UNIVERSIDADE FEDERAL DA FRONTEIRA SUL
CAMPUS CHAPECÓ
CURSO DE CIÊNCIA DA COMPUTAÇÃO

WAGNER FRANA

PROPOSTA DE MODELO ÚNICO PARA PRIORIZAÇÃO DE DÍVIDA TÉCNICA

CHAPECÓ
2018

WAGNER FRANA

PROPOSTA DE MODELO ÚNICO PARA PRIORIZAÇÃO DE DÍVIDA TÉCNICA

Trabalho de conclusão de curso de graduação apresentado como requisito para obtenção do grau de Bacharel em Ciência da Computação da Universidade Federal da Fronteira Sul.

Orientadora: Prof. Dra. Graziela Simone Tonin

CHAPECÓ
2018

PROGRAD/DBIB - Divisão de Bibliotecas

Frana, Wagner
Proposta de Modelo Único para Priorização de Dívida Técnica/ Wagner Frana. -- 2018.
87 f.:il.

Orientadora: Graziela Simone Tonin.
Trabalho de conclusão de curso (graduação) - Universidade Federal da Fronteira Sul, Curso de Ciência da Computação , Chapecó, SC, 2018.

1. Dívida Técnica. 2. Priorização de Dívida Técnica. 3. Smells de Código. 4. Qualidade de Software. I. Tonin, Graziela Simone, orient. II. Universidade Federal da Fronteira Sul. III. Título.

Elaborada pelo sistema de Geração Automática de Ficha de Identificação da Obra pela UFFS com os dados fornecidos pelo(a) autor(a).

WAGNER FRANA

PROPOSTA DE MODELO ÚNICO PARA PRIORIZAÇÃO DE DÍVIDA TÉCNICA

Trabalho de conclusão de curso de graduação apresentado como requisito para obtenção do grau de Bacharel em Ciência da Computação da Universidade Federal da Fronteira Sul.

Orientador: Prof. Dra. Graziela Simone Tonin

Este trabalho de conclusão de curso foi defendido e aprovado pela banca em: 04/07/2018

BANCA EXAMINADORA:

Dra. Graziela Simone Tonin - UFFS

Dra. Raquel Aparecida Pegoraro - UFFS

Me. Ana Marcia Debiasi Duarte - UNOESC

AGRADECIMENTOS

Agradeço inicialmente a Deus, por ter me guiado durante toda jornada acadêmica, concedendo-me força e determinação para enfrentar os diversos obstáculos vivenciados.

À minha família, meus pais e minha irmã por toda a base disponibilizada e pelo incentivo repassado ao decorrer de minha graduação, não medindo esforços para me propiciar as melhores condições necessárias.

Agradeço à minha orientadora e professora Graziela Simone Tonin, por me conduzir durante a elaboração deste trabalho de conclusão de curso, fornecendo dicas e compartilhando conhecimentos extremamente importantes. Muito obrigado profe!

À minha banca de defesa, composta pelas professoras Raquel Pegoraro e Ana Marcia Duarte pelas dicas e sugestões para o desenvolvimento de um trabalho melhor.

Agradeço aos desenvolvedores que ofereceram um pouco do seu tempo para auxiliar na validação da proposta desenvolvida e na obtenção dos dados e opiniões necessárias.

À Universidade Federal da Fronteira Sul e aos demais professores que estiveram presentes durante a minha formação, por me fornecer, além do conhecimento repassado, um enriquecimento pessoal e amadurecimento em inúmeros aspectos, guardarei todos em meu coração.

Agradeço aos meus amigos e colegas pelo companheirismo e apoio durante os anos de graduação, seja em momentos de estudo ou de descontração.

E por fim, agradeço a todos que contribuíram de alguma maneira para a realização deste trabalho.

“A persistência é o caminho do êxito.”
— CHARLES CHAPLIN

RESUMO

Com o passar dos anos, os *softwares* tornaram-se cada vez mais presentes no cotidiano de pessoas e empresas. As indústrias desenvolvedoras, visando atender as necessidades dos seus clientes investem frequentemente em otimização, buscando melhorias na qualidade do produto final. Porém, as indústrias de desenvolvimento de *software* lidam com valores limitados de tempo e recursos, fazendo com que essas tenham que aplicá-los de forma a gerar um retorno financeiro a curto prazo e ao mesmo tempo, desenvolvendo funcionalidades que possam satisfazer os clientes. Com isso, aspectos internos de qualidade são alvos de indecisão por parte dos gerentes e desenvolvedores, retratando o contexto da dívida técnica, em que para se estabelecer um equilíbrio entre os objetivos, recursos e funcionalidades do produto, atalhos de desenvolvimento podem ser tomados a curto prazo. Em 1992, o termo dívida técnica foi citado por Ward Cunningham para refletir o cenário em que para acelerar o desenvolvimento de *software* seria necessário a escrita de um código imaturo, gerando-se assim uma dívida. Essa metáfora estendeu-se gradualmente a outras partes do *software*, refletindo de maneira geral aos artefatos imaturos, inadequados ou incompletos presentes no ciclo de vida de desenvolvimento. Devido aos recursos limitados, os itens de dívida técnica identificados devem ser priorizados, buscando classificar e ranquear as dívidas a partir de fatores ou necessidades técnicas. Embora já existam alguns estudos sobre priorização, ainda existem muitos desafios em como definir a prioridade de um item de dívida técnica. Por isso, é necessária a elaboração de novos modelos para priorizar os itens com sucesso, buscando auxiliar na tomada de decisão e visando esclarecer aos empresários os reais benefícios vinculados às melhorias técnicas. O objetivo deste trabalho é identificar e categorizar as abordagens de priorização de dívida técnica existentes na literatura e por fim, propôs-se um modelo único de priorização baseado nas características oriundas desses estudos. A abordagem desenvolvida tem como objetivo priorizar classes afetadas por *code smells*, que são problemas de *design* ao nível do código-fonte de um sistema e podem indicar pontos de dívida técnica. O modelo possui seis fases de classificação, sendo que cada fase representa métricas específicas de ranqueamento. No final do processo de priorização, obtém-se as classes *smelly* com maior prioridade de correção em relação aos critérios considerados. O modelo proposto foi validado com especialistas na área a fim de verificar sua contribuição e relevância.

Palavras-chave: Dívida Técnica. Priorização de Dívida Técnica. *Smells* de código. Qualidade de *software*.

ABSTRACT

Over the years, software has become more and more present in the people's and compani'es daily life. The software development companies aiming to understand customer needs, often investing in optimization, improving the product quality. However, software development companies have been dealing with limited amounts of time and resources, which have to be applied to generate in a short term financial gains and in addition they also have to develop features that satisfy the customers. Therefore, managers and developers focus other aspects than quality to establish a balance between the objectives, features and functionalities of the product, development can be taken shortcuts because of the short time to prioritize this over the quality. In 1992, the term technical debt was cited by Ward Cunningham to reflect the scenario in which to accelerate the development of software would require the writing of an immature code, thereby generating a debt. This metaphor gradually extended to other parts of the software, generally reflecting the immature, inadequate, or incomplete artifacts present in the development life cycle. Due to limited resources, identified technical debt items should be prioritized, seeking to classify and rank debts from technical factors or needs. Although there are some studies on technical debt prioritization, there are still many gaps on how to prioritize a technical debt item. So, it is important to elaborate new models to help developers to prioritize technical debt items, seeking to assist in decision making and aiming to clarify to entrepreneurs the real benefits linked to technical improvements. The objective of this study is to identify and categorize the technical debt prioritization approaches in the literature and, finally, a unique model of prioritization was proposed based on the characteristics derived from these studies. The approach developed is intended to prioritize classes affected by code smells, which are design problems at the source code level of a system and can indicate technical debt points. The model has six phases of classification, each phase representing specific ranking metrics. At the end of the prioritization process, the classes smelly with the highest priority of correction are obtained in relation to the considered criteria. The proposed model was validated with specialists in the area in order to verify its contribution and relevance.

Keywords: Technical Debt. Technical Debt Prioritization. Code smells. Software quality.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ACM	*Association for Computing Machinery*
DT	Dívida Técnica
OOPSLA	*Objected Oriented Programming System, Languages & Application*
DTA	Dívida Técnica de Arquitetura
SATD	Dívida Técnica Auto-Admitida
POs	*Product Owners*
RCR	*Ranking* do Custo de Refatoração das Classes
RPM	*Ranking* baseado na Probabilidade de Mudança das Classes
RPD	*Ranking* baseado na Probabilidade de Defeito das Classes
RRSDA	*Ranking* da Relevância dos Smells e da Densidade de Anomalias das Classes
RPJ	*Ranking* baseado no Principal e no Juros
RIN	*Ranking* do Impacto Negativo das Classes *Smelly*
RFP	*Ranking* Final de Priorização das Classes
ENT	Entrevistado na fase de validação do modelo de priorização

SUMÁRIO

1 INTRODUÇÃO

1.1 Contextualização do problema

Em décadas anteriores, existia um número reduzido de empresas de desenvolvimento de *software*, onde as mesmas desenvolviam produtos distintos e com isso, não geravam uma grande competição entre elas. Atualmente, os *softwares* estão cada vez mais presentes no cotidiano de pessoas e empresas, fazendo com que as empresas desenvolvedoras invistam com frequência em fatores de otimização, como custo, tempo e qualidade, buscando satisfazer as necessidades dos clientes.

Apesar do aumento dos investimentos voltados à qualidade, principalmente quando se trata do processo de desenvolvimento do *software*, os projetos de *softwares* são realizados com valores finitos de tempo e recursos. Desse modo, um mau gerenciamento dos recursos pode comprometer a qualidade do produto final [46].

O termo dívida técnica foi citado pela primeira vez em 1992 nas pesquisas envolvendo qualidade de *software*. Essa metáfora foi empregada para retratar situações em que muitas vezes para acelerar o desenvolvimento de *software* seria necessário a implementação de um código imaturo a ser reescrito futuramente, para corrigir e atender todas as suas especificações [16]. Apesar da metáfora da dívida técnica no início se referir apenas a imperfeições a nível de código, ela desenvolveu-se e estendeu-se para outros artefatos de *software*.

Segundo Kruchten *et al.* [43], os desenvolvedores e gerentes de empresas de *software* discordam regularmente de decisões de como aplicar os recursos escassos em projetos, especialmente quando se trata de aspectos internos de qualidade, os quais são fundamentais para a vida do produto, mas permanecem invisíveis para os gestores e clientes, além de não produzirem valor a curto prazo. Dentre os aspectos, incluem-se melhorias no código, na documentação, na qualidade do projeto e nos testes.

Esse cenário retrata o contexto da dívida técnica, em que, para se estabelecer um equilíbrio entre os objetivos, recursos e funcionalidades do produto, atalhos de desenvolvimento podem ser tomados a curto prazo. Contudo, contrair uma dívida para obter benefícios momentâneos, pode causar maiores custos e influenciar na qualidade do *software* em longo prazo.

Entretanto, Allman [3] enfatiza que adquirir uma dívida técnica é quase inevitável, portanto, o enfoque principal deve estar no controle por meio do seu gerenciamento. Devem-se

aplicar atividades e procedimentos de gestão capazes de reduzir o seu risco futuro e eventualmente fazer sua correção.

Uma solução simples para a dívida técnica seria fazer sua correção ou reembolso[1] antes que mais problemas ocorram. No entanto, o mercado de *software* altamente competitivo faz com que as empresas tenham que trabalhar em prazos apertados para entregar o *software* ao cliente. Dessa forma, cria-se uma pressão constante sobre a equipe de desenvolvimento para entregar recursos funcionais aos clientes dentro do prazo, evitando frustrações, limitando assim, a execução de reparos no *software* [4].

Segundo Yli-Huumo, Maglyas e Smolander [75], surge a necessidade de identificar e desenvolver processos para que as empresas lidem com a dívida técnica, para saber como, o que e quando deve ser reembolsada.

Contudo, devido aos recursos limitados, os itens de dívida técnica devem ser identificados e posteriormente priorizados [15], trazendo-os até a etapa de reembolso. Por exemplo, somente as dívidas que possam causar algum benefício a qualidade do *software* ou aquelas que maximizam o risco de evolução e manutenção do mesmo.

Segundo Li *et al.* [47], a priorização tem o papel de classificar os itens de dívida técnica a partir de um conjunto de regras predefinidas e suportar a decisão de quais dívidas devem ser pagas primeiro e quais podem ser toleradas até lançamentos futuros.

A priorização da dívida técnica é uma parte essencial do processo de gerenciamento, porque pode ser implementada de forma a refletir não só as prioridades técnicas, mas também as de negócios, e estas, em última instância têm precedência na prática [12].

Essa atividade de gestão foi sugerida baseada em várias áreas, como o domínio financeiro, mas permanece sendo um desafio no desenvolvimento de *software* [15]. Segundo Yli-Huumo, Maglyas e Smolander [75], atualmente há uma lacuna quando se trata da priorização de dívida técnica, porque está faltando o conhecimento dos problemas mais importantes a serem corrigidos, o que pode resultar em decisões erradas.

[1] reembolso segundo Li *et al.* [47], é a atividade de gerenciamento que resolve ou atenua a dívida técnica em um sistema de *software* por técnicas como reengenharia e refatoração.

1.2 Objetivos

1.2.1 Objetivo Geral

O objetivo deste trabalho é identificar as abordagens de priorização de dívida técnica existentes e suas características e a partir disso, propor um modelo único de priorização.

1.2.2 Objetivos Específicos

- Identificar as principais abordagens de priorização de dívida técnica existentes e identificar e analisar as características inerentes em cada abordagem estudada;
- Categorizar as abordagens de priorização a partir das métricas utilizadas;
- Propor um modelo único de priorização de dívida técnica.

1.3 Justificativa

Devido aos recursos limitados para lidar com a dívida técnica, a etapa de priorização é uma atividade fundamental [15]. Geralmente, os recursos de refatoração disponíveis são menores do que os custos para reembolsar toda a dívida presente no sistema, sendo extremamente útil fazer uma análise de impacto, buscando entender quando um item de dívida técnica deve ser refatorado em vez de desenvolver novas funcionalidades ao cliente [54]. Torna-se importante, avaliar a quantidade de recursos e esforços que a fixação da dívida técnica exige [74].

A priorização segue sendo um dos principais desafios no gerenciamento da dívida técnica, pois não há soluções adequadas para entender e explicar por que alguns itens de dívida devem ter maior prioridade em relação a outros, além de faltar modelos e métodos para priorizar os problemas técnicos com sucesso [74]. Além disso, o desenvolvimento de novos modelos ou abordagens para a priorização das dívidas ajudaria as equipes de desenvolvimento de *software* a esclarecer aos empresários os reais benefícios vinculados às melhorias técnicas, com base em valores precisos, como por exemplo, qualidade, tempo e valor comercial. A compreensão dos benefícios e riscos vinculados à dívida técnica, também impactaria na melhora da tomada de decisões [74].

Codabux [15], reforça que os desenvolvedores e gerentes de projetos enfrentam problemas para decidir qual dívida técnica contém a maior prioridade e qual deve ser abordada

primeiro. Esse processo de tomada de decisão não é padronizado e atualmente mantém-se dependente do seu contexto na maioria das empresas.

Portanto, os efeitos de gravidade da dívida técnica podem depender do contexto da organização e as estimativas usadas podem ser subjetivas [53]. Além disso, uma atividade de priorização inadequada de uma determinada dívida técnica pode levar a uma crise de desenvolvimento de *software*, seguida de grandes atividades de refatoração que impedem a entrega contínua de funcionalidades [55].

1.4 Delimitação do trabalho

Este trabalho se delimita em uma proposta de abordagem de priorização de dívida técnica composta pela junção de critérios de classificação oriundos de estudos relacionados ao tema encontrados na literatura, a fim de obter-se um modelo único e de maior abrangência. O modelo proposto visa o ranqueamento de classes afetadas por *smells* de código, ou seja, por problemas de *design* do código-fonte do *software*, a partir de seis fases de classificação, buscando obter as classes com maior prioridade de correção em relação aos fatores considerados. Por fim, o modelo desenvolvido é validado por especialistas na área com o intuito de analisar sua relevância e contribuição.

1.5 Estruturação do trabalho

No capítulo 2 é apresentada a contextualização da área de dívida técnica, das atividades de identificação, de priorização e pagamento, além da descrição de algumas ferramentas utilizadas para o gerenciamento das dívidas, de acordo com o levantamento bibliográfico. Já no capítulo 3, é discorrido sobre os trabalhos relacionados ao tema da priorização de dívida técnica identificados na literatura. No capítulo 4, há um detalhamento dos métodos científicos que serviram de norte para a construção deste trabalho, além das etapas de trabalho realizadas no transcorrer deste estudo. O capítulo 5, refere-se à categorização das abordagens de priorização encontradas na literatura a partir das métricas utilizadas em cada modelo. No capítulo 6, consta a proposta de modelo único de priorização de dívida técnica elaborada durante este trabalho e após isso, apresenta-se a avaliação da abordagem por meio da opinião dos entrevistados. E por fim, o capítulo 7 apresenta as considerações finais e as sugestões para trabalhos futuros.

2 REFERENCIAL TEÓRICO

2.1 A Metáfora da Dívida Técnica

A metáfora da dívida técnica (DT), foi citada pela primeira vez por Ward Cunningham em 1992 no *Objected Oriented Programming System, Languages & Application* (OOPSLA) da *Association for Computing Machinery* (ACM). Esse termo, foi utilizado para retratar a situação em que, para acelerar o desenvolvimento de *software* é necessário o desenvolvimento de um código imaturo à ser reescrito futuramente, gerando assim uma dívida. Segundo o autor, o perigo ocorre quando essa dívida não é reembolsada, pois cada minuto que o código é mantido em inconformidade, juros são acrescidos, tornando-a cada vez mais difícil de ser paga [16].

De acordo com Martin Fowler [29], a maneira de realizar atividades voltadas ao desenvolvimento de *software* de forma rápida e imperfeita nos fornece uma dívida técnica, que é semelhante a uma dívida financeira. Assim como uma dívida financeira, a dívida técnica incorre em pagamento de juros, que neste caso, vêm na forma do esforço extra que devemos dedicar em um futuro desenvolvimento para corrigir tais práticas.

Fowler [28], ilustra a seguinte situação para o cálculo do juros associado a uma dívida técnica:

> Quando uma equipe completa um recurso, peça-lhes para lhe dizer quanto tempo demorou (o esforço real) e quanto tempo eles pensam que teriam demorado se o sistema fosse devidamente implementado. A diferença entre os dois é o juros da dívida técnica. (Então, se levou 5 dias, mas eles acham que teriam levado 3 dias com um sistema adequado, logo foi pago 2 dias de esforço como juros em sua dívida técnica) (tradução livre).

Atualmente, a metáfora da dívida técnica foi estendida para outras partes do *software*, não somente ao código, incluindo a arquitetura, *design*, e até mesmo a documentação, requisitos e testes [11]. Refletindo de maneira geral, os artefatos imaturos, incompletos ou inadequados presentes no ciclo de vida de desenvolvimento do *software* [66].

Embora esse termo tenha sido originado há mais de duas décadas, apenas tem recebido significativa relevância dos pesquisadores nos últimos anos [47]. Uma definição atual da dívida técnica está descrita a seguir: “um *design* ou uma abordagem de construção que é implementada a curto prazo, mas que devido ao contexto técnico o mesmo trabalho pode custar mais ao ser refatorado futuramente do que custaria agora (incluindo o aumento do custo ao longo do tempo)” ([57], tradução livre).

Cada item de dívida técnica pode representar um problema técnico [14]. Estes, podem

ser categorizados de acordo com suas características. Em Li *et al.* [47], os itens de dívida são classificados em dez tipos, conforme descrito a seguir.

