorizada.

Tarefa DTAA	Exemplo
Principal	<i>//TODO: Hack to work around Bug</i>
Secundária	<i>//FIXME: duplicated from the method with same name in ...</i>
Complexa	<i>//This test is a bit stupid. We cannot introspect</i>
Esperada	<i>//HACK!! This should be done using a separate renderer</i>
Acelerada	<i>//The workaround is to fake a query string match if ...</i>

Fonte: Adaptado de Mensah et al. (2018).

Tabela 3.7: Comparativo quantitativo de instâncias DTAA classificadas em tarefas principais e secundárias.

Tarefas DTAA	ArgoUML	Chromium	Eclipse	Apache
Principal	33,3%	30,6%	42,2%	38,5%
Secundária	66,7%	69,4%	57,8%	61,5%

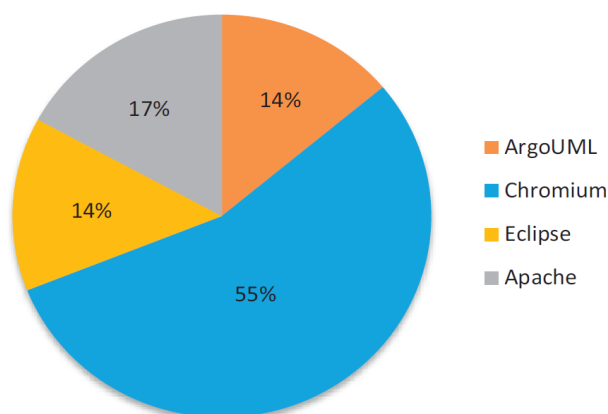
Fonte: Adaptado de Mensah et al. (2018).

Com o agrupamento manual das instâncias de DTAA em tarefas priorizadas, os autores analisaram quantitativamente cada um dos conjuntos obtidos. Por meio da análise do conjunto de tarefas DTAA principais e secundárias, foi descoberto que a maioria das instâncias de DTAA pertencentes a esses dois conjuntos são relacionadas as tarefas secundárias, aproximadamente 62–70% – observe os percentuais apresentados na Tabela 3.7, enquanto o conjunto de tarefas principais representam aproximadamente 30–38% distribuídas nos quatro projetos de softwares analisados.

Em seguida, Mensah et al. (2018) analisaram os comentários pertencentes ao grupo de tarefas complexas e perceberam que cerca de 4% a 12% dessas tarefas complexas não foram resolvidas antes da liberação dos projetos software para uso. Portanto, enfatizaram que há uma necessidade de que essas tarefas complexas não resolvidas recebam atenção crítica e sejam tratadas especialmente por desenvolvedores mais experientes durante o desenvolvimento, assim como também enfatiza Potdar e Shihab (2014). O gráfico ilustrado na Figura 3.6 apresenta a distribuição dessas tarefas complexas entre os quatro projetos de software estudados.

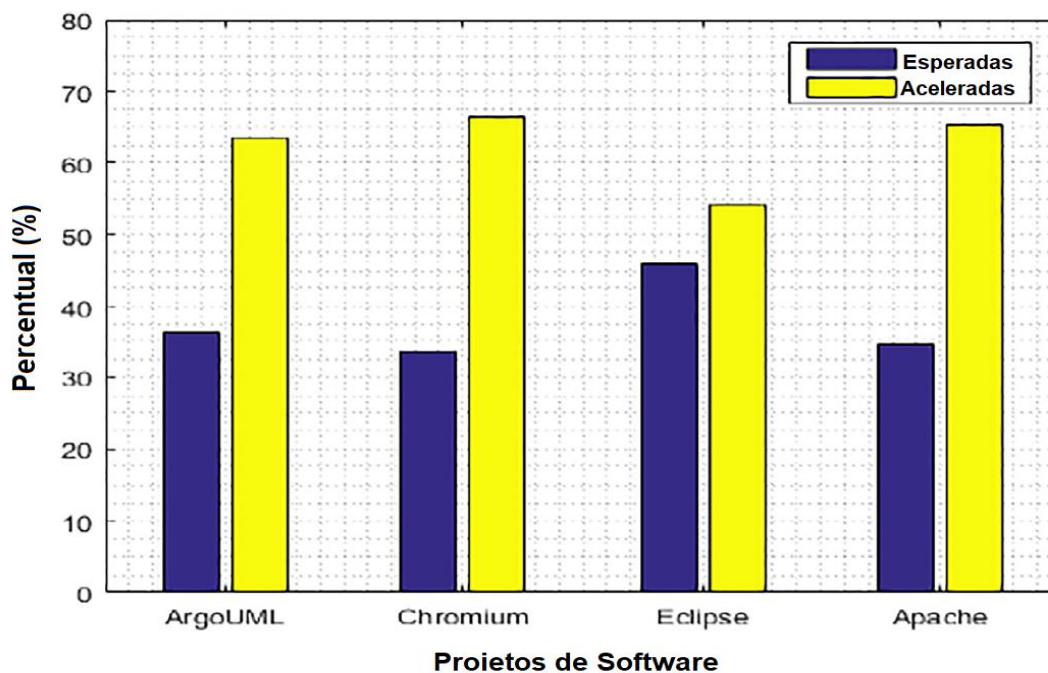
Na Figura 3.7 é apresentada a distribuição das tarefas esperadas e aceleradas extraídas dos comentários do código-fonte dos quatro projetos estudados. Os resultados indicam que a maioria das tarefas realizadas pelos desenvolvedores dos quatro projetos de software de código-fonte aberto foram aceleradas, com percentuais de 63,6% no ArgoUML, 66,3% no Chromium, 54,2% no Eclipse e 65,5% no Apache. Como a maioria dessas tarefas está sendo acelerada, existe a necessidade da equipe de desenvolvimento prestar muita atenção a essas tarefas e usar as abor-

Figura 3.6: Distribuição do conjunto de tarefas DTAA complexas entre os quatro projetos de softwares estudados.



Fonte: Adaptado de Mensah et al. (2018).

Figura 3.7: Comparativo quantitativo de instâncias DTAA classificadas em tarefas esperadas e aceleradas.



Fonte: Adaptado de Mensah et al. (2018).

dagens esperadas apropriadas para retificá-las, a fim de minimizar falhas lógicas e de funcionamento dos componentes (Mensah et al., 2018).

Em seguida, visando a identificar as causas plausíveis de DTAA presentes no conjunto de comentários, cada comentário DTAA foi analisado e foram identificados sete causas plausíveis das tarefas DTAA, cujo os resultados são sumarizados na Tabela 3.8. Em destaque, os principais fatores que causam DTAA encontrados durante a análise exploratória são: *code smells*, tarefas complicadas e complexas, teste de código inadequado e desempenho de código inesperado. Segundo Mensah et al. (2018), as causas plausíveis das tarefas DTAA são altamente dominantes nas poucas tarefas vitais identificadas.

O restante do estudo foi direcionado para validar o algoritmo de Mineração Textual. Para

Tabela 3.8: Causas plausíveis de DTAA identificadas ao analisar os comentários de código-fonte dos quatro software estudados.

Causas de DTAA	Percentual
<i>Code Smells</i>	23,2%
Tarefas muito complicadas e complexas	22,0%
Teste de código inadequado	21,2%
Desempenho de código inesperado	17,4%
Incapaz de não lembrar	8,1%
Restrição de tempo	4,6%
Comunicação inconsistente entre desenvolvedores	3,5%

Fonte: Adaptado de Mensah et al. (2018).

validar o algoritmo de Mineração Textual e possibilitar sua utilização, foi realizada uma comparação entre os resultados obtidos com a exploração manual, e consequentemente aplicação da abordagem de priorização manualmente, com os resultados obtidos ao aplicar a priorização de modo automatizado por meio de Mineração Textual. O conjunto de entrada para o algoritmo de mineração foram os comentários obtidos do código-fonte e os identificadores textuais para classificar as DTAA em cada uma das tarefas priorizadas.

Mensah et al. (2018) validaram empiricamente a abordagem de priorização proposta utilizando o algoritmo de Mineração Textual usando medidas de precisão e revocação. Além disso, foram realizados testes estatísticos usando o *Welch's t-test* e o teste *Cliff's δ effect*, conforme recomendado por Kitchenham et al. (2017) para confirmar a significância estatística dos resultados.

Os resultados após a aplicação das métricas estatísticas indicaram uma alta precisão para classificação das instâncias de DTAA nas tarefas pertencentes a abordagem. Para as tarefas principais e secundárias, a precisão obtida foi de 80,3% em média e a revocação obtido foi de 75,7%, considerando as tarefas esperadas e aceleradas a precisão foi em média 76,1% e a revocação foi de 71,2%, por fim, para as tarefas vitais e triviais a precisão e a revocação foram de 81,05% e 80,9% respectivamente. O teste *Cliff's δ effect* também foi aplicado em todos os casos não indicando alta diferença entre a classificação manual e a classificação por meio do algoritmo de Mineração Textual.

3.8 Considerações Finais

Neste capítulo, foi apresentado estudos direcionados para obter uma compreensão sobre DTAA. Somente com a análise do comportamento e da evolução da DTAA em projetos de software pode-se compreender melhor suas características e impactos gerados por sua presença e acúmulo.

Na perspectiva de evolução da DTAA em diversos projetos de software, foi identificado que o impacto da DTAA não possui correlação clara com o número de defeitos encontrado no software, e que a qualidade interna do código-fonte, considerando aspectos de complexidade, acoplamento e legibilidade, não influenciam na introdução de novas instâncias de DTAA. Contudo, foi constatado que sob a ótica de mudanças no processo de evolução do software, as mudan-

ças relacionadas diretamente com DTAA em geral são mais complexas de serem realizadas, se comparadas com as mudanças que não estão relacionadas com DTAA.

Além disso, foi identificado que desenvolvedores costumam pagar as instâncias de DTAA introduzidas no projeto de software. Contudo, não seguem um método para realizar essa atividade, e muito raramente realizam a documentação dessa remoção, quando o fazem, é por meio de mensagens de *commits*.

Entender as características da DTAA, seus impactos e o ciclo de vida durante o desenvolvimento do software, pode ajudar a desenvolver novas abordagens para gerenciar DTAA. Uma vez constatado que a DTAA dificulta modificações no código, as quais elas aparecem, surge a necessidade de elaborar mecanismos que apoiem o pagamento dessas dívidas de modo eficiente. Nesse sentido, Mensah et al. (2018) analisou instâncias de DTAA para extrair um conjunto de identificadores textuais utilizados para classificar tais instâncias em tarefas que devem ser priorizadas. O esquema de priorização é composto por 6 etapas e ao final de sua aplicação é possível estimar o esforço de retrabalho ao pagar DTAA.

O esquema de priorização de 6 etapas de Mensah et al. (2018) busca ajudar o processo de tomada de decisões antes do lançamento do software para o mercado, provendo apoio na tentativa de minimizar as tarefas relacionadas ao DTAA que, de outra forma, resultariam em uma diminuição no custo necessário para etapas de manutenção. Além do esquema de priorização, os autores disponibilizaram um algoritmo de Mineração Textual que possibilita aplicar as etapas da esquematização na prática de modo automático. O esquema de priorização foi testado e validado comparando seus resultados com os resultados da abordagem aplicada manualmente.

Embora o esquema de Mensah et al. (2018) tenha apresentado resultados benéficos para a atividade de priorização de DTAA, esse processo ainda é restringido por observar apenas os comentários que indicam DTAA e os padrões textuais extraídos deles. O código-fonte em si e os possíveis problemas nele presentes não são considerados em sua abordagem de priorização, sendo esse um ponto em que este trabalho buscou explorar, por meio de uma abordagem de priorização detalhada no capítulo seguinte.

Abordagem de Priorização para Dívida Técnica Auto-Admitida

4.1 Considerações Iniciais

*D*iante do contexto apresentado sobre Dívida Técnica Auto-Admitida, é notório que os trabalhos presentes na literatura são concentrados na identificação e análise desse fenômeno. Contudo, um estudo realizado por Mensah et al. (2018) foi direcionado para priorização de DTAA, com o objetivo de auxiliar a etapa de pagamento da Dívida Técnica que foi auto-admitida pelos desenvolvedores de software.

Como apresentado anteriormente, a abordagem de priorização para DTAA descrita por Mensah et al. (2018) utiliza as informações presentes nas descrições de comentários que indicam DTAA para inferir sua priorização. Contudo os autores não consideram os possíveis problemas existentes no trecho de código em que está localizada uma instância de DTAA para indicar sua priorização, sendo está a perspectiva que este trabalho buscou explorar. Nesse contexto, esta seção apresenta uma Abordagem de Priorização desenvolvida por este trabalho, que por sua vez, têm o propósito de contribuir para o fortalecimento da atividade de priorização de DTAA.

