Um modelo baseado em dados históricos para a estimação da dívida técnica

Jandisson Soares de Jesus

Texto da Tese de Doutorado apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para
obtenção do título
de
Doutor em Ciências

Programa: Ciência da Computação

Orientadora: Profa. Dra. Ana Cristina V. de Melo

Durante o desenvolvimento deste trabalho o autor recebeu auxílio financeiro da CAPES

São Paulo, Julho de 2016

Um modelo baseado em dados históricos para a estimação da dívida técnica

Esta é a versão original da tese elaborada pelo candidato Jandisson Soares de Jesus, tal como submetida à Comissão Julgadora.

Resumo

Jesus, J. S. **Um modelo baseado em dados históricos para a estimação da dívida técnica**. 2016. 120 f. Tese (Doutorado) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2016.

Negligenciar o gerenciamento da dívida técnica traz consequências negativas para os projetos de desenvolvimento de software. Caso a dívida técnica atinja patamares muito altos, é possível que a continuidade do projeto se torne inviável. Uma das atividades do gerenciamento é estimar o esforço adicional, causado pela existência da dívida técnica, para realizar as futuras atividades de desenvolvimento. Esse esforço adicional é chamado de juros da dívida técnica. Apesar de sua importância, pouco se sabe a respeito dele. Essa falta de informação dificulta o gerenciamento, pois a estimativa dos juros é essencial para a priorização do pagamento da dívida técnica. Caso uma dívida apresente juros muito baixos, não faz sentido que seu pagamento seja priorizado. Semelhantemente, caso uma dívida tenha os juros muito alto, o pagamento dela deve ser priorizado. Além disso, saber quais tipos de dívida apresentam maiores juros permitiria a definição de estratégias para evitar a criação desses tipos de dívida técnica. Neste projeto iremos propor um modelo para estimar o comportamento dos juros da dívida técnica. Esse modelo irá utilizar dados de projetos presentes em repositórios de software para analisar o impacto da dívida técnica no desenvolvimento e evolução dos projetos. Iremos estimar, para um determinado tipo de dívida técnica, o quanto de esforço extra será necessário para realizar as atividades de desenvolvimento e evolução do software em decorrência da existência dessa dívida.

Palavras-chave: dívida técnica, repositório de software, estimativa de software.

Abstract

Jesus, J. S. **A repository-based model to estimate technical debt.**. 2016. 120 f. Tese (Doutorado) - Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2016.

An insuficient technical debt management can bring bad consequences to software development projects. If the technical debt reaches a too high level, it is possible that the continuity of the project becomes unfeasible. One of the management activities is the estimation of the additional effort to make future development activities. We call this additional effort as the interest of the technical debt. Despite its importance, little is known about him. This lack of information difficults its management because the estimation of the interest is essential for the prioritization of the technical debt payment. If a technical debt presents a very low interest, is payment do not neeed to be prioritized. Similarly, if a debt has very high interest, the payment it should be prioritized. Also, finding which debt types have higher interest rates would allow the development of strategies to prevent the creation of these kinds of technical debt. In this project we propose a model to estimate the behavior of technical debt interest. This model will use data of projects in software repositories to analyze the impact of technical debt in the development and evolution of projects. We estimate, for a given type of technical debt, how much extra effort will be needed to carry out development activities and progress of the software due to the existence of that debt.

Keywords: techncial debt, software repository, software estimation.

Sumário

Lista de Abreviaturas

TD Dívida Técnica (*Technical Debt*)

Lista de Figuras

Lista de Tabelas

Capítulo 1

Dívida Técnica

Neste capítulo, descrevemos o que é uma dívida técnica, seus principais tipos e formas de classificação juntamente com as atuais abordagens de gerenciamento.

1.1 Introdução

Desenvolver software é uma atividade complexa por diversas razões. Entre elas estão as dificuldades em gerenciar requisitos muitas vezes ambíguos e até mesmo conflitantes, a imprevisibilidade do contexto no qual o software está inserido e as particularidades das tecnologias utilizadas. Nem sempre todos esses fatores poderão ser devidamente tratados nos projetos de software. Em algumas circunstâncias, pode não ser possível lidar com todos eles de forma satisfatória devido ao seu número excessivo e a falta de recursos disponíveis tais como a quantidade de membros na equipe e o tempo disponível para realizar as tarefas. Essa falta de recursos pode fazer com que seja necessário realizar algumas escolhas para que um projeto possa ser viabilizado. Existem algumas opções para tornar viável um projeto que tenha recursos incompatíveis. A solução mais óbvia é conseguir mais recursos. Outra opção é a eliminação ou simplificação de determinadas funcionalidades e com isso diminuir o esforço necessário para realizar o projeto. Naturalmente, nem sempre é possível que uma dessas duas opções possa ser seguida. Isso pode levar a uma situação onde algumas das atividades do projeto tenham de ser realizadas utilizando menos recursos. Essa redução nos recursos necessários pode ser alcançada melhorando a eficiência dos processos ou diminuindo a qualidade na qual eles são realizados. Um aperfeiçoamento na eficiência dos processos é algo que, apesar de positivo, pode não ser alcançável. Enquanto isso, quase sempre é possível diminuir a qualidade na qual um processo é realizado. Isso faz com que essa diminuição na qualidade seja a solução mais fácil para resolver o problema da falta de recursos. Uma dívida técnica é a diminuição na qualidade

de algum aspecto do projeto de software para viabilizá-lo e que gerará dificuldades adicionais para desenvolvê-lo no futuro.

Uma das formas de definir uma dívida técnica é como **algo** no software ou no seu processo de desenvolvimento que não está ideal e que por causa disso poderá haver algum tipo de **dificuldade adicional**. Esse **algo** pode ser a existência de código de má qualidade, um design inadequado, uma tecnologia ultrapassada dentro outros. A **dificuldade adicional** é o aumento de esforço necessário para realizar alguma atividade relacionada ao software no futuro. Esse aumento de esforço não existiria caso o **algo** também não existisse. Essa é uma definição propositalmente ambígua já que o termo dívida técnica foi demasiadamente estendido e aplicado em diversas situações tornando desafiadora a tarefa de defini-lo precisamente.

Para ilustrarmos o que é uma dívida técnica forneceremos alguns exemplos. O primeiro deles é baseado no código da Figura 1.1. Nele, há um trecho de uma classe chamada RelatorioV1. Essa classe tem a função de receber os nomes e endereços de algumas pessoas e gerar um relatório em algum formato previamente estabelecido. A função adicionarLinha é acionada por outras classes toda vez que uma nova pessoa tiver sido obtida da fonte de dados. A função gerarRelatorioFormatado é executada quando todas as pessoas tiverem sido obtidas. Apesar da simplicidade dessa classe, ela contém uma dívida técnica. Caso uma nova coluna tenha de ser incluída no relatório, seria necessário alterar ao menos o método adicionarLinha e todas as classes que o utilizam. Agora se analisarmos a classe RelatorioV2 podemos ver que esse problema foi resolvido. Nessa versão, é utilizado um vetor com todos os campos que deverão ser incluídos no relatório. Assim, caso um novo campo tivesse que ser adicionado, como por exemplo o telefone da pessoa, não seria necessária nenhuma alteração no método adicionarLinha. Entretanto, fica claro que é necessário um maior esforço para escrever a classe RelatorioV2 do que a classe RelatorioV1. É mais rápido escrever a classe RelatorioV1, porém todas as vezes que for necessário adicionar um novo campo ao relatório, haverá um esforço maior. Optar pela classe RelatorioV1 ao invés da classe RelatorioV2 é adquirir uma dívida técnica. Há um ganho imediato de tempo já que a implementação é substancialmente mais simples. Entretanto, haverá uma dificuldade adicional para evoluir esse software devido a essa escolha.

```
class RelatorioV1
{
  separador = "," ;
  novaLinha = "\n";
  linhas = "";
  function adicionarLinha(nome,endereco)
  {
    this.linhas = this.linhas + nome + separador + endereco + novaLinha;
  }
  function gerarRelatorioFormatado()
  {
    /* Gera o relatorio com todas as linhas adicionadas. */
  }
}
```

```
class RelatorioV2
{
  separador = "," ;
  novaLinha = "\n";
  linhas = "";
  function adicionarLinha(campos)
  {
     if(campos.size > 0 )
     {
       for(i=0;i<campos.size;i++)
       {
          this.linhas = this.linhas + campos[i] + separador;
       }
       this.linhas = substring(this.linhas,0,this.linhas.size - separador.size );
       this.linhas = this.linhas+novaLinha;
     }
}
function gerarRelatorioFormatado()
{
   /* Gera o relatorio com todas as linhas adicionadas. */
 }
}
```

Figura 1.1: *Exemplo de dívida técnica em código fonte.*

Nosso segundo exemplo de dívida técnica está relacionado com o banco de dados de uma aplicação. De acordo com [EN16] as restrições de integridade são mecanismos que os sistemas gerenciadores de banco de dados fornecem para que os usuários possam definir regras a respeito dos dados armazenados de forma que eles se mantenham consistentes e representem corretamente a realidade modelada. Um tipo de restrição muito comum são as de integridade referencial. Nesse tipo de restrição, basicamente, o banco de dados garante que para cada linha em uma relação onde exista uma chave estrangeira, sempre exista a linha correspondente na relação associada. O princial papel desse tipo de restrição de integridade é evitar situações no qual uma chave estrangeira faça referência a um dado que não existe na tabela associada. Na Figura 1.2 há um exemplo de um modelo em que as restrições de integridade seriam úteis para garantir a consistência dos dados. Esse modelo contém três tabelas: Aluno, Curso e Histórico. Na tabela Histórico temos duas chaves estrangeiras chamadas de $ALUNO\_ID$ e $CURSO\_ID$. Caso alguns alunos ou cursos, que estejam presentes na tabela Histórico, sejam removidos, as linhas que contém referências a esses elementos não mais farão sentido dentro do modelo e indicarão a existência de dados inconsistentes. Uma forma de evitar esse problema é criar uma restrição de integridade de tal forma que, antes de remover alguma linha nas tabelas Aluno e Curso, o próprio sistema gerenciador do banco de dados verifique se isso não gerará linhas órfãs na tabela Histórico. Nesse exemplo é simples identificar qual restrição de integridade é necessária para garantir a consistência do modelo. Entretanto, em situações reais, a quantidade de tabelas e restrições necessárias podem ser muito grandes. Em algumas situações nem todas as restrições são devidamente mapeadas e implementadas no banco de dados devido à alguma restrição de recurso. Quando isso acontece, há a aquisição de uma dívida técnica. A dificuldade adicional gerada por essa dívida ocorre quando é necessário incluir alguma restrição que não foi anteriormente aplicada. Isso pode levar a necessidade de adaptar e testar um número grande de partes do sistema que de alguma forma utilizam as tabelas relacionadas com a restrição. É possível inclusive que seja necessário realizar alterações nessas partes para que elas se adequem a inclusão das novas restrições de integridade. Isso pode levar a um custo substancial gerado pela necessidade de um conjunto de alterações em cascata.