- DT de Requisitos: refere-se às falhas na especificação de requisitos ou o distanciamento entre uma especificação ideal.
- DT de Arquitetura: é causada por decisões de arquitetura que afetam aspectos de qualidade interna do *software*, como a manutenção.
- DT de *Design*: refere-se aos atalhos técnicos no *design* do *software*.
- DT de Código: é causada pela escrita de código que viola as regras e as boas práticas de codificação.
- DT de Teste: refere-se à deficiência de testes, como por exemplo, a falta de testes unitários.
- DT de Construção: refere-se às falhas no sistema de compilação ou no processo de construção do *software*, tornando o desenvolvimento difícil e complexo.
- DT de Documentação: diz respeito à documentação insuficiente, incompleta ou desatualizada referente ao desenvolvimento do *software*.
- DT de Infraestrutura: refere-se às configurações inadequadas de processos de desenvolvimento, tecnologias e ferramentas de suporte.
- DT de Versionamento: refere-se aos problemas encontrados no controle de versão do código-fonte.
- DT de Defeito: refere-se aos defeitos ou erros localizados em sistemas de *software*.

A dívida técnica é uma metáfora que reflete compromissos técnicos que podem resultar em benefícios a curto prazo, mas que podem prejudicar a qualidade de um sistema de *software* a longo prazo [11]. Ou seja, a aquisição de uma dívida pode estar relacionada a entrega do *software* no prazo, ou ao aceleramento no desenvolvimento de novas funcionalidades, agregando valor positivo, mas em contrapartida, se mal geridas essas dívidas podem trazer danos ou complicações futuras.

O maior problema em questão não é adquirir uma dívida técnica, mas sim, deixá-la fugir do controle. Conforme ressalta Allman [3], a dívida técnica é inevitável, sendo assim, o enfoque principal não é tentar eliminá-la, mas mantê-la sob controle por meio de seu gerenciamento. Além disso, o autor destaca que é melhor ter um protótipo do sistema, mesmo que não esteja completo ou perfeito, para que os clientes possam ir se adaptando ao *software*. Em contrapartida, é importante pensar em um processo de remediação ou plano de mudança, seguindo assim, a filosofia de programação ágil.

Segundo Avgeriou *et al.* [7], a dívida técnica pode ser causada a partir de um processo, por uma decisão, uma ação ou pela falta dela, ou por um evento que desencadeia a sua existência, como pela pressão de programação ou de cronograma, pela indisponibilidade de uma pessoa-chave ou pela falta de informações sobre um recurso técnico.

2.2 Classificação de Dívida Técnica

Segundo McConnell [56], dependendo do modo que a dívida técnica é adquirida ela pode ser classificada em intencional ou sem intenção.

2.2.1 Dívida técnica intencional

Esse tipo de dívida técnica ocorre quando as pessoas envolvidas no projeto inserem a dívida de forma intencional, ou seja, sabe-se que um problema ou imperfeição técnica está sendo imposta. Uma dívida intencional pode ser gerada por meio de uma decisão estratégica da equipe, pela priorização de outras funcionalidades ou também pela pressão do cronograma.

McConnell [56], exemplifica a dívida técnica intencional, como se segue: “Não tivemos tempo para escrever todos os testes unitários para o código que escrevemos nos últimos 2 meses do projeto. Vamos corrigir esses testes após o lançamento” (tradução livre).

2.2.2 Dívida técnica sem intenção

Uma dívida técnica sem intenção, é aquela adquirida de forma involuntária ou sem nenhum planejamento. Ela pode ser causada a partir de más processos ou decisões da equipe. Esse tipo de dívida pode tornar-se muito perigosa, pois não é causada de forma proposital, podendo permanecer invisível, fora de controle e gerenciamento.

Um exemplo dado por McConnell [56] para a categorização dessa dívida é: “sua em-

presa pode adquirir uma empresa que acumulou uma dívida técnica significativa que você não identifica até mesmo depois da aquisição" (tradução livre).

Martin Fowler [30] também criou uma categorização de dívida técnica que pode ser observada a seguir.

2.2.3 Quadrante de Martin Fowler

Fowler [30] desenvolveu um quadrante para classificar a dívida técnica. O autor classifica a dívida técnica em duas dimensões. Na primeira dimensão separam-se as dívidas originadas de forma estratégica (intencional) das que surgem de forma involuntária (sem intenção), assim como McConnell [56]. Já na segunda dimensão separam-se as dívidas em imprudentes e prudentes, conforme ilustrado na figura 2.1.

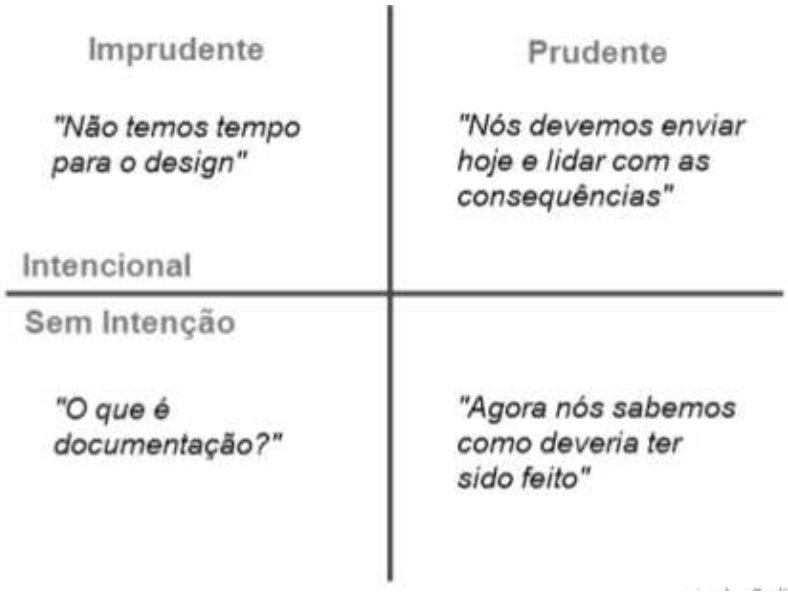

Figura 2.1 – Classificação de dívida técnica [30]. Tradução livre.

A dívida imprudente e intencional é aquela que os desenvolvedores estão conscientes das consequências, e ocorre principalmente pela falta de tempo ou por outros motivos. É uma dívida perigosa se for esquecida, pois pode resultar em altos juros. A dívida prudente e intencional é comum para equipes que sabem como usar a dívida técnica em sua vantagem, pode-se citar o seguinte exemplo: "Amanhã faremos a documentação que não fizemos hoje". Opostamente, a dívida imprudente e sem intenção também é chamada de "código confuso" e pode ser produzida por desenvolvedores que ignoram boas práticas de *design*. Já a dívida prudente e sem intenção pode ocorrer quando uma equipe de excelentes profissionais está aprendendo durante o processo de desenvolvimento do sistema. Por enquanto, eles acham que fizeram um bom

trabalho e o cliente está satisfeito com a qualidade do sistema, mas depois de algum tempo eles percebem que deveriam ter feito algo diferente [30]. Cabe resaltar que é sempre aconselhável ter dívidas técnicas prudentes, pois tem-se consciência sobre o risco assumido e seu possível impacto futuramente.

2.3 Gerenciamento da Dívida Técnica

A gestão da dívida técnica, inclui atividades que são usadas para controlar e reduzir a dívida técnica em um projeto de desenvolvimento de *software*. Com o gerenciamento e a inclusão de diferentes processos, técnicas e ferramentas, empresas visam a redução e a prevenção de atalhos e soluções não ideais com sucesso [47].

Li *et al.* [47], divide a gestão da dívida técnica em oito atividades principais. As atividades são: reembolso/pagamento, prevenção, representação/documentação, identificação, medição, monitoramento, comunicação e a priorização.

2.3.1 Identificação de Dívida Técnica

Segundo Li, Avgeriou e Liang [47], o papel da identificação de dívida técnica é detectar dívidas causadas por decisões técnicas intencionais ou não intencionais em um sistema de *software* através de um conjunto de técnicas específicas. Sendo que o principal objetivo de identificar as dívidas é facilitar a tomada de decisão [67].

Existem várias maneiras de identificar dívidas técnicas de diferentes tipos. Algumas estratégias envolvem simplesmente tornar explícitas as dívidas, e em alguns casos quantificá-las. Outros métodos de identificação fazem o uso de ferramentas para analisar estaticamente o código-fonte ou outros artefatos, buscando encontrar dívidas ocultas, como por exemplo, código mal estruturado e problemas de arquitetura. Cada técnica de identificação associa-se a algum indicador, como uma métrica ou outro parâmetro, que pode ser usado para indicar áreas com tipos específicos de dívida [4].

Conforme destaca Zazworka *et al.* [80], o uso de ferramentas para a identificação é essencial para tornar as dívidas técnicas gerenciáveis, permitindo um maior controle de sua situação. As abordagens de identificação podem ser categorizadas como aquelas que provocam instâncias de seres humanos (desenvolvedores) e ferramentas automatizadas para detectar possíveis dívidas técnicas. Segundo o autor, as técnicas humanas e manuais de identificação pro-

vavelmente serão mais demoradas em relação às abordagens automatizadas, mas na teoria têm duas vantagens. Uma delas é a precisão, ou seja, métodos manuais são propensos à identificar dívidas mais significativas, enquanto análises automatizadas podem revelar muitas anomalias sem importância. A outra vantagem é que pode-se fornecer informações contextuais adicionais importantes relacionadas a cada instância de dívida, como por exemplo, estimativas de esforço e impacto, que é difícil ou até mesmo impossível de obter com ferramentas de análise.

Para localizar indícios de dívida técnica no código-fonte as ferramentas de identificação tem como objetivo detectar defeitos e falta de padrões no código de um projeto de forma automatizada. Para isso, as ferramentas utilizam alguns métodos, dentre os mesmos estão a análise estática de código, violações de modularidade e *code smells* [77].

2.3.1.1 Análise estática de código

Esse método busca analisar o código-fonte ou o código compilado de um projeto em busca de violações das práticas de programação recomendadas que podem causar falhas ou comprometer algumas dimensões da qualidade do *software*, como por exemplo, a manutenção e a eficiência. Algumas dessas inconformidades devem ser removidas através da refatoração para evitar problemas futuros [77].

Com o uso da análise estática de código muitas violações podem ser identificadas, sendo elas relevantes ou não, com isso, é importante analisar o contexto do *software* desejado. Isso pode ser configurado nas ferramentas com base no fator de qualidade que é definido para encontrar as possíveis dívidas técnicas [78].

2.3.1.2 Violações de modularidade

Em *softwares*, módulos representam subsistemas que são projetados para evoluir de forma independente [77]. Esses módulos podem estar relacionados e dependentes entre si. As dependências indicam como os módulos devem evoluir juntos, quando isso não ocorre, pode-se gerar violações de modularidade e consequentemente dívida técnica. Uma violação de modularidade ocorre por exemplo, quando um módulo pai é modificado e seus módulos filhos não sofrem alterações necessárias para manter sua interação [78].

2.3.1.3 *Code smells*

O termo *code smell* (*smell* de código) foi introduzido por Fowler [31] e descreve escolhas em sistemas orientados a objetos que não seguem bons princípios de programação, como por exemplo, encapsulamento e uso de herança.

Segundo Yamashita e Moonen [73], *code smell* é um indicador de uma falha de *design* ou problema no código-fonte que pode afetar aspectos importantes de manutenção.

Code smells podem apontar para uma variedade de problemas potenciais no *software*. Com isso, é necessário que várias estratégias sejam adicionadas ao método para identificá-los, além de requerer da decisão do desenvolvedor de quais devem ser considerados [78]. Se aplicado corretamente, *code smells* podem ser resolvidos através de atividades de refatoração.

Segundo Zazworka e Seaman [78], dois dos *code smells* mais importantes relacionados à dívida técnica são: *God Class* (Classe Deus) e *Dispersed Coupling* (Acoplamento Disperso). Várias abordagens automáticas de identificação foram implementadas para detectar *code smells* em código orientado a objetos, como por exemplo:

- Abordagens de detecção baseadas nas estratégias de Radu Marinescu.
- Para Java: *CodeVizard* e *Marple*.
- Para C#: *ReSharper*, *CodeRush*, *Gendarme* e *FxCop*.

2.3.2 Priorização de Dívida Técnica

A etapa de priorização consiste no ranqueamento de itens de dívida técnica seguindo regras ou fatores estabelecidos, visando determinar sua prioridade de correção em relação a outros itens e atividades. Gong e Lyu [35], descrevem que a atividade de priorização, trata da classificação das dívidas técnicas identificadas e da decisão de qual delas reembolsar primeiro.

Para Zazworka e Seaman [79], uma abordagem de priorização de dívida técnica deve considerar o custo de refatoração das dívidas concomitantemente com o ganho de qualidade para o *software* em realizar as correções.

A figura 2.2, ilustra de maneira geral as principais etapas que compõem um processo de priorização. Inicialmente, tem-se como entrada um conjunto de problemas gerais referentes ao *software*, em seguida, é extraído somente as dívidas técnicas. A partir disso, essas dívidas serão

ranqueadas seguindo algum fator estabelecido, por fim, tem-se uma lista de dívidas técnicas priorizadas.

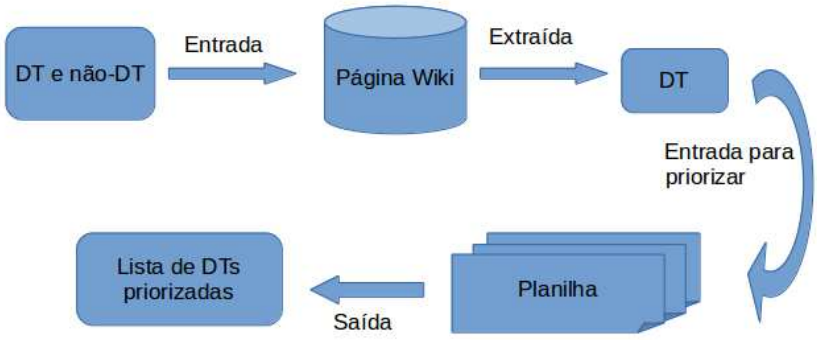

Figura 2.2 – Processo de priorização de dívida técnica [35]. Tradução livre.

2.3.3 Pagamento de Dívida Técnica

O pagamento de dívida técnica visa resolver ou mitigar a dívida técnica presente em um sistema [35]. Segundo Gong e Lyu [35], o pagamento está fortemente relacionado à outras atividades de gestão da dívida, pois deve-se tomar a decisão de qual deve ser paga e quando, visando corrigir problemas mais urgentes e que possuem um maior risco associado ao mantê-los no *software*.

Gong e Lyu [35] descrevem algumas técnicas existentes e suas ações para efetuar o pagamento ou a correção da dívida técnica em um sistema, entre elas estão:

- Refatoração: alterações no código, no *design* ou na arquitetura do *software* visando melhorar a qualidade interna sem afetar o comportamento do sistema.
- Reescrita: reescrever o código que contém dívida técnica.
- Automação: automatizar trabalhos repetidos manualmente, como testes e compilações manuais.
- Reengenharia: evolução do *software* existente para exibir novos recursos e melhorar a sua qualidade operacional.
- Reempacotamento: agrupar módulos coesos, para simplificar as dependências entre os mesmos, visando melhorar a capacidade de manutenção do sistema.

- Resolução de *Bugs*: resolver erros identificados no *software*.
- Tolerância a falhas: adicionar estrategicamente exceções de tempo de execução no local que contêm dívida técnica.

2.4 Ferramentas

Nesta sessão serão descritas algumas ferramentas que visam de forma automatizada auxiliar nas atividades de gestão da dívida técnica.

2.4.1 *CodeVizard*

A ferramenta *CodeVizard* examina os repositórios de *software* avaliando o histórico do código e as métricas ao longo do tempo. O objetivo da ferramenta é oferecer ao usuário uma maior visibilidade em relação às mudanças e a qualidade de um projeto de *software* [76].

CodeVizard suporta código nas linguagem Java e C#. Além disso, calcula mais de 70 métricas de *software* para cada versão do projeto analisado, oferecendo o uso dessas informações para identificar *smells* de código [18] .

2.4.2 *FindBugs*

FindBugs é um *software* livre, distribuído sob os termos da *Lesser GNU Public License* [24]. É uma ferramenta de análise estática que visa identificar defeitos de codificação em programas Java. Em sua versão atual, o *FindBugs* reconhece mais de 300 erros de programação utilizando técnicas simples de análise. O *FindBugs* também usa técnicas de análise mais sofisticadas, planejadas para ajudar a identificar efetivamente certos problemas, como a *Null Pointer Dereference* [8].

A ferramenta permite indicar quais padrões de erro serão utilizados na análise. Após isso, os mesmos são listados e agrupados em categorias, como más práticas de programação e desempenho, além de atribuir prioridade a cada defeito [40].

2.4.3 MARPLE

MARPLE (*Metrics and Architecture Reconstruction Plug-in for Eclipse*) suporta a identificação de padrões de *design* de *software* através do uso de elementos básicos e métricas ex-

traídas automaticamente do código-fonte [5].

As principais atividades realizadas através do MARPLE são retratadas na figura 2.3, sendo elas: o processo geral de extração de dados, detecção de padrões de *design* (DPD), reconstrução da arquitetura do *software* (SAR) e consequente a visualização dos resultados (*Visualization*) [5].

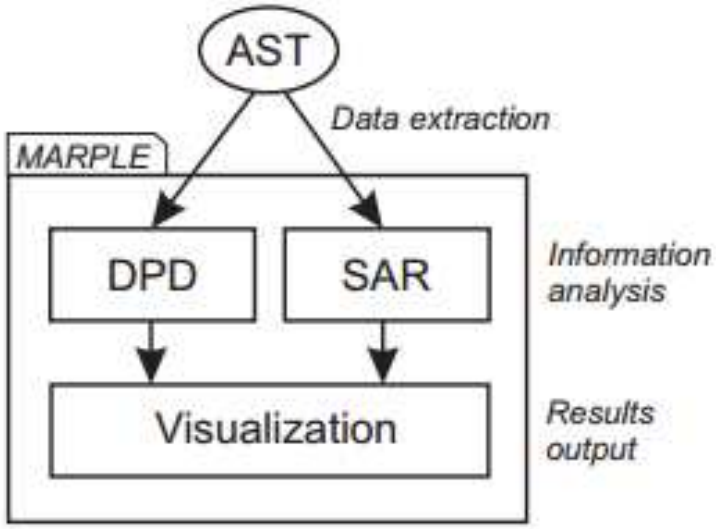

Figura 2.3 – Processo geral do MARPLE [5].

Segundo Arcelli *et al.* [5], a arquitetura do MARPLE é constituída por cinco módulos principais, que interagem uns aos outros através da transferências de dados XML. Os cinco módulos são os seguintes:

- Detector de Informações: coleta elementos básicos e métricas a partir de uma representação AST do código-fonte do projeto verificado;
- *Joiner*: extrai as arquiteturas do projeto que podem combinar com os padrões de *design*, baseando-se nas informações obtidas pelo Detector de Informações;
- Classificador: Este módulo ajuda a detectar possíveis falsos positivos identificados pelo *Joiner* e busca avaliar a similaridade com padrões de *design* canônicos;
- Reconstrução da Arquitetura do *Software*: obtém abstrações do projeto-alvo baseando-se nos elementos e métricas extraídas pelo Detector de Informações;
- Geração de saída: através desta atividade, o usuário visualiza os resultados produzidos pela detecção de padrões de *design*.

2.4.4 *JDeodorant*

O *JDeodorant* é um *plug-in* do Eclipse que tem como função identificar problemas de *design* de *software*, conhecidos como *smells*. Além de identificar esses problemas, a ferramenta resolve-os aplicando refatorações apropriadas [21].

Atualmente, o *JDeodorant* tem a capacidade de identificar cinco tipos de *smell* de código, sendo eles: *Feature Envy, Type Checking, Long Method, God Class* e *Duplicated Code*.