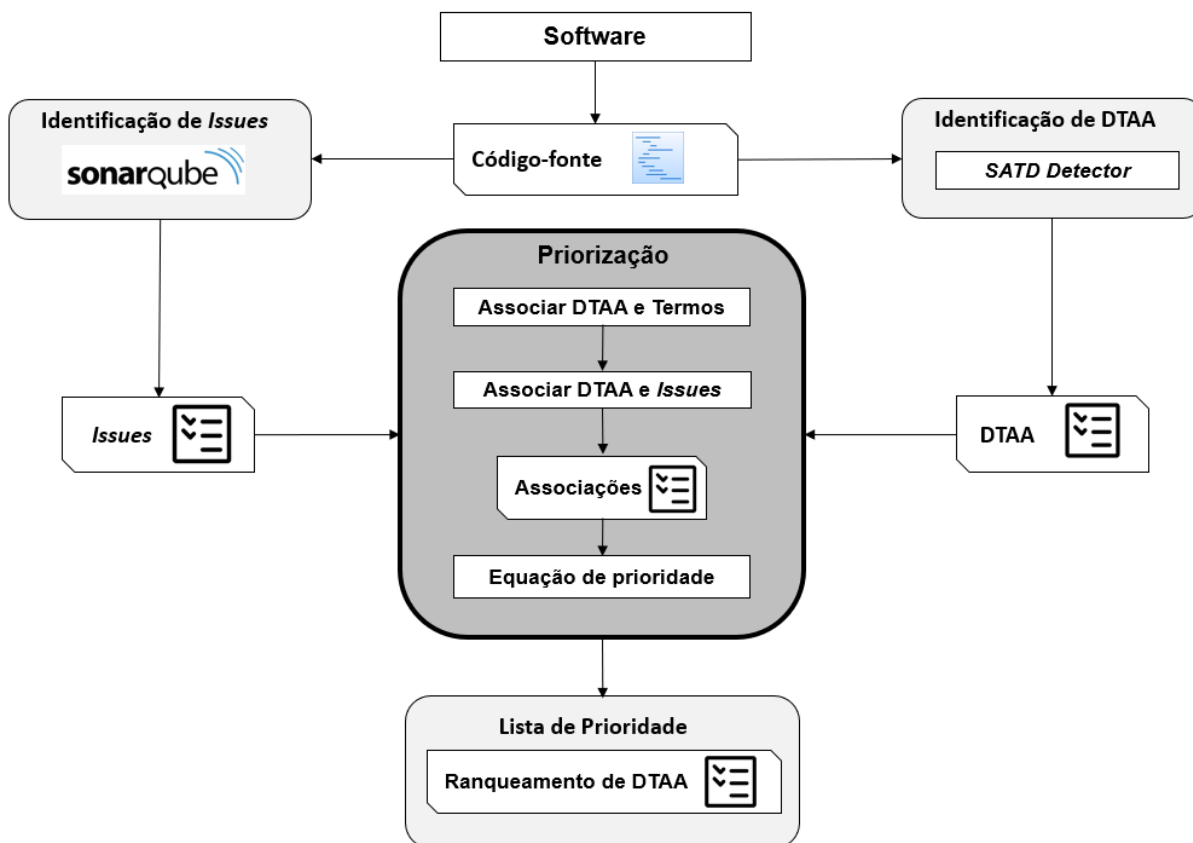
4.2 Abordagem de Priorização

O esquema da Abordagem de Priorização para pagamento de DTAA, desenvolvida por este trabalho, é ilustrado na Figura 4.1. Para a aplicação da abordagem de priorização, dado como entrada o código-fonte de um software, primeiramente é executado a atividade de *Identificação de DTAA* e *Identificação de Issues* (problemas presentes no código-fonte), assim é obtido uma lista de instâncias de DTAA e uma lista de *Issues*.

Após a identificação, é iniciada a etapa de priorização, em que ocorre o processo de *Associação*, Termos de Prioridade (frases ou palavras que indicam prioridade, detalhados na Subse-

ção 4.2.3) e as *Issues* são associadas a cada instância de DTAA. Todas as instâncias de DTAA que possuem associações com termos de prioridade ou *Issues* são consideradas instâncias de DTAA que devem ter prioridade em uma possível etapa de pagamento de Dívida Técnica. Em seguida, todas as instâncias de DTAA que possuem associações e são consideradas prioritárias, são submetidas para uma *Equação de prioridade*, de modo a indicar quais instâncias de DTAA prioritárias tendem a ser mais graves. Por fim, uma *Lista de prioridade* é formada com todas as instâncias de DTAA ordenadas por prioridade.

Figura 4.1: Esquema da Abordagem de Priorização para DTAA.



Fonte: Elaborada pelo autor.

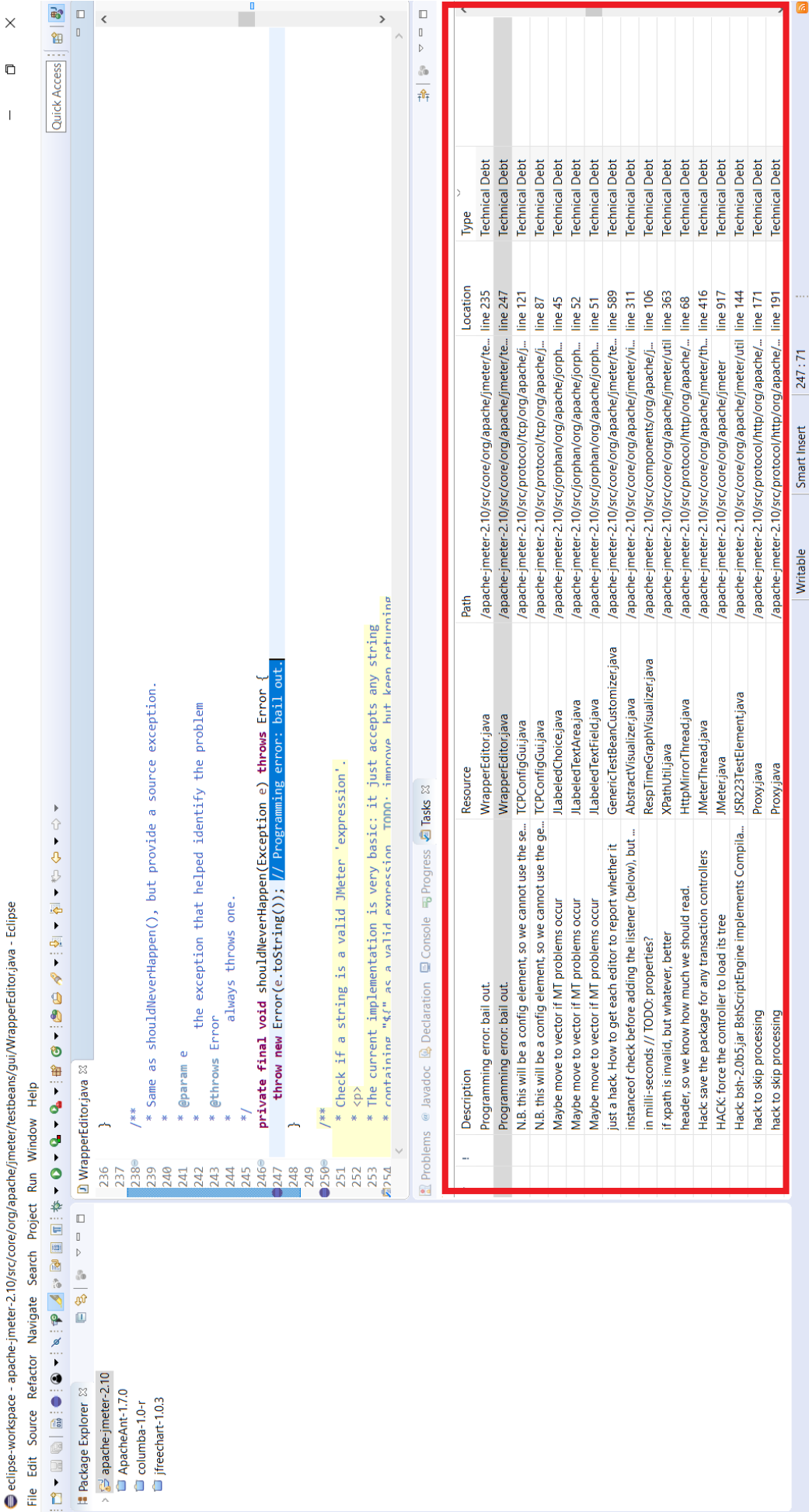
4.2.1 Identificação de DTAA

A etapa de identificação de DTAA faz uso da técnica baseada em Classificação Composta idealizada por Huang et al. (2018), apresentada no Capítulo 2. Essa técnica de identificação de DTAA, foi escolhida por apresentar melhor desempenho que as demais presentes na literatura. A técnica de identificação baseada em Classificação Composta é materializada por meio da ferramenta denominada *SATD Detector*¹ (Liu et al., 2018).

A *SATD Detector* coloca em prática a técnica de identificação baseada em Classificação Composta apresentada por Huang et al. (2018) através de um *plugin* desenvolvido para a *IDE Eclipse*. O usuário da *SATD Detector* apenas faz uso da ferramenta já treinada, assim ao importar um projeto Java na *IDE Eclipse* a *SATD Detector* realiza automaticamente a classificação

¹github.com/Tbalm/SATDDetector-Core

Figura 4.2: Instantâneo da interface da ferramenta SATD Detector utilizada para identificar instâncias de DTAA.



Fonte: Elaborada pelo autor.

e identifica todas as instâncias de DTAA encontradas no projeto alvo. O resultado da classificação é visualizado por meio de uma listagem simples de todos os comentários classificados como instâncias de DTAA.

Na Figura 4.2 é ilustrado um instantâneo da interface do usuário da *SATD Detector*. Nesse exemplo, a identificação foi aplicada no software Apache JMeter, observe o retângulo em vermelho que destaca a listagem das instâncias de DTAA identificadas após a execução da *SATD Detector*. Os comentários identificados como instâncias de DTAA são listados e cada um deles apresenta uma descrição do comentário, classe em que ele foi identificado, a localização em que ele situa-se no código-fonte e seu tipo, que no caso é simplesmente uma *tag* indicando Dívida Técnica.

4.2.2 Identificação de *Issues*

O código-fonte de um projeto de software é submetido a uma Análise Estática Automática (ASA) realizada pela ferramenta SonarQube² e possui como resultado uma lista de *Issues*, ou seja, problemas e violações de boas práticas de programação identificadas no código-fonte. Existem algumas ferramentas disponíveis para essa finalidade, porém o SonarQube é utilizado neste trabalho para aplicação da abordagem, por ser uma ferramenta utilizada em outros trabalhos relacionados (Jesus e Melo, 2017, Digkas et al., 2017, Martini et al., 2018). Além disso, o SonarQube possui a característica de detectar um grande número de violações e tipos de problemas relacionados ao código-fonte (Digkas et al., 2018).

A inspeção do código-fonte realizada pelo SonarQube aponta um problema (*Issue*) presente no código-fonte. Na Figura 4.3 é ilustrado um exemplo de *Issue* identificada após a ASA, no software Apache JMeter. Neste exemplo, a *Issue* descreve que nível de complexidade cognitiva está acima do permitido (22 quando deveria ser 15 ou menos por convenção) para o método *mousePressed* do software Apache JMeter. A alta complexidade cognitiva dificulta o entendimento do trecho de código pertencente ao método em questão e impacta negativamente em uma possível etapa de manutenção.

Uma *Issue* identificada possui um nível de severidade: quanto maior a severidade, maior é o impacto causado na qualidade do código-fonte. Além disso, o SonarQube indica uma estimativa de Tempo de Retrabalho (TR), expresso em minutos, necessário para solucionar o problema descrito pela *Issue* (SonarSource, 2016). A seguir são descritos os níveis de severidade:

- **Blocker:** *Bug* com alta probabilidade de impactar o comportamento do software, deve ser corrigido imediatamente.
- **Critical:** *Bug* com baixa probabilidade de afetar o comportamento do software, código deve ser revisto.
- **Major:** Falha de qualidade que pode afetar gravemente a produtividade do desenvolvedor.
- **Minor:** Falha de qualidade que pode afetar ligeiramente a produtividade do desenvolvedor.

²<https://www.sonarqube.org>

Figura 4.3: Exemplo de *Issue* identificada no software Apache JMeter: Problema de alta complexidade cognitiva.



Fonte: Elaborada pelo autor.

4.2.3 Associação de Termos de Prioridade

Após a identificação das instâncias de DTAA, cada uma delas pode possuir associações com Termos de Prioridade. Os Termos de Prioridade são termos simples ou compostos que indicam prioridade e gravidade e são utilizados para identificar quais instâncias de DTAA devem ser pagas prioritariamente. Esses termos estão comumente presentes na descrição do comentário que indica uma instância de DTAA.

Os Termos de Prioridade são sumarizados na Tabela 4.1. Essa lista de termos foi obtida inspirada no trabalho de Potdar e Shihab (2014) que apresentou padrões textuais que indicam DTAA, de modo análogo, foram inspecionados manualmente, sem esforço de consenso, um pouco mais de 4 mil instâncias de DTAA presentes em diferentes projetos de software, sendo eles: Hibernate, ArgoUML, JRuby, Apache JMeter e JFreeChart, com o objetivo de identificar termos que indiquem: urgência, imediatismo, prioridade, gravidade, problemas severos, erros, resultados incorretos ou inconsistentes, falta de testes ou falha em testes realizados.

Essas indicações destacam os comentários que reapresentam instâncias de DTAA que devem ser priorizadas em uma futura etapa de pagamento de Dívida Técnica. Esse mesmo princípio de associação com termos de prioridade também foi utilizado pela abordagem de priorização já existente na literatura.

4.2.4 Associação de *Issues* por contexto

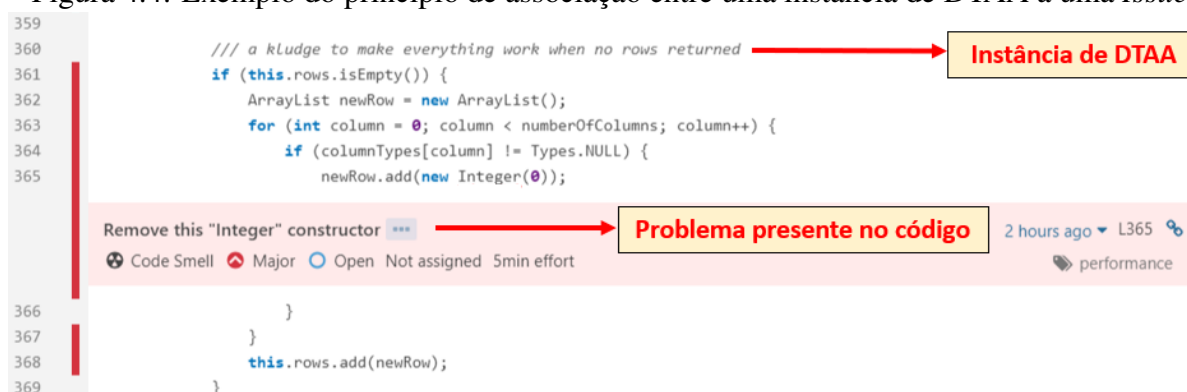
Após as instâncias de DTAA e as *Issues* serem identificadas, é dado início a etapa de associação entre instâncias de DTAA e *Issues*, com o objetivo de utilizar tais associações para indicar prioridade para o pagamento de DTAA. A associação entre as instâncias de DTAA e as *Issues* ocorre como no exemplo ilustrado na Figura 4.4.