Utilizaremos um exemplo relacionado às atividades de testes durante o processo de desenvolvimento de software para concluir nossa ilustração a respeito das dívidas técnicas. Existem diversos tipos de testes que podem ser realizados em um software. Dentre esses tipos, os testes unitários são aqueles que têm como objetivo validar se as menores unidades estão funcionando individualmente como o esperado[CL02, Run06]. Esses testes consistem, basicamente, em acionar essas unidades fornecendo uma entrada e verificar se a saída corresponde ao que foi especificado. Essas unidades

Aluno

ALUNO_ID	Nome
101	João
102	Henrique
103	Matheus

Curso

CURSO_ID	Curso
1	Matemática
2	História
3	Física

Histórico

HISTORICO_ID	ALUNO_ID	CURSO_ID	NOTA
301	**101**	**1**	**A**
302	**101**	**2**	**B**
303	**101**	**3**	**A**

Figura 1.2: *Exemplo de dívida técnica em banco de dados.*

podem ser métodos, classes, funcionalidades ou módulos. Em um cenário perfeito, todas as unidades do software deveriam ser testadas para todas as entradas possíveis. Naturalmente, devido à quantidade de recursos necessários, isso não é possível em todos os casos. Sendo assim, existe a necessidade de selecionar quais testes serão criados. Essa seleção pode ser feita de diversas formas. Seja pela priorização das unidades mais importantes ou pela escolha das entradas que são mais prováveis de serem fornecidas durante o funcionando do sistema. Além disso, é necessário que haja uma compatibilização entre a quantidade de testes que serão criados e a quantidade de recursos disponíveis. Haverá a aquisição de uma dívida técnica caso o número de testes criados não seja compatível com o nível de qualidade necessário para o software e no futuro seja necessário criar mais testes. A dificuldade adicional gerada pela existência dessa dívida técnica é causada pelo fato de que possivelmente seja necessário realizar refatorações[MKAH14, MSM+16] nas unidades do sistema para facilitar ou até mesmo tornar possível a criação desses testes. Isso ocorre porque a estrutura dessas unidades pode ter sido construída de forma que seja impossível testar determinados comportamentos. Na Figura 1.3 exibimos duas versões de uma mesma classe. Na primeira versão, existe uma dificuldade em testar o método calculaImposto. Isso acontece, pois, essa versão do método tem duas responsabilidades: calcular o imposto e inserir o resultado no banco de dados. Caso um teste unitário fosse escrito para essa versão, seria necessário fornecer uma conexão de

banco de dados válida. Além disso, a inserção de dados em um banco de dados iria de encontro com os objetivos dos testes unitários já que o teste abrangeria um escopo maior do que uma unidade. Esses problemas não são encontrados na versão 2 do método calculaImposto. Nessa versão, a conexão com o banco de dados é um parâmetro do método. Assim, é possível criar um teste onde fosse utilizada uma conexão fictícia de banco de dados ou um Mock[MFC00, Kac12, Ach14]. Esse teste verificaria se o método calculaImposto está tendo um comportamento conforme a especificação do software. A refatoração que levou o método da versão 1 para a versão 2 também exigirá que todas as referências ao método calculaImposto sejam alteradas. Logo, haverá uma dificuldade adicional para realizar essa alteração se compararmos a dificuldade de realizá-la no momento em que a versão 1 foi criada.

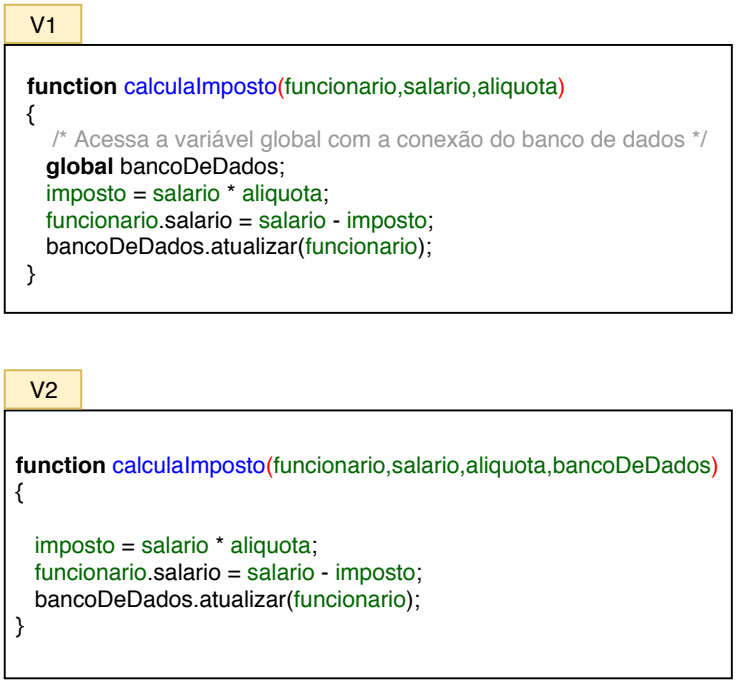

Figura 1.3: *Exemplo de refatoração para viabilizar a criação de um teste unitário.*

.

1.2 Definição da dívida técnica

Em 1992 Cunningham[Cun93] criou o termo dívida técnica da seguinte forma:

"*Although immature code may work fine and be completely acceptable to the customer, excess quantities will make a program unmasterable, leading to extreme specialization of programmers and finally an inflexible product. Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. Objects make the cost of this transaction tolerable. The danger occurs when the debt is not repaid. Every minute spent on not-*

quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object- oriented or otherwise."

Apesar de Cunningham ser considerado o criador da metáfora, outros autores escreveram previamente a respeito das dificuldades de manutenção e evolução causadas por problemas de design. Um exemplo é o conjunto de leis da evolução de software criadas por Lehman [Leh80, Leh96] em 1980. Nesse trabalho, o autor analisa estudos prévios sobre processos de programação e acompanha a evolução do sistema operacional OS/300 durante um período de 20 anos. Com os dados obtidos, são formuladas 8 leis que descrevem algumas características observadas durante a evolução de um software. Algumas dessas leis apresentam conceitos claramente muito semelhantes aos utilizados para descrever a metáfora da dívida técnica. Especialmente, as leis II e VII, transcritas a seguir.

II - *Increasing Complexity*

"As a program is evolved its complexity increases unless work is done to maintain or reduce it."

VII - *Declining Quality*

"E-type programs will be perceived as of declining quality unless rigorously maintained and adapted to a changing operational environment."

A lei II descreve de forma indireta a dívida técnica de design e arquitetura. Esses tipos de dívida técnica serão descritos nas seções 1.5.2 e 1.5.6 respectivamente. Enquanto isso, a lei VII está relacionada às dívidas técnicas de tecnologia, descritas na seção 1.5.8. Um sistema chamado de *E-type* é aquele efetivamente utilizado em um contexto real, ou seja, as leis descritas por Lehman não se aplicam a sistemas que não operam em um contexto sujeito a mudanças. Apesar de haver contestações a respeito do uso do termo "leis", este trabalho permitiu observar como a percepção dos usuários e programadores muda à medida que o tempo passa e o software evolui.

Apesar da existência de relatos semelhantes na década de 80 e da criação da metáfora em 1992, apenas a partir do ano de 2006 [MJ06] que a analogia voltou a ser discutida e estudada cientificamente. Inicialmente houve um esforço por parte dos pesquisadores em criar uma definição precisa do que é uma dívida técnica. Essa definição inicial indicava que uma dívida técnica deveria ser algo invisível para o usuário final do software. A Figura 1.4 apresenta uma adaptação da representação gráfica dessa definição.

Além da expansão do conceito para incluir diversos tipos de dívida conforme veremos na seção 1.5, houve, com o objetivo de torná-la menos ambígua, também um aprimoramento da definição

em si. Essa evolução permitiu diferenciá-la do simples déficit de qualidade de um software, tornar o conceito mais claro, e principalmente, facilitar a identificação do que não se trata de uma dívida técnica. Na Figura 1.4 há um resumo a respeito desse estágio de evolução do conceito. Nele, houve uma tentativa de separar as dívidas técnicas de outros aspectos de qualidade do software. O resultado foi que apenas os problemas de qualidade que são invisíveis para o usuário final foram definidos como dívida técnica. Objetivo dessa separação é evitar a diluição do conceito. Essa diluição faria com que uma quantidade demasiadamente grande de situações fossem consideradas dívidas técnicas. Isso dificultaria a criação de pesquisas mais aplicadas a respeito de técnicas e ferramentas para o gerenciamento da dívida técnica. Com isso, ficou definido que as dívidas técnicas são problemas de qualidade interna do software. A qualidade interna engloba os aspectos que não são visíveis para o usuário final do software[AQ10]. Na Figura 1.5 há uma indicação do lugar das dívidas técnicas dentro do contexto de qualidade de software.