O *JDeodorant* engloba vários recursos inovadores, como por exemplo, a transformação de conhecimento especializado em processos totalmente automatizados, a pré-avaliação do efeito para cada solução sugerida e a orientação do usuário na compreensão dos problemas de *design*. Além disso, a ferramenta possui uma fácil utilização [21].

2.4.5 *AnaConDebt*

AnaConDebt é uma ferramenta que foi projetada para auxiliar na avaliação e na comunicação do juros da dívida técnica entre desenvolvedores, arquitetos e gerentes [53].

Essa ferramenta possui uma interface *web* com a função de receber valores atribuídos pelos usuários em relação ao impacto negativo das dívidas técnicas, ou seja, estimativas de fatores vinculados ao juros das dívidas. A partir desses, a ferramenta calcula e atribui a cada item uma pontuação de gravidade entre 1 e 10, sendo que valores próximos a 10 representam as dívidas com um maior impacto negativo aos critérios analisados.

Por fim, a *AnaConDebt* mostra as dívidas classificadas pelo seu valor de pontuação, auxiliando os desenvolvedores na tomada de decisão referente à priorização e à correção das mesmas.

2.4.6 *DebtFlag*

A ferramenta *DebtFlag* foi criada com o objetivo de capturar, rastrear e resolver a dívida técnica em projetos de *software*. Essa ferramenta integra-se ao ambiente de desenvolvimento e fornece aos desenvolvedores vincular as dívidas às partes correspondentes na implementação através de árvores de dependências [39].

O formato de documentação da dívida técnica com *DebtFlag* baseia-se na estrutura introduzida por Seaman *et al.* [66], [37]. Entretanto, na ferramenta essa estrutura é estendida visando apresentar adequadamente a dívida ao nível de implementação. Primeiramente, uma

lista é preenchida com Itens de Dívida Técnica (TDIs), que correspondem a ocorrências atômicas únicas de dívida no *software* [39].

Cada item mantém um conjunto de informações atreladas, como descrições contendo seu tipo, sua localização e o raciocínio vinculado a sua aquisição. Além disso, conta com estimativas do principal e da probabilidade e quantidade de juros esperados. No mecanismo *DebtFlag*, um TDI pode consistir em um único ou vários elementos *DebtFlag*. Um elemento *DebtFlag* é um *link* entre uma observação de dívida técnica e uma implementação correspondente, como por exemplo, um pacote, uma classe ou um método [39].

A primeira implementação do mecanismo *DebtFlag* foi projetada para suportar a linguagem de programação Java. A ferramenta *DebtFlag* é um sistema de duas partições que consiste em um *plug-in* para o Ambiente de Desenvolvimento Integrado (IDE) do Eclipse e um aplicativo *web*. O *plug-in DebtFlag* é responsável por captar a dívida técnica através do ambiente de desenvolvimento, rastreando sua propagação e apoiando seu gerenciamento. O aplicativo *web* fornece uma apresentação dinâmica da Lista de Dívida Técnica compilada a partir das informações produzidas com os *plug-ins DebtFlag* [39].

Segundo Holvitie e Leppänen [39], os principais benefícios do uso da ferramenta *DebtFlag* são:

- *DebtFlag* captura observações feitas pelo desenvolvedor: a ferramenta permite fornecer aos desenvolvedores um raciocínio adicional a suas observações, garantindo que as informações sobre a dívida técnica sejam precisas e bem definidas.

- A ferramenta documenta a estrutura da dívida técnica: as implementações de *software* são estruturas complexas, hierárquicas e interconectadas. O mecanismo *DebtFlag* capta a dívida técnica como itens. Os TDIs são formados como um conjunto de elementos *DebtFlag* para os quais o mecanismo *DebtFlag* resolve automaticamente os caminhos de propagação. Essa forma estruturada permite acompanhar a dívida técnica durante um desenvolvimento contínuo, além de propiciar a aplicação de várias abordagens de avaliação aos diferentes níveis da hierarquia adquirida.

- *DebtFlag* apresenta a dívida técnica ao nível de implementação: ao projetar todas as observações sobre a dívida técnicas ao nível de implementação, o mecanismo *DebtFlag* garante que o desenvolvimento seja conduzido enquanto estiver ciente da presença da dívida técnica. Isso permite que os desenvolvedores evitem aumentar involuntariamente

o valor da dívida técnica por meio de dependências para as áreas afetadas ou reduzam seu valor, resolvendo-a em áreas onde o desenvolvimento é atualmente conduzido.

2.4.7 TEDMA

A TEDMA é uma ferramenta de código aberto utilizada para auxiliar na gestão da dívida técnica por meio da execução de métricas de gerenciamento, levando em conta a evolução histórica do *software* [23].

Segundo Fernández-Sánchez *et al.* [23], a TEDMA facilita a integração de novas ferramentas e foi projetada para suportar diferentes implementações de modelos de gerenciamento de dívida técnica. As informações geradas são armazenadas em bancos de dados que podem ser explorados por ferramentas externas, incluindo o acesso à informações específicas usando linguagens de consulta. Os dados são organizados segundo a evolução do *software*, portanto, a TEDMA facilita a implementação de algoritmos eficientes para percorrer o histórico de arquivos e revisões.

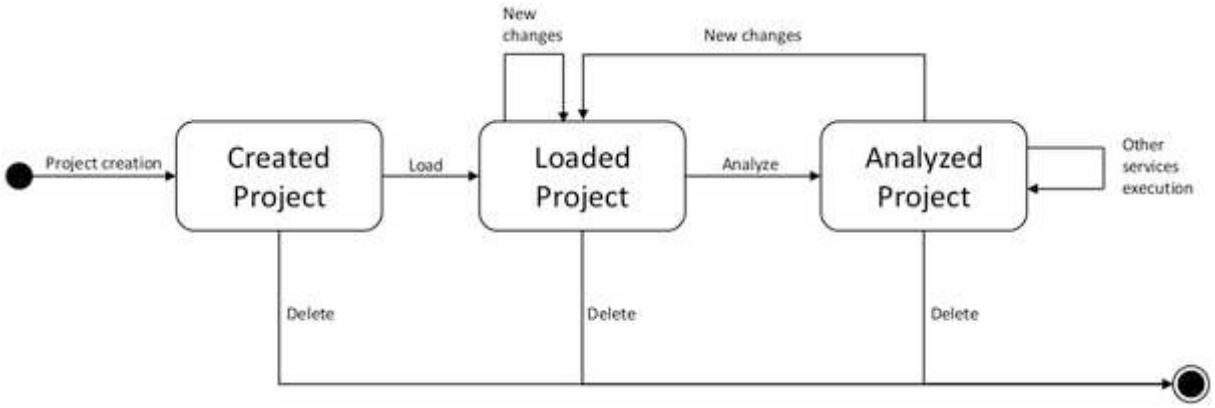

Figura 2.4 – Ciclo de vida de um projeto na ferramenta TEDMA [23].

A figura 2.4, mostra o ciclo de vida de um projeto na TEDMA. Inicialmente um projeto deve ser adicionado à ferramenta informando um nome, uma descrição e a localização do repositório do código-fonte (local ou remoto). O próximo passo é carregar os dados do projeto no banco de dados da TEDMA. A partir dessas informações básicas, o *software* está apto a ser analisado, com isso, quando novas alterações são carregadas no repositório, as mesmas podem ser analisadas na ferramenta. Além disso, a qualquer momento um projeto pode ser retirado da TEDMA para liberar recursos [23].

As informações são armazenadas na TEDMA para cada revisão do *software* e para cada arquivo várias métricas são guardadas, como por exemplo, métricas referentes ao tamanho do

arquivo em bytes, número de linhas e, se o arquivo foi modificado, o tipo e o tamanho da alteração [23].

Para Fernández-Sánchez [23] a TEDMA é uma ferramenta útil, pois permite que as organizações gerenciem a dívida técnica em seus projetos de *software* com uma perspectiva de evolução, permitindo a integração de ferramentas de terceiros e a seleção de métricas desejadas.

2.4.8 *CodeScene*

CodeScene é uma ferramenta desenvolvida pela *Empear AB*, que visa identificar e priorizar a dívida técnica de projetos de *software* a partir da identificação de padrões na evolução do código-fonte. Essa ferramenta permite localizar o código que é de difícil evolução e propenso à defeitos, a fim de sugerir refatorações, melhorando sua qualidade [22].

Segundo Adam Tornhill [22], a análise de código feita pela *CodeScene* leva apenas alguns minutos, mas evita futuras implicações, problemas de manutenção e consequentemente custos elevados.

A *CodeScene* tem como principal fonte de dados os repositórios de controle de versão do sistema analisado. Além da identificação e da priorização de dívidas de código, essa ferramenta utiliza uma abordagem baseada em dados comportamentais e sociais para verificar como os desenvolvedores interagem com a base de dados, permitindo identificar gargalos de produtividade da equipe [22].

Conforme descrito na documentação da *CodeScene* [22], os únicos pré-requisitos para sua utilização são: possuir o código-fonte do projeto de *software* em um ou mais repositórios Git e esses, estarem hospedados no *GitHub*. O plano gratuito da *CodeScene* permite analisar somente projetos de código aberto, já os planos pagos, permitem analisar repositórios abertos, privados e pertencentes a uma organização na qual a pessoa licenciada é membro.

3 TRABALHOS RELACIONADOS

No trabalho de Zazworka e Seaman [79], foi proposta uma abordagem para a priorização de dívidas técnicas baseada no custo/benefício da refatoração. Nesse trabalho, calcula-se o principal e o juros vinculado à correção da dívida em *God Classes*[2] (Classes Deus). Buscou-se apontar a priorização de problemas que são baratos de refatorar, mas que têm uma importância significativa para a qualidade do *software*. Para determinar o custo da refatoração das *God Classes*, foram utilizadas três métricas: a métrica WMC (Método Ponderado por Classe), representa a medida de complexidade de uma classe, TCC (Coesão de Classe Estreita), representa a quantidade relativa de métodos diretamente conectados em uma classe e ATFD (Acesso aos Dados Estrangeiros), é uma métrica que mede coesão e está relacionada ao número de atributos de classes externas que são acessados por uma classe. A partir disso, as classes receberam valores relacionados a cada métrica, conforme visto na figura 3.1. Como pode-se notar, dependendo de quanto o valor atribuído é mais próximo do limiar de cada métrica, menor será a classificação atribuída a classe, ou seja, mais barata em ser refatorada. As classes mais fáceis de correção ficaram com números menores de classificação no *ranking* final desta tabela [79].

God Class Name	WMC (>46)		TCC (<0.33)		ATFD (>5)		Overall Score and Rank	
	Value	Rank	Value	Rank	Value	Rank	Rank Sum	Rank
GodClass1	49	3	0.0	8	20	6	17	6
GodClass2	87	8	0.005	7	28	7	22	7
GodClass3	107	9	0.0	8	28	7	24	9
GodClass4	69	7	0.026	6	34	9	22	7
GodClass5	49	3	0.065	5	9	3	11	3
GodClass6	60	5	0.177	4	19	4	13	4
GodClass7	47	1	0.219	1	7	1	3	1
GodClass8	48	2	0.199	2	7	1	5	2
GodClass9	61	6	0.192	3	19	4	13	4

Figura 3.1 – *Ranking* do esforço de refatoração com base em métricas de detecção [79].

[2] God Class é uma dívida de código que descreve as classes que centralizam funcionalidades e têm muitas responsabilidades em um sistema [63].

O *ranking* do impacto da refatoração foi determinado a partir de duas características de qualidade, sendo elas, a correção, representada pela probabilidade de defeito e a manutenção, representada pela probabilidade de mudança de cada classe. Em relação a correção, determinou-se a cada classe, o número de defeitos corrigidos dentro de um período de tempo em relação ao número total de correções no sistema, obtendo assim a sua probabilidade. Já em relação a manutenção, foi estimado a frequência com que uma classe é alterada. Nesse *ranking* do impacto na qualidade, as classes mais interessantes de serem corrigidas receberam os maiores números de classificação, conforme visto na figura 3.2 [79].

God Class Name	Change Likelihood		Defect Likelihood		Overall Score and Rank	
	Value	Rank	Value	Rank	Rank Sum	Rank
GodClass1	0.016	1	0.0	1	2	1
GodClass2	0.097	8	0.0	1	9	4
GodClass3	0.102	9	0.029	5	14	9
GodClass4	0.068	7	0.177	6	13	7
GodClass5	0.040	3	0.0	1	4	3
GodClass6	0.0455	4	0.133	7	11	5
GodClass7	0.0458	5	0.133	7	12	6
GodClass8	0.052	6	0.133	7	13	7
GodClass9	0.027	2	0.0	1	3	2

Figura 3.2 – *Ranking* de qualidade baseado na probabilidade de defeito e mudança [79].

Por fim, se fez a compensação entre o custo/benefício em fazer a refatoração nessas classes. A figura 3.3, demonstra o ranking final obtido nesse estudo, onde os maiores valores de *overall score* representam as classes com maior valor de esforço/impacto associado. No trabalho de Zazworka e Seaman [79], obteve-se uma abordagem de priorização das classes mais atrativas a serem trabalhadas primeiro, ou seja, com um menor custo de retrabalho associado e com um maior impacto na qualidade do sistema. Essa proposta limitou-se principalmente por abordar somente classes afetadas pelo *smell God Class*, além de requerer de uma avaliação empírica para comprovar sua eficácia.

Class	Effort Rank	Impact Rank	Overall Score
GodClass7	1	6	5
GodClass8	2	7	5
GodClass6	4	5	1
GodClass3	9	9	0
GodClass4	7	7	0
GodClass5	3	3	0
GodClass2	7	5	-2
GodClass9	4	2	-2
GodClass1	6	1	-5

Figura 3.3 – *Ranking* de priorização final [79].

Guo e Seaman [36], basearam-se na relação entre dívida técnica e o domínio financeiro, para propor uma abordagem de portfólio que determina a coleção de itens de dívida que devem ser reembolsados ou mantidos. Na visão financeira, um portfólio refere-se a um conjunto de ativos mantidos por um investidor, como ações, títulos e outros produtos que podem render um retorno por meio do seu investimento [36]. O gerenciamento do portfólio é um processo de decisões que envolve a determinação dos tipos dos ativos que devem ser investidos ou alienados e quando fazer isso. Esse processo de gerenciamento é semelhante a gestão que ocorre na dívida técnica, onde os gerentes precisam tomar decisões sobre quais itens devem ser pagos e quais devem ser mantidos. Se tratando do investimento, os itens de dívida técnica são os ativos que o projeto pode investir para obter lucros. O elemento principal da abordagem formulada é uma lista de dívida técnica composta por itens, representando tarefas incompletas que podem causar problemas futuramente. A partir da sua identificação, cada item inclui algumas informações como a localização da dívida, a razão pela qual é considerada uma dívida técnica, a pessoa responsável, uma estimativa de principal e estimativas do montante de juros esperados e do desvio padrão de juros, além de estimativas das correlações desse item com outras dívidas técnicas. O principal refere-se ao esforço necessário para completar a tarefa. O juro é o trabalho extra que será necessário para corrigir a dívida caso ela não seja reembolsada. Como é incerto que será necessário esforço extra, usa-se o montante esperado de juros e o desvio padrão de juros para capturar a incerteza. Nessa abordagem proposta, as estimativas do principal e do juros são obtidas através da mensuração da dívida técnica, feitas subjetivamente de acordo com a experiência dos desenvolvedores. Nesse trabalho, foi usado o dia/pessoa como unidade das

métricas. As correlações de um item com outros itens de dívida, também são propriedades a serem estimadas. É usada a ideia de coeficientes de correlação para representar a relação entre dois itens de dívida técnica. Os coeficientes de correlação utilizados podem variar de -1 (uma correlação perfeitamente negativa entre dois itens), 0 e +1 (uma correlação perfeitamente positiva). Feitas as estimativas, os itens de dívida são tratados como ativos, determinando se é melhor mantê-lo ou corrigi-lo. Para tomar essa decisão, é determinado o benefício líquido do item, que é calculado a partir do principal menos o valor de juros esperado. Após isso, é preciso equilibrar o benefício líquido com o risco de que o item não produza um benefício. Esse valor é representado pelo desvio padrão de juros. Dessa forma, o trabalho de Guo e Seaman [36], buscou vincular o domínio financeiro com o domínio da dívida técnica, porque o problema da tomada de decisão no gerenciamento dessas duas áreas se correspondem. Trouxe assim, uma ideia de abordagem que pode ser utilizada para medir o custo/benefício dos diferentes itens de dívida e consequentemente fazer sua priorização, mas necessita-se de uma melhor avaliação do desempenho dessa proposta através da aplicação de casos de teste reais.

Arcoverde *et al.* [6], apresentaram e avaliaram quatro heurísticas diferentes para ajudar os desenvolvedores a priorizar anomalias de código, com base em sua contribuição potencial para a degradação da arquitetura do *software*. Essas heurísticas exploraram diferentes características de um projeto de *software*, buscando classificar automaticamente os elementos de código que devem ser corrigidos de acordo com sua potencial relevância arquitetônica. Primeiramente, as anomalias de código foram encontradas nos sistemas alvo. Na segunda fase, aplicaram as heurísticas propostas e calcularam pontuações para cada anomalia detectada. A saída desta fase foi uma lista ordenada com anomalias de alta prioridade. Finalmente, na terceira fase, os autores compararam os resultados das heurísticas com *rankings* anteriormente definidos por desenvolvedores de cada sistema. A heurística de densidade de mudança, buscou classificar as anomalias de código com base no número de mudanças realizadas nesses elementos, assim, quanto maior o número de mudanças, maior seria a prioridade do elemento. Essa informação foi extraída a partir do log de alterações dos sistemas de controle de versão. A heurística de densidade de erro, baseou-se na ideia de que os elementos de código que tem um número elevado de erros observados durante a evolução do sistema podem ser considerados de alta prioridade. A heurística de densidade de anomalias, tratou da ideia de que cada elemento de código pode ser afetado por muitas anomalias. Além disso, um elevado número de elementos anômalos concentrados em um único componente indicam um problema de manutenção mais

profundo. Basicamente, se calculou o número de anomalias encontradas por elemento de código. Por fim, a heurística de função de arquitetura foi definida da seguinte forma: dado um elemento de código c, essa heurística examina a função arquitetônica r executada por c. Com isso, se r é definido como uma função de arquitetura relevante e é executada por c, o elemento de código c será classificado com alta prioridade. Contudo o trabalho de Arcoverde *et al.* [6], buscou demonstrar uma abordagem de priorização para anomalias de código relevantes para a arquitetura, pois essas geralmente levam ao declínio da qualidade da arquitetura do *software*. Além disso, a correção dessas anomalias críticas não é devidamente priorizada, principalmente devido à incapacidade das ferramentas atuais para identificar e classificar as anomalias de código relevantes nesse contexto. A abordagem formulada nesse estudo limitou-se pois focou apenas no impacto de dívidas técnicas arquitetônicas e os resultados estão voltados ao escopo de quatro sistemas alvo, com isso, a fim de generalizar os resultados, ainda é necessária uma pesquisa empírica.

Fontana *et al.* [27], enfatizam que a refatoração é uma atividade cara e que a priorização pode ajudar na economia de tempo ao permitir considerar primeiro as dívidas técnicas mais críticas. Partindo deste pressuposto, esse trabalho focou na priorização de dívidas de seis tipos de *smells*[3] de código a partir da aplicação de um índice de intensidade individual. Primeiramente, a estratégia consiste na atribuição de valores de intensidade às métricas utilizadas na identificação de cada *smell*. Esses valores referem-se a cinco graduações de criticidade: muito baixa, baixa, média, alta e muito alta. Em seguida, foi calculada a média dos valores das métricas, resultando assim, em um único valor. Por fim, o valor resultante foi classificado em um dos cinco rótulos de intensidade, sendo esse o índice referente à dívida técnica associada. As limitações desse trabalho, são que o mesmo restringiu-se ao ranqueamento de classes afetadas por seis tipos de *smell* de código e que utilizou-se somente das métricas de detecção dos *smells* para a priorização das dívidas [27].