Neste exemplo, tem-se um comentário indicando que o trecho de código foi implementado

Tabela 4.1: Termos que indicam prioridade para um pagamento futuro de Dívida Técnica Auto-Admitida

Termos de Prioridade	
<i>serious</i>	<i>junk</i>
<i>urgent</i>	<i>not work</i>
<i>fundamental</i>	<i>need the correctness</i>
<i>terrible</i>	<i>wrong</i>
<i>fugly</i>	<i>implement properly</i>
<i>horrible</i>	<i>improve this a lot</i>
<i>add it now</i>	<i>fix</i>
<i>system malfunctioning</i>	<i>killing</i>
<i>fail</i>	<i>slow</i>
<i>fatal</i>	<i>inefficient</i>
<i>problematic</i>	<i>optimize</i>
<i>issue</i>	<i>do this more efficiently</i>
<i>bug</i>	<i>check this result</i>
<i>error</i>	<i>nasty</i>
<i>broken</i>	<i>might hang</i>
<i>exception</i>	<i>fragile</i>
<i>test fail</i>	<i>ugliest</i>
<i>fix testcase</i>	<i>hack</i>
<i>untested</i>	<i>fugly code</i>
<i>needs tested</i>	<i>code violates</i>
<i>more testes</i>	<i>silly</i>
<i>improve this test</i>	<i>don't like</i>
<i>yuck</i>	<i>does not make sense</i>
<i>ugly</i>	<i>this needs work</i>
<i>stupid</i>	<i>find a cleaner way to do this</i>

Fonte: Elaborada pelo autor.

Figura 4.4: Exemplo do princípio de associação entre uma instância de DTAA a uma *Issue*.

Fonte: Elaborada pelo autor.

como uma solução alternativa, deslegante e ineficiente (*kludge*), para fazer o código funcionar na situação em que nenhuma linha é retornada. Esse comentário claramente indica uma DTAA, e nesse trecho de código em que foi admitido uma Dívida Técnica também é encontrado uma *Issue*, mais especificamente um *Code Smell*, que por sua vez, seguindo a abordagem de priorização, é então associado a instância de DTAA. De forma análoga, cada instância de DTAA

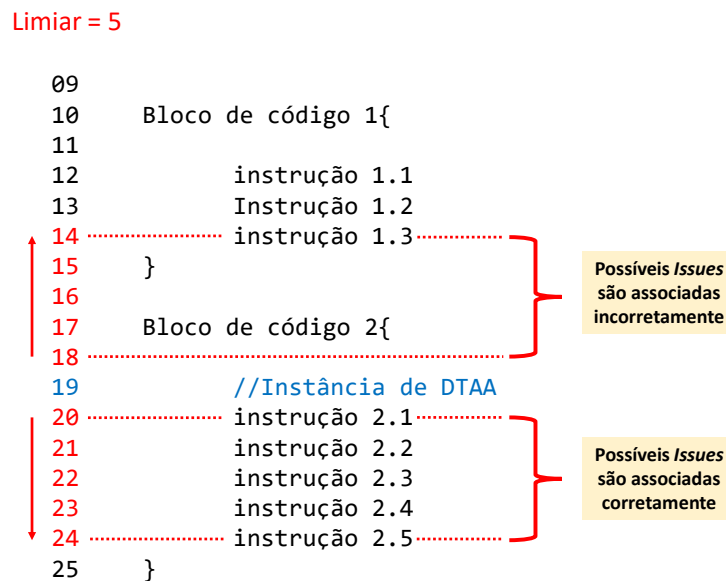
identificada é associada com as possíveis *Issues* encontradas no trecho de código correspondente à DTAA.

A associação descrita anteriormente permite utilizar as características das *Issues* para indicar quais instâncias de DTAA devem ser pagas prioritariamente. De modo que, quanto mais *Issues* estão associadas a uma instância de DTAA específica, maior será a prioridade dessa instância em uma possível etapa de pagamento de Dívida Técnica. Além disso, outras características de uma *Issue*, como sua severidade e tempo de retrabalho, são utilizadas para guiar o desenvolvedor no estabelecimento de prioridade da instância de DTAA associada, em uma atividade de pagamento futuro.

Uma maneira de associar uma instância de DTAA com as *Issues* relacionadas, seria utilizando associação por proximidade. A associação por proximidade consiste em associar as *Issues* localizadas no código-fonte em linhas próximas a localização da instância de DTAA, com base em um limiar, que vai indicar até que distância em linhas no código-fonte que pode-se associar *Issues* com a instância de DTAA.

Entretanto, esse tipo de associação não é utilizado pela abordagem de priorização, uma vez que, pode ocorrer a associação de *Issues* que não são relacionadas com uma instância de DTAA específica. Considere o exemplo apresentado na Figura 4.5, a instância de DTAA está localizada na linha 19 do código-fonte, no Bloco de código 2, usando a associação por proximidade, com limiar = 5, as *Issues* presentes no Bloco de código 1 são associadas a instância de DTAA incorretamente, por estarem em um bloco de código desassociado com a instância de DTAA. Além disso, a dificuldade de definir de um limiar preciso para indicar o deslocamento de associação é outro fator impossibilita o uso deste tipo de associação.

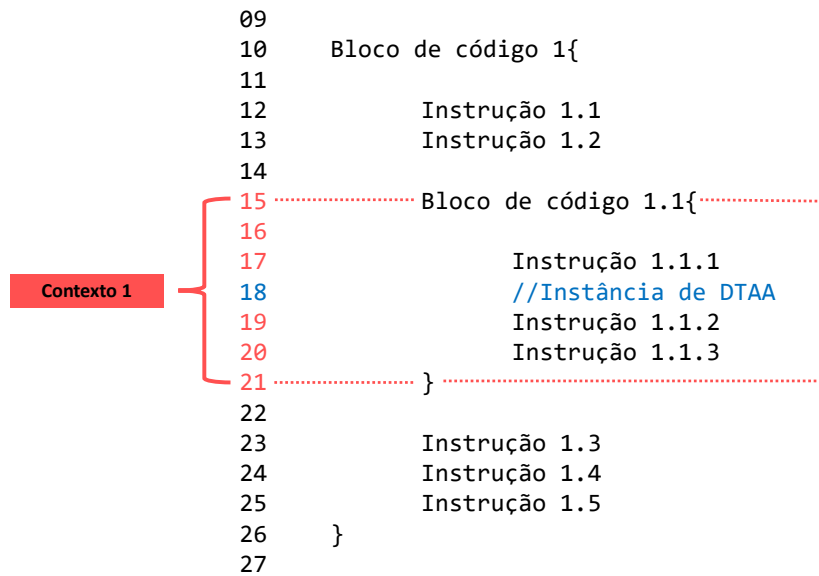
Figura 4.5: Associação entre uma instância de DTAA e *Issues* por proximidade.



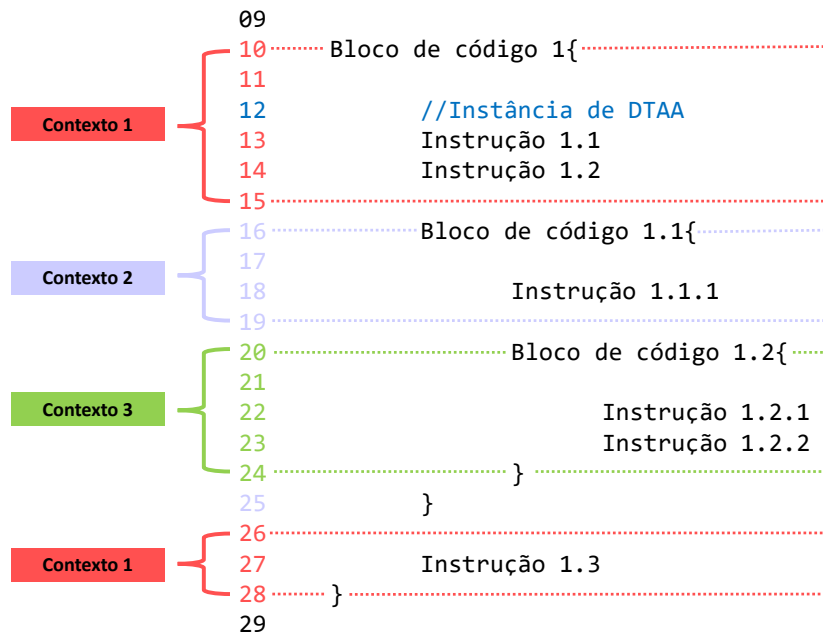
Fonte: Elaborada pelo autor.

Desse modo, a Abordagem de Priorização utiliza a associação de *Issues* por contexto, onde

são associadas somente as *Issues* que estão no mesmo contexto, ou seja, mesmo bloco de código que a instância de DTAA está localizada, na Figura 4.6 são ilustrados dois exemplos desse modelo de associação. Na Figura 4.6(a) a instância de DTAA está localizada na linha 18 e seu contexto é o Bloco de Código 1.1, ou seja, somente as *Issues* presentes dentro desse bloco de código, entre as linhas 15 e 21, são associadas a essa instância de DTAA. Isso justifica-se, pois a situação problema descrita pela instância de DTAA sempre estará localizado dentro do contexto



(a)



(b)

Figura 4.6: Associação entre uma instância de DTAA e *Issues* por contexto.
Fonte: Elaborada pelo autor.

no qual essa instância de DTAA pertence.

Outro exemplo, é apresentado na Figura 4.6(b), neste caso, o contexto da instância de DTAA está entre as linhas 10 e 28 que compõem o Bloco de Código 1, contudo existem outros blocos de código no interior do Bloco de Código 1 que também fazem parte do contexto em que a instância de DTAA e, por sua vez, as *Issues* presentes em todos os blocos de código encadeados são associadas a instância de DTAA em questão. Foi implementado um *Parser* que analisa o código-fonte de uma Classe (considerando o Paradigma de Programação Orientação a Objetos) e realiza a identificação do contexto da instância de DTAA. Após a identificação do contexto o *Parser* associa as *Issues* presentes nesse contexto a instância de DTAA correspondente ao mesmo. O *Parser* foi implementado utilizando a linguagem de programação JAVA e está integrado na ferramenta apresentada na Seção 4.3.

4.2.5 Equação e Lista de Prioridade

Após a formação das associações e consequentemente a obtenção das instâncias de DTAA consideradas como prioritárias, cada uma dessas instâncias são submetidas a uma Equação de Prioridade – Equação 4.1. A Equação de Prioridade tem como objetivo apenas indicar quais são as instâncias mais graves dentre as instâncias de DTAA prioritárias.

A equação considera a descrição da instância de DTAA, na qual possui termos (palavras ou frases) e pode estar associada com um ou mais Termos de Prioridade, descritos na subseção 4.2.3. Como visto, os Termos de Prioridade indicam urgência na resolução da instância de DTAA ou impacto que ela representa, assim, os Termos de prioridade possuem peso 1 na equação, os demais termos possuem peso nulo. Além da descrição das instâncias de DTAA, as características das *Issues* presentes no código-fonte associadas a DTAA, também fazem parte da Equação de prioridade, no caso as características utilizadas são: severidade, tempo de retrabalho e nível de contextos.

$$NP = \sum_{i=1}^n (TP_i) * \sum_{j=1}^m \frac{(SeveridadeIssue_j * TRIssue_j)}{ContextoIssue_j} \quad (4.1)$$

- **NP**: Nível de Prioridade de uma instância de DTAA.
- **TP**: Termo de prioridade presente na descrição da instância de DTAA, todos os termos de prioridade possuem peso 1.
- **SeveridadeIssue**: Severidade da *Issue* associada a instância de DTAA. Severidade *Blocker* peso 4, *Critical* peso 3, *Major* peso 2, *Minor* peso 1.
- **TRIssue**: Tempo de retrabalho da *Issue* associada a instância de DTAA, expresso em horas.
- **ContextoIssue**: Nível do contexto em que a *Issue* foi associada a instância de DTAA.

Em situações na qual a instância de DTAA submetida a Equação de Prioridade não possuir associação com Termos de Prioridade, são consideradas somente as características das *Issues*

associadas a essa instância de DTAA para indicar sua gravidade. De modo análogo, quando uma instância de DTAA não possuir associações com *Issues*, mas possuir associação com um ou mais Termos de Prioridade, somente os Termos de Prioridade são considerados pela equação para indicar a gravidade dessa instância de DTAA.

Na Equação de Prioridade o contexto diminui a gravidade indicada por uma *Issue*. Isso ocorre, devido a localização da *Issue do no contexto*, considere o exemplo de associação por contexto apresentado na Figura 4.6(b), neste caso, as *Issues* localizadas no contexto 2 e 3, estão mais distantes da instância de DTAA e podem afetar o problema descrito por ela de forma menos relevante.

4.3 SATDPriority

A Abordagem de Priorização foi implementada, sendo materializada na ferramenta *SATDPriority*³. A ferramenta foi implementada utilizando a linguagem de programação JAVA com o auxílio da IntelliJ IDEA e do *Framework Spring Boot* fazendo uso do padrão de projeto MVC. Na Figura 4.7 é ilustrado um instantâneo da Ferramenta *SATDPriority*.

Dado como entrada as instâncias de DTAA e as *Issues* detectadas, a Ferramenta *SATDPriority* realiza a identificação de contexto para cada instância de DTAA e aplica Abordagem de Priorização efetivando as associações com Termos de Prioridade e com as *Issues* correspondentes. A ferramenta possibilita ao usuário observar a lista de instâncias de DTAA priorizadas e não priorizadas, provendo todos os detalhes das associações de cada instância o que permite ao usuário inspecionar essas características para tomada de decisão final.

Além disso, o usuário fazendo o uso da ferramenta, pode indicar qual ou quais instâncias já foram pagas, ou mesmo marcar uma instância de DTAA como não prioridade, caso o julgamento formado pela opinião de sua equipe de desenvolvimento considerar que uma instância de DTAA indicada como prioridade pela abordagem não represente uma situação de real prioridade por parte da equipe.