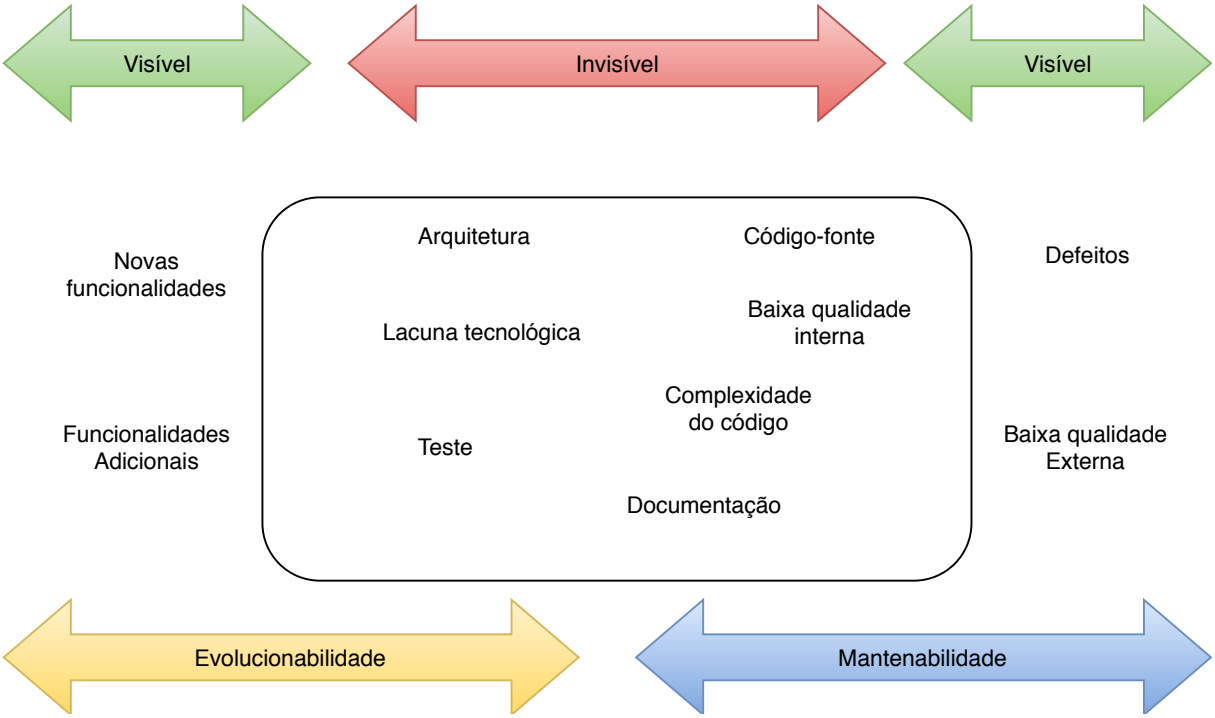

Figura 1.4: *Panorama da dívida técnica. Adaptado de [KNOF13].*

.

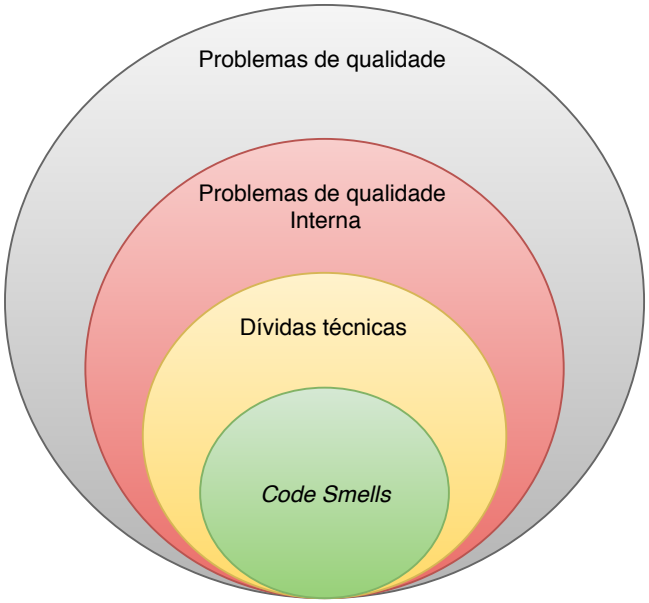

Figura 1.5: *Localização da dívida técnica na hierarquia dos problemas de qualidade de software.*

.

Com o aumento da popularidade dessa metáfora, novos tipos de dívida técnica surgiram. Alguns desses novo tipos não se enquadravam nessa definição, como, por exemplo, as dívidas de usabilidade e os defeitos. Isso fez com que os pesquisadores tivessem que abandonar a procura por uma definição precisa. Ao invés disso, foi adotada uma definição mais flexível e que permitisse que essa metáfora pudesse ser utilizada em diferentes situações. Um dos resultados desse esforço em busca de uma expansão do conceito foi a seguinte definição criada por Kruchten et al.[KNOF13]:

"A unifying perspective is emerging of technical debt as the invisible results of past decisions about software that affect its future. The affect can be negative in the form of poorly managed risks but if properly managed can be seen in a positive light to add value in the form of deferred investment opportunities. "

Fica evidente, nessa definição, a preocupação em não mais restringir as dívidas técnicas a problemas presentes no código fonte do software. Um dos motivos disso é a necessidade de diferenciá-las do conceito de *Bad Smells* criado por Fowler[Fow09] e que se tornou muito popular na comunidade de desenvolvimento de software[OCBZ09]. Um "*smell*" é uma violação à algum princípio do desenho de software orientado a objetos. Alguns exemplos de *smells* são código duplicado, classes muito extensas ou muito curtas e métodos demasiadamente longos[VEM02]. Essencialmente, um *Bad Smell* é uma dívida técnica no código orientado a objetos[P+15]. Como evidenciado na Figura 1.5 todos os *Bad Smells* são dívidas técnicas, mas nem toda dívida técnica é um *Bad Smell*.

Tendo sido superada essa fase de definição do termo, a comunidade agora procura por teo-

rias e técnicas comprovadamente eficazes para o gerenciamento e identificação da dívida técnica [FKNO14].

Sinônimos

A dívida técnica, por ser um fenômeno aparentemente onipresente nos projetos de software[LTS12, BCG+10], foi percebida e chamada de diversas formas pelos profissionais e pesquisadores. Existem ao menos duas razões para que esses sinônimos sejam devidamente documentados. A primeira delas está relacionada com a pesquisa bibliográfica a respeito do tema. Existem diversos trabalhos com resultados relevantes e que não usam diretamente o termo dívida técnica[Fow18, LK94, Lin12, SGHS11]. A segunda razão é a de permitir que seja traçado um retrospecto a respeito do assunto incluindo informações anteriores à definição do termo em 1992 por Cunningham[Cun93]. Poliakov realizou uma revisão sistemática a respeito dos sinônimos da dívida técnica[P+15]. Um dos resultados dessa revisão sistemática é um catálogo com os sinônimos para a dívida técnica encontrados na literatura. Na Tabela 1.1, apresentados um resumo desse catálogo.

Sinônimos
Shortcut
Code Smells / Design principles violation
Workaround / Hack
Grime
Software aging
Spaghetti code

Tabela 1.1: *Sinônimos do termo dívida técnica. Adaptado de [P+15]*

1.3 A metáfora

A metáfora dívida técnica surgiu inicialmente como uma forma de explicar a necessidade de evitar que código de má qualidade se espalhe pelo software a ponto de tornar sua evolução inviável[Cun93]. Uma das vantagens da utilização dessa metáfora é a sua capacidade de facilitar a justificativa para a disponibilização de recursos para a realização de atividades que não estejam diretamente ligadas à adição de novas funcionalidades ou correção de defeitos. A utilização de uma analogia com aspectos financeiros pode ser eficaz para explicar para pessoas sem conhecimento em desenvolvimento de sistemas a necessidade de empregar recursos para evitar o acúmulo de juros.

Ainda assim, apesar de ser apropriada, a metáfora dívida técnica tem diversas diferenças em relação a sua contrapartida no contexto financeiro e essas diferenças precisam ser consideradas durante seu gerenciamento. Uma delas é a impossibilidade de calcular previamente os juros a serem pagos. É difícil calcular com exatidão qual o esforço extra necessário para evoluir e manter o software devido a existência de uma dívida técnica. Apesar de algumas contribuições relevantes[SSK14, SG11, CSS12], até mesmo a criação de estimativas é um desafio devido à imprevisibilidade a respeito do contexto no qual o software será desenvolvido no futuro. Isso acontece principalmente porque, quase sempre, não é possível determinar se uma parte do código-fonte será ou não alterada. A quantidade de juros será proporcional à frequência de alterações relacionadas na parte do código com dívida técnica. Além disso, mesmo que existam dívidas técnicas nessa parte, não haverá incidência de juros caso não haja nenhuma alteração no futuro. Devido a essa incapacidade de prever se uma dívida técnica gerará ou não juros, alguns autores como Schmid, K[Sch13] diferenciam as dívidas técnicas como efetivas ou potenciais. Uma dívida potencial é aquela que está associada a uma expectativa de existência de juros. Ou seja, ela ainda não trouxe nenhum esforço adicional para o desenvolvimento do software. Enquanto isso, uma dívida técnica efetiva é aquela que já está gerando dificuldades adicionais nas tarefas de desenvolvimento e manutenção do software. Apesar das diferenças, a metáfora da dívida técnica é uma forma eficaz de evidenciar a necessidade de manter um equilíbrio entre a existência de recursos limitados e a preocupação de manter viável a evolução do software a médio e longo prazo.