Martini e Bosh [52], fizeram um trabalho para investigar, através de uma combinação de entrevistas, observações e uma pesquisa, quais informações eram necessárias aos *Product Owners (POs)* e arquitetos de *software*, a fim de priorizar a refatoração de itens de dívida técnica arquitetônica em relação ao desenvolvimento de novos recursos. Inicialmente, foi feito um questionário com os POs e arquitetos para classificar e avaliar a frequência de utilização e a importância de alguns aspectos utilizados para a fase de priorização, como por exemplo, vanta-

[3] *Smells* são sintomas de problemas no nível de código ou *design*.

gem competitiva, prazo de entrega, custo de manutenção e outros. Após isso, um questionário buscou coletar quais efeitos da dívida técnica arquitetônica seriam úteis para a atividade de priorização. Em seguida, foram mapeados três efeitos mais relevantes que dariam mais informações para avaliar um determinado aspecto, desta forma, pode-se entender para qual aspecto a informação dos efeitos da dívida arquitetônica seria útil. Por fim, através dos dados coletados fez-se uma classificação dos aspectos de priorização, onde a partir do grau de importância e da frequência de utilização os mesmos foram ranqueados. Os efeitos da dívida técnica de arquitetura foram classificados a partir do grau geral da sua utilidade para a priorização. Outras informações que obteve-se com o estudo dos dados, mostra a conexão entre os efeitos da DTA (Dívida Técnica de Arquitetura) e os aspectos de priorização. Pode-se ser visto também, o quanto cada efeito das DTAs contribuem para a atividade geral de priorização. Esse trabalho, contribuiu com informações sobre quais aspectos são relevantes para a priorização de dívidas técnicas arquitetônicas e a relação dos mesmos com os efeitos gerados pela dívida, que também são muito importantes e levados em consideração durante essa fase. O presente estudo limitou-se principalmente por abordar apenas dívidas técnicas de arquitetura e por utilizar uma amostra pequena de entrevistados [52].

Malhotra, Chug e Khosla [48], desenvolveram um modelo de priorização de classes afetadas por *smells* de código que necessitam refatoração imediata a partir do cálculo da Regra de Índice de Depreciação de Qualidade (QDIR) de cada classe. Para identificar as classes cujo a qualidade está comprometida são utilizadas seis métricas, cuja elas são: WMC, RFC, CBO, LCON, DIT e NOC. A ferramenta CKJM é responsável pelo cálculo do valor das métricas e os limites das mesmas são sugeridos por Shatnawi *et al.* [68]. Nessa abordagem são considerados quatro tipos de *smells* de código, sendo eles: *God Class*, *Long Method*, *Type Checking* e *Feature Envy*. O valor QDIR das classes é estabelecido através da seguinte fórmula:

$$QDIR = \frac{BoB}{2} + \frac{BoM}{2} \tag{3.1}$$

O valor de BoB (*Calculation of Base of Bad Smell*) presente na fórmula (3.1), calcula-se da seguinte forma: a cada tipo de *smell* de código presente em uma classe soma-se 0.25. Por fim, esse valor é dividido por 4 (valor referente à quantidade de tipos de *smell* considerados nessa abordagem). Já o valor de BoM (*Calculation of Base of Metric*) de cada classe é determinado em relação aos valores das métricas e de seus limites. Primeiramente, é calculado o MV (*Metric Value*) para cada uma das seis métricas das classes. O MV é obtido através da divisão do valor

da métrica pelo seu limite. Em seguida, os valores de MV de todas as métricas são somados e posteriormente divididos por 6 (quantidade total de métricas) resultando no valor de BoM de uma determinada classe. A partir dos cálculos, as classes são ranqueadas em ordem decrescente baseando-se no valor do QDIR e por fim, é atribuído rótulos de gravidade às classes da seguinte forma: gravidade crítica para classes cujo valor de QDIR exceda 3, alta gravidade para aquelas com valor de QDIR maior que 0.40, gravidade leve para as classes com QDIR maior que 0.15 e as demais classes são consideradas de baixa gravidade. Portanto, a abordagem desenvolvida por Malhotra, Chug e Khosla [48], contribuiu para a priorização de classes infectadas por *smells* de código baseada no cálculo do valor do QDIR, buscando identificar as classes que podem ser as mais prejudiciais para a qualidade do *software* e que devem ser refatoradas primeiro, entretanto, esse modelo restringiu-se ao ranqueamento de classes afetadas por quatro tipos de *smell*.

Akbarinasaji [2], sugeriu uma abordagem de priorização de dívida técnica baseada na aprendizagem por reforço. O aprendizado por reforço é um tarefa ligada à aprendizagem de máquina que visa obter um comportamento ótimo de um agente a partir da seleção de ações, tendo como objetivo final maximizar a quantidade total de recompensas [71]. Basicamente, o aprendizado por reforço é um problema de um agente que interage com o meio e tenta alcançar um objetivo ótimo. A cada momento, o agente com base em um estado escolhe uma ação. A partir da ação ele recebe uma recompensa e também é transferido para um novo estado. Com isso, o agente pode avaliar seu desempenho com base na recompensa obtida. Akbarinasaji [2], modelou a tomada de decisão vinculada à priorização de dívida técnica como uma aprendizagem por reforço. Segundo o autor, o maior problema em questão é que no ambiente da dívida técnica enfrenta-se muitas mudanças e com isso, um cronograma com políticas fixas não funcionaria. Portanto, é preciso usar técnicas de aprendizado, fazendo com que um algoritmo considere os efeitos das ações e aprenda com isso, além de considerar a incerteza presente no efeito futuro da dívida técnica. O cenário sugerido pelo autor em relação a dívida técnica é o seguinte: suponha que o desenvolvedor seja o agente no modelo e ele tem que ajustar a política a fim de minimizar a quantidade de juros da dívida. O conjunto de estados no ambiente pode ser a quantidade de tempo restante antes da próxima versão e o conjunto de ações determina se deve-se pagar a dívida no próximo lançamento ou não. A recompensa é igual a quantidade de juros economizados ao pagar a dívida no momento em vez de adiá-la para a próxima versão. Portanto, o melhor resultado das decisões retornaria a sequência de dívidas técnicas que os desenvolvedores deveriam consertar para economizar a quantidade máxima de juros. A ideia

proposta por Akbarinasaji [2], buscou vincular a tomada de decisão presente na dívida técnica com um modelo de aprendizagem por reforço, pois essa técnica é bastante realista e próxima da forma como os humanos aprendem, podendo auxiliar os desenvolvedores no momento da priorização das dívidas. Entretanto, cabe ressaltar que é difícil de implementar um algoritmo de aprendizado para esse contexto. Além disso, há a necessidade de consultar os desenvolvedores para avaliar melhor essa abordagem formulada [2].

Sommerhoff e Harun [70], construíram uma abordagem custo/benefício de priorização de itens de dívida técnica baseada na modularidade do *software*. A partir da definição de métricas de modularidade e de seus limites estabeleceu-se uma função que equilibra o principal em relação à quantidade e a probabilidade de juros de cada dívida.

$$Prioridade(Item) = \frac{QTJ * PJ}{principal} \tag{3.2}$$

Na função (3.2), o principal define o custo de correção do item de dívida técnica, e o custo de não corrigir é definido pela quantidade de juros (QTJ) e pela probabilidade de juros (PJ). O valor de probabilidade de juros é importante para estabelecer maior prioridade às dívidas que provavelmente sofrerão alterações no futuro. A obtenção desses valores parte de estimativas, que dependem de fatores organizacionais e técnicos, além da habilidade dos desenvolvedores, portanto, não é uma tarefa trivial. Essa estratégia proposta é útil para representar o custo/benefício dos itens de dívida técnica, usando dos valores obtidos para um *ranking* de priorização e posteriormente seu reembolso. Entretanto, a abordagem apresentada limitou-se à métricas de modularidade do *software* [70].

A estratégia proposta por Codabux e Williams [15], buscou priorizar classes de código defeituosas que são mais críticas para o futuro do *software*, ou seja, que são mais propensas à mudanças e que devem ser abordadas primeiro. A partir de um conjunto de métricas relacionadas a volume, complexidade e orientação a objetos, construiu-se uma rede Bayesiana para determinar a probabilidade de propensão da dívida técnica de cada classe. As métricas utilizadas foram as seguintes: SLOC (Linhas de Código de Origem), CBO (Acoplamento entre Objetos), NOC (Número de Filhos), RFC (Resposta de uma Classe), DIT (Profundidade na Árvore de Herança), LCOM (Falta de Coesão em Métodos) e WMC (Método Ponderado por Classe). Após estabelecer a probabilidade, os itens de dívida (classes) são classificados em baixo, médio e alto risco. Feito isso, uma decisão subjetiva pode ser tomada pelo gerente do projeto ou pelos desenvolvedores para decidir quais itens devem ser considerados. Uma decisão

poderia envolver a escolha de apenas itens de dívida de alta prioridade e outra escolha poderia ser uma mistura de itens de dívida de classificação baixa, média e alta, como por exemplo, itens de baixa prioridade que exigem um esforço mínimo para consertar. Conforme destacado por Codabux e Williams [15], essa é a primeira abordagem aprofundada de priorização baseada na criticidade de itens de dívida técnica para o futuro de um projeto. Portanto, requer-se casos de teste reais para avaliar o desempenho do estudo.

Em seu trabalho Choudhary e Singh [13], reforçam que a atividade de refatoração é essencial para a vida de um sistema, no entanto, ela não é prontamente adotada pelas equipes de desenvolvimento, principalmente devido aos prazos rígidos dos projetos e aos recursos limitados. Portanto, é feito principalmente refatorações que incorrem em despesas mínimas e que ao mesmo tempo produzam benefícios à qualidade do *software*. Baseando-se nisso, foi proposta uma abordagem de priorização de dívida técnica que foca na correção de classes relevantes para a arquitetura do *software*. O modelo de ranqueamento das classes segue um processo de três passos, sendo eles: a análise de versões, a análise de classes relevantes para a arquitetura e a geração da classificação. Na etapa de análise de versões, é feito a priorização das classes que foram frequentemente refatoradas no passado, pois essas classes são mais propensas a refatorações no futuro, seguindo assim a hipótese de Girba *et al.* [34]. Na fase de análise de classes relevantes para a arquitetura, a partir da versão atual do sistema são identificadas as classes relevantes para a arquitetura que possuem *code smells*, pois essas classes são consideradas pilares de um projeto de *software* e se não forem corrigidas podem causar efeitos negativos para a qualidade do sistema. Em seguida, os dados das duas etapas acima são combinados e o conjunto comum de classes, ou seja, as classes que são tanto arquitetonicamente relevantes como frequentemente refatoradas são fornecidas como entrada para a etapa de geração da classificação. Na geração da classificação, as classes críticas e favoráveis à refatoração são ordenadas de acordo com seu impacto na qualidade do sistema, conforme a equação (3.3).

$$\textit{Class Score} = F * S * \sum (S(xi) * I(xi)) \tag{3.3}$$

Na equação acima, o valor de F representa a frequência de refatoração de uma classe, ou seja, toda vez que uma classe é encontrada para ser refatorada ao comparar duas versões do *software* subsequentes, sua pontuação de frequência é incrementada em 1. S é a pontuação de severidade de uma classe, mede o impacto negativo de uma classe sobre os atributos de qualidade do sistema. É determinado pela exploração de várias métricas de *software* como tamanho,

coesão, acoplamento e complexidade. S(xi) é a pontuação de gravidade de um *code smell* particular xi para uma determinada classe. Já I(xi), representa o número de instâncias de um *code smell* xi presente em uma classe. Por fim, as classes são classificadas em ordem decrescente pelas suas pontuações de *Class Score*. Portanto, quanto maior o valor de *Class Score*, maior a necessidade de refatoração da classe. Essa abordagem de priorização de oportunidades de refatoração pode ser útil para as equipes de desenvolvimento que possuem prazos curtos de tempo e orçamentos limitados, além de poder ajudar os desenvolvedores a acabar com a necessidade de processar grandes conjuntos de recomendações de refatoração. Esse estudo limitou-se a um número finito de *smells* de código, além disso, os experimentos incluem somente projetos de pequeno e médio porte [13].

Vidal, Marcos e Díaz-Pace [72], criaram uma ferramenta semiautomatizada chamada Identificação Inteligente de Oportunidades de Refatoração (SpIRIT) para priorizar *smells* de código de um sistema de acordo com sua criticidade. A avaliação dos *smells* de código é determinada por três fatores: (i) a estabilidade dos componentes relacionados (SRC), (ii) a relevância do *smell* de código (RCS) e (iii) o impacto de um *smell* em cenários de modificação relacionados (RMS). Primeiramente, para se analisar um *software* no SpIRIT deve ser carregado um arquivo MSE, sendo esse, um arquivo genérico que guarda informações relacionadas a um sistema, tais como classes, métodos e atributos. Em seguida, o SpIRIT segue algumas etapas a fim de priorizar os *smells* encontrados. Na primeira etapa, a ferramenta faz a identificação dos *smells* seguindo estratégias de detecção apresentadas em Lanza e Marinescu [45]. Na segunda etapa é iniciada a priorização dos *smells* de código seguindo os três critérios de avaliação. A estabilidade de componentes relacionados (SRC) busca verificar se o componente em que o *smell* foi encontrado sofreu muitas mudanças ao longo do histórico da aplicação. A suposição feita é que os *smells* que aparecem em classes que mudaram muitas vezes devem ser corrigidos primeiro. A ferramenta SpIRIT permite carregar o histórico de um sistema como um conjunto de versões e cada versão é carregada usando seu próprio arquivo MSE. Uma vez que o histórico é carregado, o SpIRIT calcula automaticamente o valor SRC para cada classe afetada por um *smell* de código. A relevância do *smell* de código (RCS), é usada para determinar o quão relevante é um tipo de *smell* para o desenvolvedor. O SpIRIT permite ao desenvolvedor escolher uma escala ordinal de 1 a 5 para cada tipo de *smell*. Neste contexto, 1 significa que o *smell* de código não é relevante para o sistema e 5 significa que é muito relevante. Além disso, o desenvolvedor pode selecionar indiretamente quais problemas tratar, como o acoplamento, a

coesão ou a complexidade. O critério do impacto de um *smell* em cenários de modificação (RMS) ajuda a focar nos *smells* que afetam os cenários de modificação do sistema, permitindo que o desenvolvedor se concentre nas partes do sistema que afetam a qualidade da arquitetura através da análise de modificações específicas. Os cenários podem ser partes do sistema ou podem descrever algum uso antecipado ou desejado do sistema, por exemplo, um cenário de modificação que descreve a mudança de um mecanismo de visualização 3D. Para cada cenário, o desenvolvedor deve especificar à ferramenta SpIRIT os recursos, pacotes e/ou classes que o compõem. Por exemplo, um cenário I pode ser mapeado para as classes A, B e C. Além disso, para distinguir a importância de cada cenário, o desenvolvedor pode selecionar entre uma escala ordinal de 0 a 1, sendo 1 o mais importante. Com isso, usando cenários de modificação relacionados (RMS), é realizado uma análise de impacto de mudança dos *smells* em relação aos cenários, ou seja, determina-se quais classes afetadas por um *smell* também são mapeadas por um cenário. O valor RMS para um *smell* de código é calculado com base no número de componentes que pertencem aos cenários afetados pelo *smell*. Primeiro, o valor RCS é multiplicado pela importância do cenário que inclui a classe afetada se a classe em que o *smell* é identificado é mapeada por pelo menos um cenário. Então, para cada classe afetada pelo *smell*, somasse o valor RCS multiplicado pela importância do cenário e esse valor é dividido pelo número de classes que são mapeadas por pelo menos um cenário. Nos casos em que mais de um cenário mapeia uma classe, o cenário com a maior importância é utilizado. A intuição por trás desse cálculo é dar maior importância aos *smells* de códigos cujas classes principais e classes afetadas estão diretamente envolvidas em cenários. Ao final do processo de priorização, a ferramenta calcula o *ranking* final pela agregação de SRC, RCS e RMS. Especificamente, um valor de classificação é determinado para cada *smell* da seguinte maneira:

$$\textit{Ranking} = \alpha * (SRC * RCS) + (1 - \alpha) * RMS \tag{3.4}$$

Na equação (3.4), o valor α permite que o desenvolvedor pese a contribuição de uma determinada parte da equação para a classificação final. Além disso, para criar o *ranking*, o SpIRIT permite que o desenvolvedor use todos os cenários definidos ou apenas um subconjunto deles. Contudo, o trabalho de Vidal, Marcos e Díaz-Pace [72], trouxe uma ideia para a priorização de *smells* de código baseada no cálculo de três métricas e com auxílio de uma ferramenta semiautomatizada chamada SpIRIT, auxiliando os desenvolvedores a refatorar primeiro os *smells* considerados mais críticos seguindo essa abordagem e critérios.

Sae-Lim, Hayashi e Saeki [65], propuseram uma abordagem de priorização de *smells* de código com base em um contexto escolhido pelos desenvolvedores e considerando uma lista de mudanças ou problemas a serem resolvidos em um sistema alvo. Tais mudanças são implementadas modificando ou ampliando módulos existentes no código-fonte, portanto, a técnica sugerida busca atribuir maior prioridade aos *smells* de código localizados nos módulos considerados relevantes para os desenvolvedores. A abordagem usa primeiro uma análise de impacto para obter uma lista de módulos que provavelmente serão direcionados a partir das descrições de mudança, ou seja, calcula-se a probabilidade que os métodos ou classes do sistema possuem de estarem relacionados a cada mudança sugerida. Em seguida, através de um detector gera-se uma lista de *smells* de código existente no código-fonte do projeto. Após isso, para cada *smell* de código da lista, calcula-se o Índice de Relevância do Contexto (CRI) com base nos valores obtidos na análise de impacto. O valor do atributo CRI de cada *smell* é calculado pela soma das probabilidades dos módulos correspondentes a esse contexto e afetados pelo *smell*. Finalmente, é emitido uma lista priorizada de *smells* ordenados pelo valor CRI. Para executar a abordagem proposta, os autores implementaram uma ferramenta automatizada. Ela foi projetada para se conectar a uma ferramenta de análise de impacto existente. Quando executada, a ferramenta desencadeia um detector de *smells* de código e após isso a mesma calcula o CRI. Para a análise de impacto, foi utilizado ferramentas propostas por Dit *et al.* [19] juntamente com uma solução baseada em TraceLab [41], [20]. Para a identificação dos *smells*, utilizaram inFusion ver. 1.9.0. Contudo, a abordagem de Sae-Lim, Hayashi e Saeki [65], focou na identificação e na priorização de *smells* de código que estão localizados em módulos relevantes para o contexto escolhido pelos desenvolvedores e assim, proporcionou distingui-los facilmente dos demais *smells* presentes no projeto. Esse trabalho restringiu-se ao ranqueamento de dívidas de código e necessita-se a realização de estudos de caso para confirmar que os *smells* relevantes, conforme definidos neste contexto, são úteis para os desenvolvedores.

Visando analisar os efeitos negativos e econômicos do juros relacionados aos itens de dívida técnica, Martini e Bosch [53] criaram uma abordagem de priorização das dívidas a partir da estimativa de sete fatores de gravidade. Com o auxílio de uma ferramenta chamada AnaConDebt (que foi projetada para auxiliar a avaliação sistemática do juros da dívida técnica, pela comunicação entre desenvolvedores, arquitetos e gerentes) e pela análise dos (sete fatores), acontecem os cálculos, seguindo algumas etapas. Na primeira etapa, os desenvolvedores de forma intuitiva repassam à ferramenta uma avaliação quantitativa entre 1 a 10, em relação ao

impacto negativo de cada dívida técnica. A segunda etapa, consiste na avaliação dos sete fatores relacionados ao juros das dívidas, ou seja, os desenvolvedores informavam à ferramenta um valor de 0 (sem gravidade) a 5 (alta gravidade) a cada fator. Em seguida, a ferramenta retornava um valor de 1 a 10, representando a gravidade do item de dívida técnica analisado. E como última etapa, as pessoas interessadas podem fazer uma avaliação entre os valores obtidos pela ferramenta e a análise intuitiva dos desenvolvedores, permitindo a priorização de itens considerados mais graves em relação aos aspectos desejados. Essa abordagem limitou-se a avaliação do efeito negativo das dívidas em relação ao juros, além disso, o estudo foi feito com apenas uma empresa e com um conjunto de nove itens de dívida técnica [53].