4.4 Considerações Finais

Neste Capítulo foi descrita a Abordagem de Priorização desenvolvida para apoiar o pagamento de DTAA. A abordagem apresentada neste trabalho, pode contribuir para a atividade de priorização de DTAA, ainda pouco explorada. A Abordagem de Priorização consiste em realizar associações com Termos de prioridade presentes na descrição de um comentário DTAA e associações com *Issues*, problemas presentes no código-fonte situados no mesmo contexto em que a instância de DTAA está localizada. A abordagem foi implementada na ferramenta *SATDPriority* para possibilitar sua utilização por desenvolvedores e gestores de Dívida Técnica.

No Capítulo seguinte, são apresentadas os resultados das análises realizadas após a aplicação da Abordagem de Priorização em um conjunto de diferentes repositórios de software, bem como, um Estudo Experimental que teve como objetivo verificar a eficácia da Abordagem de Priorização, comparando seu resultado com o que os desenvolvedores priorizam na prática.

³github.com/brunoslima/SATDPriority

Figura 4.7: Instantâneo da interface da ferramenta SATDPriority.



Fonte: Elaborada pelo autor.

Análises e Estudo Experimental

5.1 Considerações Iniciais

A Abordagem de Priorização de DTAA baseada em associações com Termos de Prioridade e problemas presentes no código-fonte, foi apresentada no Capítulo anterior. A abordagem foi materializada na ferramenta *STADPriority* com o objetivo de possibilitar sua utilização por parte da comunidade acadêmica e em um futuro pela comunidade de desenvolvedores, com seu uso na prática.

Neste Capítulo, é descrita uma Análise Exploratória, com a aplicação da Abordagem de Priorização em um conjunto de repositórios de softwares distintos, com o objetivo de apresentar as características das associações e instâncias de DTAA priorizadas resultantes da abordagem. Além disso, foi executado um Estudo Experimental para verificar a eficácia da abordagem de priorização.

5.2 Análise Exploratória

Uma Análise Exploratória foi realizada considerando os dados obtidos após a aplicação da Abordagem de Priorização para DTAA nos repositórios de software sumarizados na Tabela 5.1. O conjunto de repositórios de software foi escolhido pelas seguintes características: ser de código-fonte aberto, seguindo o paradigma de Orientação a Objetos, implementados na linguagem Java, de diferentes domínios de aplicação, variando em tamanho, em número de comentários e em número de contribuintes. Devido às características, eles são frequentemente utilizados em estudos relacionados à DTAA (Mensah et al., 2016, Zampetti et al., 2017, Huang et al., 2018) e, por esta razão, foram escolhidos para terem seus dados coletados e fazerem parte das análises e do Estudo Experimental.

A Análise Exploratória acerca dos dados resultantes da abordagem foi realizada com o objetivo de verificar as possíveis características e padrões que possam existir no contexto da Abordagem de Priorização em diferentes repositórios de software. Deste modo, a análise foi aplicada

Tabela 5.1: Sumário dos repositórios utilizados no processo de Análise Exploratória.

Repositório	Domínio	Desenvolvedores	LOC	Taxa de comentários
Apache Ant	<i>Building Tool</i>	101	233K	Alta
Apache JMeter	<i>Performance Test</i>	41	354K	Média
jEdit	<i>Text Editor</i>	56	314K	Média
JFreeChart	<i>Char Library</i>	19	317K	Alta
Squirrel	<i>SQL Client</i>	50	725K	Média

Fonte: Elaborada pelo autor.

investigando quantitativamente as associações formadas, os Termos de Prioridade frequentemente associados com DTAA, instância de DTAA com *Issues* presentes no código-fonte, a distribuição dos grupos de severidade das *Issues* associadas, apresentando também quais *Issues* associadas são mais frequentes e quais delas possuem maior severidade.

5.2.1 Associações: Análise Quantitativa

Após a aplicação da Abordagem de Priorização para DTAA em cada um dos repositórios de software, foi investigado primeiramente, a quantidade de instâncias de DTAA que possuem associações, descritas na Tabela 5.2.

Em geral, os repositórios de softwares utilizados no estudo tiveram entre 32,9% e 67,4% das instâncias de DTAA associadas com no mínimo um Termo de Prioridade ou uma Issue presente em seu contexto. No total, considerando todos os repositórios ocorreram 49,3% de instâncias de DTAA com associações após aplicação da Abordagem de Priorização. Apenas o projeto JFreeChart não possui uma proporção de instâncias associadas com *Issues* ou Termos de Prioridade próxima a proporção total de associações, no caso desse repositório, foram 32,9% das instâncias de DTAA que possuem associações. Os demais repositórios de software atingiram proporções próximas ou superiores ao total geral de associações que foi de 49,3%.

No total 28,3% das instâncias de DTAA possuem associações com Termos de Prioridade, enquanto as associações com *Issues* correspondem a 31,8% do total de associações. A quantidade de associações entre instâncias de DTAA e *Issues* presentes em seu contexto, sustenta a existência de correlação entre ambas e possibilita a aplicação da Abordagem de Priorização. Ou seja, se a quantidade de *Issues* associadas com instâncias de DTAA fossem mínima, não existiria correlação clara entre elas, fator que não justificaria aplicação da Abordagem de Priorização.

Tabela 5.2: Quantidade de associações com instâncias de DTAA.

Repositório	Instâncias de DTAA	Associação Abordagem	Associação com Issues	Associação com Termos
Apache Ant	131	74 (56,5%)	51 (38,9%)	48 (36,6%)
Apache JMeter	440	219 (49,8%)	147 (33,4%)	117 (26,6%)
jEdit	135	91 (67,4%)	56 (41,5%)	54 (40,0%)
JFreeChart	152	50 (32,9%)	28 (18,4%)	33 (21,7%)
Squirrel	243	109 (44,9%)	68 (28,0%)	60 (24,7%)
Total	1101	543 (49,3%)	350 (31,8%)	312 (28,3%)

Fonte: Elaborada pelo autor.

5.2.2 Termos de Prioridades frequentemente associados com DTAA

A Abordagem de Priorização utiliza os problemas presentes no código-fonte no contexto de instâncias de DTAA para indicar prioridade. Além disso, são utilizados Termos de Prioridade presentes na descrição dos comentários classificados como DTAA, para indicar quais instâncias de DTAA devem ser pagas prioritariamente. A utilização de Termos de Prioridade em conjunto com problemas presentes no código-fonte é a base da abordagem desenvolvida.

Neste contexto, foi analisado quais Termos de Prioridade são frequentemente associados com instâncias de DTAA e as levam a serem classificadas como prioritárias. Para essa análise, listamos os dez Termos de Prioridade mais frequentes associados com DTAA em cada um dos repositórios de softwares que fazem parte desse estudo, os Termos de Prioridade mais frequentes são apresentados na Tabela 5.3.

Foi analisado manualmente cada Termo frequentemente associado – Tabela 5.3, e foi destacado em negrito os Termos de Prioridade que aparecem com frequência nas instâncias de DTAA priorizadas simultaneamente nos repositórios de softwares estudados. Dentre esses termos, os mais frequentes associados com DTAA simultaneamente em todos os repositórios de softwares são: "*hack*"; "*implement*", "*fix*". Seguidos dos Termos: "*ugly*"; "*check*"; "*exception*"; "*error*"; "*problem*". Desse modo, os Termos de prioridade com maior frequência de associação com DTAA descrevem um erro ou problema específico, codificação classificada como "feia" pelo desenvolvedor, necessidade de verificar e consertar um trecho de código e implementar alguma tarefa que deveria ter sido feita.

Tabela 5.3: Termos de prioridade mais frequentes associadas com instâncias de DTAA.

Apache Ant	Apache JMeter	jEdit	JFreeChart	Squirrel
<i>exception</i>	<i>error</i>	<i>hack</i>	<i>implement</i>	<i>problem</i>
<i>hack</i>	<i>implement</i>	<i>stupid</i>	<i>properly</i>	<i>implement</i>
<i>implement</i>	<i>check</i>	<i>fix</i>	<i>check</i>	<i>bug</i>
<i>ugly</i>	<i>bail out</i>	<i>bug</i>	<i>hack</i>	<i>fix</i>
<i>check</i>	<i>hack</i>	<i>problem</i>	<i>this needs work</i>	<i>check</i>
<i>error</i>	<i>improve</i>	<i>check</i>	<i>fix</i>	<i>hack</i>
<i>bail out</i>	<i>fix</i>	<i>implement</i>	<i>find a cleaner way to do this</i>	<i>ugly</i>
<i>silly</i>	<i>problem</i>	<i>exception</i>	<i>ugly</i>	<i>error</i>
<i>problem</i>	<i>more tests</i>	<i>properly</i>	<i>not work</i>	<i>fail</i>
<i>fix</i>	<i>fail</i>	<i>ugly</i>	<i>wrong</i>	<i>exception</i>

Fonte: Elaborada pelo autor.

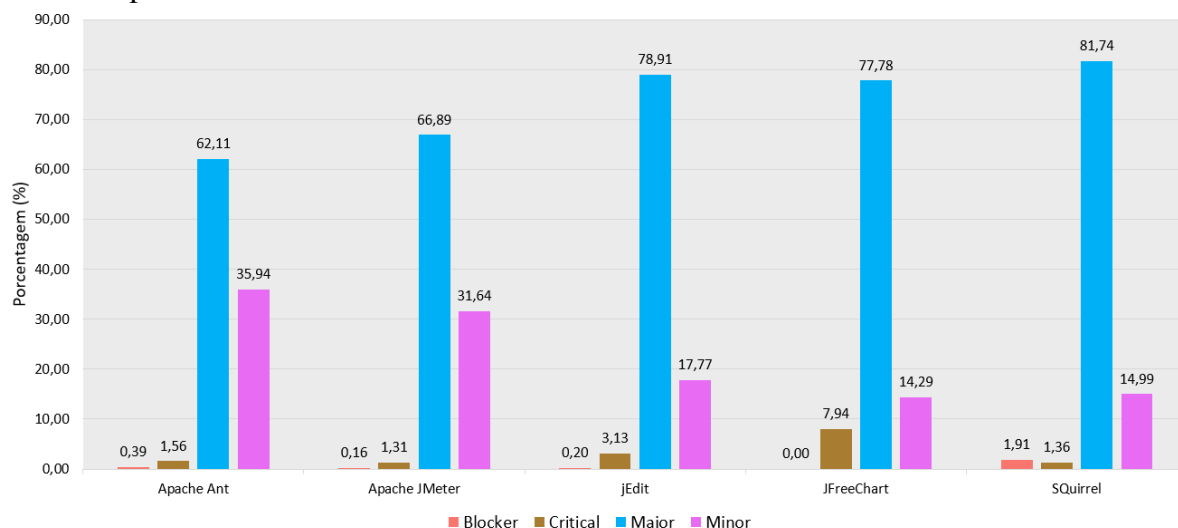
5.2.3 Severidade: Análise Quantitativa

Conforme descrito anteriormente, os níveis de severidade mais altos para *Issues* são *Blocker* e *Critical*, problemas que possuem essas severidades podem afetar o funcionamento do software, bem como sua eficiência ou aspectos de segurança. Os níveis de severidade *Major* e *Minor* estão presentes em problemas que afetam a manutenibilidade do software, embora menos graves, ainda assim são prejudiciais.

Todas as *Issues* que possuem associações com instâncias de DTAA foram agrupadas por nível de severidade, Figura 5.1. Analisando as *Issues* agrupadas por severidade é identificado

um padrão em todos os software, a maioria das *Issues* associadas com DTAA são de severidade *Major* e *Minor* respectivamente. Esse padrão indica que a maioria das instâncias de DTAA priorizadas estão relacionadas com problemas que prejudicam a manutenibilidade do software. Em contrapartida, *Issues* com níveis de severidade mais altos não estão presentes em quantidade significativa, em que a severidade *Blocker* quase não está presente nas associações com DTAA considerando a maioria dos repositórios de software. Isso é esperado, uma vez que problemas de alta severidade geralmente resultam em alto impacto para o software e seu funcionamento, assim, costumam ser solucionados rapidamente pelos desenvolvedores. Apesar dessas situações, as *Issues* possuem relevância quantitativa nas associações formadas nos repositórios de softwares estudados, como apresentado na Tabela 5.2. A seguir, são descritos os tipos e as características dos problemas mais frequentes e de maior severidade associados com instâncias de DTAA consideradas como prioridade pela abordagem.

Figura 5.1: Distribuição das *Issues* associadas com instâncias de DTAA por grau de severidade em cada repositório de software.



Fonte: Elaborada pelo autor.

5.2.4 *Issues* frequentemente associadas com DTAA

Considera-se *Issues* associadas frequentemente os tipos de *Issues* que aparecem em maior quantidade e estão associadas com instâncias de DTAA de acordo com seu contexto. Na Tabela 5.4 estão agrupadas os seis tipos de *Issues* associadas com maior frequência a instâncias de DTAA presentes em cada um dos repositórios de software que compõe o estudo.

Ao observar os diferentes repositórios de softwares e as *Issues* associadas frequentemente com instâncias de DTAA, pode-se notar que em geral uma das *Issues* com maior frequência de associação com DTAA nos repositórios de software estudados é intitulada: "*Refactor this code to not nest more than 3 if/for/while/switch/try statements – ID:738*", que descreve o problema em que um trecho de código possui muitos encadeamentos de instruções de seleção ou de repetição, no caso, mais de três encadeamentos, o que pode resultar na ineficiência da execução do software, além de prejudicar a manutenção deste bloco de código. Outra *Issue* frequentemente

associada com instâncias de DTAA na maioria dos softwares estudados é a intitulada: *"This block of commented-out lines of code should be removed – ID:683"*, reflete a existência de trechos de código comentados que podem ser não funcionais e que devem ser removidos. Note que, é preciso cuidado ao solucionar problemas desse tipo, pois o código pode estar comentado por ser uma implementação futura ainda não completa, que não necessariamente precisa ser removida, isso pode explicar seu alto tempo de retrabalho, o qual é discutido posteriormente.