1.3.1 Termos da metáfora

Assim como na área financeira, os principais conceitos relacionados a dívida técnica são o principal e os juros. Entretanto, além desses conceitos, existem outros. De acordo com Li. et al.[LAL15], existe uma lista de conceitos utilizados para descrever a dívida técnica e suas consequências. A seguir iremos descrever alguns desses conceitos.

Principal

O principal corresponde ao resultado gerado pelas atividades feitas de forma não ideal e que, consequentemente, não apresentam um nível de qualidade compatível com o projeto. No caso da dívida técnica no código, o principal será o trecho ou trechos do código que não estão de acordo com as boas práticas de desenvolvimento ou que não estão de acordo com critérios de qualidade adotados pela equipe de desenvolvimento. O valor do principal é equivalente ao esforço necessário para corrigir algum aspecto do software que não esteja adequado. Usando novamente o exemplo

da dívida técnica no código, o valor do principal é equivalente ao esforço necessário para alterar o código, de forma que ele fique de acordo com os padrões de qualidade necessários para o projeto.

Uma característica importante a respeito do valor do principal é que ele se altera com o passar do tempo. Isso acontece por diversas razões. Uma delas é a adição de novos artefatos ao software a medida que ele evolui. Com isso, nos casos em que esses artefatos também terão de ser alterados devido ao pagamento do principal, o esforço total necessário será maior. Outra razão para a variação temporal do esforço necessário para eliminar o principal é a possibilidade de que, devido ao tempo passado, a equipe já não esteja tão habituada com a parte do software onde a mudança precisa ser feita. As regras de negócios associadas com o código a ser alterado podem ter sido discutidas em um período de tempo muito anterior ao momento onde o pagamento do principal será feito. Além disso, é possível que até mesmo a tecnologia utilizada possa já não ser dominada pela equipe como era no momento em que o principal foi inserido no código.

Há uma semelhança entre a variação temporal do valor do principal e o conceito financeiro de correção monetária de uma dívida. De acordo com Schmidt[SSFM07], a correção monetária é um método de tornar real o valor monetário das contas permanentes das demonstrações contábeis. Essa correção é realizada por meio de algum índice como a inflação acumulada em um determinado período de tempo. Em ambos os casos, há uma tendência de aumento da dívida com o passar do tempo. Caso esse aumento não seja gerenciado, pode chegar a um cenário onde seu pagamento se torne inviável.

Juros

No contexto financeiro, os juros são os valores a serem pagos a um credor após a aquisição de um empréstimo. Dessa forma, podem ser definidos como o preço a ser pago para permanecer com uma determinada quantia. Enquanto essa quantia não for paga para o credor, os juros serão pagos na forma de uma porcentagem relativa ao valor ainda devido. Essa porcentagem não é igual para todos os credores. Assim como qualquer outro produto, o preço do empréstimo varia, ou seja, existem empréstimos com um preço maior ou menor que outros. Assim como existem dívidas técnicas que produzem mais ou menos juros.

No contexto da dívida técnica, os juros são todo o esforço adicional nas atividades de desenvolvimento de software causado pela existência da dívida técnica. Por exemplo, no caso da dívida técnica de arquitetura, os juros serão toda a dificuldade causada por uma característica da arquitetura do software que não esteja de acordo com os padrões de qualidade definidos para o software. Essa dificuldade pode estar relacionada com o tempo necessário para se adicionar um novo ele-

mento na arquitetura por exemplo. Enquanto o principal não for pago, isto é, enquanto o problema arquitetural não for resolvido, a equipe terá de lidar com as dificuldades causadas pelos juros. A Figura 1.6 ilustra essa definição dos juros da dívida técnica.

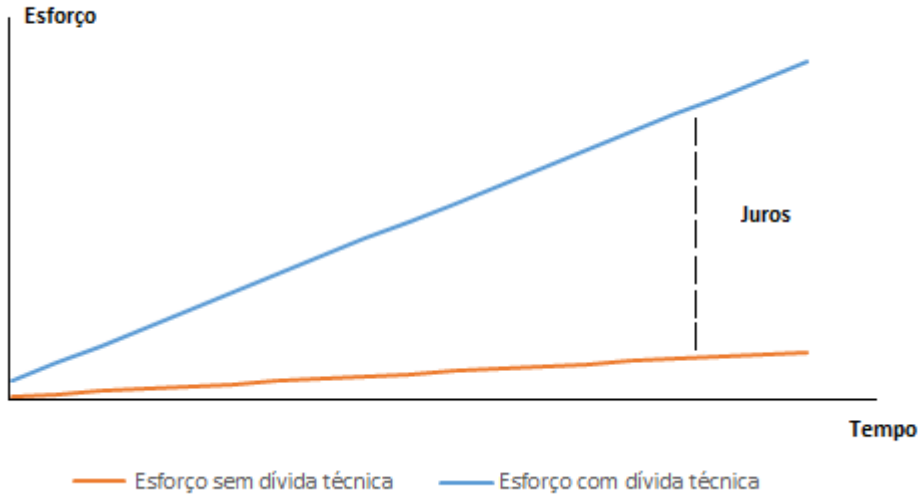

Figura 1.6: *Representação dos juros como o esforço adicional causado pela dívida técnica.*

.

Probabilidade dos juros

Muitas vezes existe uma incerteza em relação aos juros causados pela existência da dívida técnica. Essa incerteza está na ocorrência ou não desses juros como também em seu valor. Isso acontece, pois o esforço futuro necessário para desenvolver o software depende de fatores que não são conhecidos *a priori*. Até mesmo não é possível afirmar que uma determinada parte do software, relacionada a uma dívida técnica, terá de ser alterada no futuro.

Ponto de quebra

É chamado ponto de quebra o instante no tempo onde os juros da dívida técnica se acumula de tal forma que torna inviável a realização das atividades de evolução do software [CAAA15]. Em alguns casos, ao atingir esse ponto, a equipe de desenvolvimento cogita a hipótese de abandonar o projeto atual e recomeçar um novo do início. Esse processo é algumas vezes chamado de *like-to-like migration*. Apesar de parecer uma alternativa plausível, essa opção apresenta uma série de pontos fracos. Alguns deles são o custo elevado de reconstrução do software, a necessidade de considerar atualizações no conjunto de requisitos originais e a necessidade de utilizar tecnologias mais atuais e que consequentemente exigem treinamento e adaptação[Ste10]. Ou seja, o esforço necessário para

a criação de uma solução do tipo *like-to-like* normalmente é subestimada e irá custar mais do que o imaginado.

1.4 Classificações da dívida técnica

Uma forma recorrentemente encontrada na literatura de classificar a dívida técnica é como intencional e não intencional[Ste10, BCG+10, KTWW11]. A dívida técnica não intencional é aquela que as pessoas relacionadas ao desenvolvimento e evolução do software não sabem que a estão gerando. Esse desconhecimento pode ser devido à falta de experiência, conhecimento ou cuidado. Por outro lado, a dívida técnica intencional é aquela devidamente documentada e usada como uma ferramenta para alcançar um objetivo de curto prazo e que não seria possível caso a atividade fosse realizada de forma a atender o padrão de qualidade estabelecido. Além disso, existe um planejamento indicando como e quando essa dívida técnica será paga. Enquanto que a dívida técnica não intencional sempre é negativa e expõe deficiências nas atividades de desenvolvimento do software, a dívida técnica intencional pode ser uma aliada estratégica para aumentar a produtividade de uma equipe. O gerenciamento da dívida técnica não tem como objetivo eliminar completamente a existência da dívida técnica nos projetos de software. Ao invés disso, ela pode ser resumida como a busca pelo equilíbrio entre a aquisição e o pagamento da dívida de forma que ela se mantenha controlada e benéfica para o projeto.

Dentre as dívidas técnicas intencionais, podemos definir uma subcategoria chamada *self-admitted technical debt*(SATD). Essa é uma forma de dívida técnica onde o programador indica explicitamente nos comentários do programa que seu código contém dívida técnica. Essas dívidas são descritas no próprio código da aplicação por meio de comentários. É chamada de *self-admitted* porque o próprio programador admite que a solução apresentada não é adequada. Um dos trabalhos que abordam essa forma de dívida técnica[MS15] disponibilizou um banco de dados com 33k+ comentários com *self-admitted technical debt* em projetos de código livre.

As dívidas técnicas *self-admitted* ganharam certa atenção na literatura. Um exemplo disso é o trabalho de Maldonado et al. [MS15]. Nesse estudo os autores investigaram quais são os tipos de SATD e quantificam a ocorrências desses tipos em projetos de software livre. Identificar os tipos de SATD é importante pois, 1) ajuda a comunidade a entender suas limitações, 2) complementa as técnicas existentes de detecção de dívida técnica, 3) fornece uma melhor compreensão do ponto de vista dos desenvolvedores sobre a dívida técnica. Para a realização deste estudo foram analisados 5 projetos de código fonte aberto de diferentes domínios de aplicação. A extração dos comentários

do código dessas aplicações foi realizada em duas etapas. Na primeira, foram excluídos automaticamente, por meio de um programa, os comentários que tinham baixa chance de conter informações a respeito da dívida técnica. Como resultado, foram selecionados 33.093 comentários. Em seguida, os comentários restantes foram analisados manualmente por um experiente desenvolvedor de software. Após a análise manual, foram selecionados 2.457 comentários que estavam realmente relacionados com a dívida técnica nos 5 projetos analisados. Esses comentários então foram inseridos em um banco de dados disponibilizado publicamente pelos autores. Como resultado, foram identificadas 5 categorias de SATD: Design, defeitos, documentação, requisitos e testes. Os tipos design e requisitos são os mais presentes nos projetos analisados. Os SATD de design representam de 42% a 84% dos comentários enquanto que os SATD de requisitos representam de 5% a 45% dos comentários. Os autores indicam a utilização de técnicas de processamento de linguagem natural como um possível caminho para avançar na pesquisa sobre SATD.