Rani e Chhabra [62], propuseram um esquema de priorização de classes afetadas por *smells* de código de acordo com o seu nível de acoplamento, ou seja, a partir de sua interação com outras classes do sistema. A abordagem proposta é dividida em duas fases. Na primeira fase, as classes afetadas por *smells* são identificadas utilizando uma ferramenta chamada JDeodorant [25]. JDeodorant [25] é um *plug-in* do Eclipse que detecta principalmente cinco tipos de *smells*, sendo eles: *Feature Envy, Duplicate Code, Type Checking, Long Method* e *God Class*. Após a identificação, inicia-se a segunda fase, nela é obtido o índice de impacto de cada uma das classes. A ideia principal é que a classe que tenha alta interação com outras classes possua um alto índice de impacto. Para obter o índice de impacto das classes, é analisado o histórico de alterações do *software*, onde é verificado através das mudanças ocorridas o acoplamento de uma classe com o resto do sistema. O histórico de alterações é registrado e mantido por sistemas de controle de versão, como por exemplo o CSV, SVN ou GIT. O índice de impacto de cada classe é calculado da seguinte forma: número total de classes acopladas à classe analisada dividido por $(n-1)*m$, onde n é o número de classes do sistema e m é o número de *commits* [4] verificados. Como pode-se perceber, o valor de impacto sempre pertence ao intervalo [0, 1], onde 1 (um) representa as classes com maior prioridade de refatoração. Portanto, o trabalho de Rani e Chhabra [62], colaborou com o desenvolvimento de uma abordagem de priorização de classes *smelly*, sendo útil para a redução de custos de refatoração, pois permite aos desenvolvedores obter as classes mais graves em relação ao seu nível de interação com o restante do sistema, limitando-se a cinco tipos de *smells* específicos.

No trabalho de Mensah *et al.* [58], apresentou-se um esquema de priorização que compreende principalmente na identificação, na análise e na estimativa do esforço de retrabalho

[4] *Commit* é a operação que encerra uma transação salvando permanentemente todas as alterações (*updates, deletes, inserts*) realizadas em um sistema.

de dívidas técnicas auto-admitidas (SATD), para assim, tomar a decisão de quais devem ser corrigidas antes da liberação do *software*. O esquema de priorização proposto é baseado em indicadores textuais capturados nos comentários do código-fonte que fornecem informações das causas das dívidas e de sua gravidade, sendo esses, identificados no trabalho de Potdar e Shihab [61]. A abordagem formulada consiste em seis etapas, sendo elas: a identificação da lista de tarefas SATD, análise da complexidade das tarefas e sua importância, distinção entre tarefas esperadas e expedições aceleradas, classificação entre tarefas triviais e vitais, identificação das causas das dívidas e por fim, *break-point* para a decisão de liberação. Inicialmente, com o auxílio de um algoritmo de mineração de texto é identificado instâncias de tarefas SATD analisando os comentários a partir de um vocabulário de indicadores textuais extraídos de Potdar e Shihab [61]. Após isso, analisando os comentários é feita a separação entre as principais tarefas das restantes, em virtude da urgência, da seriedade e do significado das mesmas. Na segunda etapa, é realizado a classificação das tarefas em virtude de sua complexidade e importância, ou seja, em relação à dificuldade em se desenvolver e na relevância das tarefas para o funcionamento efetivo do sistema, respectivamente. Na terceira etapa, através dos comentários presentes no código-fonte são extraídos indicadores textuais buscando classificar e diferenciar as tarefas SATD em esperadas e aceleradas. As tarefas aceleradas são aquelas onde os desenvolvedores normalmente utilizam soluções alternativas para serem concluídas, podendo influenciar diretamente na qualidade do *software*. Na quarta etapa é feito a separação entre tarefas triviais e vitais a partir da extração de indicadores textuais. As tarefas triviais referem-se às correções comuns dentro de um sistema, já as tarefas vitais podem resultar em um sistema de mau funcionamento se não forem resolvidas antes da versão do *software*. Na quinta etapa, após a extração das tarefas triviais e vitais são feitas atribuições das possíveis causas das tarefas examinando os comentários. A identificação das possíveis causas das tarefas SATD é importante a fim de minimizá-las durante o desenvolvimento do *software*. Por fim, com base na sua classificação, nas possíveis causas das tarefas e na estimativa do esforço de retrabalho de cada uma delas, decisões podem ser tomada. A estimativa do esforço de retrabalho das tarefas é calculada em virtude da quantidade de linhas de código (LOC) que requerem modificações ou correções em relação à quantidade total de linhas do código-fonte do projeto. Com isso, tem-se as tarefas SATD que necessitam de um maior custo de retrabalho associado. Portanto, o trabalho de Mensah *et al.* [58], buscou auxiliar na tomada de decisão e na priorização das dívidas técnicas auto-admitidas (SATD) a partir de um esquema de seis etapas, baseando-se nos comentários

feitos pelos desenvolvedores no código fonte, na tentativa de minimizar as tarefas relacionadas à SATD que podem resultar em custos de manutenção mais elevados. Esse estudo limitou-se a quatro projetos de código aberto, portanto, os resultados obtidos não podem ser generalizados com total confiança.

O estudo de Plösch *et al.* [60], buscou priorizar correções de dívidas de código de um projeto de *software* a partir da identificação de falhas no uso de práticas de desenvolvimento adequadas. As melhores práticas de *design* são os itens medidos pela abordagem e a dívida é operacionalizada pelas violações e pela não conformidade das práticas recomendadas no código-fonte. A partir da identificação das violações, o modelo busca analisar as dívidas que são mais importantes a serem corrigidas. Para isso, é realizada uma avaliação baseada em portfólio. Essa técnica agrupa as práticas de programação em quatro áreas denominadas de investimento, ou seja, atribui-se classificações às práticas de modo a recomendar determinadas correções, visando obter ganhos específicos, como por exemplo, priorizar a correção de débitos de *design* que possuem um alto retorno de investimento e que elevem a qualidade do sistema. Entretanto, o trabalho de Plösch *et al.* [60] limita-se ao abordar somente dívidas e práticas de desenvolvimento relacionadas ao código, além disso, a abordagem foi aplicada somente em um projeto de código aberto, necessitando mais testes na prática.

4 METODOLOGIA

Segundo Marconi e Lakatos [49], todas as ciências caracterizam-se pela utilização de métodos científicos, ou seja, sem a adição de métodos científicos não há ciência. Neste capítulo, expõem-se os métodos científicos utilizados no desenvolvimento deste trabalho.

Um método pode ser definido como "[...] o conjunto das atividades sistemáticas e racionais que, com maior segurança e economia, permitem alcançar o objetivo - conhecimentos válidos e verdadeiros -, traçando o caminho a ser seguido, detectando erros e auxiliando as decisões do cientista." [49]. Dessa forma, o método utilizado deve assegurar que a hipótese seja testada e que ao final se possa concluir se ela é verdadeira ou não [49].

Sob o ponto de vista dos objetivos, a pesquisa pode ser classificada como: exploratória, descritiva ou explicativa. Neste estudo, a pesquisa é definida como sendo exploratória, pois parte de um levantamento bibliográfico para a compreensão sobre o tema e assim, delineamento do problema proposto [33].

Quanto à abordagem do problema, Silva e Menezes [69] classificam a pesquisa como quantitativa ou qualitativa. A pesquisa quantitativa utiliza técnicas e procedimentos estatísticos, pois tudo nela é quantificável, isto é, as informações são transformadas em números. Na pesquisa qualitativa há a interpretação dos fenômenos e atribuição de significados sem sua tradução em números. Este estudo, por sua vez, possui características da pesquisa qualitativa, tendo em vista que foram identificados e analisados estudos existentes e a partir disso, interpretados os dados presentes.

Para o desenvolvimento deste trabalho foram seguidas algumas etapas, as mesmas estão descritas a seguir.

4.1 Busca dos artigos

O seguinte trabalho iniciou-se pela revisão bibliográfica da área para identificar os trabalhos existentes relacionados a priorização de dívida técnica. Inicialmente, para encontrar os estudos foi definido a *string* de busca *"technical debt" and "prioritization"* e em seguida, os artigos foram pesquisados nos repositórios da ACM, *IEEE Xplore Digital Library*, entre outros.

Como resultado da pesquisa inicial obteve-se vários trabalhos relacionados ao tema. A partir disso, iniciou-se a seleção dos artigos considerados mais relevantes à este trabalho, partindo da análise do título, da leitura do *abstract*, da introdução e da conclusão dos mesmos.

Durante a etapa de análise, muitos foram descartados pois não condiziam com o assunto proposto, restando 16 (dezesseis) trabalhos, sendo esses, lidos e analisados por completo, visando identificar as métricas utilizadas, o foco de cada método e como cada abordagem faz o cálculo de priorização. No gráfico 4.1, consta o número de artigos lidos de forma integral por repositório durante esta etapa.

Figura 4.1 – Quantidade de trabalhos analisados por repositório. Elaborado pelo autor.

4.2 Categorização das abordagens de priorização de dívida técnica

Após a busca dos artigos, de suas leituras e análise, os trabalhos foram categorizados em relação às métricas utilizadas em cada um deles, ou seja, buscou-se primeiramente identificar as variáveis que compõe os métodos de ranqueamento das dívidas técnicas e a partir disso agrupá-los em categorias.

Com isso, permitiu-se ter uma visão mais ampla das características de cada estudo e principalmente da conectividade entre elas, servindo de apoio para o desenvolvimento do modelo de priorização proposto neste trabalho.

4.3 Desenvolvimento da proposta de modelo de priorização de dívida técnica

Nesta etapa, após esmiuçar as peculiaridades presentes nos trabalhos relacionados à priorização de dívida técnica obtidos na literatura, foi desenvolvido uma proposta de abordagem de ranqueamento de dívidas baseado nas características e métricas oriundas desses estudos.

Para isso, buscou-se interligar os aspectos inerentes nas abordagens encontradas a fim de construir um modelo de ranqueamento mais completo, ou seja, com mais critérios de classificação associados.

4.4 Validação do trabalho

A validação deste trabalho foi realizada por meio de entrevistas com especialistas e conhecedores na área de dívida técnica. Para isso, foram entrevistados quatro desenvolvedores. Segundo Marconi e Lakatos [49], "[...] a entrevista é o encontro entre duas pessoas, a fim de que uma delas obtenha informações a respeito de um determinado assunto, mediante uma conversação de natureza profissional". Dentre os diferentes tipos de entrevistas Marconi e Lakatos [49] classificam em estruturada, não estruturada e painel, de acordo com o propósito do entrevistador.

Neste trabalho, utilizei entrevistas semiestruturadas, sendo que essas mesclam características das entrevistas estruturadas e não estruturadas. Nas entrevistas estruturadas segue-se um roteiro previamente estabelecido, onde o pesquisador não é livre para adaptar suas perguntas, em sua maioria são utilizadas perguntas fechadas. Já nas entrevistas não-estruturadas, o entrevistador tem liberdade para desenvolver cada situação e em geral, as perguntas são abertas e podem ser respondidas através de uma conversação informal [49].

4.4.1 Organização das entrevistas

Foram utilizados dois blocos de questões durante a validação do modelo de priorização, os mesmos encontram-se no apêndice A deste trabalho. Primeiramente, a partir das questões presentes no Bloco I foi identificado o perfil de cada entrevistado e em seguida, foram aplicadas as perguntas pertencentes ao Bloco II para efetuar a avaliação da proposta de modelo desenvolvida.

4.4.2 Coleta dos dados dos entrevistados

Para efetuar a coleta dos dados foram aplicadas entrevistas individuais, onde as respostas foram captadas por meio de gravações ou anotações manuais, na qual a decisão referente ao modo do registro dos dados foi estabelecido de acordo com a autorização dos entrevistados.

4.4.3 Análise e interpretação dos dados

Para realizar a análise das respostas dos entrevistados, primeiramente os dados obtidos foram organizados em tabelas estruturadas por grupos de análise, conforme proposto por Gibbs [32]. Neste estudo, os grupos de análise referem-se à identificação do perfil dos participantes da entrevista e à avaliação da proposta de priorização elaborada perante suas respostas. Em seguida, os dados coletados foram interpretados, para isso, Gibbs [32] propõe considerar as associações e diferenças nos comentários feitos pelos entrevistados para cada questão aplicada, avaliando de forma qualitativa a proposta de modelo de priorização desenvolvida neste trabalho.

5 CATEGORIZAÇÃO DOS ESTUDOS RELACIONADOS À PRIORIZAÇÃO DE DÍVIDA TÉCNICA

Segundo Moraes [59], a categorização é um procedimento de agrupar dados, classificando-os por semelhanças ou analogias, seguindo critérios previamente estabelecidos ou definidos durante o processo. Além disso, um processo de categorização deve ser entendido essencialmente como uma técnica de redução de dados, onde as categorias devem representar aspectos importantes nesse procedimento [64].

Neste capítulo, consta a categorização dos trabalhos relacionados à priorização de dívida técnica a partir das métricas utilizadas em cada uma das abordagens, ou seja, em relação às variáveis e características principais que compõem os métodos ou sugestões de ranqueamento das dívidas nos estudos analisados, conforme visto na tabela 5.1.

Trabalho	Métricas																							
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**
Prioritizing design debt investment opportunities [79].	X	X		X	X																			
A portfolio approach to technical debt management [36].						X																		
Prioritization of code anomalies based on architecture sensitiveness [6].				X	X		X	X																
Towards a prioritization of code debt: A code smell intensity index [27].									X															
Towards prioritizing architecture technical debt: information needs of architects and product owners [52].										X														
Prioritization of classes for refactoring: A step towards improvement in software quality [48].	X										X		X											
Toward measuring defect debt and developing a recommender systemfor their prioritization [2].														X										
Towards a modularity-based technical debt prioritization approach [70].						X																		
Technical debt prioritization using predictive analytics [15].	X		X								X													
Minimizing refactoring effort through prioritization of classes based on historical, architectural and code smell information [13].															X	X	X	X						
An approach to prioritize code smells for refactoring [72].				X													X		X					
Context-based code smells prioritization for prefactoring [65].																				X	X			
The magnificent seven: towards a systematic estimation of technical debt interest [53].												X												
Prioritization of Smelly Classes: A Two Phase Approach [62].																							X	
On the value of a prioritization scheme for resolving self-admitted technical debt [58].			X																			X		
Design Debt Prioritization: A Design Best Practice-Based Approach [60].																								X

Tabela 5.1 – Categorização dos trabalhos relacionados à priorização de dívida técnica. Elaborado pelo autor.

Legenda das métricas presentes na tabela 5.1:

1: WMC (Método Ponderado por Classe).

2: TCC (Coesão de Classe Estreita) e ATFD (Acesso aos Dados Estrangeiros).

3: LOC (Número de Linhas de Código).

4: Probabilidade de mudança de uma classe.

5: Probabilidade de erros corrigidos em uma classe.

6: Principal e Juros das dívidas.

7: Densidade de anomalias (número de anomalias encontradas em um componente).

8: Relevância à arquitetura do *software*.

9: Métricas utilizadas para identificação de 6 smells de código [44]: *God Class*, *Data Class*, *Brain Method*, *Shotgun Surgery*, *Dispersed Coupling* e *Message Chains*.

10: Entrevista com desenvolvedores a fim de obter informações sobre a atividade de priorização de dívida técnica.

11: Conjunto de métricas utilizadas: CBO (Acoplamento entre Objetos), NOC (Número de Filhos), RFC (Resposta de uma Classe), DIT (Profundidade na Árvore de Herança) e LCON (Falta de Coesão em Métodos).

12: Estimativas feitas pelos desenvolvedores de 7 fatores de gravidade relacionados ao juros da dívida, sendo eles: velocidade de desenvolvimento reduzida, *bugs* relacionados ao item TD, qualidade, custos extras, usuários afetados, porção do sistema afetado e frequência do impacto negativo da dívida técnica.

13: $QDIR($*Regra de Índice de Depreciação de Qualidade*$) = BoB/2 + BoM/2$, onde BoB é obtido da seguinte forma: a cada *smell* presente em uma classe soma-se 0.25. Por fim, esse valor é dividido por 4 (total de tipos de *smell* considerados). Já BoM é calculado assim: primeiramente, é feito a soma da divisão do valor das métricas de identificação dos *smells* pelo seu limite para cada classe. Após, esse valor é dividido por 6 (total de métricas de identificação utilizadas).

14: Esse trabalho, propõe uma ideia de priorização baseada na aprendizagem por reforço, visando melhorar a tomada de decisão referente à correção de dívidas técnicas e minimizando o valor do juros acumulado.

15: Frequência de refatoração de uma classe.

16: Severidade de uma classe: é uma pontuação baseada no impacto negativo de uma classe infectada por *smells* para a qualidade do *software*.

17: Pontuação de importância de um *smell* de código.

18: Número de instâncias de um *smell* de código em uma classe.

19: (RMS): Impacto dos *smells* em cenários de modificação.

20: Probabilidade que módulos de um sistema possuem de estarem relacionados à mudanças sugeridas.

21: CRI (Índice de Relevância do Contexto): A cada *smell* de código, soma-se as probabilidades dos módulos correspondentes a um contexto (por exemplo, referente a uma funcionalidade) de serem afetados pelo *smell*.

22: Indicadores textuais relacionados à dívidas SATD.

23: Número de classes acopladas em relação a uma classe.

24: Identificação de violações das práticas de desenvolvimento recomendadas.

Como percebe-se na tabela 5.1, os estudos relacionados ao tema foram categorizados a partir de 24 (vinte e quatro) métricas distintas. Com isso, propiciou-se ter uma visão mais clara da conectividades entre eles e dos fatores utilizados pelos mesmos para a priorização das dívidas técnicas, contribuindo para o desenvolvimento da proposta de modelo elaborada durante o capítulo 6. Para isso, buscou-se interligar métricas e características a fim de propor-se um modelo mais abrangente, composto por critérios considerados importantes na atribuição da prioridade aos problemas técnicos associados.

6 DESENVOLVIMENTO DA PROPOSTA DE MODELO ÚNICO DE PRIORIZAÇÃO DE DÍVIDA TÉCNICA

Um modelo ou abordagem de priorização de dívida técnica tem como objetivo principal ranquear e classificar dívidas seguindo algum critério determinado, visando assim, oferecer aos desenvolvedores uma visão mais clara desses problemas, auxiliando-os na tomada de decisão de quais devem ser corrigidos primeiro, além de manter as dívidas sob gerenciamento.

Neste capítulo, será descrita a construção de uma proposta de abordagem de priorização referente à dívida técnica baseada nas características identificadas e analisadas nos trabalhos de priorização encontrados na literatura.

Como ponto de partida, toda atividade de priorização tem a necessidade de um determinado item a ser ranqueado. O modelo proposto tem como objetivo priorizar prováveis dívidas de código, através de classes infectadas por *smells*, sendo esse, o tipo de dívida técnica mais relevante na indústria [4].

Os *smells* de código são sintomas de problemas ao nível de código ou design de *software* e podem ser usados para capturar sintomas do decaimento da qualidade do código e possíveis problemas de manutenção, além disso, podem ser vistos como uma fonte importante de dívida técnica [27], [26].