Após analisar as *Issues* mais frequentes, pode-se observar que, em sua maioria, estão relacionadas com problemas que dificultam uma possível etapa de manutenção, ou seja, possuem severidade *Major* e *Minor*. Esse fator que leva a concluir que os desenvolvedores costumam deixar muitos comentários que caracterizam DTAA em situações que são relacionadas com problemas agravantes a manutenibilidade do código-fonte, e consequentemente do software. Foi constatado apenas um caso isolado, que ocorre no software JFreeChart, em que uma *Issue* frequentemente associada com DTAA possui alta severidade, no caso, severidade *Critical*, nesse caso, consiste no problema de realização de testes de igualdade incorretamente.

Por fim, tem-se ainda que em geral as *Issues* frequentemente associadas com instâncias de DTAA não apresentam um alto Tempo de Retrabalho (TR), ou seja, os problemas ou apontamentos descritos por elas podem ser solucionados sem demandar muito tempo de correção, segundo os dados obtidos do SonarQube. Como descrito na Tabela 5.4 o tempo de retrabalho comum para as *Issues* frequentemente associadas com instância de DTAA varia entre 1 à 20 minutos. Apenas um tipo de *Issue* destoa desse intervalo de tempo de retrabalho, a *Issue* intitulada: *"This block of commented-out lines of code should be removed – ID:683"*, descrevendo o problema da presença de linhas de código comentadas e não removidas, com tempo de retrabalho estimado em 60 minutos.

5.2.5 *Issues* de maior severidade associadas com DTAA

Embora em quantidade muito menor, como apresentado na Figura 5.1, as *Issues* de maior severidade também estão associadas com instâncias de DTAA em todos os softwares estudados. Na Tabela 5.5 são descritas todos os tipos de *Issues* que possuem os mais altos níveis de severidade, *Blocker* e *Critical*, associadas com instâncias de DTAA em cada um dos softwares utilizados nesse estudo.

Dentre as *Issues* associadas de severidade *Blocker*, destaca-se como a mais comum a intitulada: *"Catch Exception instead of Throwable – ID:746"*, referente ao problema de utilizar *Throwable* ao invés de tratar a exceção corretamente. Os problemas com tratamento de exceções são comuns entre as *Issues* associadas com DTAA, além do descrito anteriormente, tem-se o problema: *"Use a logger to log this exception – ID:827"*, que possui severidade *Critical*. Os demais tipos de problemas de severidade *Critical* não apresentam um padrão comum e estão presentes diversificadamente entre os softwares utilizados no estudo. Ao observar o Tempo de Retrabalho (TR) das *Issues* associadas que possuem maior severidade é notado que o tempo de resolução desses tipos de problemas são baixos, pertencentes ao intervalo de resolução entre 5 à 20 minutos. A *Issue* intitulada: *"Remove this hard-coded password – ID:704"* é a única desse grupo possui TR maior que 20 minutos.

Tabela 5.4: *Issues* frequentemente associadas com instâncias de DTAA.

Repositório	Nº	ID	Descrição	Severidade	TR
Apache Ant	1º	738	Refactor this code to not nest more than 3 if/for/while/switch/try statements.	MINOR	20
	2º	665	Move this variable to comply with Java Code Conventions.	MINOR	1
	3º	804	Replace the synchronized class "Vector" by an unsynchronized one such as "ArrayList" or "LinkedList".	MAJOR	20
	4º	835	Move the string literal on the left side of this string comparison.	MAJOR	10
	5º	710	Add the "@Override" annotation above this method signature	MAJOR	5
	6º	727	Either log or rethrow this exception.	MAJOR	10
Apache JMeter	1º	738	Refactor this code to not nest more than 3 if/for/while/switch/try statements.	MINOR	20
	2º	683	This block of commented-out lines of code should be removed.	MAJOR	60
	3º	727	Either log or rethrow this exception.	MAJOR	10
	4º	665	Move this variable to comply with Java Code Conventions.	MINOR	1
	5º	680	Define a constant instead of duplicating this literal "stupid" 3 times.	MINOR	10
	6º	844	Rename this constant name to match the regular expression '[A-Z][A-Z0-9]*([A-Z0-9]+)*\$'.	MAJOR	20
jEdit	1º	699	Replace this usage of System.out or System.err by a logger.	MAJOR	10
	2º	649	Introduce a new variable instead of reusing the parameter "pl".	MAJOR	5
	3º	738	Refactor this code to not nest more than 3 if/for/while/switch/try statements.	MINOR	20
	4º	710	Add the "@Override" annotation above this method signature	MAJOR	5
	5º	665	Move this variable to comply with Java Code Conventions.	MINOR	1
	6º	842	Rename this field name to match the regular expression '[a-z][a-zA-Z0-9]*\$'.	MAJOR	20
JFreeChart	1º	683	This block of commented-out lines of code should be removed.	MAJOR	60
	2º	803	Define and throw a dedicated exception instead of using a generic one.	MAJOR	20
	3º	839	Take the required action to fix the issue indicated by this comment.	MAJOR	20
	4º	785	Equality tests should not be made with floating point values.	CRITICAL	5
	5º	649	Introduce a new variable instead of reusing the parameter "value".	MAJOR	5
	6º	700	Cast one of the operands of this subtraction operation to a "double".	MAJOR	5
Squirrel	1º	842	Rename this field name to match the regular expression '[a-z][a-zA-Z0-9]*\$'.	MAJOR	20
	2º	683	This block of commented-out lines of code should be removed.	MAJOR	60
	3º	831	At most one statement is allowed per line, but 2 statements were found on this line.	MAJOR	1
	4º	819	Move this closing curly brace to the next line.	MAJOR	1
	5º	738	Refactor this code to not nest more than 3 if/for/while/switch/try statements.	MINOR	20
	6º	680	Define a constant instead of duplicating this literal "varchar" 3 times.	MINOR	10

Fonte: Elaborada pelo autor.

Tabela 5.5: Issues de maior severidade associadas com instâncias de DTAA.

Repositório	ID	Descrição	Severidade	TR
Apache Ant	746	Catch Exception instead of Throwable.	BLOCKER	20
	757	Remove this call to "wait" or move it into a "while" loop.	CRITICAL	20
	827	Use a logger to log this exception.	CRITICAL	10
	720	Either remove this useless object instantiation of class "name" or use it	CRITICAL	5
	689	Make this array "private".	CRITICAL	N/A
Apache JMeter	746	Catch Exception instead of Throwable.	BLOCKER	20
	704	Remove this hard-coded password.	CRITICAL	30
	824	Remove this call to "exit" or ensure it is really required.	CRITICAL	20
	827	Use a logger to log this exception.	CRITICAL	10
	785	Equality tests should not be made with floating point values.	CRITICAL	5
jEdit	671	Add a call to super.finalize() at the end of this Object.finalize() implementation.	BLOCKER	5
	647	Do not override the Object.finalize() method.	CRITICAL	20
	824	Remove this call to "exit" or ensure it is really required.	CRITICAL	20
	732	End this switch case with an unconditional break, continue, return or throw statement.	CRITICAL	20
	827	Use a logger to log this exception.	CRITICAL	10
	720	Either remove this useless object instantiation of class "name" or use it	CRITICAL	5
	785	Equality tests should not be made with floating point values.	CRITICAL	5
JFreeChart	785	Equality tests should not be made with floating point values.	CRITICAL	5
Squirrel	746	Catch Exception instead of Throwable.	BLOCKER	20
	704	Remove this hard-coded password.	CRITICAL	30
	824	Remove this call to "exit" or ensure it is really required.	CRITICAL	20
	827	Use a logger to log this exception.	CRITICAL	10

Fonte: Elaborada pelo autor.

5.3 Estudo Experimental: Eficácia da Abordagem de priorização

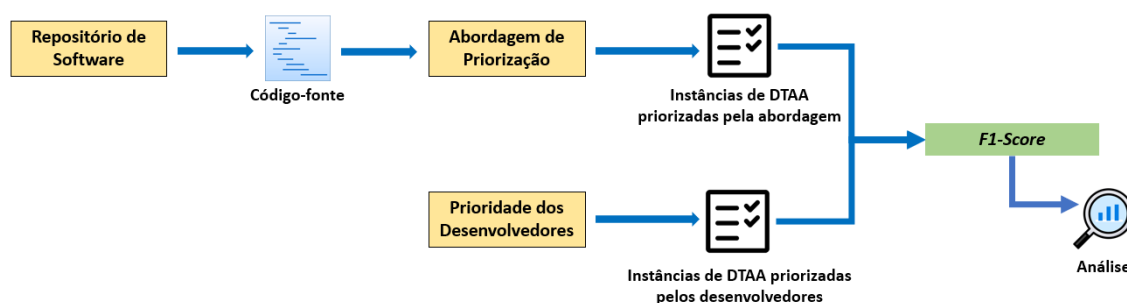
Após a Análise Exploratória, um Estudo Experimental foi conduzido para verificar a eficácia da Abordagem de Priorização. Na Figura 5.2 é ilustrado uma visão geral do Estudo Experimental realizado. O Estudo Experimental foi aplicado e repetido para cada um dos repositórios de softwares descritos anteriormente na Tabela 5.1.

O Estudo Experimental consiste em dado um repositório de software e seu código-fonte em uma versão antiga no processo de desenvolvimento, chamada de versão base, é então identificado o conjunto de instâncias de DTAA e posteriormente aplicada a Abordagem de Priorização. Após essa etapa, é obtida uma listas de instâncias de DTAA que foram consideradas como prioritárias pela Abordagem de Priorização. Em seguida, essa lista é comparada com uma lista de instâncias de DTAA consideradas como prioridade por parte dos desenvolvedores que contribuem no repositório de software utilizado.

O processo de obtenção da lista de instâncias de DTAA consideradas pelos desenvolvedores como prioridade em uma etapa de pagamento de Dívida Técnica é ilustrado na Figura 5.3. Primeiramente, é identificado as instâncias de DTAA presentes na versão base (Figura 5.3 – versão n), utilizando a ferramenta *SATD Detector*. Em seguida, também são identificadas as instâncias de DTAA presentes em cada uma das versões subsequentes a versão base (Figura 5.3 – versão $n+1$, versão $n+2$, ..., versão $n+k$). Desse modo, são obtidas diferentes listas de instâncias de DTAA, uma para a versão base e uma para cada versões subsequentes do software.

Por fim, são identificadas as instâncias de DTAA que foram pagas pelos desenvolvedores ao longo do desenvolvimento do software e disponibilização de suas múltiplas versões. Para isso, são comparadas as instâncias de DTAA presentes na versão base, com as instâncias de DTAA presentes nas versões subsequentes do software. As instâncias de DTAA que estão na versão base, mas não estão em versões subsequentes do softwares, são instâncias consideradas pagas prioritariamente por parte dos desenvolvedores do software.

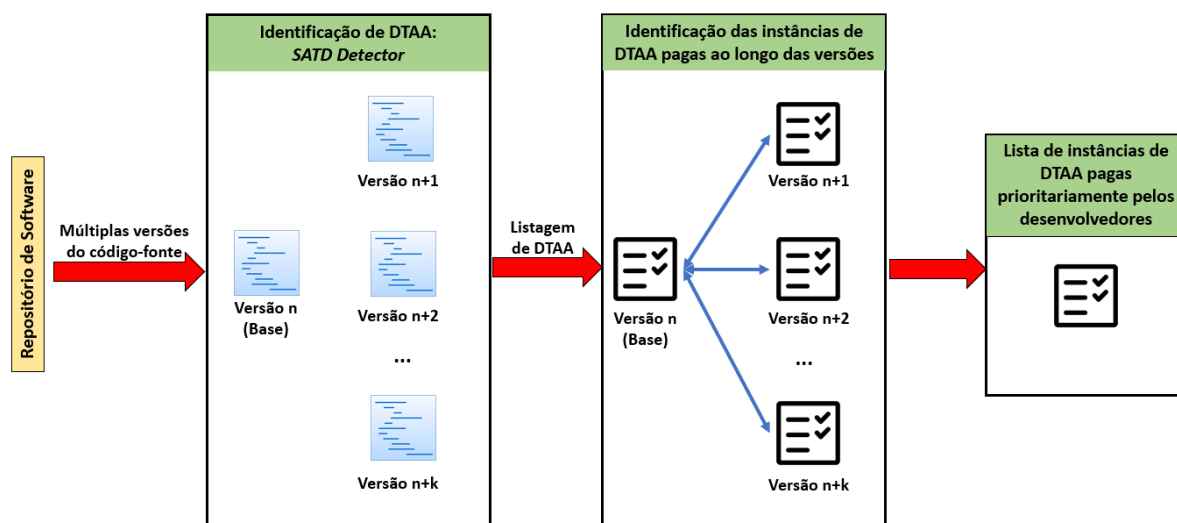
Figura 5.2: Visão geral do Estudo Experimental: Verificação da eficácia da abordagem de priorização.



Fonte: Elaborada pelo autor.

Após a obtenção da lista de instância de DTAA prioritárias resultante da Abordagem de

Figura 5.3: Processo de obtenção das instâncias de DTAA pagas prioritariamente pelos desenvolvedores ao longo do desenvolvimento do software.



Fonte: Elaborada pelo autor.

priorização e da lista de instâncias de DTAA consideradas prioritárias pelos desenvolvedores, ambas foram comparadas por meio da métrica *F1-Score*. A *F1-Score* é obtida pela combinação de duas outras métricas, Precisão e Revocação.

Para a aplicação das métricas *F1-Score*, Precisão e Revocação, primeiramente foram calculadas quatro estatísticas básicas: os verdadeiros positivos e negativos, falsos positivos e negativos. Na Figura 5.4 é ilustrado a modelagem para obtenção dos verdadeiros positivos e negativos, falsos positivos e negativos. Nesse contexto, considera-se Verdadeiro Positivo (VP) – a instância de DTAA priorizada pela Abordagem de Priorização que realmente é prioridade para os desenvolvedores em etapas de pagamento. Assim, entende-se Falso Positivo (FP) – a instância de DTAA que é considerada prioritária pela Abordagem de Priorização, mas não é prioridade na visão dos desenvolvedores. Denomina-se Falso Negativo (FN) – a instância de DTAA não classificada como prioridade pela Abordagem de priorização, mas que realmente é prioridade para os desenvolvedores. Por fim, Verdadeiro Negativo (VN) – a instância de DTAA que não é prioridade tanto para a Abordagem de Priorização quanto para os desenvolvedores.