1.5 Tipos de dívida técnicas

O termo dívida técnica foi inicialmente utilizado para descrever problemas relacionados ao código. Entretanto, posteriormente, foi percebido que ele pode ser utilizado para descrever problemas relacionados com outros aspectos do desenvolvimento de software. A seguir, iremos descrever alguns dos tipos de dívida técnica mais citados na literatura.

1.5.1 Código

A dívida técnica de código está relacionada aos problemas de organização e qualidade encontrados no código do software. Esses problemas são mais simples do que os apresentados na seção 1.5.2. Normalmente estão associados a não conformidade com o estilo de código definido ou reconhecidamente adequado pela comunidade que utiliza a linguagem de programação do projeto. Além disso, o impacto negativo causado pelas dívidas técnicas de código tem o escopo reduzido à classe, ao método ou ao bloco de código onde elas se encontram. Podemos citar, como exemplo de dívida técnica relacionada ao código, a existência de duplicação, complexidade desnecessária e a não aderência aos padrões de estilos definidos para o software. A ferramenta Sonar Qube [CP13] possui um plugin capaz de detectar uma grande quantidade de tipos diferentes de dívida técnica de código. Essas dívidas são organizadas em categorias como testabilidade, reusabilidade e segurança.

1.5.2 Design

Durante a história da programação orientada a objetivos, foram sendo observadas a existência de certas propriedades para que um código orientado a objetos possa ser entendido e alterado mais facilmente. Dentre essas diversas propriedades, podemos destacar a necessidade de baixo acoplamento entre os elementos do software e a necessidade de alta coesão. Uma dívida técnica de design é caracterizada pela ocorrência de código que viola esses princípios de padrões reconhecidos como corretos para o desenvolvimento de software orientado a objetos.

1.5.3 Testes

As dívidas técnicas de testes podem ocorrer em duas situações. A primeira delas é quando há uma quantidade insuficiente de testes. Isso faz com mudanças futuras no software possam se tornar mais difíceis devido à necessidade de realização de testes de regressão. A segunda situação que pode gerar dívida técnicas de testes é quando o código dos testes é escrito de forma inadequada. Segundo Wiklund et al.[WESL12], isso acontece porque as organizações geralmente negligenciam a qualidade do código, design e documentação quando produzem o software responsável pela automatização dos testes. Isso gera um acúmulo da dívida técnica causando problemas na utilização, extensão e manutenção desses sistemas. Ainda segundo Wiklund et al., existem quatro principais razões para a aquisição de dívidas técnicas no código dos testes automatizados. A primeira é a de que o reuso e o compartilhamento das ferramentas de automatização de testes são assuntos importantes e precisam ser considerados no gerenciamento da dívida técnica. A segunda observação está relacionada a infraestrutura do ambiente de automatização dos testes. Diferenças no ambiente de testes e no ambiente de produção podem causar resultados incorretos. A terceira observação se refere à excessiva generalidade das ferramentas de automação. A existência de muitas configurações induz o usuário a cometer erros na utilização dessas ferramentas. Por fim, as práticas de desenvolvimento de código para automatização de testes são menos rigorosas quando comparadas às utilizadas no código das outras partes do software. Esse fato naturalmente faz com que o acúmulo da dívida técnica nos sistemas de automação de testes seja maior.

1.5.4 Documentação

A dívida técnica de documentação é caracterizada pela inexistência de documentação, documentação desatualizada ou documentação inadequada. Esse tipo de dívida técnica pode trazer impactos negativos para o projeto de software nos casos onde funcionalidades precisem ser adicio-

nadas ou alteradas. O tempo necessário para realizar essas atividades pode ser sensivelmente maior em comparação com o cenário onde haja documentação adequada. Isso ocorre, pois, os responsáveis por realizar essas atividades precisarão reservar tempo para procurar em fontes não estruturadas as informações necessárias para concluí-las. Além disso, existe o risco que sejam assumidas soluções baseadas em documentação incorreta ou atualizada. Isso pode gerar atrasos e aumentar a quantidade de recursos utilizados.

1.5.5 Defeitos

Assim como existem na literatura divergências a respeito do que deve ser considerado como dívida técnica, existem divergências a respeito do que não deve ser considerado como dívida técnica. Alguns estudos explicitamente definem que defeitos devem ser considerados como dívida técnica [Dav13, GS11, XHJ12]. Esses autores restringem esses defeitos a aqueles que não apresentam grandes dificuldades para o usuário ou possuem algum caminho alternativo para contorná-los. Já alguns autores argumentam que devem ser considerados como dívida técnicas apenas elementos que não estejam visíveis para o usuário final[KNOF13].

1.5.6 Arquitetura

A dívida técnica arquitetural é uma violação no código do software em relação a alguma característica arquitetural pré-definida tal tais como modularidade, portabilidade e escalabilidade. Além disso, são consideradas dívidas técnicas de arquitetura as violações de restrições impostas pelos desenvolvedores de software tais como o isolamento entre módulos específicos e a proibição de acesso direto ao banco de dados. Um exemplo de dívida técnica arquitetural é a presença de dependências proibidas entre dois componentes. Martini et. al. [MBC14] apresentam uma taxonomia para as causas do acúmulo da dívida técnica arquitetural e um modelo para o processo de acúmulo e pagamento. Para criar essa taxonomia e esse modelo foi realizado um estudo de caso envolvendo cinco empresas de desenvolvimento de software. O estudo de caso foi realizado em duas fases. A primeira fase consistiu em um estudo preliminar envolvendo apenas três das cinco empresas. Nesse estudo, foram realizados workshops com diferentes membros dessas empresas para identificar os principais desafios no gerenciamento da dívida técnica arquitetural. A segunda fase envolveu, além de representantes das cinco empresas, a análise de documentos e a realização de entrevistas mais informais. Ao final, os resultados foram apresentados e discutidos com 15 representantes das cinco empresas. Os fatores que levam ao acúmulo da dívida técnica arquitetural foram divididos em oito categorias: negócios, falta de documentação, utilização de código open source ou legado, desen-

volvimento em paralelo, incerteza a respeito dos efeitos da refatoração, refatorações incompletas, evolução da tecnologia e fatores humanos. O estudo de caso mostrou que os modelos de acúmulo e pagamento da dívida técnica devem considerar que haverá um ponto no tempo onde o acúmulo excessivo da dívida gerará uma crise no desenvolvimento do software de tal forma que o pagamento da dívida não poderá ser mais postergado. Ao atingir esse ponto crítico, é necessário que a dívida técnica seja paga. As equipes podem realizar pagamentos parciais ou totais antes de atingir esse ponto crítico para que ele seja adiado ou totalmente evitado.

1.5.7 Construção

Algumas atividades comuns à tarefa de construção do software, tais como compilação de arquivos fonte e a análise e importação de dependências, atualmente são realizadas automaticamente por ferramentas de construção como Apache Ant[dOBdAFT15], Maven[GK07] e Gradle[Mus14]. Essas ferramentas são capazes de obter da internet as dependências necessárias, executar e analisar os resultados de testes unitários, além de poderem ser configuradas para emitir relatórios com detalhes sobre o processo de construção. Entretanto, é necessário que o software seja estruturado de forma a utilizar adequadamente as funcionalidades dessas ferramentas de automatização de construção. A dívida técnica de construção é caracterizada pela existência de características no software que não permitam a utilização das facilidades oferecidas por essas ferramentas. Logo, muitas das atividades de construção deverão ser realizadas manualmente, aumentando o esforço necessário para completá-las. A busca por soluções para este tipo de dívida técnica tem se tornado popular no âmbito profissional devido à inerente busca por automatização dos processos de desenvolvimento de software[MGSB12].

1.5.8 Tecnologia

Uma outra forma de dívida técnica está relacionada à evolução da tecnologia utilizada nos projetos. Com o passar do tempo, novas ferramentas e tecnologias são criadas para tornar o desenvolvimento de software mais eficaz e eficiente. Logo, as tecnologias utilizadas se tornarão obsoletas com o passar do tempo. Quando essas tecnologias não são atualizadas ou substituídas, o esforço necessário para o desenvolvimento é maior em relação ao cenário onde as tecnologias mais recentes são utilizadas. Isso é o que caracteriza a existência de uma dívida técnica.

1.6 Gerenciamento da dívida técnica

O gerenciamento da dívida técnica se assemelha ao gerenciamento de projetos. Existe uma série de atividades que precisam ser desempenhadas como identificação, análise, priorização e monitoramento. Apesar dessas atividades serem comuns ao gerenciamento da dívida técnica, existem diversas formas de desempenhá-las. Assim como no gerenciamento de projetos existem diversas técnicas diferentes que podem ser usadas em diversas atividades.