Bertrán [10], constatou através de uma análise empírica que *smells* de código podem constituir uma forma de dívida técnica, pois se alguns *smells* são introduzidos nos primeiros estágios de desenvolvimento e não forem removidos o mais rápido possível, eles podem originar problemas mais graves. Além disso, a combinação de alguns *smells* pode ser particularmente prejudicial [1], pois eles podem ser propensos à mudanças e/ou propensos à falhas.

Do ponto de vista do gerenciamento da dívida técnica, os *smells* de código têm a vantagem de geralmente apontarem para locais e problemas distintos, além de estarem associados à refatorações específicas [27].

Partindo do pressuposto que *smells* podem representar problemas na qualidade do código e estão diretamente vinculados à dívida técnica, este trabalho abordou um modelo de priorização de classes afetadas por *smells* de código a partir de critérios de classificação, a fim de ranquear os problemas a serem corrigidos primeiro através da refatoração. Fontana, Ferme e Spinelli [26], afirmam que a refatoração é uma atividade cara e que a priorização pode economizar tempo, permitindo considerar apenas (ou primeiro) os problemas mais relevantes ao

sistema.

Smells de código geralmente são identificados com auxílio de ferramentas e outras técnicas de detecção. Neste estudo, a abordagem de priorização sugerida baseou-se nas estratégias de detecção de *smells* definidas pelos estudos de Lanza e Marinescu [45] e Marinescu [50], [51]. Nessas estratégias, é definido um conjunto de métricas e seus limites para cada *smell* a ser localizado no sistema.

6.1 *Smells* considerados na abordagem

O presente estudo, buscou priorizar classes infectadas por sete tipos de *smell* de código, sendo eles: *God Class, Data Class, Shotgun Surgery, Misplaced Class*, *Feature Envy*, *Dispersed Coupling* e *Brain Method*. A seguir consta a descrição de cada um deles, além de suas estratégias de identificação.

6.1.1 *God Class*

O problema de *design God Class* refere-se às classes que centralizam funcionalidades e têm muitas responsabilidades em um sistema [63].

Segundo Lanza e Marinescu [45], "*God Class* é potencialmente prejudicial para o *design* de um sistema porque é uma agregação de diferentes abstrações e faz o uso de outras classes (geralmente meras detentoras de dados) para executar sua funcionalidade" (tradução livre). Além disso, quebram na maioria das vezes os princípios básicos do *design* orientado a objetos, que diz que uma classe deve ter uma responsabilidade, podendo assim, dificultar a evolução de um sistema [45].

Lanza e Marinescu [45] definem a estratégia de detecção de classes infectadas por esse *smell* baseada em três métricas, conforme visto na regra abaixo:

$$\textit{God Class} = (ATFD > 4)\ \textit{and}\ (WMC >= 47)\ \textit{and}\ (TCC < 0,33) \tag{6.1}$$

Na regra (6.1), a métrica ATFD (Acesso aos Dados Estrangeiros) representa o número de atributos de outras classes, acessados de forma direta ou por meio de métodos de acesso pela classe analisada. A métrica WMC (Método Ponderado por Classe) representa a soma das complexidades de todos os métodos de uma classe. Já a métrica TCC (Coesão de Classe

Estreita) corresponde ao número relativo de métodos de uma classe que acessam pelo menos um atributo em comum.

6.1.2 *Data Class*

Esse problema de *design* corresponde às classes que não possuem funcionalidades complexas, mas outras classes do sistema mantém-se dependentes delas [63].

Os princípios de encapsulamento e ocultação de dados são fundamentais para obter um bom *design* orientado a objetos. As *Data Classes* quebram esses princípios e permitem que outras classes vejam e manipulem seus dados, causando a redução da capacidade de manutenção, da testabilidade e da compreensão de um sistema [45].

Lanza e Marinescu [45] definem a estratégia de detecção de *Data Class* com base em suas características, através da seguinte regra:

$$\begin{aligned} \textit{Data Class} = ((NOPA + NOAM > 2 \textit{ and } WMC < 31) \textit{ or} \\ (NOPA + NOAM > 7 \textit{ and } WMC < 47)) \textit{ and } WOC < 0,33 \end{aligned} \tag{6.2}$$

Na regra (6.2), a métrica WOC (Peso da Classe) representa a porcentagem de métodos que correspondem a serviços oferecidos por uma classe em relação ao total de métodos públicos que a classe possui, visando assim, verificar se a classe dispõe de muitos dados e poucos serviços. A métrica WMC (Método Ponderado por Classe) é designada como em *God Class*. A métrica NOPA (Número de Atributos Públicos) representa o número de atributos públicos de uma classe. E NOAM (Número de Métodos de Acesso) representa o número de métodos de acesso que uma classe possui, exceto os herdados.

A cláusula (NOPA + NOAM > 2) *and* (WMC < 31) é responsável por verificar se uma classe não é complexa, mas apresenta muitos dados e (NOPA + NOAM > 7) *and* (WMC < 47) verifica se a classe analisada apresenta uma complexidade considerável e também possui uma grande quantidade de dados.

6.1.3 *Shotgun Surgery*

Esse problema de *design* está relacionado a um forte acoplamento entre as classes de um sistema [45]. Segundo Fowler e Beck [31], *Shotgun Surgery* corresponde às classes cuja a sua modificação resulta em outros pequenos ajustes em várias outras classes.

Uma operação afetada por *Shotgun Surgery* tem muitas outras entidades de *design* dependendo dela. Com isso, se ocorrer uma mudança em uma operação afetada por esse *smell*, vários outros métodos e classes também precisarão ser alterados, entretanto, é fácil perder ou esquecer uma mudança necessária, causando assim, problemas de manutenção [45].

Lanza e Marinescu [45] definem a estratégia de identificação de classes afetadas por esse *smell* com base em duas características.

(i) quantidade de métodos que poderão sofrer alterações a partir de mudanças em uma determinada classe.

(ii) quantidade de classes que deverão ser verificadas devido a mudanças realizadas em uma classe analisada.

A regra de detecção de *Shotgun Surgery* é a seguinte:

$$\textit{Shotgun Surgery} = CM > 10 \textit{ and } CC > 5 \tag{6.3}$$

Na regra (6.3), as métricas CM (Métodos Mudando) e CC (Classes Mudando) são responsáveis por verificar a primeira e a segunda característica do *smell* respectivamente.

6.1.3.1 *Misplaced Class*

O problema de *design Misplaced Class* diz respeito às classes em que o grau de interdependência com as demais classes do sistema definidas no mesmo pacote é baixo, além de dependerem muito de classes definidas em outros pacotes [9].

Segundo Bertrán [9], "uma classe que utiliza principalmente as funcionalidades definidas em um pacote diferente do seu, deveria provavelmente ser movida para esse outro pacote" (tradução livre).

Herold *et al.* [38] destacam que o problema *Misplaced Class* pode afetar negativamente fatores de qualidade de um sistema ao longo prazo, como modularidade, adaptabilidade e robustez.

Marinescu [50] define a estratégia para a detecção de classes afetadas por esse *smell* baseado nas três características a seguir.

(i) classes que dependem muito de classes definidas em pacotes diferente ao seu.

(ii) classes que dependem mais das classes definidas em outros pacotes que das definidas no seu.

(iii) classes cujas dependências com outras classes estão distribuídas em diferentes pacotes.

Com isso, a regra de detecção para o problema *Misplaced Class* é definido da seguinte forma:

$$\textit{Misplaced Class} = NOED > 6 \textit{ and } CL < 0,33 \textit{ and } DD > 3 \tag{6.4}$$

Na regra (6.4), a métrica NOED (Número de Dependências Externas) é responsável pelo número de classes de outros pacotes que a classe avaliada depende, a métrica CL (Localidade da Classe) refere-se à porcentagem de dependências que uma classe possui em seu próprio pacote em relação ao total de dependências da mesma, sendo que uma classe X depende de uma classe Y se a classe X invoca algum método, e/ou acessa algum atributo e/ou herda da classe Y. A métrica DD (Dispersão de Dependências) refere-se ao número de pacotes diferentes do seu que a classe avaliada depende, onde uma classe tem dependência de um pacote se a mesma necessita de alguma classe do mesmo.

6.1.4 *Feature Envy*

O problema de *design Feature Envy* refere-se aos métodos que acessam mais dados de outras classes do que da própria classe [31]. Segundo Lanza e Marinescu [45], como esses métodos acessam muitos dados de outras classes de forma direta ou através de métodos de acesso, esse cenário representa um indício que os mesmos deveriam ser movidos para outras classes do sistema.

O *smell Feature Envy* pode elevar o acoplamento entre classes e causar dependências desnecessárias, como por exemplo, centralizar os atributos acessados em poucas classes ou apenas em uma [45].

Lanza e Marinescu [45] identificam os métodos com essa anomalia através da seguinte regra:

$$\textit{Feature Envy} = ATFD > 4 \textit{ and } LAA < 3 \textit{ and } FDP <= 3 \tag{6.5}$$

Na regra (6.5), a métrica ATFD (Acesso aos Dados Estrangeiros) é designada como em *God Class*. A métrica LAA (Acesso aos Atributos Locais) representa o número de atributos

acessados por um método que pertencem a sua classe. Já a métrica FDP (Provedores de Dados Estrangeiros) diz respeito ao número de classes externas onde o método utiliza atributos.

6.1.5 *Dispersed Coupling*

O problema *Dispersed Coupling* refere-se aos métodos ou operações que estão excessivamente relacionadas com outros métodos do sistema, sendo que esses métodos provedores estão dispersos em muitas classes. Além disso, a comunicação dos métodos afetados por essa anomalia com cada uma das classes não é muito intensa, ou seja, a operação chama um ou apenas alguns métodos de cada classe [45].

Segundo Lanza e Marinescu [45], operações *Dispersed Coupling* trazem efeitos indesejáveis de ondulação, pois uma mudança em um método afetado por esse *smell* ocasiona mudanças em todas suas classes acopladas e dependentes.

A estratégia de identificação de *Dispersed Coupling* é definida por Lanza e Marinescu [45] da seguinte forma:

$$\textit{Dispersed Coupling} = CINT > 7 \textit{ and } CDISP >= 0.5 \textit{ and } MAXNESTING > 1 \quad (6.6)$$

Na regra 6.6, a métrica CINT (Acoplamento Intensivo) refere-se ao número de métodos invocados pela operação pertencentes a outras classes, a métrica CDISP (Acoplamento Disperso) diz respeito ao grau de dispersão dos métodos invocados pela operação analisada. A métrica MAXNESTING (Nível Máximo de Aninhamento) é utilizada para definir que a operação possua um nível de aninhamento não trivial, para garantir que os casos irrelevantes, como funções inicializadoras, sejam ignorados.

6.1.6 *Brain Method*

A anomalia *Brain Method* refere-se aos métodos que centralizam as funcionalidades de uma classe [45]. Lanza e Marinescu [45] ilustram a seguinte situação: “um método origina como um método “normal”, mas com o tempo, cada vez mais funcionalidades são adicionadas a ele, ficando fora de controle, tornando-o difícil de manter e entender” (tradução livre).

Segundo Lanza e Marinescu [45], um método deve evitar extremidades de tamanho (Regra de Proporção). Os *Brain Methods* são muito longos, dificultando seu entendimento e depuração, e praticamente impossíveis de reutilizar. Além disso, um método bem escrito deve

ter uma complexidade adequada, tendo concordância com a sua finalidade (Regra de Implementação).

A identificação dessa anomalia de código é baseada nas três características a seguir:

(i) Métodos longos: são indesejáveis, afetam a capacidade de compreensão e de teste do código. Métodos longos tendem a fazer mais de uma funcionalidade e, portanto, utilizam muitas variáveis e parâmetros temporários, tornando-os mais propensos a erros.

(ii) Ramificação excessiva: o uso intensivo de instruções *switch* (ou if - else - if), é na maioria dos casos, um sintoma claro de um *design* não orientado a objetos, no qual o polimorfismo é ignorado.

(iii) Muitas variáveis utilizadas: o método utiliza muitas variáveis locais, mas também muitas variáveis de instância.

Lanza e Marinescu [45] definem a estratégia de detecção de *Brain method* da seguinte forma:

$$\begin{aligned} \textit{Brain Method} = LOC > 50 \textit{ and } CYCLO >= 3.1 \textit{ and} \\ MAXNESTING >= 3 \textit{ and } NOAV > 7 \end{aligned} \tag{6.7}$$

Na regra 6.7, a métrica LOC (Número de Linhas de Código) é responsável por verificar se o método analisado é excessivamente grande, através do número de linhas de código que o mesmo possui. A métrica CYCLO (Complexidade Ciclomática) verifica as ramificações condicionais do método verificado. A métrica NOAV (Número de Variáveis Acessadas) é responsável pela contagem das variáveis utilizadas pelo método analisado, incluindo variáveis locais, parâmetros, atributos e variáveis globais. Por fim, a métrica MAXNESTING (Nível Máximo de Aninhamento) verifica o nível de aninhamento do método, ou seja, o nível máximo de aninhamento de estruturas de controle que o mesmo possui.

6.2 Processo de priorização das classes afetadas pelos *smells* de código

Como citado anteriormente, o objeto de priorização neste estudo são classes infectadas por pelo menos um dos sete tipos de *smell* de código. A partir disso, inicia-se o processo de priorização das mesmas através de seis fases, sendo que cada fase representa critérios específicos de classificação.

Em cada etapa, as classes são ranqueadas em ordem crescente em relação ao fator que está sendo analisado, ou seja, se existem n classes consideradas, serão atribuídos n valores entre

[1, n], onde os valores de classificação próximos a 1 (um) pertencem às classes com maior importância ou atratividade de correção em relação ao fator verificado.

Após as fases de classificação é obtido o *Ranking* Final de Priorização das Classes (RFP). O mesmo é calculado a partir da soma dos valores de classificação de cada classe durante as etapas do modelo dividido pelo total de fases. Por fim, as classes com os menores valores de RFP representam aquelas com maior prioridade de correção em relação aos critérios analisados.

A seguir, serão descritas as 6 (seis) fases de classificação referentes ao modelo de priorização proposto neste trabalho.

6.2.1 1ª Fase: *Ranking* do Custo de Refatoração das Classes (RCR)

O *Ranking* do Custo de Refatoração das Classes (RCR) foi baseado no trabalho de Zazworka e Seaman [79]. Esta fase de classificação utiliza as métricas de detecção de cada tipo de *smell* de código a fim de determinar as classes mais fáceis de refatorar.

Através do valor referente às métricas de cada uma das classes e dos limites de detecção dos *smells* determinados no estudo de Lanza e Marinescu [45], determina-se o custo de refatoração das classes afetadas.

O valor de classificação RCR é obtido a cada conjunto de classes infectadas pelos sete tipos de *smell*, ou seja, cada tipo de problema origina um ranqueamento individual. O RCR de cada *smell* é calculado da seguinte forma: a cada classe, é avaliada a diferença entre os valores de suas métricas em relação aos seus limites propostos na regra de identificação do *smell* associado, a partir disso, quanto maior for essa distância, maior será a classificação da classe no *ranking* da métrica analisada. Em seguida, são somadas as posições de cada classe nos *rankings* das suas respectivas métricas. Por fim, é feita a comparação dos valores obtidos pelas mesmas, classificando-as em ordem crescente, ou seja, os menores valores resultantes referem-se às classes com menor custo de correção em relação ao *smell* vinculado.

A seguir consta um exemplo fictício representando o *ranking* do custo de refatoração em classes afetadas pelo *smell God Class*.

Classe	WMC (>= 47)	Rank 1	ATFD (> 4)	Rank 2	TCC (< 0,33)	Rank 3	Soma dos Ranks	Rank Final RCR
Cl1	49	2	5	1	0,11	3	6	2
Cl2	50	3	7	2	0,20	2	7	3
Cl3	47	1	5	1	0,25	1	3	1

Figura 6.1 – *Ranking* RCR para classes afetadas pelo *smell God Class*. Elaborado pelo autor.

Como visto na figura 6.1, a classe Cl3 possui o menor custo de refatoração associado, pois os valores de suas métricas estão mais próximos dos limites estipulados na regra (6.1).

O ranqueamento das classes infectadas por outros tipos de *smell* segue a mesma ideia das *God Classes*, exceto as afetadas por *Data Class*. Nessas classes, como a estratégia de detecção desse *smell* possui duas cláusulas possíveis para as métricas NOPA, NOAM e WMC, o limite usado para a obtenção do *ranking* é baseado na média dos valores limite de cada métrica das duas cláusulas, resultando em: (NOPA + NOAM > 4,5 *and* WMC < 39). Entretanto, isso não modifica a regra de detecção do *smell*, somente para a obtenção do RCR. Na figura 6.2, conta um exemplo fictício de *ranking* RCR em classes *Data Class*.

Classe	NOPA+NOAM (> 4,5)	Rank 1	WMC (<39)	Rank 2	WOC (< 0,33)	Rank 3	Soma dos Ranks	Rank Final RCR
Cl7	4	1	20	3	0,15	3	7	3
Cl10	9	3	40	1	0,23	2	6	2
Cl11	6	2	25	2	0,28	1	5	1

Figura 6.2 – *Ranking* RCR para classes afetadas pelo *smell Data Class*. Elaborado pelo autor.

No caso de uma mesma classe ser afetada por mais de um tipo de *smell*, o seu valor final de RCR é obtido através da soma de suas classificações RCR nos *rankings* dos *smells*.

6.2.2 2ª Fase: *Ranking* baseado na Probabilidade de Mudança das Classes (RPM)

Esta fase de classificação tem como objetivo atribuir alta prioridade às classes afetadas pelos *smells* que possuem maior probabilidade de mudança, ou seja, que são modificadas com maior frequência em relação à outras classes do *software*.

Esta classificação foi baseada nos trabalhos de Zazworka e Seaman [79], Arcoverde *et al.* [6], Vidal, Marcos e Díaz-Pace [72] e Choudhary e Singh [13]. Segundo Zazworka e Seaman [79], classes com alta probabilidade de mudança devem permitir que os desenvolvedores façam

alterações no menor tempo possível. Como nesse caso as classes são afetadas por *smells*, geralmente são mais complexas e difíceis de entender, portanto necessitam ser corrigidas primeiro em relação as demais classes, evitando problemas maiores de manutenção.

Outra hipótese levada em consideração ao propor este *ranking* é que as classes que foram frequentemente modificadas no passado são mais propensas a serem alteradas no futuro [79]. Isso corresponde ao princípio da dívida técnica, em que apenas dívidas baseadas nesse contexto devem ser pagas, visando trazer um impacto positivo ao *software*. Além disso, uma classe afetada por *smells* de código que não foi modificada desde sua implementação pode não representar um real problema [17].

Uma sugestão para obter o ranqueamento das classes através do critério RPM é extrair o *log* de alterações realizadas no *software* a partir do sistema de controle de versão e após isso, calcular a probabilidade de mudança de cada classe baseando-se nas alterações feitas nas mesmas em relação ao total de modificações executadas no sistema.

Por fim, têm-se as classes ordenadas pelo valor da probabilidade, onde classes com maior probabilidade de mudança associada representam as primeiras colocadas no *ranking* RPM.

6.2.3 3ª Fase: *Ranking* baseado na Probabilidade de Defeito das Classes (RPD)

Esta fase de classificação foi embasada nos trabalhos de Zazworka e Seaman [79] e Arcoverde *et al.* [6], e visa ranquear as classes *smelly* [5] partindo do pressuposto de que elementos de código que têm um número elevado de erros observados durante a evolução do sistema podem ser considerados de alta prioridade [6].

O valor RPD de cada classe afetada pelos *smells* pode ser obtido a partir do número de defeitos corrigidos na mesma em relação ao total de defeitos corrigidos no sistema em um determinado período de tempo. Quanto maior o valor resultante, mais provável que um defeito se manifeste na classe analisada [79].