A seguir são formalmente definidas as métricas: Precisão (P) – Equação 5.1, maior Precisão indica que as instâncias de DTAA classificadas como prioritárias pela abordagem são mais prováveis de serem verdadeiras instâncias de DTAA prioritárias, situação que pode auxiliar os desenvolvedores a identificar com mais eficácia quais instâncias de DTAA são prioritárias, evitando assim o prejuízo de tempo com uma verificação manual. Revocação (R) – Equação 5.2, possuir um alto Revocação indica que a Abordagem de Priorização pode identificar um maior número de instâncias de DTAA que realmente são prioritárias, esse cenário contribui para identificar um maior conjunto de instâncias de DTAA que devem ser priorizadas e evitar perder instâncias importantes que deveriam ser consideradas prioridades em uma etapa de pagamento

Figura 5.4: Tabela demonstrativa para Verdadeiros positivos (VP), Verdadeiros negativos (VN), Falsos positivos (FP) e Falsos negativos (FN).

		PRIORIDADE DOS DESENVOLVEDORES	
		DTAA Prioritária	DTAA Não Prioritária
ABORDAGEM DE PRIORIZAÇÃO	DTAA Prioritária	Verdadeiro Positivo (VP)	Falso Positivo (FP)
	DTAA Não Prioritária	Falso Negativo (FN)	Verdadeiro Negativo (VN)

Fonte: Elaborada pelo autor.

futuro.

Por fim, *F1-Score* consiste na relação da média harmônica entre Precisão e Revocação, expressa pela Equação 5.3. Todas as métricas resultam em valores pertencentes ao intervalo 0,0 e 1,0, quanto mais próximo do valor de referência 1,0 melhor a precisão, Revocação e *F1-Score*.

$$P = VP / (VP + FP) \quad (5.1)$$

$$R = VP / (VP + FN) \quad (5.2)$$

$$F1 = (2 \times P \times R) / (P + R) \quad (5.3)$$

O Estudo Experimental para verificação da eficácia da Abordagem de Priorização foi aplicado em cada um dos repositórios de software apresentados anteriormente na Tabela 5.1. Os resultados do Estudo Experimental são sintetizados na Tabela 5.6 que por sua vez, descreve o valor obtido de cada uma das métricas de *F1-Score*, Precisão e Revocação, a quantidade de instâncias de DTAA priorizadas pela abordagem, pelos desenvolvedores, além expor quantitativamente as estatísticas básicas: Verdadeiro Positivo (VP), Falso Positivo (FP), Falso Negativo (FN) e Verdadeiro Negativo (VN).

Ao analisar os resultados descritos na Tabela 5.6 obtidos após o Estudo Experimental,

Tabela 5.6: Eficácia da Abordagem de Priorização.

Repositório de Software	Priorizadas Abordagem	Priorizadas Desenvolvedores	VP	FP	FN	VN	Precisão	Revocação	F1-Score
Apache Ant	74	49	30	44	18	39	0,405	0,625	0,492
Apache JMeter	219	176	102	117	85	136	0,466	0,545	0,502
jEdit	91	61	43	48	20	24	0,473	0,683	0,558
JFreeChart	50	57	33	17	27	75	0,660	0,550	0,600
Squirrel	109	188	99	10	90	44	0,908	0,524	0,664
Média							0,582	0,585	0,563

Fonte: Elaborada pelo autor.

nota-se que a pontuação *F1-Score* média obtida pela Abordagem de Priorização foi de 0,563, com Precisão e Revocação em média de 0,582 e 0,585 respectivamente. Pode-se concluir que a Abordagem de Priorização não possui alta eficácia, ou seja, os resultados obtidos com a aplicação das métricas *F1-Score*, Precisão e Revocação não estão totalmente próximos ao valor de referência 1,0, que indica priorização próxima a realidade considerando as métricas. Entretanto, a Abordagem de Priorização não pode ser descartada, uma vez que, tanto em Precisão quanto em Revocação e consequentemente em pontuação *F1-Score* seus valores resultante estão mais próximos de 1,0 do que do valor de referência 0,0, este último representa uma pior eficácia.

Um dos pontos de maior interesse é a Precisão da Abordagem de Priorização, uma vez que, no contexto de Dívida Técnica não é viável o pagamento de todas ou da maioria das instâncias da Dívida Técnica. Consequentemente, o Revocação baixo pode ser tolerado, contudo obter uma alta Precisão é fundamental para possibilitar a utilização prática da abordagem. Nesse caso, por ser uma abordagem automatizada e pela escassez (até o presente momento deste trabalho) de abordagens para priorização de DTAA, a Precisão obtida pela Abordagem de Priorização embora não ideal, pode ser importante para tornar a Abordagem de Priorização não como uma indicação de decisão final e incontestável, mas sim como um guia para auxiliar os desenvolvedores na etapa de pagamento de DTAA, reduzindo a quantidade de instâncias de DTAA que o desenvolvedor deve analisar para definir a priorização e deixando sempre a responsabilidade e decisão final para os próprios desenvolvedores.

Após a execução do Estudo Experimental e análise dos resultados, a Abordagem de Priorização resultou em muitos Falsos Positivos (FP) em alguns projetos de software dentre eles: Apache Ant, Apache JMeter e jEdit (Tabela 5.6), isso foi um fator determinante por diminuir a pontuação da Precisão obtida pela Abordagem de Priorização. Entretanto, para verificação de eficácia, o Estudo Experimental comparou os resultados da Abordagem de Priorização com as instâncias de DTAA que os desenvolvedores realmente pagaram prioritariamente ao longo do desenvolvimento do software. Neste contexto, sabe-se que não é possível pagar toda a Dívida Técnica e ainda tem-se relatos na literatura que o número de instâncias de DTAA são removidas em baixa quantidade ao longo do desenvolvimento (Maldonado et al., 2017, Zampetti et al., 2018). Desse modo, alguns Falsos Positivos (FP) resultantes da Abordagem de Priorização podem ser Verdadeiros Positivos (VP), ou seja, podem ser instâncias de DTAA que realmente sejam prioridade, mas que os desenvolvedores por falta de conhecimento ou por falta de tempo, devido a prazos apertados, não conseguiram dedicar seu pagamento, isso caracteriza uma pos-

sível ameaça para a validação.

Visando mitigar essa possível ameaça, foi realizada uma inspeção nos Falsos Positivos (FP) obtidos identificando quais deles possuem associações com *Issues* de severidade mais alta, *Blocker* e *Critical*. Os Falso Positivos (FP) que possuem associação com problemas com uma dessas severidades foram considerados Verdadeiros Positivos (VP), uma vez que, problemas de alta severidade possuem resolução de vital importância para o bom funcionamento do software, e por isso são prioritários. Os resultados após essa inspeção e tratamento dos Falsos Positivos são descritos na Tabela 5.7, foi observado que a Precisão da Abordagem de Priorização passou de 0,582 para 0,614, a Revocação passou de 0,585 para 0,598 e o *F1-Score* passou de 0,563 para 0,587, pontuações relativamente acrescidas, contudo ainda distantes da ideal ou ótima considerando o valor de referência 1,0.

Tabela 5.7: Eficácia da Abordagem de Priorização com tratamento de Falsos Positivos (FP).

Repositório de Software	Priorizadas Abordagem (FP)	Priorizadas Desenvolvedores	VP	FP	FN	VN	Precisão	Revocação	<i>F1-Score</i>
Apache Ant	74	49	32	42	18	39	0,432	0,640	0,516
Apache JMeter	219	176	103	116	85	136	0,470	0,548	0,506
jEdit	91	61	49	42	20	24	0,538	0,710	0,613
JFreeChart	50	57	35	15	27	75	0,700	0,565	0,625
Squirrel	109	188	101	8	90	44	0,927	0,529	0,673
Média							0,614	0,598	0,587

Fonte: Elaborada pelo autor.

Adicionalmente, foi aplicado em cada software a Abordagem de Priorização realizando a etapa de associações somente com Termos de prioridade e sem associar *Issues*, problemas presentes no código-fonte. Em contrapartida, também foi aplicado a Abordagem de Priorização realizando a etapa de associação somente com *Issues* e sem associar Termos de prioridade. O objetivo consiste em analisar se a Abordagem de priorização completa efetuando associações com Termos de prioridade e *Issues* possui melhores resultados do que associar somente Termos de prioridade ou somente *Issue*. Para isso, a mesma configuração e processo do Estudo Experimental para verificar a eficácia da abordagem foi utilizado.

Na Tabela 5.8 são apresentados os resultados ao utilizar a Abordagem de Priorização associando somente Termos de Prioridade com DTAA e na Tabela 5.9 são apresentados os resultados ao utilizar a Abordagem de Priorização associando somente *Issues* com DTAA. Em média, a diferença entre ambas, considerando o *F1-Score*, Precisão e Revocação, é mínima. Contudo a Abordagem utilizando somente associações com *Issues* possui vantagem, considerando os valores resultantes das métricas. A Abordagem associando somente Termos de prioridade com DTAA possui *F1-Score* de 0,411, Precisão de 0,554 e Revocação de 0,350, enquanto a Abordagem associando somente *Issues* possui *F1-Score*, Precisão e Revocação ligeiramente superiores, 0,437, 0,578 e 0,381 respectivamente. Portanto, ao associar somente *Issues* com DTAA visando priorização desta última melhora a identificação em todas as métricas, ou seja, identifica com maior precisão as instâncias de DTAA que devem ser priorizadas e contempla um conjunto maior de instâncias de DTAA que devem ser realmente priorizadas.

Ao comparar os resultados obtidos na utilização da Abordagem de Priorização completa

Tabela 5.8: Eficácia da Abordagem de Priorização associando somente Termos de prioridade.

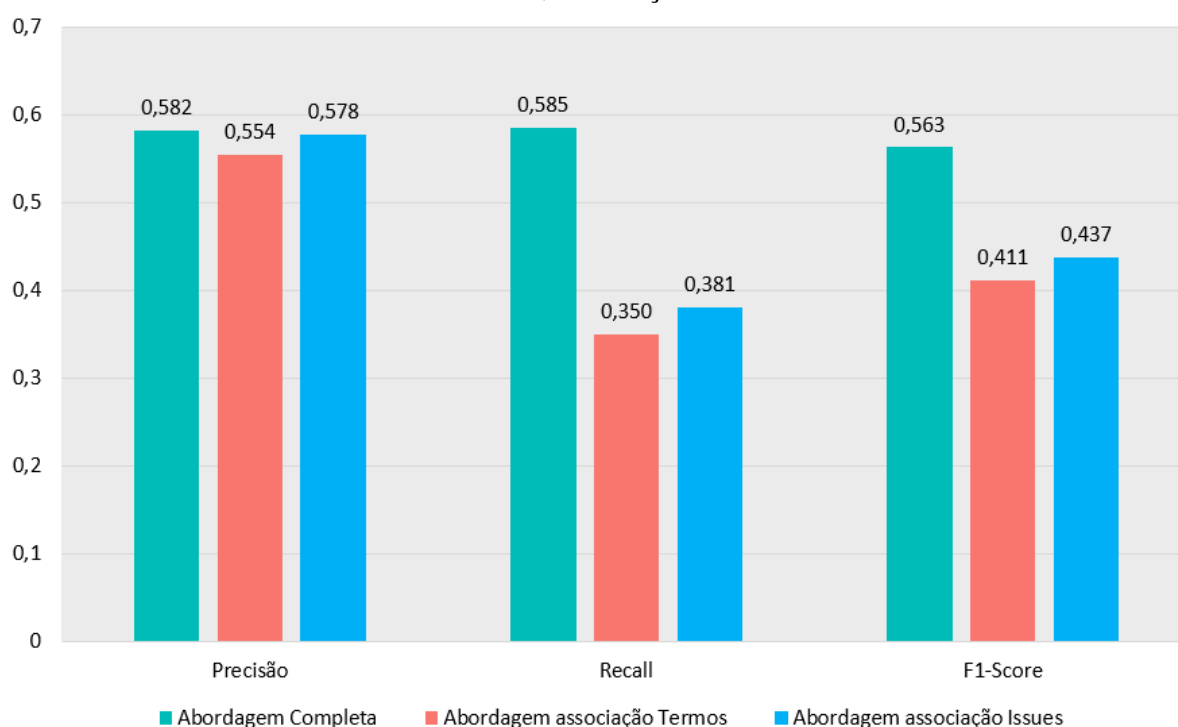
Repositório de Software	Priorizadas Abordagem (Termos)	Priorizadas Desenvolvedores	VP	FP	FN	VN	Precisão	Revocação	F1-Score
Apache Ant	48	49	18	30	30	53	0,375	0,375	0,375
Apache JMeter	117	176	39	78	95	228	0,333	0,291	0,311
jEdit	54	61	29	25	34	47	0,537	0,460	0,496
JFreeChart	33	57	20	13	40	79	0,606	0,333	0,430
Squirrel	60	188	55	5	133	50	0,917	0,293	0,444
		Média					0,554	0,350	0,411

Fonte: Elaborada pelo autor.

Tabela 5.9: Eficácia da Abordagem de Priorização associando somente *Issues*.

Repositório de Software	Priorizadas Abordagem (Issues)	Priorizadas Desenvolvedores	VP	FP	FN	VN	Precisão	Revocação	F1-Score
Apache Ant	51	49	24	27	24	56	0,471	0,500	0,485
Apache JMeter	147	176	43	104	93	200	0,293	0,316	0,304
jEdit	56	61	24	32	38	41	0,429	0,387	0,407
JFreeChart	28	57	22	6	37	87	0,786	0,373	0,506
Squirrel	68	188	62	6	127	48	0,912	0,328	0,482
		Média					0,578	0,381	0,437

Fonte: Elaborada pelo autor.

Figura 5.5: Comparação entre a Abordagem Priorização completa, Abordagem associando somente Termos de prioridade e Abordagem associando somente *Issues*, considerando o valor em média de cada uma das métricas: Precisão, Revocação e F1-Score.

Fonte: Elaborada pelo autor.