Caso o principal da dívida técnica não seja pago, os juros podem crescer a ponto de tornar a evolução do software insustentável. Logo, é importante que a dívida técnica seja devidamente gerenciada a fim de evitar que isso ocorra[Pow13]. O gerenciamento da dívida técnica é um conjunto de atividades realizadas com o intuito de controlá-la de forma que ela não comprometa o desenvolvimento e evolução do software. De acordo com Zengyang Li. et. al.[LAL15], ele pode ser dividido em três atividades principais: a prevenção, identificação e o balanceamento entre aquisição e o pagamento da dívida. A decisão de realizar uma atividade de forma a gerar uma dívida técnica normalmente é feita devido à limitação de recursos disponíveis para a realização da atividade. Os responsáveis pelo gerenciamento do desenvolvimento e evolução do software precisam avaliar se o impacto causado pela aquisição da dívida será menor do que o benefício causado pela realização da atividade.

As abordagens de gerenciamento da dívida técnica podem ser divididas em dois grupos. As abordagens que adaptam métodos financeiros e as abordagens específicas, especialmente criadas para o gerenciamento da dívida técnica. A seguir, descreveremos algumas das abordagens encontradas na literatura.

1.6.1 Abordagens adaptadas da área financeira

A dívida técnica nada mais é do que uma metáfora que compara deficiências nos projetos de software com conceitos financeiros. Conforme descrito na seção 1.3.1, muitos dos conceitos utilizados para descrever a dívida técnica são originários da área financeira. Por causa disso, algumas das abordagens para o gerenciamento da dívida financeira foram adaptadas para serem utilizadas para gerenciar a dívida técnica. Um exemplo dessas abordagens é o gerenciamento de portfólio. Em [GS11] Guo, et al., sugerem uma abordagem baseada em portfólios para gerenciar a dívida técnica. Um portfólio é o conjunto de investimentos que uma determinada empresa ou pessoa possui. Cada um desses investimentos possui um risco associado. Esse risco é uma medida para o quão provável é que o investimento não traga o retorno esperado. Quanto maior o risco, maiores as chances de

o investimento não trazer o retorno esperado. O objetivo do gerenciamento de portfólio é escolher os itens desse conjunto de forma que o retorno seja o maior possível e o risco esteja dentro de um patamar pré-estabelecido.

Apesar de a metáfora dívida técnica ter se mostrado útil como uma ferramenta de comunicação, existem muitas diferenças entre a dívida financeira e a dívida técnica que fazem com que a adaptação de abordagens de gerenciamento oriundas da área financeira seja de difícil realização. Em alguns casos, essa dificuldade é observada pelo fato de a área financeira utilizar métodos matemáticos complexos. Além disso, alguns conceitos financeiros, utilizados nessas abordagens, são de difícil mapeamento para o contexto de desenvolvimento de software. Um exemplo é o caso da técnica de precificação de opções definida por Black and Scholes [Chr96]. Essa técnica foi adaptada para ser utilizada em projetos de software [BK99, AB13, AR15]. Entretanto, mostrou-se de difícil utilização por pessoas sem um grande conhecimento na área financeira e em modelos matemáticos de análise. Mesmo tendo em vista a complexidade na utilização de algumas dessas abordagens, é possível que elas possam ser adaptadas com sucesso em projetos reais quando forem criadas ferramentas que automatizem seus cálculos e procedimentos.

1.6.2 Abordagens específicas

Além das abordagens de gerenciamento originárias da área financeira, foram criadas abordagens específicas para o gerenciamento da dívida técnica. Essas formas de gerenciamento foram criadas observando as necessidades específicas dos projetos de software em manter suas dívidas técnicas em níveis aceitáveis.

Fernández-Sánchez, C et al.[FSGY15] sugerem uma proposta para o gerenciamento da dívida técnica. Nesta pesquisa, os autores iniciam a definição de um arcabouço para o gerenciamento da dívida técnica. Por meio de um mapeamento sistemático da literatura, são definidos os elementos utilizados para o gerenciamento da dívida técnica e como esses elementos são considerados pelos diferentes pontos de vista dos stakeholders. Os elementos mapeados pelos estudos foram:

- *Identification of technical debt items.*
- *Principal Estimation.*
- *Interest estimation.*
- *Interest probability estimation.*
- *Technical debt impact estimation.*

- *Automated estimates.*
- *Expert Opinion.*
- *Scenario analysis.*
- *Time-to-market.*
- *When to implement decisions.*
- *Tracking technical debt over time.*
- *Visualizing technical debt.*

Os pontos de vista identificados foram engenharia, gerenciamento da engenharia e gerenciamento do negócio. A engenharia inclui processos de design e construção de software. O gerenciamento da engenharia envolve as atividades relacionadas ao planejamento e monitoramento. Por fim, as atividades de gerenciamento do negócio envolvem as estratégias, objetivos e planejamento organizacional. O mapeamento sistemático também mostrou que todos os pontos de vistas estão majoritariamente focados nas técnicas de estimação da dívida técnica. Os autores acreditam que ao identificar quais conceitos estão relacionados ao gerenciamento da dívida técnica e como esses conceitos são utilizados, poderão, em trabalhos futuros, criar modelos concretos que auxiliem o gerenciamento da dívida técnica.

A abordagem mais madura de gerenciamento da dívida técnica encontrada na literatura é a definida por Seaman, C e Guo, T[SG11]. A Figura 1.7 ilustra a estrutura básica dessa abordagem. A TD list (*Technical Debt List*) é um catálogo com as informações de todas as dívidas técnicas presentes no projeto. Esse catálogo inclui a data de identificação, descrição, localização, responsável, tipo, estimativa do principal, estimativa dos juros e estimativa da probabilidade dos juros ser efetivamente exercido. Todas as atividades de gerenciamento da dívida técnica atualizam ou obtêm informações desta lista. A abordagem criada pelos autores possui três atividades: Identificação, medição e monitoramento.

As atividades de identificação e medição geram uma lista que contém, além das dívidas técnicas, informações sobre o principal e os juros. Na atividade de identificação, o software e os processos utilizados em sua construção são analisados a fim de encontrar dívidas técnicas. Cada ocorrência de dívida técnica é, então, inserida na TD List. Na identificação não é realizada uma análise mais profunda a respeito do principal e juros das dívidas. Ao invés disso, são utilizadas apenas estimativas superficiais como simples, médio e difícil para designar o esforço necessário para correção. Na

atividade de medição, é realizado um estudo mais completo a respeito do custo para pagamento das dívidas e dos juros relacionados. Esse estudo é realizado considerando as informações atuais a respeito do software e as futuras atividades de desenvolvimento e manutenção. Essa divisão entre identificação e medição é realizada pelo fato de a medição ser uma atividade cara. Logo, não faz sentido que ela seja realizada em dívidas que não serão pagas no curto prazo ou não tenham alto impacto no software.

A atividade de monitoramento consiste em observar o ciclo de vida do software e incorporar ou remover da lista as dívidas a medida que elas forem sendo pagas ou novas dívidas forem sendo criadas. Além disso, no monitoramento é feito um acompanhamento da evolução da dívida técnica do software a fim de manter o nível dentro de patamares aceitáveis.

Além dessas atividades, os autores descrevem como deve ser o processo de decisão a respeito de quais dívidas devem ser pagas em uma release. Devem primeiro ser selecionadas as dívidas que estejam relacionadas ao componente ou componentes que serão alterados na release. Além disso, deve ser considerado o esforço, a probabilidade de que os juros tenham de ser pagos e os benefícios do pagamento da dívida. Esse framework não define explicitamente quais métodos ou ferramentas devem ser utilizados em cada atividade. Essa escolha é deixada para as pessoas que irão aplicá-lo. Apesar disso, os autores fornecem sugestões.

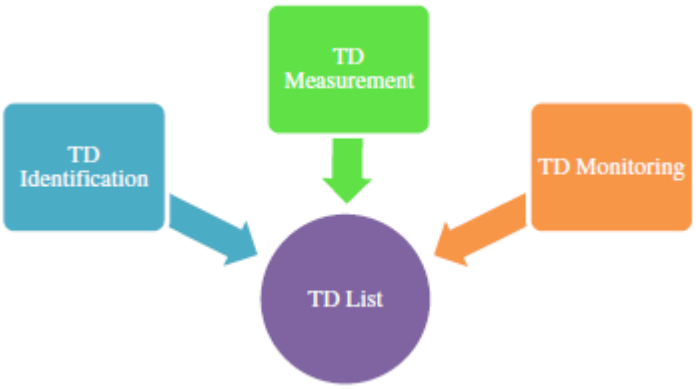

Figura 1.7: *Framework para gerenciamento da dívida técnica. Adaptado de [SG11].*

Para verificar a viabilidade de aplicação do framework, Seaman, C et. al., realizou um estudo onde são investigados os custos necessários para aplicar o arcabouço de gerenciamento [GSS16]. Nesse trabalho, os autores investigaram o custo necessário para realizar o gerenciamento da dívida técnica em um projeto de software e como as informações sobre a dívida técnica influenciam o processo de decisão. Para realizar essa investigação é conduzido um estudo de caso em uma pequena empresa de desenvolvimento de software. O objetivo desse estudo de caso é identificar quais são os custos necessários para gerenciar a dívida técnica e como as informações sobre a dívida técnica

contribuem para o processo de decisão. O estudo de caso foi desenvolvimento em uma empresa que produz software empresarial, realiza consultoria e fornece serviços de treinamento. Uma equipe formada por nove profissionais foi monitorada durante o desenvolvimento de um software web de gerenciamento de embarcações. Foram realizadas as atividades de gerenciamento da dívida técnica propostas pelos autores. O foco nesse monitoramento estava no processo de decisão e como ele era influenciado pelas informações da dívida técnica. Por meio do estudo de caso, foram possíveis identificar quatro itens ou categorias de custo para o gerenciamento da dívida técnica. Essas categorias foram: a. identificação, b. análise e avaliação, c. comunicação e d. documentação. Sendo que a análise e avaliação foi a categoria com o custo mais alto. O custo do planejamento do projeto aumentou 70% após a inclusão das atividades de gerenciamento da dívida técnica. Entretanto, este aumento significativo foi causado pelo custo de atividades iniciais que não serão repetidas em futuros projetos ou que terão o custo diminuído, como, por exemplo, as atividades de treinamento sobre a dívida técnica. O segundo objetivo do estudo foi identificar como as informações da dívida técnica influenciam o processo de decisão. O estudo de caso mostrou que as necessidades do cliente, a quantidade de recursos disponíveis, os juros da dívida técnica, o nível de qualidade do módulo e o impacto em outras funcionalidades são os fatores que influenciam o processo de decisão. Os autores concluem que o gerenciamento da dívida técnica na empresa do estudo de caso trouxe diversos benefícios. Isso foi evidenciado pelo fato de que o líder do projeto continuou utilizando a abordagem mesmo após o fim do estudo de caso. Além disso, o benefício causado pelo gerenciamento da dívida técnica se mostrou maior do que o custo de executá-lo.