Para obter esses dados, Arcoverde *et al.* [6] propõem extraí-los a partir da inspeção do *log* de registro de mudanças do *software* buscando termos relacionados a *bug* ou correção, esses são encontrados geralmente em mensagens de *commit*. Essa técnica foi aplicada com sucesso em outros estudos relevantes [42].

Por fim, as classes são ordenadas decrescentemente pelo valor RPD, onde as primeiras

[5] *Smelly* refere-se às classes afetadas por *smells*.

colocadas possuem maior importância de correção em relação ao fator verificado nesta fase de ranqueamento.

6.2.4 4ª Fase: *Ranking* da Relevância dos *Smells* e da Densidade de Anomalias das Classes (RRSDA)

Esta fase de classificação das classes é baseada em dois critérios de ranqueamento. Inicialmente, os desenvolvedores visando atribuir maior prioridade à problemas de *design* específicos, concedem a cada tipo de *smell* um valor numérico entre 1 e 7, onde *smells* com valor de relevância 7 (sete) representam problemas com maior gravidade ou importância.

Portanto, esse critério permite atender as preferências do desenvolvedor a fim de priorizar a correção de problemas que o mesmo acredita ser os mais significativos para o *software*, conforme ressaltado nos trabalhos de Vidal, Marcos e Díaz-Pace [72] e Choudhary e Singh [13].

A métrica da densidade de anomalias é baseada na ideia de que cada elemento de código pode ser afetado por muitos problemas, além disso, um elevado número de anomalias concentradas em um único componente indicam um problema de manutenção mais profundo [6], [13].

A partir desse fator objetiva-se priorizar as classes afetadas por vários *smells* de código, a fim de evitar a propagação de problemas ao *software*. A densidade de anomalias é atribuída da seguinte forma: a cada classe verifica-se quais instâncias de *smell* ela possui, ou seja, por quais tipos de problema de *design* a mesma é infectada.

Após a obtenção dos valores de relevância dos tipos de *smell* considerados e da detecção dos problemas em cada uma das classes, realiza-se o cálculo do RRSDA. O mesmo é obtido a partir da fórmula abaixo:

$$\textit{RRSDA(classe)} = \sum(R(i)) \tag{6.8}$$

Na equação (6.8), o valor RRSDA de cada classe é originado pelo somatório da variável R(i), a mesma representa o valor de relevância dos *smells* presentes na classe analisada. Por fim, as classes são ranqueadas decrescentemente pelo valor RRSDA, onde as primeiras colocadas possuem maior prioridade de correção.

6.2.5 5ª Fase: *Ranking* baseado no Principal e no Juros (RPJ)

Esta fase é baseada no custo/benefício vinculado às classes afetadas pelos *smells*, tendo como objetivo atribuir maior prioridade àquelas que possuem um maior retorno positivo, ou seja, mais atrativas a serem corrigidas. Esta etapa de classificação baseou-se nos trabalhos de Guo e Seaman [36] e Sommerhoff e Harun [70].

As métricas utilizadas são o Principal, o Juros e o RPM. O Principal refere-se ao custo atual para corrigir os problemas na classe analisada, o Juros representa o custo extra ao corrigir as imperfeições presentes na classe no futuro e o valor RPM refere-se à probabilidade de mudança da mesma, oriundo da segunda fase de classificação. Nesse caso, o valor RPM é utilizado para atribuir menor prioridade às classes que provavelmente nunca serão alteradas no futuro.

Os valores referentes ao Principal e o Juros são obtidos a partir da estimativa dos desenvolvedores e a unidade de medida utilizada é horas/pessoa. Sommerhoff e Harun [70] ressaltam que essa não é uma tarefa trivial, porque as estimativas dependem de fatores organizacionais e técnicos, além da habilidade dos desenvolvedores, interdependências entre os itens e mudanças projetadas. Portanto, as estimativas devem ser realizadas por cada organização de forma individual [70].

$$RPJ(classe) = \frac{Juros * RPM}{Principal} \tag{6.9}$$

Após obter os valores das três métricas, aplica-se a fórmula (6.9) a cada uma das classes, onde divide-se o benefício pelo custo em realizar as correções nas mesmas. Por fim, as classes com maior valor RPJ associado são as que detêm maior custo/benefício ao serem refatoradas, ocupando assim as primeiras posições nesta fase de classificação.

6.2.6 6ª Fase: *Ranking* do Impacto Negativo das Classes *Smelly* (RIN)

A sexta fase de classificação foi baseada no trabalho de Martini e Bosch [53] e tem como objetivo ranquear as classes *smelly* a partir de quatro aspectos de gravidade vinculados a esses indícios de dívida técnica.

Os aspectos analisados representam efeitos negativos provocados ao sistema pelos problemas presentes nas classes. Os efeitos de gravidade considerados nesta fase são:

- *Bugs* relacionados ao item: se muitos *bugs* estão relacionados à classe afetada pelos *smells*

de código e os desenvolvedores suspeitam de uma conexão entre os problemas de *design* e os erros, esse cenário indica que a classe está afetando a capacidade de manutenção do sistema e portanto, deve-se realizar correções na mesma.

- Redução da velocidade de desenvolvimento: se a velocidade de desenvolvimento for reduzida, os *smells* que afetam a classe estão interferindo na capacidade de evolução e manutenção do sistema.

- Porção do sistema afetado: busca avaliar a relação da classe *smelly* com o restante do sistema, sendo que, quanto maior for a parte do *software* afetada pela possível dívida maior será sua gravidade, pois a mesma pode desencadear efeitos negativos em cascata para as partes relacionadas.

- Usuários afetados: esse aspecto visa atribuir maior prioridade às classes infectadas por *smells* que estão relacionadas a funcionalidades do sistema mais acessadas pelos usuários.

O *ranking* do impacto negativo das classes (RIN) é calculado da seguinte forma: a cada classe, os desenvolvedores atribuem valores quantitativos entre 1 e 5 avaliando de forma intuitiva o seu impacto em relação aos efeitos de gravidade listados acima. Sendo que os valores pertencentes ao intervalo [1, 5] estão associados respectivamente aos rótulos de gravidade: muito baixa (1), baixa (2), média (3), alta (4) e muito alta (5) em relação aos critérios considerados. Após isso, soma-se os valores estimados a cada classe e em seguida, divide-se o valor resultante por quatro, obtendo assim, o valor médio de gravidade das mesmas.

Por fim, as classes são ordenadas decrescentemente pelo valor RIN vinculado, onde as primeiras colocadas referem-se às classes com maior prioridade de correção em relação aos aspectos considerados nesta fase.

6.2.7 *Ranking* Final de Priorização das Classes (RFP)

Após as seis fases de classificação, as classes *smelly* recebem a sua posição final de ranqueamento referente a esta abordagem de priorização. A mesma é atribuída a partir da seguinte fórmula:

$$RFP(classe) = \frac{RCR + RPM + RPD + RRSDA + RPJ + RIN}{6} \tag{6.10}$$

Na fórmula (6.10), os valores de RCR, RPM, RPD, RRSDA, RPJ e RIN referem-se às posições ocupadas pelas classes nas fases de classificação deste modelo de priorização. Portanto, para obter o valor RFP de cada classe é somado suas classificações nos *rankings* desta abordagem e em seguida, o valor resultante é dividido por seis (número de fases de classificação).

Por fim, as classes são ordenadas crescentemente pelo valor RFP, ou seja, as classes com os menores valores de RFP associados são as que detêm maior prioridade de correção considerando os critérios analisados durante as etapas de priorização desta proposta.

Após a aplicação do questionário de validação da proposta de modelo de priorização, constatou-se a partir do relato dos entrevistados perante a questão 3 oriunda do Bloco II da entrevista, que seria importante que a fórmula (6.10) fosse modificada, de modo a permitir que os desenvolvedores atribuam diferentes pesos às fases de ranqueamento do modelo, visando assim, captar problemas específicos ou que possuem maior atratividade de correção em relação a fatores determinados. Com isso, foi elaborada a seguinte fórmula para a obtenção do *ranking* RFP:

$$RFP(classe) = \frac{(u * RCR) + (v * RPM) + (w * RPD) + (x * RRSDA) +}{u + v + w + x +}$$

$$\frac{+(y * RPJ) + (z * RIN)}{+y + z} \tag{6.11}$$

Na fórmula (6.11), as variáveis *u, v, w, x, y,* e *z* são utilizadas para a atribuição das ponderações às fases do modelo. Apesar dessas mudanças, as classes com os menores valores de RFP associados continuam representando àquelas com maior prioridade de correção.

6.3 Validação da proposta de modelo de priorização

Conforme descrito na Seção 4.4, a proposta de abordagem de priorização desenvolvida neste trabalho foi validada por quatro conhecedores na área, os entrevistados ocupam o cargo de desenvolvedor em empresas de desenvolvimento de *software*. Para a coleta dos dados foram aplicados dois blocos de questões, os mesmos encontram-se no Apêndice A deste estudo. O Bloco I é composto por perguntas cujo o objetivo é capturar o perfil dos entrevistados e o

Bloco II possui questões elaboradas e direcionadas ao objeto desenvolvido, visando analisar de maneira qualitativa o modelo proposto e suas características.

Antes da aplicação das questões, foi realizada a apresentação dos objetivos do presente trabalho e da proposta de modelo de priorização desenvolvida, fazendo com que os participantes da validação compreendessem o que foi elaborado neste estudo e qual a sua finalidade.

A tabela 6.1 apresenta os resultados obtidos a partir do questionário oriundo do Bloco I, já as tabelas 6.2 e 6.3, trazem os relatos dos entrevistados perante as questões abordadas no Bloco II. Nas tabelas citadas anteriormente, os entrevistados são identificados pela sigla ENTs.

	Questões referentes à identificação do perfil dos entrevistados		
Entrevista	Qual o seu cargo?	Há quanto tempo exerce essa função?	Conhece os termos dívida técnica e a atividade de priorização de dívida técnica? A empresa ou organização que você trabalha utiliza algum modelo de priorização?
ENT1	Desenvolvedor	2 anos e 5 meses	"Sim conheço, os termos tem ganhado muita atenção nos últimos anos. A empresa não utiliza nenhum modelo de priorização."
ENT2	Desenvolvedor	5 anos e 2 meses	"Sim conheço, sei do que se trata. A minha empresa não utiliza modelo de priorização referente à dívida técnica."
ENT3	Desenvolvedor	7 anos	"Sim, conheço os termos e sei do que se trata. Na minha empresa não utilizamos modelos de priorização. Os reparos referentes à problemas no código são realizados de forma não padronizada, ou seja, quando os mesmos estão dificultando o desenvolvimento durante algumas versões do *software*, buscamos encaixá-los com novas implementações para realizar as correções necessárias".
ENT4	Desenvolvedor	8 anos e 4 meses	"Sim, já conheço os termos, tenho ciência do que se trata. Na empresa não utilizamos modelos de priorização referente à dívida técnica. As correções são realizadas conforme aparecem os problemas no dia a dia de desenvolvimento, principalmente o que está dificultando a evolução e a manutenção do sistema."

Tabela 6.1 – Respostas dos entrevistados perante as questões '1', '2' e '3' do Bloco I da entrevista. Elaborado pelo autor.

Através da análise das respostas presentes na tabela 6.1, pode-se confirmar que os entrevistados possuem conhecimento sobre o tema deste estudo, além de estarem diretamente relacionados ao ambiente de desenvolvimento de *software* e à questões similares àquelas retratadas neste trabalho, podendo assim, contribuir para a avaliação qualitativa da proposta de modelo de priorização e de suas características.

	Questões 1 a 3 referente à validação da proposta de modelo de priorização		
Entrevista	O modelo de priorização desenvolvido neste estudo é relevante? Quais suas principais contribuições?	Os critérios das fases de classificação são importantes a serem considerados?	A fórmula presente na fase do Ranking Final de Priorização das Classes (RFP) é adequada? Algo poderia ser modificado?
ENT1	"Sim é relevante, o modelo ajuda a priorizar os problemas que são mais graves e atrativos baseando-se nos fatores considerados, buscando assim, ajudar na tomada de decisão de quais corrigir primeiro."	"Sim, são fatores importantes e fazem sentido serem abordados."	"Acredito que sim, mas necessita testes na prática para verificar os resultados da priorização. Uma sugestão seria testar fórmulas com ponderações nos critérios."
ENT2	"É relevante. O modelo contribui com a priorização de classes afetadas pelos sete tipos de anomalias de código através das métricas utilizadas, visando a partir das mesmas distinguir os problemas mais importantes a serem corrigidos dos restantes. Acho importante também o fato do modelo possuir *rankings* que contam com a participação direta dos desenvolvedores na determinação da prioridade dos problemas."	"Sim, são métricas importantes, que fazem sentido ao abordar esses tipos de problemas, contribuindo para suas priorizações."	"Ela é adequada ao considerar a priorização das classes através de suas classificações em cada um dos seis *rankings*, mas requer testes reais pra comprovar de fato o resultado da priorização dos problemas. Uma sugestão seria modificar a fórmula, a fim de permitir que os desenvolvedores inserissem a ponderação de cada fase de classificação."

	Questões 1 a 3 referente à validação da proposta de modelo de priorização		
Entrevista	O modelo de priorização desenvolvido neste estudo é relevante? Quais suas principais contribuições?	Os critérios das fases de classificação são importantes a serem considerados?	A fórmula presente na fase do Ranking Final de Priorização das Classes (RFP) é adequada? Algo poderia ser modificado?
ENT3	"Sim é relevante, os *smells* considerados e as métricas das fases de classificação são importantes. O modelo ajuda na priorização e na distinção dos problemas mais graves a serem corrigidos primeiro, baseando-se nos critérios de ranqueamento abordados, auxiliando os desenvolvedores na tomada de decisão e na justificação dos reparos que precisam ser realizados."	"Sim, são critérios importantes e válidos a serem considerados ao retratar esses problemas técnicos. Os mesmos refletem aspectos reais, presentes no dia a dia do desenvolvimento de *software*."	"Sim, a partir da forma como a fórmula está construída irá obter-se a prioridade dos problemas de modo igualitário ou geral em relação aos *rankings* utilizados. Uma sugestão seria permitir que a fórmula considerasse a ponderação dos critérios, para assim, possibilitar aos desenvolvedores ajustar a prioridade de alguma fase de ranqueamento específica. Outra sugestão é realizar testes de priorização com diferentes fórmulas para comparar os resultados e obter conclusões."
ENT4	"Sim é relevante, a proposta desenvolvida é composta por problemas importantes e que fazem parte do cotidiano de desenvolvimento. As fases utilizadas no modelo contribuem para ter-se um melhor conhecimento dos mesmos, de forma a gerenciá-los e priorizá-los através dos critérios associados, visando sugerir as correções mais urgentes e com um melhor retorno ao sistema. Além disso, as métricas utilizadas auxiliam na previsão da gravidade das classes para o futuro do *software*, permitindo a participação direta dos desenvolvedores no processo de priorização."	"Sim são critérios importantes, que fazem sentido ao priorizar a correção desses problemas."	"A fórmula atual atribui os mesmos pesos aos *rankings* de priorização. Uma sugestão seria permitir a ponderação dos critérios pelos desenvolvedores, permitindo captar problemas específicos, ou seja, atribuir maior prioridade à problemas desejados. Mas há a necessidade de testar na prática a real eficácia das fórmulas."

Tabela 6.2 – Respostas dos entrevistados perante as questões '1', '2' e '3' do Bloco II da entrevista. Elaborado pelo autor.

	Questões 4 a 6 referente à validação da proposta de modelo de priorização		
Entrevista	Quanto a compreensão da proposta, ela é simples de ser entendida?	Quanto a complexidade de execução da proposta, ela é fácil de ser aplicada?	Avaliando de modo geral, indique uma classificação para a importância da contribuição da proposta de modelo de priorização de classes *smelly* desenvolvida neste trabalho:
ENT1	"Muito simples e fácil de entender."	"Acredito que para aplicar o modelo de maneira fácil seria necessário automatizar o processo de priorização, para captar os dados necessários com maior rapidez e assim fazer os cálculos, além disso, seria muito importante a dedicação e o apoio do time de desenvolvimento durante todo processo."	( ) Extremamente importante (X) Muito importante ( ) Importante ( ) Pouco importante ( ) Sem importância
ENT2	"Sim, é fácil de entender o que está sendo proposto."	"Para facilitar a aplicação da proposta, deveria-se construir e interligar ferramentas para auxiliar na captura dos dados e informações necessárias para efetuar a priorização dos problemas, além do engajamento da empresa."	( ) Extremamente importante (X) Muito importante ( ) Importante ( ) Pouco importante ( ) Sem importância
ENT3	"É simples e fácil de entender o modelo e suas características."	"Acredito que para a proposta ser fácil de ser aplicada, deve-se ter um envolvimento da empresa de um modo geral, adequando o modelo de priorização ao ciclo de desenvolvimento de *software*. Além disso, acredito que seria importante automatizar o processo de priorização, buscando capturar e calcular as informações que o modelo necessita de forma agilizada e prática, auxiliando nas fases de ranqueamento e na gestão dos problemas identificados."	( ) Extremamente importante (X) Muito importante ( ) Importante ( ) Pouco importante ( ) Sem importância
ENT4	"Sim, é fácil de entender, pois é composta por conceitos que fazem parte do cotidiano de desenvolvimento."	"Para a proposta ser aplicada por uma empresa, acredito que necessitaria de um forte engajamento da mesma e de todo time, adequando-a ao processo de desenvolvimento. Os benefícios do modelo devem ser repassados à organização, justificando sua aplicação. Além disso, precisa-se encontrar formas para captar os dados necessários ao modelo com facilidade, automatizando o processo de priorização."	( ) Extremamente importante (X) Muito importante ( ) Importante ( ) Pouco importante ( ) Sem importância

Tabela 6.3 – Respostas dos entrevistados perante as questões '4', '5' e '6' do Bloco II da entrevista. Elaborado pelo autor.

Através dos relatos dos entrevistados perante as questões das tabelas 6.2 e 6.3, pode-se chegar às seguintes conclusões:

- Segundo os entrevistados, o modelo contribui para a priorização dos problemas considerados na abordagem através dos critérios das fases de classificação, sendo que os *smells* avaliados refletem anomalias importantes e as métricas são relevantes em relação aos itens a serem ranqueados. Além disso, o modelo auxilia na tomada de decisão referente à correção e na distinção dos problemas técnicos identificados, atribuindo maior atratividade àqueles que ocupam as melhores posições durante as fases do modelo. Outro ponto positivo levantado pelos mesmos, é que o modelo permite que os desenvolvedores participem diretamente na atribuição da prioridade das classes.

- Segundo a adequação da fórmula aplicada na fase final de priorização, os entrevistados relataram que a proposta necessita de testes na prática a fim de verificar os resultados da priorização alcançados pelo modelo, para assim, ter uma conclusão mais precisa referente a sua eficácia. Foi sugerido também, futuros testes com fórmulas que considerem ponderações dos critérios de classificação ou que permitam que os desenvolvedores possam fazê-las, propiciando atribuir pesos diferenciados às fases do modelo e consequentemente obter problemas específicos.

- Os entrevistados consideraram de forma unânime que a proposta de modelo de priorização e as métricas associadas são fáceis e simples de entender.

- Quanto a complexidade de execução da proposta, os entrevistados consideraram que para facilitar a aplicação do modelo necessitaria a automatização do processo de priorização, contando com o auxílio de ferramentas para capturar, cadastrar e calcular as informações necessárias. Além disso, acreditam que para colocar em prática o modelo de priorização seria de suma importância o envolvimento da empresa e o comprometimento do time de desenvolvimento.

- Todos os participantes da avaliação do modelo, consideraram a partir da questão 6 do Bloco II da entrevista, que de modo geral, a proposta de modelo de priorização desenvolvida neste estudo é muito importante em relação a sua contribuição para o ranqueamento dos problemas considerados.