(associando Termos de Prioridade e *Issues*) – Tabela 5.6 com os resultados obtidos ao utilizar a Abordagem de Priorização somente associando Termos de Prioridade – Tabela 5.8 e com os resultados obtidos ao utilizar a Abordagem de Priorização somente associando *Issues* – Ta-

bela 5.9, pode-se observar uma melhor eficácia para priorizar DTAA ao utilizar a Abordagem de Priorização completa, na Figura 5.5 é ilustrado um gráfico comparativo entre a eficácia de cada uma dessas situações. Observando as três métricas utilizadas, em média, a Abordagem de Priorização completa foi superior em todos os aspectos considerados (*F1-Score*: 0,563; *Precisão*: 0,582; *Revocação*: 0,585), se comparado com a Abordagem de priorização associando somente Termos de prioridade (*F1-Score*: 0,411; *Precisão*: 0,554; *Revocação*: 0,350) e com a Abordagem de priorização associando somente *Issues* (*F1-Score*: 0,437; *Precisão*: 0,578; *Revocação*: 0,381).

Portanto, considerando o objetivo dessa última análise pode-se concluir, com base nessa configuração do estudo, é mais eficiente utilizar a descrição de um comentário que indica uma instância de DTAA e dos termos presentes nele, em conjunto com os problemas presentes no código-fonte em seu contexto para indicar prioridade de pagamento de uma instância de DTAA, do que utilizar somente a descrição ou somente os problemas presentes no código-fonte para indicar prioridade. Isso, demonstra que o princípio de associação utilizado pela Abordagem de Priorização desenvolvida por este trabalho deve ser considerado em pesquisas futuras relacionadas com a atividade de priorização de DTAA.

5.4 Considerações Finais

Neste Capítulo foram apresentadas análises e um Estudo Experimental para verificar a eficácia da Abordagem de Priorização. Primeiramente foi realizado uma Análise Exploratória acerca dos conjuntos de dados para evidenciar características e padrões das associações formadas.

A Análise Exploratória revelou que os Termos de Prioridade mais frequentes associados com DTAA descrevem um erro ou problema, codificação "feia", necessidade de consertar e implementar um trecho de código. Além disso, foi descoberto que as *Issues* associadas com DTAA são referentes a problemas que afetam a manutenibilidade do código-fonte, dentre elas a existência de muitos blocos de códigos encadeados e trechos de códigos comentados que devem ser removidos ou modificados. Por fim, a análise também demonstrou que as *Issues* de maior severidade associadas com DTAA estão relacionadas com o tratamento inadequado de exceções e que em geral as *Issues* associadas possuem um pequeno tempo de retrabalho para sua resolução.

Considerando a eficácia da Abordagem de Priorização, o Estudo Experimental evidenciou que a abordagem apesar de não obter um alto *F1-Score* não pode ser descartada, pois possui uma precisão e uma pontuação de revocação considerável, a qual possibilita o uso da Abordagem de Priorização como um guia para desenvolvedores na priorização de DTAA em etapas de pagamento de Dívida Técnica. Por fim, uma vez que, Abordagem de Priorização faz uso da descrição de comentários DTAA e das características dos problemas presentes no código-fonte para priorizar DTAA, o Estudo Experimental destacou a abordagem como promissora, pois ela obteve maior precisão e revocação ao priorizar DTAA quando comparada com a priorização utilizando somente a descrição de comentários DTAA e com a priorização utilizando somente as características dos problemas presentes no código-fonte associados com DTAA.

Considerações Finais

O acúmulo de Dívida Técnica é comprovadamente prejudicial à qualidade do software e à sua capacidade de manutenção (Brown et al., 2010, Seaman et al., 2015, Rios et al., 2018). Com o passar dos anos, um novo conceito envolvendo essa metáfora foi apresentado na literatura por Potdar e Shihab (2014), que descreveu a situação na qual comentários presentes no código-fonte introduzidos por desenvolvedores indicam instâncias de Dívida Técnica, as quais foram conceitualmente definidas como Auto-Admitidas. Os estudos envolvendo DTAA são recentes, e necessitam de maior investigação, até o momento são concentrados em três vertentes: Identificação, Análise e Priorização de DTAA.

A identificação eficiente de instâncias da Dívida Técnica foi o primeiro passo para possibilitar mais estudos envolvendo DTAA. As abordagens para identificação de DTAA são, até o momento baseadas em: Padrões Textuais (Potdar e Shihab, 2014, Maldonado e Shihab, 2015, de Freitas Farias et al., 2020), Processamento de Linguagem Natural (NLP) (Maldonado et al., 2017) e por Classificação Composta (Huang et al., 2018).

Com a possibilidade de identificação de modo eficiente de instâncias de DTAA, um conjunto de estudos foram direcionadas para analisar as características, evolução e impactos causados pela presença de DTAA em repositórios de software. Na perspectiva de impacto, foi constatado que a quantidade de defeitos presentes no software não está relacionada com a DTAA, além disso, na perspectiva de qualidade de código, os aspectos complexidade, acoplamento e legibilidade não aparentam influenciar na introdução de novas instâncias de DTAA. Contudo, foi identificado por Wehaibi et al. (2016) que as mudanças relacionadas com DTAA são mais difíceis de serem realizadas, fator prejudicial a capacidade de manutenção do software.

Devido ao fato da presença de DTAA prejudicar a manutenibilidade do software, surge a necessidade de possibilitar mecanismos que apoiem o processo de pagamento da dívida formada. Nesse contexto, Mensah et al. (2018) apresentaram uma abordagem de priorização em 6 etapas, a qual utiliza de identificadores textuais para possibilitar apontar tarefas de DTAA que devem ser priorizadas em uma possível etapa de pagamento de Dívida Técnica.

Como discutido anteriormente, apesar de que a abordagem de priorização de Mensah et al. (2018) tenha apresentado resultados benéficos para a atividade de priorização de DTAA, esse processo ainda é restrito por observar apenas a descrição textual dos comentários que indicam DTAA para apontar a prioridade de cada instância de DTAA em uma possível etapa de pagamento de Dívida Técnica. Contudo, o código-fonte e os possíveis problemas nele presentes não são considerados em sua abordagem de priorização. Observar as características dos problemas presentes no trecho do código-fonte na qual foi cometida a DTAA é um fator importante para mensurar a prioridade dessa na etapa de pagamento de Dívida Técnica.

Neste contexto, este trabalho apresentou uma Abordagem de Priorização que faz uso de termos presentes nas descrições de comentários DTAA e de problemas presentes no bloco de código associado aos comentários DTAA para indicar priorização da Dívida Técnica. Dado instâncias de DTAA identificadas pela ferramenta *SATD Detector e Issues*, problemas de código-fonte identificados pelo SonarQube, a Abordagem de Priorização identifica e realiza associações de Termos de Prioridade e *Issues* para indicar quais instâncias de DTAA devem ser priorizadas em uma futura etapa de pagamento de Dívida Técnica.

6.1 Contribuições

Até o presente momento, a abordagem de priorização para DTAA presente na literatura apresentada por Mensah et al. (2018) utiliza apenas a descrição de uma instâncias de DTAA e os termos presentes nela para indicar priorização. A Abordagem de Priorização apresentada por este trabalho faz uso da descrição de instâncias de DTAA e dos termos presentes nela, mas também utiliza problemas presentes no código-fonte para priorizar DTAA.

Após a realização de experimentos para verificar a eficácia da abordagem desenvolvida, foi constatado que utilizar a descrição de comentários DTAA e os problemas presentes no código-fonte associados a esse comentário resulta em maior precisão para priorizar DTAA se comparado com a utilização de somente a descrição do comentário DTAA ou de somente os problemas associados com DTAA. Esta caracteriza-se como a principal contribuição deste trabalho, uma vez que, possibilita e torna promissora pesquisas futuras para priorizar DTAA levando em consideração também os possíveis problemas nos trechos de código em que situa-se a instância de DTAA.

A Abordagem de Priorização apresentada por este trabalho também é uma contribuição, embora não possua uma precisão suficiente para ser utilizada como direcionamento inquestionável, por sua vez, pode ser utilizada para auxiliar desenvolvedores e gestores a pagar instâncias de DTAA, pois seu uso diminui o conjunto de instâncias que devem ser analisada em uma etapa manual e escolhidas para priorização.

6.2 Limitações e Ameaças à Validade do Estudo

A principal limitação deste trabalho consiste em fazer uso de um conjunto de dados obtidos após inspeção de projetos de software implementados utilizando a linguagem JAVA. Desse modo, a abordagem foi idealizada e avaliada em um primeiro momento considerando softwares imple-

mentados em linguagem JAVA. Essa é uma limitação comum entre os trabalhos presentes na literatura no contexto de DTAA, o que leva a indicar um futuro de estudos que ampliem as abordagens e avaliações considerando outras linguagens.

Assim como toda pesquisa, existem riscos com relação ao desenvolvimento da abordagem que podem resultar em ameaças para sua validação. O primeiro fator de risco está relacionado ao conjunto de dados, uma vez que foi selecionado o código-fonte de alguns repositórios de softwares. Essa amostragem de repositórios de software pode impactar na análise final do estudo, bem como a generalização dos resultados com a aplicação da abordagem. Para reduzir esse risco, foram escolhidos repositórios de software de diferentes domínios, porte, quantidade de desenvolvedores e com taxas de comentários variantes entre: *muito altas*, *altas*, *médias* e *baixas*, com base na mensuração indicada pelo OpenHub. Apesar desta medida, o risco ainda está presente.

Além disso, em cada código-fonte utilizado podem existir blocos de código não utilizáveis, chamados de "código morto", comentados para remoção futura. Alguns desses blocos de código podem possuir instâncias de DTAA consideradas prioritárias pela abordagem e assim impactar no resultado final do estudo, uma vez que, blocos de código com essa característica não precisam ser priorizados, pois já é conhecido pelos desenvolvedores e apenas devem ser removidos juntamente com o código descartado.

6.3 Disseminação do Trabalho

Com o proposito de possibilitar a disseminação deste trabalho, uso e aprimoramento da Abordagem de Priorização desenvolvida, assim como a replicação do estudo, a ferramenta *SATDPriority* é disponibilizada para a comunidade científica e está situada em um repositório¹ no GitHub. O *Parser* responsável pela identificação de contexto e associação de instâncias de DTAA com *Issues* está integrado na ferramenta *SATDPriority* e também está disponível no repositório.

Adicionalmente, o conjunto de dados² utilizado e resultante por este trabalho também está disponível para uso por parte da comunidade científica em pesquisas futuras. O conjunto de dados, o *Parser* e a ferramenta *SATDPriority* são contribuições secundárias deste trabalho.

6.4 Trabalhos Futuros

Como trabalhos futuros, pode-se incrementar e, conseqüentemente, melhorar o dicionário de Termos de Prioridade fazendo análise manual de outros conjuntos de instâncias de DTAA identificadas em outros projetos de softwares que não os explorados neste trabalho. Outro trabalho futuro, pode-se em conjunto com grupos de pesquisa e desenvolvedores, realizar a obtenção de um conjunto de comentários DTAA classificados manualmente como prioridade em comum acordo considerando opiniões de pesquisadores e desenvolvedores. Desse modo, obter um conjunto de dados que pode ser utilizado como um oráculo para estudos futuros relacionados com priorização e pagamento de DTAA.

¹github.com/brunoslima/SATDPriority

²github.com/brunoslima/SATDPriority-Dataset

Além disso, pode-se realizar um estudo em larga escala, ou seja, fazer uso de um maior conjunto de dados envolvendo diversos repositórios ou ecossistemas de software, e assim, subsidiar a identificação de um grande conjunto de instâncias de DTAA. Essa ação pode ajudar a alcançar uma proximidade de generalização dos resultados obtidos pela Abordagem de Priorização.

Por fim, como última sugestão de trabalho futuro pode-se melhorar a implementação da Abordagem de Priorização facilitando a utilização do usuário, por meio do uso de abordagens visuais para facilitar o entendimento das instâncias de DTAA apontadas como prioridade, uma vez que, atualmente essas são apresentadas na forma de listagem simples. Desse modo, em paralelo resulta em uma contribuição para a área de Visualização de Dívida Técnica, na qual estudos ainda são incipientes (Rios et al., 2018).

Adicionalmente, na fase inicial do trabalho, foi realizada uma Revisão Sistemática da Literatura (RSL). A RSL visá apresentar para a comunidade científica o panorama atual acerca dos estudos centrados em Dívidas Técnicas Auto-Admitidas, bem como as atividades de Gestão de Dívida Técnica aplicadas em DTAA, abordagens existentes e conjunto de dados utilizados e resultantes de diversos estudos. Um artigo que apresenta os resultados da RSL será disponibilizado no futuro.

Referências

ALVES, N. S.; MENDES, T. S.; DE MENDONÇA, M. G.; SPÍNOLA, R. O.; SHULL, F.; SEAMAN, C. Identification and management of technical debt: A systematic mapping study. *Information and Software Technology*, v. 70, p. 100 – 121, 2016.

Disponível em <http://www.sciencedirect.com/science/article/pii/S0950584915001743>

ALVES, N. S. R.; RIBEIRO, L. F.; CAIRES, V.; MENDES, T. S.; SPÍNOLA, R. O. Towards an ontology of terms on technical debt. In: *2014 Sixth International Workshop on Managing Technical Debt*, 2014, p. 1–7.

ARISHOLM, E.; BRIAND, L. C.; FUGLERUD, M. Data mining techniques for building fault-proneness models in telecom java software. In: *Proceedings of the The 18th IEEE International Symposium on Software Reliability*, ISSRE '07, Washington, DC, USA: IEEE Computer Society, 2007, p. 215–224 (ISSRE '07,).

Disponível em <http://dx.doi.org/10.1109/ISSRE.2007.22>

AVGERIOU, P.; KRUCHTEN, P.; OZKAYA, I.; SEAMAN, C. Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162). *Dagstuhl Reports*, v. 6, n. 4, p. 110–138, 2016.

Disponível em <http://drops.dagstuhl.de/opus/volltexte/2016/6693>

BAVOTA, G.; RUSSO, B. A large-scale empirical study on self-admitted technical debt. In: *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, 2016, p. 315–326.

BREIMAN, L. Bagging predictors. *Mach. Learn.*, v. 24, n. 2, p. 123–140, 1996.