Neste trabalho[OGS15] é realizada uma pesquisa-ação com o objetivo de avaliar em um cenário real a aplicação do framework de gerenciamento da dívida técnica proposto por Seaman, C e Guo, T[SG11]. A pesquisa-ação foi realizada em duas empresas brasileiras. A primeira desenvolve sistemas de gerenciamento de benefícios previdenciários. A segunda desenvolve sistemas de apoio para seguradoras. Ambas as empresas apresentaram indícios de existência de dívida técnica em seus projetos. A pesquisa-ação é caracterizada pela interação entre pesquisadores e profissionais com o objetivo de resolver um problema real e ainda assim contribuir para uma área de pesquisa. A pesquisa-ação executada neste estudo foi realizada em cinco estágios: diagnose, planejamento, intervenção, avaliação e registro de aprendizado. Foram realizados três seminários com cada uma das empresas. Nesses seminários foram apresentados os conceitos da dívida técnica e detalhes sobre framework que seria utilizado. Como fonte de dados para a obtenção dos resultados desta pesquisa, foram utilizadas as anotações realizadas durante os seminários e questionários enviados aos participantes ao final de cada ciclo. Foram observadas pelos pesquisadores e, confirmadas nos

questionários, dificuldades por parte dos profissionais em mensurar as dívidas técnicas. Especialmente os juros, pois dependem da previsão de como a dívida afetará o projeto com o passar do tempo. Essa previsão é de difícil realização especialmente quando não há dados históricos. Apesar disso, o estudo mostrou que os participantes acreditam que, ainda assim, essa avaliação, juntamente com a priorização da dívida técnica, precisa ser realizada por todo o time de desenvolvimento. A maior parte das dívidas identificadas foram de design e código. Isso se justifica pelo fato de grande parte dos participantes do estudo serem arquitetos de software ou programadores. Os participantes também concordaram que o tempo necessário para a inclusão de uma dívida técnica na *technical debt list* foi razoável. Os autores concluem que as empresas irão continuar utilizando a estratégia de gerenciamento da dívida técnica utilizada mesmo após a finalização da pesquisa.

1.6.3 Dívida técnica como uma ferramenta estratégica

Os estudos sobre gerenciamento da dívida técnica normalmente não consideram os aspectos estratégicos do projeto. Grande parte dos trabalhos encontrados na literatura consideram que a dívida já existe e precisa ser gerenciada. Existe uma ausência de trabalhos que abordem a possibilidade da dívida técnica ser utilizada como uma ferramenta estratégica no gerenciamento de projetos de software. Como exemplo, muitas vezes uma dívida técnica é criada devido à uma necessidade de disponibilizar rapidamente uma funcionalidade indispensável. Pode ser feita uma relação entre essa situação e a utilização de técnicas de prototipagem evolutiva e desenvolvimento incremental que também permitem a disponibilização do software de forma incompleta ou não aderente aos padrões de qualidades exigidos para o produto final. Ainda assim, são alternativas válidas para o projeto de desenvolvimento de software. Logo, o gerenciamento da dívida técnica deveria ser feito de forma que haja um balanceamento entre essas possíveis necessidades estratégicas e as necessidades técnicas do projeto. Entretanto, as abordagens encontradas não consideram esses aspectos estratégicos. Inclusive, não consideram a etapa de decisão a respeito da aquisição ou não da dívida técnica. Nas abordagens encontradas na literatura[SG11, GS11, FSGVY15], o gerenciamento da dívida técnica se inicia nas atividades de identificação. Ou seja, após as dívidas já terem sido criadas.

Capítulo 2

Método de pesquisa

Neste capítulo descreveremos os métodos de pesquisa que serão utilizados. Inicialmente, haverá uma breve introdução a respeito do modelo de estimação da dívida técnica proposto seguido por uma explicação a respeito de como iremos avaliá-lo usando um estudo de caso quantitativo.

2.1 Introdução

Nesta pesquisa proporemos um modelo para a estimação dos juros da dívida técnica em projetos de desenvolvimento de software. Nesse modelo, consideramos os juros como a variação negativa, na produtividade do projeto, causada pela existência da dívida técnica. Conforme ilustrado na Figura 2.1, esse modelo possui dois níveis de abstração. O primeiro é conceitual e baseia-se em uma definição abstrata tanto da produtividade de um projeto quanto do principal da dívida técnica. No segundo nível, já há uma definição das métricas que serão utilizadas para a estimação dos juros em projetos reais. O segundo nível de abstração é na verdade uma instância do modelo de primeiro nível. Futuramente podem ser definidas diversas instâncias do modelo de primeiro nível, cada uma terá de ser criada de acordo com as características dos projetos que serão avaliados. Por exemplo, em uma situação onde deseja-se estimar os juros da dívida técnica de um projeto web provavelmente serão utilizadas métricas de produtividade diferentes das métricas de um projeto de software. No Capítulo **??** forneceremos uma definição precisa a respeito do modelo de estimação dos juros e seus níveis de abstração.

Para avaliarmos a aplicabilidade tanto do modelo de primeiro nível quanto de segundo nível, realizaremos um estudo de caso quantitativo e exploratório utilizando dados de 1.870 projetos hospedados publicamente na plataforma GitHub. O objetivo dessa avaliação é verificar, por meio de métodos estatísticos, os resultados da aplicação do modelo nesses 1.870. Os seguintes itens serão

avaliados:

- Existência de uma correlação entre a quantidade de dívida técnica de um projeto e a sua produtividade. A existência dessa correlação é uma evidência de que seja viável a estimação dos juros da dívida técnica por meio de modelos baseados em métricas de produtividade.
- Existência de uma consistência no modelo específico criado para estimar a dívida técnica de projetos de software livre. Ou seja, vamos avaliar a aplicação de uma instância do modelo de estimação. Essa instância foi criada observando as particularidades dos projetos de software livre e também observando as limitação nos dados que temos disponíveis da plataforma GitHub.

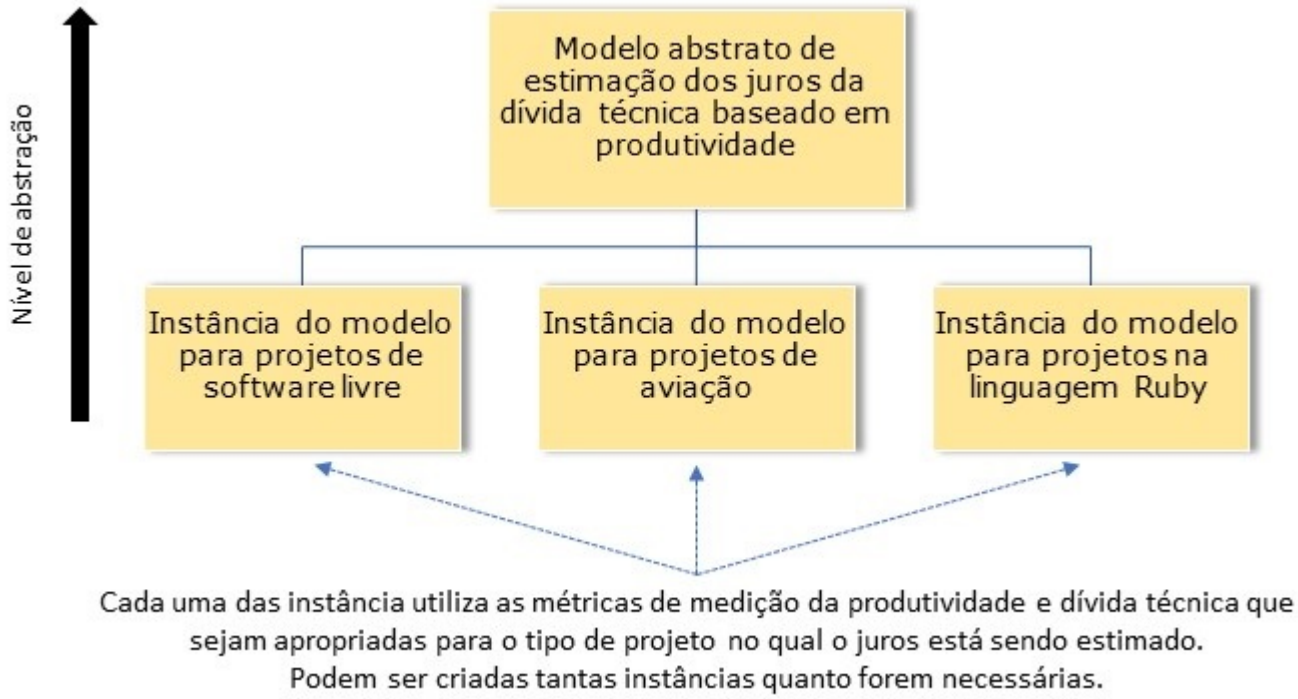

Figura 2.1: *Níveis de abstração do modelo es estimação dos juros da dívida técnica.*