7 CONSIDERAÇÕES FINAIS

Os *softwares* estão cada vez mais presentes no cotidiano das pessoas e das empresas, tornando-as dependentes dos mesmos para realizar diversas atividades do dia a dia. Com isso, os sistemas precisam estar em constante aprimoramento e evolução, para poderem suprir as necessidades dos clientes com eficácia e qualidade.

Entretanto, para permitir que os *softwares* continuem evoluindo é preciso que problemas vinculados aos mesmos sejam corrigidos ou mantidos sob gerenciamento, já que não adquiri-los é uma tarefa praticamente impossível. Dentre esses, está a dívida técnica, que refere-se de maneira geral aos artefatos de *software* que não se adequam às melhores condições de desenvolvimento e podem ocasionar maiores complicações ao sistema ao longo e médio prazo.

Dentre as implicações originadas pelas dívidas está a perda da qualidade do sistema em fatores como compreensão, testabilidade, manutenção e evolução. Além disso, ao longo do tempo as imperfeições técnicas podem incorrer no aumento do custo para corrigi-las, tornado a situação ainda mais problemática.

Uma solução simples para eliminar débitos técnicos em um sistema seria efetuar seus pagamentos, ou seja, realizar suas correções. Entretanto, as equipes de desenvolvimento possuem prazos de entrega apertados e dispõem de recursos limitados para aplicarem nos projetos, tornando difícil a realização dessas melhorias.

Nesse sentido, torna-se importante a atividade de priorização, que tem como objetivo classificar as dívidas técnicas identificadas em um sistema baseando-se em critérios pré-estabelecidos, visando assim, a correção de dívidas que realmente ofereçam ao *software* ganhos em sua qualidade e/ou que podem maximizar o seu risco de evolução e manutenção. Essa atividade também é responsável por manter as dívidas sob gestão, auxiliando os desenvolvedores e funções afins na tomada de decisão de quais problemas devem ser considerados e por quê.

Baseando-se nisso, o presente trabalho apresentou inicialmente a descrição dos trabalhos relacionados à priorização de dívida técnica encontrados na literatura. Posteriormente, os mesmos foram analisados e categorizados a partir das métricas e características que os compõem. Em seguida, foi proposto um modelo de priorização de classes afetadas por *smells* de código, cujo os critérios de ranqueamento foram oriundos dos estudos relacionados ao tema.

O modelo buscou relacionar possíveis dívidas técnicas ao nível do código-fonte a partir de *smells* presentes nas classes de um suposto sistema, levando em consideração que estes

representam indícios importante de dívidas. Além disso, as características que fazem parte das fases de ranqueamento da abordagem contribuem para que problemas consideravelmente importantes possuam uma maior prioridade de correção associada, distinguindo-os dos restantes. Além, de propiciar a priorização ou a consideração inicial de problemas que possuem um melhor custo/benefício de correção vinculado.

O modelo desenvolvido foi validado por conhecedores na área, onde constatou-se que a proposta é de fácil entendimento e contribui na priorização dos problemas técnicos através dos fatores de ranqueamento considerados, sendo esses, importantes e relevantes a serem utilizados nesse contexto, auxiliando assim, na justificação e na tomada de decisões vinculadas à correção dos mesmos, conforme descrito durante a Seção 6.3. Percebeu-se também, que a automatização do processo de priorização proposto pelo modelo seria fundamental para a aplicabilidade do mesmo, auxiliando na identificação, na coleta e nos cálculos das informações necessárias com maior rapidez e eficácia.

Considera-se assim, que os objetivos deste estudo foram atendidos, pois conseguiu-se chegar em um resultado que atende de forma satisfatória o que foi estipulado inicialmente.

7.1 Trabalhos futuros

O presente trabalho apresentou o desenvolvimento de uma proposta de modelo para a priorização de classes afetadas por *code smells*, que referem-se à problemas ao nível do código e podem indicar dívidas técnicas em um sistema. A abordagem construída possui seis fases de classificação, onde cada fase representa critérios específicos de ranqueamento, tendo como objetivo priorizar as classes mais significativas em relação aos mesmos para sua eventual correção.

Sugere-se como trabalho futuro a avaliação e a aplicação do modelo de priorização desenvolvido em casos de testes reais, a fim de comprovar na prática sua contribuição e eficácia. Além disso, para facilitar a aplicação dos critérios sugeridos na abordagem seria de suma importância a implementação e a junção de ferramentas com o intuito de automatizar o processo de priorização, buscando capturar os dados necessários e realizar os cálculos associados com maior rapidez e eficiência.

Como o presente trabalho limita-se ao ranqueamento de classes infectadas por sete tipos de *smell*, sugere-se futuramente ampliar a gama de *smells* considerados no modelo e utilizar mais critérios de classificação, tornando a proposta mais abrangente. Também recomenda-se

a construção de uma abordagem que considere dívidas técnicas e métricas oriundas de outros artefatos de *software*, não somente relacionadas ao código-fonte.

REFERÊNCIAS

[1] M. Abbes, F. Khomh, Y.-G. Gueheneuc, and G. Antoniol. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *Software maintenance and reengineering (CSMR), 2011 15th European conference on*, pages 181–190. IEEE, 2011.

[2] S. Akbarinasaji. Toward measuring defect debt and developing a recommender system for their prioritization. In *Proceedings of the 13th International Doctoral Symposium on Empirical Software Engineering*, pages 15–20, 2015.

[3] E. Allman. Managing technical debt. *Communications of the ACM*, 55(5):50–55, 2012.

[4] N. S. Alves, T. S. Mendes, M. G. de Mendonça, R. O. Spínola, F. Shull, and C. Seaman. Identification and management of technical debt: A systematic mapping study. *Information and Software Technology*, 70:100–121, 2016.

[5] F. Arcelli, C. Tosi, M. Zanoni, and S. Maggioni. The marple project: A tool for design pattern detection and software architecture reconstruction. In *1st International Workshop on Academic Software Development Tools and Techniques (WASDeTT-1)*, pages 325–334, 2008.

[6] R. Arcoverde, E. Guimarães, I. Macía, A. Garcia, and Y. Cai. Prioritization of code anomalies based on architecture sensitiveness. In *Software Engineering (SBES), 2013 27th Brazilian Symposium on*, pages 69–78. IEEE, 2013.

[7] P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman. Managing technical debt in software engineering (dagstuhl seminar 16162). In *Dagstuhl Reports*, volume 6. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[8] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *IEEE software*, 25(5), 2008.

[9] I. M. Bertrán. *Avaliação da qualidade de software com base em modelos uml*. PhD thesis, PUC-Rio, 2009.

[10] I. M. Bertrán. *On the detection of architecturally-relevant code anomalies in software systems*. PhD thesis, PhD thesis, Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Rio de Janeiro, Brazil, 2013.

[11] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, et al. Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 47–52. ACM, 2010.

[12] F. Buschmann. To pay or not to pay technical debt. *IEEE software*, 28(6):29–31, 2011.

[13] A. Choudhary and P. Singh. Minimizing refactoring effort through prioritization of classes based on historical, architectural and code smell information. In *QuASoQ/TDA@ APSEC*, pages 76–79, 2016.

[14] R. Coanda and M. F. Harun. A study of cost-benefit analysis of technical debt. *Full-scale Software Engineering/The Art of Software Testing*, page 62, 2017.

[15] Z. Codabux and B. J. Williams. Technical debt prioritization using predictive analytics. In *Software Engineering Companion (ICSE-C), IEEE/ACM International Conference on*, pages 704–706. IEEE, 2016.

[16] W. Cunningham. The wycash portfolio management system. *Proc. OOPSLA, ACM*, 1992.

[17] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-oriented reengineering patterns*. Elsevier, 2002.

[18] T. Dingsøyr, N. B. Moe, R. Tonelli, S. Counsell, C. Gencel, and K. Petersen. *Agile Methods. Large-Scale Development, Refactoring, Testing, and Estimation: XP 2014 International Workshops, Rome, Italy, May 26-30, 2014, Revised Selected Papers*, volume 199. Springer, 2014.

[19] B. Dit, A. Holtzhauer, D. Poshyvanyk, and H. Kagdi. A dataset from change history to support evaluation of software maintenance tasks. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 131–134. IEEE Press, 2013.

[20] B. Dit, E. Moritz, and D. Poshyvanyk. A tracelab-based solution for creating, conducting, and sharing feature location experiments. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 203–208. IEEE, 2012.

[21] Eclipse-Marketplace. Jdeodorant. https://marketplace.eclipse.org/content/jdeodorant. [Online; Acesso em: 20/03/2018].

[22] Empear. Codescene, 2015. https://codescene.io/. [Online; Acesso em: 07/06/2018].

[23] C. Fernández-Sánchez, H. Humanes, J. Garbajosa, and J. Díaz. An open tool for assisting in technical debt management. In *Software Engineering and Advanced Applications (SEAA), 2017 43rd Euromicro Conference on*, pages 400–403. IEEE, 2017.

[24] FindBugs. Find bugs in java programs, 2015. http://findbugs.sourceforge.net/. [Online; Acesso em: 13/03/2018].

[25] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou. Jdeodorant: Identification and removal of feature envy bad smells. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pages 519–520. IEEE, 2007.

[26] F. A. Fontana, V. Ferme, and S. Spinelli. Investigating the impact of code smells debt on quality code evaluation. In *Proceedings of the Third International Workshop on Managing Technical Debt*, pages 15–22. IEEE Press, 2012.

[27] F. A. Fontana, V. Ferme, M. Zanoni, and R. Roveda. Towards a prioritization of code debt: A code smell intensity index. In *Managing Technical Debt (MTD), 2015 IEEE 7th International Workshop on*, pages 16–24. IEEE, 2015.

[28] M. Fowler. Estimated interest, 2008. https://martinfowler.com/bliki/EstimatedInterest.html. [Online; Acesso em: 10/02/2018].

[29] M. Fowler. Technical debt, 2009. https://martinfowler.com/bliki/TechnicalDebt.html. [Online; Acesso em: 10/11/2017].

[30] M. Fowler. Technical debt quadrant, 2009. http://www.martinfowler.com/bliki/TechnicalDebtQuadrant.html. [Online; Acesso em: 10/09/2017].

[31] M. Fowler and K. Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[32] G. Gibbs. *Análise de dados qualitativos: coleção pesquisa qualitativa*. Bookman Editora, 2009.

[33] A. C. Gil. Como elaborar projetos de pesquisa. *São Paulo*, 5(61):16–17, 2002.

[34] T. Girba, S. Ducasse, and M. Lanza. Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 40–49. IEEE, 2004.

[35] Z. Gong and F. Lyu. Technical debt management in a large-scale distributed project: An ericsson case study, 2017.

[36] Y. Guo and C. Seaman. A portfolio approach to technical debt management. In *Proceedings of the 2nd Workshop on Managing Technical Debt*, pages 31–34. ACM, 2011.

[37] Y. Guo, C. Seaman, R. Gomes, A. Cavalcanti, G. Tonin, F. Q. Da Silva, A. L. Santos, and C. Siebra. Tracking technical debt—an exploratory case study. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 528–531. IEEE, 2011.

[38] S. Herold, M. English, J. Buckley, S. Counsell, and M. Ó. Cinnéide. Detection of violation causes in reflexion models. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 565–569. IEEE, 2015.

[39] J. Holvitie and V. Leppänen. Debtflag: Technical debt management with a development environment integrated tool. In *Managing Technical Debt (MTD), 2013 4th International Workshop on*, pages 20–27. IEEE, 2013.

[40] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, pages 672–681. IEEE Press, 2013.

[41] E. Keenan, A. Czauderna, G. Leach, J. Cleland-Huang, Y. Shin, E. Moritz, M. Gethers, D. Poshyvanyk, J. Maletic, J. Huffman Hayes, et al. Tracelab: An experimental workbench for equipping researchers to innovate, synthesize, and comparatively evaluate traceability solutions. In *Proceedings of the 34th international conference on Software Engineering*, pages 1375–1378. IEEE Press, 2012.

[42] M. Kim, D. Cai, and S. Kim. An empirical investigation into the role of api-level refactorings during software evolution. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 151–160. ACM, 2011.

[43] P. Kruchten, R. L. Nord, I. Ozkaya, and J. Visser. Technical debt in software development: from metaphor to theory report on the third international workshop on managing technical debt. *ACM SIGSOFT Software Engineering Notes*, 37(5):36–38, 2012.

[44] M. Lanza, S. Ducasse, H. Gall, and M. Pinzger. Codecrawler: an information visualization tool for program comprehension. In *Proceedings of the 27th international conference on Software engineering*, pages 672–673. ACM, 2005.

[45] M. Lanza, R. Marinescu, and S. Ducasse. Object oriented metrics in practice: using software metrics to characterize. *Evaluate, and Improve the Design of Object-Oriented Systems*, page 220, 2006.

[46] M. M. Lehman and L. A. Belady. *Program evolution: processes of software change*. Academic Press Professional, Inc., 1985.

[47] Z. Li, P. Avgeriou, and P. Liang. A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, 101:193–220, 2015.

[48] R. Malhotra, A. Chug, and P. Khosla. Prioritization of classes for refactoring: A step towards improvement in software quality. In *Proceedings of the Third International Symposium on Women in Computing and Informatics*, pages 228–234. ACM, 2015.

[49] M. d. A. Marconi and E. M. Lakatos. *Fundamentos de metodologia científica*. 5. ed.-São Paulo: Atlas, 2003.

[50] R. Marinescu. Measurement and quality in object oriented design. 2002.

[51] R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 350–359. IEEE, 2004.

[52] A. Martini and J. Bosch. Towards prioritizing architecture technical debt: information needs of architects and product owners. In *Software Engineering and Advanced Applications (SEAA), 2015 41st Euromicro Conference on*, pages 422–429. IEEE, 2015.

[53] A. Martini and J. Bosch. The magnificent seven: towards a systematic estimation of technical debt interest. In *Proceedings of the XP2017 Scientific Workshops*, page 7. ACM, 2017.

[54] A. Martini and J. Bosch. On the interest of architectural technical debt: Uncovering the contagious debt phenomenon. *Journal of Software: Evolution and Process*, 2017.

[55] A. Martini, J. Bosch, and M. Chaudron. Architecture technical debt: Understanding causes and a qualitative model. In *Software Engineering and Advanced Applications (SEAA), 2014 40th EUROMICRO Conference on*, pages 85–92. IEEE, 2014.

[56] S. McConnell. Technical debt, 2007. http://www.construx.com/10x_Software_Development/Technical_Debt/. [Online; Acesso em: 30/08/2017].

[57] S. McConnell. Managing technical debt (slides). In *Workshop on Managing Technical Debt (part of ICSE 2013): IEEE*, 2013.

[58] S. Mensah, J. Keung, J. Svajlenko, K. E. Bennin, and Q. Mi. On the value of a prioritization scheme for resolving self-admitted technical debt. *Journal of Systems and Software*, 135:37–54, 2018.

[59] R. Moraes. Análise de conteúdo. *Revista Educação, Porto Alegre*, 22(37):7–32, 1999.

[60] R. Plösch, J. Bräuer, M. Saft, and C. Körner. Design debt prioritization - a design best practice-based approach. In *International Conference on Technical Debt (TechDebt) 2018, Gothenburg, Sweden, May 27 - June 3rd, 2018*. ACM, 2018.

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

[62] A. Rani and J. K. Chhabra. Prioritization of smelly classes: A two phase approach (reducing refactoring efforts). In *Computational Intelligence & Communication Technology (CICT), 2017 3rd International Conference on*, pages 1–6. IEEE, 2017.

[63] A. J. Riel. *Object-oriented design heuristics*. Addison-Wesley Publishing Company, 1996.

[64] J. Ruiz, M. A. Ispizua, et al. La descodificación de la vida cotidiana. métodos de investigación cualitativa. *Bilbao: Universidad de Deusto*, 1989.

[65] N. Sae-Lim, S. Hayashi, and M. Saeki. Context-based code smells prioritization for prefactoring. In *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*, pages 1–10. IEEE, 2016.

[66] C. Seaman and Y. Guo. Measuring and monitoring technical debt. *Advances in Computers*, 82(25-46):44, 2011.

[67] C. Seaman, Y. Guo, C. Izurieta, Y. Cai, N. Zazworka, F. Shull, and A. Vetrò. Using technical debt data in decision making: Potential decision approaches. In *Proceedings of the Third International Workshop on Managing Technical Debt*, pages 45–48. IEEE Press, 2012.

[68] R. Shatnawi, W. Li, J. Swain, and T. Newman. Finding software metrics threshold values using roc curves. *Journal of Software: Evolution and Process*, 22(1):1–16, 2010.

[69] E. L. d. Silva and E. M. Menezes. Metodologia da pesquisa e elaboração de dissertação. 2001.

[70] P. Sommerhoff and M. F. Harun. Towards a modularity-based technical debt prioritization approach. *Full-scale Software Engineering/Current Trends in Release Engineering*, page 45, 2016.

[71] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[72] S. A. Vidal, C. Marcos, and J. A. Díaz-Pace. An approach to prioritize code smells for refactoring. *Automated Software Engineering*, 23(3):501–532, 2016.

[73] A. Yamashita and L. Moonen. Do code smells reflect important maintainability aspects? In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 306–315. IEEE, 2012.

[74] J. Yli-Huumo et al. The role of technical debt in software development. *Acta Universitatis Lappeenrantaensis*, 2017.

[75] J. Yli-Huumo, A. Maglyas, and K. Smolander. How do software development teams manage technical debt?–an empirical study. *Journal of Systems and Software*, 120:195–218, 2016.

[76] N. Zazworka. Tool development, 2013. http://www.nicozazworka.com/tool-development/. [Online; Acesso em: 11/03/2018].

[77] N. Zazworka, C. Izurieta, S. Wong, Y. Cai, C. Seaman, F. Shull, et al. Comparing four approaches for technical debt identification. *Software Quality Journal*, 22(3):403–426, 2014.

[78] N. Zazworka and C. Seaman. Identifying and managing technical debt, 2012. http://www.slideshare.net/zazworka/identifying-and-managing-technical-debt. [Online; Acesso em: 02/03/2018].

[79] N. Zazworka, C. Seaman, and F. Shull. Prioritizing design debt investment opportunities. In *Proceedings of the 2nd Workshop on Managing Technical Debt*, pages 39–42. ACM, 2011.

[80] N. Zazworka, R. O. Spínola, A. Vetro, F. Shull, and C. Seaman. A case study on effectively identifying technical debt. In *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*, pages 42–47. ACM, 2013.

APÊNDICES

APÊNDICE A – Entrevista de validação da proposta de modelo de priorização elaborada

Esta entrevista visa através do Bloco I identificar o perfil dos entrevistados a serem utilizados para validar a proposta de modelo de priorização de classes afetadas por *smells* de código desenvolvida neste trabalho. Já as perguntas referentes à avaliação do modelo elaborado encontram-se no Bloco II.

Bloco I - Questões referentes ao perfil do entrevistado

1. Qual o seu cargo?
2. Há quanto tempo exerce essa função?
3. Conhece os termos dívida técnica e a atividade de priorização de dívida técnica? A empresa ou organização que você trabalha utiliza algum modelo de priorização?

Bloco II - Questões referentes à validação da proposta de modelo de priorização de dívida técnica

1. O modelo de priorização desenvolvido neste estudo é relevante? Quais suas principais contribuições?
2. Os critérios das fases de classificação são importantes a serem considerados?
3. A fórmula presente na fase do Ranking Final de Priorização das Classes (RFP) é adequada? Algo poderia ser modificado?
4. Quanto a compreensão da proposta, ela é simples de ser entendida?
5. Quanto a complexidade de execução da proposta, ela é fácil de ser aplicada?
6. Avaliando de modo geral, indique uma classificação para a importância da contribuição da proposta de modelo de priorização de classes *smelly* desenvolvida neste trabalho:

   ( ) Extremamente importante
   ( ) Muito importante
   ( ) Importante

( ) Pouco importante

( ) Sem importância