Disponível em <http://dx.doi.org/10.1023/A:1018054314350>

BROWN, N.; CAI, Y.; GUO, Y.; KAZMAN, R.; KIM, M.; KRUCHTEN, P.; LIM, E.; MACCORMACK, A.; NORD, R.; OZKAYA, I.; SANGWAN, R.; SEAMAN, C.; SULLIVAN, K.; ZAZWORKA, N. Managing technical debt in software-reliant systems. In: *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, New York, NY, USA: ACM, 2010, p. 47–52 (FoSER '10,).

- BUSE, R. P. L.; WEIMER, W. R. Learning a metric for code readability. *IEEE Trans. Softw. Eng.*, v. 36, n. 4, p. 546–558, 2010.
Disponível em <http://dx.doi.org/10.1109/TSE.2009.70>
- CHIDAMBER, S. R.; DARCY, D. P.; KEMERER, C. F. Managerial use of metrics for object-oriented software: an exploratory analysis. *IEEE Transactions on Software Engineering*, v. 24, n. 8, p. 629–639, 1998.
- COHEN, J. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, v. 20, n. 1, p. 37–46, 1960.
- COLLARD, M. L.; DECKER, M. J.; MALETIC, J. I. Lightweight transformation and fact extraction with the srcml toolkit. In: *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*, 2011, p. 173–184.
- CUNNINGHAM, W. The wycash portfolio management system. *SIGPLAN OOPS Mess.*, v. 4, n. 2, p. 29–30, 1992.
- DIGKAS, G.; LUNGU, M.; AVGERIOU, P.; CHATZIGEORGIOU, A.; AMPATZOGLOU, A. How do developers fix issues and pay back technical debt in the apache ecosystem? In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, p. 153–163.
- DIGKAS, G.; LUNGU, M.; CHATZIGEORGIOU, A.; AVGERIOU, P. The evolution of technical debt in the apache ecosystem. In: *Proc. of European Conference on Software Architecture*, 2017, p. 51–66.
- EICK, S. G.; GRAVES, T. L.; KARR, A. F.; MARRON, J. S.; MOCKUS, A. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, v. 27, n. 1, p. 1–12, 2001.
- FARIAS, M. A. D. F.; NETO, M. G. D. M.; SILVA, A. B. D.; SPÍNOLA, R. O. A contextualized vocabulary model for identifying technical debt on code comments. In: *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*, 2015, p. 25–32.
- FERNÁNDEZ-SÁNCHEZ, C.; GARBAJOSA, J.; YAGÜE, A.; PEREZ, J. Identification and analysis of the elements required to manage technical debt by means of a systematic mapping study. *J. Syst. Softw.*, v. 124, n. C, p. 22–38, 2017.
Disponível em <https://doi.org/10.1016/j.jss.2016.10.018>
- FLEISS, J. L., L. B.; PAIK, M. C. *The measurement of interrater agreement*, cáp. 18 p. 598–626, 2004.

DE FREITAS FARIAS, M. A.; DE MENDONÇA NETO, M. G.; KALINOWSKI, M.; SPÍNOLA, R. O. Identifying self-admitted technical debt through code comment analysis with a contextualized vocabulary. *Information and Software Technology*, v. 121, p. 106270, 2020.

Disponível em <http://www.sciencedirect.com/science/article/pii/S0950584920300203>

GRIFFITH, I.; TAFFAHI, H.; IZURIETA, C.; CLAUDIO, D. A simulation study of practical methods for technical debt management in agile software development. In: *Proceedings of the Winter Simulation Conference 2014*, 2014, p. 1014–1025.

GUO, Y.; SEAMAN, C. Measuring and monitoring technical debt. In: *Advances in Computers*, 2011, p. 25–46.

GUO, Y.; SEAMAN, C.; GOMES, R.; CAVALCANTI, A.; TONIN, G.; SILVA, F. Q. B. D.; SANTOS, A. L. M.; SIEBRA, C. Tracking technical debt - an exploratory case study. In: *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, 2011, p. 528–531.

GUO, Y.; SPÍNOLA, R. O.; SEAMAN, C. Exploring the costs of technical debt management — a case study. *Empirical Softw. Engg.*, v. 21, n. 1, p. 159–182, 2016.

Disponível em <http://dx.doi.org/10.1007/s10664-014-9351-7>

HALL, M. A. *Correlation-based feature selection for machine learning*. Relatório Técnico, The University of Waikato, 1999.

HASSAN, A. E. Predicting faults using the complexity of code changes. In: *2009 IEEE 31st International Conference on Software Engineering*, 2009, p. 78–88.

HUANG, Q.; SHIHAB, E.; XIA, X.; LO, D.; LI, S. Identifying self-admitted technical debt in open source projects using text mining. *Empirical Software Engineering*, v. 23, n. 1, p. 418–451, 2018.

IZURETA, C.; VETRO, A.; ZAZWORKA, N.; CAI, Y.; SEAMAN, C.; SHULL, F. *Organizing the technical debt landscape*. In: *Proceedings of the Third International Workshop on Managing Technical Debt*, Piscataway, NJ, USA: IEEE Press, p. 23-26(MTD '12, V.1), 2012.

JESUS, J. S. D.; MELO, A. C. V. D. Technical debt and the software project characteristics. a repository-based exploratory analysis. In: *2017 IEEE 19th Conference on Business Informatics (CBI)*, 2017, p. 444–453.

JIANG, T.; TAN, L.; KIM, S. Personalized defect prediction. In: *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering, ASE'13*, Piscataway, NJ, USA: IEEE Press, 2013, p. 279–289 (ASE'13,).

Disponível em <https://doi.org/10.1109/ASE.2013.6693087>

- KAMEI, Y.; DA S. MALDONADO, E.; SHIHAB, E.; UBAYASHI, N. Using analytics to quantify interest of self-admitted technical debt. In: *1st International Workshop on Technical Debt Analytics*, 2016.
- KITCHENHAM, B.; MADEYSKI, L.; BUDGEN, D.; KEUNG, J.; BRERETON, P.; CHARTERS, S.; GIBBS, S.; POHTHONG, A. Robust statistical methods for empirical software engineering. *Empirical Software Engineering*, v. 22, n. 2, p. 579–630, 2017.
Disponível em <https://doi.org/10.1007/s10664-016-9437-5>
- KRUCHTEN, P.; NORD, R. L.; OZKAYA, I. Technical debt: From metaphor to theory and practice. *IEEE Software*, v. 29, n. 6, p. 18–21, 2012.
- KRUCHTEN, P.; NORD, R. L.; OZKAYA, I.; FALESSI, D. Technical debt: Towards a crisper definition report on the 4th international workshop on managing technical debt. *SIGSOFT Softw. Eng. Notes*, v. 38, n. 5, p. 51–54, 2013.
- LI, Z.; AVGERIOU, P.; LIANG, P. A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, v. 101, p. 193 – 220, 2015.
Disponível em <http://www.sciencedirect.com/science/article/pii/S0164121214002854>
- LIM, E.; TAKSANDE, N.; SEAMAN, C. A balancing act: What software practitioners have to say about technical debt. *IEEE Software*, v. 29, n. 6, p. 22–27, 2012.
- LIU, Z.; HUANG, Q.; XIA, X.; SHIHAB, E.; LO, D.; LI, S. Satd detector: A text-mining-based self-admitted technical debt detection tool. In: *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, 2018, p. 9–12.
- MALDONADO, E. D. S.; ABDALKAREEM, R.; SHIHAB, E.; SEREBRENIK, A. An empirical study on the removal of self-admitted technical debt. In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, p. 238–248.
- MALDONADO, E. D. S.; SHIHAB, E. Detecting and quantifying different types of self-admitted technical debt. In: *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*, 2015, p. 9–15.
- MALDONADO, E. D. S.; SHIHAB, E.; TSANTALIS, N. Using natural language processing to automatically detect self-admitted technical debt. *IEEE Transactions on Software Engineering*, v. 43, n. 11, p. 1044–1062, 2017.
- MALDONADO, E. S.; SHIHAB, E. Detecting and quantifying different types of self-admitted technical debt. In: *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*, 2015, p. 9–15.

MANDELA, N. Lighting your way to a better future. planetarium. University of the Witwatersrand, Johannesburg, South Africa, 2003.

MANNING, C.; KLEIN, D. Optimization, maxent models, and conditional estimation without magic. In: *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology: Tutorials - Volume 5*, NAACL-Tutorials '03, Stroudsburg, PA, USA: Association for Computational Linguistics, 2003, p. 8–8 (NAACL-Tutorials '03,).

Disponível em <https://doi.org/10.3115/1075168.1075176>

MARTINI, A.; BESKER, T.; BOSCH, J. Technical debt tracking: Current state of practice: A survey and multiple case study in 15 large organizations. *Science of Computer Programming*, v. 163, p. 42 – 61, 2018.

MCCABE, T. J. A complexity measure. *IEEE Trans. Softw. Eng.*, v. 2, n. 4, p. 308–320, 1976.

Disponível em <https://doi.org/10.1109/TSE.1976.233837>

MCCALLUM, A.; NIGAM, K. A comparison of event models for naive bayes text classification. In: *In AAAI-98 Workshop on learning for text categorization*, AAAI Press, 1998, p. 41–48.

MENSAH, S.; KEUNG, J.; BOSU, M.; BENNIN, K. Rework effort estimation of self-admitted technical debt. *CEUR Workshop Proceedings*, v. 1771, p. 72–75, 2016.

MENSAH, S.; KEUNG, J.; SVAJLENKO, J.; BENNIN, K. E.; MI, Q. On the value of a prioritization scheme for resolving self-admitted technical debt. *J. Syst. Softw.*, v. 135, n. C, p. 37–54, 2018.

NAKATSU, R. T.; GROSSMAN, E. B.; IACOVU, C. L. A taxonomy of crowdsourcing based on task complexity. *J. Inf. Sci.*, v. 40, n. 6, p. 823–834, 2014.

Disponível em <http://dx.doi.org/10.1177/0165551514550140>

NUGROHO, A.; VISSER, J.; KUIPERS, T. An empirical model of technical debt and interest. In: *Proceedings of the 2Nd Workshop on Managing Technical Debt*, MTD '11, New York, NY, USA: ACM, 2011, p. 1–8 (MTD '11,).

POTDAR, A.; SHIHAB, E. An exploratory study on self-admitted technical debt. In: *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, p. 91–100.

RAHMAN, F.; POSNETT, D.; DEVANBU, P. Recalling the "imprecision" of cross-project defect prediction. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, New York, NY, USA: ACM, 2012, p. 61:1–61:11 (FSE '12,).

Disponível em <http://doi.acm.org/10.1145/2393596.2393669>

- RIOS, N.; DE MENDONÇA NETO, M. G.; SPÍNOLA, R. O. A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners. *Information and Software Technology*, v. 102, p. 117 – 145, 2018.
- SALTON, G.; WONG, A.; YANG, C. S. A vector space model for automatic indexing. *Commun. ACM*, v. 18, n. 11, p. 613–620, 1975.
Disponível em <http://doi.acm.org/10.1145/361219.361220>
- SEAMAN, C.; GUO, Y.; IZURIETA, C.; CAI, Y.; ZAZWORKA, N.; SHULL, F.; VETRE, A. *Using technical debt data in decision making: Potencial decision approaches*. In: Proceedings of the Third International Workshop on Managing Technical Debt, Piscataway, NJ, USA: IEEE Press, p. 45-48(MTD '12, V.1), 2015.
- SEBASTIANI, F. Machine learning in automated text categorization. *ACM Comput. Surv.*, v. 34, n. 1, p. 1–47, 2002.
Disponível em <http://doi.acm.org/10.1145/505282.505283>
- SHIHAB, E.; IHARA, A.; KAMEI, Y.; IBRAHIM, W.; OHIRA, M.; ADAMS, B.; HASSAN, A.; MATSUMOTO, K. Studying re-opened bugs in open source software. *Empirical Software Engineering*, v. 18, n. 5, p. 1005–1042, 2013.
- SIERRA, G.; SHIHAB, E.; KAMEI, Y. A survey of self-admitted technical debt. *Journal of Systems and Software*, v. 152, p. 70 – 82, 2019.
Disponível em <http://www.sciencedirect.com/science/article/pii/S0164121219300457>
- SONARSOURCE, S. A. Sonarqube documentation. Docs.sonarqube.org/display/SONAR/Documentation, 2016.
- TSANTALIS, N.; CHAIKALIS, T.; CHATZIGEORGIOU, A. Jdeodorant: Identification and removal of type-checking bad smells. In: *2008 12th European Conference on Software Maintenance and Reengineering*, 2008, p. 329–331.
- WEHAIBI, S.; SHIHAB, E.; GUERROUJ, L. Examining the impact of self-admitted technical debt on software quality. In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2016, p. 179–188.
- WONGTHONGTHAM, P.; CHANG, E.; DILLON, T.; SOMMERVILLE, I. Development of a software engineering ontology for multisite software development. *IEEE Transactions on Knowledge and Data Engineering*, v. 21, n. 8, p. 1205–1217, 2009.
- YANG, Y.; PEDERSEN, J. O. A comparative study on feature selection in text categorization. In: *Proceedings of the Fourteenth International Conference on Machine Learning, ICML '97*, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997, p. 412–420 (ICML '97,).
Disponível em <http://dl.acm.org/citation.cfm?id=645526.657137>

ZAMPETTI, F.; NOISEUX, C.; ANTONIOL, G.; KHOMH, F.; PENTA, M. D. Recommending when design technical debt should be self-admitted. In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, p. 216–226.

ZAMPETTI, F.; SEREBRENIK, A.; DI PENTA, M. Was self-admitted technical debt removal a real removal?: An in-depth perspective. In: *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, New York, NY, USA: ACM, 2018, p. 526–536 (*MSR '18*,).

Disponível em <http://doi.acm.org/10.1145/3196398.3196423>

TERMO DE REPRODUÇÃO XEROGRÁFICA

Autorizo a reprodução xerográfica do presente Trabalho de Conclusão, na íntegra ou em partes, para fins de pesquisa.

São José do Rio Preto, 29 / 01 / 2021



Assinatura do autor