2.2 Pesquisa quantitativa

De acordo com Creswell[WC16], uma pesquisa quantitativa tem como foco principal a quantificação de relacionamentos ou a comparação de um ou mais grupos. Adicionalmente, conforme explicado por Wohlin et al.[WHH03], as pesquisas quantitativas são apropriadas quando existe a necessidade de testar o efeito de alguma atividade ou manipulação. Segundo Wang at al. [WWAL13], a abordagem quantitativa disponibiliza uma série de ferramentas para descobrir, com um determi-

nado nível de confiança, a verdade a respeito de um objeto de estudo. O que difere substancialmente uma pesquisa quantitativa de outra qualitativa é seu grau de objetividade a respeito dos fenômenos avaliados. Na abordagem quantitativa há pouco ou nenhuma margem para que haja, durante o processo de coleta de dados, uma interpretação dos indivíduos relacionados com o evento analisado. Ao invés disso, são utilizados apenas fatos que não dependem de sensações, reflexões, intuições ou qualquer outra forma subjetiva de avaliação. Isso faz com que os dados numéricos sejam predominantes em pesquisas quantitativas. Essa característica permite que, utilizando poucos recursos, um grande volume de dados possa ser coletados e analisado. Amaratunga et al. [ABSN02] lista algumas das principais características de uma pesquisa quantitativa:

- Permitem a replicação e a comparação de resultados.
- Independência entre o observador e o objeto observado.
- A confiabilidade e validade dos resultados podem ser determinadas de forma mais objetiva.
- Enfatiza a necessidade de formular hipóteses para subsequentes verificações.

São muitas as atividades necessárias para a realização de uma pesquisa quantitativa. Essas atividades podem ser dívidas em duas fases. A primeira fase é constituídas por atividades de planejamento, obtenção e validação dos dados necessários para a avaliação do objeto da pesquisa. Esses dados podem ser obtidos de diversas formas. As mais comuns são a realização de questionário, experimentos, estudos de caso e, mais recentemente, a mineração de repositórios. A segunda fase consiste na avaliação desses dados utilizando métodos matemáticos, estatísticos ou computacionais.

Conforme argumentado por Brown, N et al.[BCG+10], há uma predominância na utilização de métodos qualitativos nas pesquisas a respeito da dívida técnica e isso pode levar a conclusões baseadas em intuições atraentes, porém não necessariamente corretas. Essas conclusões incorretas podem ser explicadas pela existência de dados obtidos por meio de declarações imprecisas. Essas declarações podem ser dadas pela dificuldade que as pessoas envolvidas com os projetos de software têm em assumir suas deficiências ou falhas. Por isso, Brown, N et al.[BCG+10] indica a necessidade da criação de modelos baseados em abordagens quantitativas para viabilizar a criação de rigorosas técnicas de gerenciamento da dívida técnica que possam ser aplicadas em projetos de larga escala. Neste trabalho iremos propor um modelo para estimação da dívida técnica. Para analisarmos a validade desse modelo, iremos realizar um estudo de caso quantitativo utilizando dados de projetos de software hospedados na plataforma GitHub.

2.2.1 Mineração de repositórios

Plataformas como GitHub, SourceForge e Bitbucket ganharam popularidade devido à evolução nas ferramentas de controle de versão e o reconhecimento, por parte da comunidade de software, das vantagens de utilizar ferramentas de colaboração. Além de ferramentas para armazenamento e organização do código, essas plataformas fornecem uma variedade de facilidades para a interação entre os colaboradores dos projetos. Com isso, essas ferramentas acumularam uma quantidade imensa de dados sobre os projetos hospedados e a forma como colaboradores interagem com esses projetos. Esses dados têm sido reconhecidos como altamente relevantes para as pesquisas quantitativas na área de engenharia de software. Foi chamado de mineração de repositórios de software [BL08] o conjunto de técnicas de investigação que utilizam informações provenientes de repositórios de software. Como exemplos de estudos que exploram essas técnicas, podemos citar aqueles envolvendo a predição de defeitos [Wan14], propagação de mudanças [WRS+15] e confiabilidade do software [dF15]. Neste trabalho, utilizaremos a mineração de repositórios de software para extrairmos os dados para o estudo de caso.

2.3 O estudo de caso

De acordo com Wohlin et al.[WHH03], um estudo de caso é um método de pesquisa onde são utilizados dados de situações reais. Diferentemente de um experimento, no estudo de caso o pesquisador tem menos ou nenhum controle sobre os acontecimentos. No contexto de projetos de software, um estudo de caso tem como objetivo monitorar as atividades realizadas durante o projeto. Segundo Yin, Robert K[Yin11], existem dois tipos de estudo de caso: os únicos e os múltiplos. Os estudos de casos únicos são aqueles onde os dados são obtidos de um único “caso”, que pode ser um projeto, uma empresa, um indivíduo ou qualquer outra unidade que seja apropriada para o estudo do objeto da pesquisa. Por outro lado, um estudo de caso múltiplo envolve diferente unidades de interesse. Ou seja, são consideras diversas empresas, projetos, indivíduos e etc. A realização de caso múltiplos é mais indicada já que os mesmos ela facilita generalização dos resultados obtidos por fornecerem múltiplas visões a respeito do objeto de pesquisa. Tendo isso em vista, para avaliarmos o modelo de estimação do comportamento dos juros da dívida técnica descrito no capítulo **??**, realizaremos um estudo de caso múltiplo envolvendo milhares de projetos armazenados em um repositório de software.

2.3.1 Etapas do estudo de caso

O estudo de caso será realizado em 5 etapas conforme resumido na Figura 2.2. Foi desenvolvida uma ferramenta para automatizar grande parte das atividades realizadas em cada etapa. Mais detalhe sobre a ferramenta de apoio serão fornecidos na seção **??**. A seguir forneceremos uma descrição panorâmica a respeito de cada uma das etapas do estudo de caso. Uma descrição detalhada de cada uma das etapas será fornecida no Capítulo [**?**].

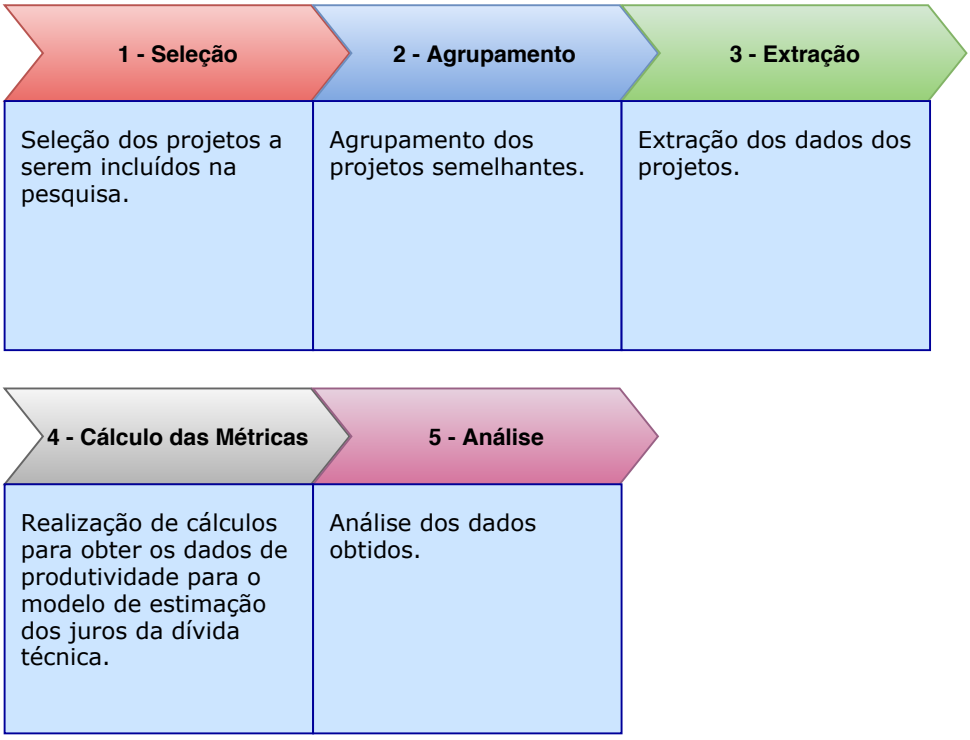

Figura 2.2: *Resumo das etapas do estudo de caso.*

Etapa 1 - Seleção dos projetos

Foram usados diversos critérios para selecionar os projetos incluídos no estudo de caso. Os primeiros deles estão relacionados com a necessidade de verificar se um repositório presente no GitHub realmente se trata de um projeto de software. Isso é necessário já que por disponibilizar a possibilidade de gratuitamente armazenar arquivos, muitas vezes o GitHub é utilizado para armazenar conteúdo que não é um projeto de software. É comum encontrar arquivos de sites, exercícios escolares, contratos e diversos outros itens que não têm relação com o objeto desta pesquisa. Na literatura, são encontradas pesquisas que fornecem heurísticas para identificar se um repositório no GitHub é um projeto de software ou não. Um exemplo é o trabalho de Kalliamvakou et. al.[KGB+14] onde são elencados diversos perigos encontrados na mineração de dados no GitHub. Outro trabalho relevante é o de Russel. M.[Rus13] onde o autor faz um análise abrangente a respeito

da mineração de dados no GitHub como também em diversas outras plataformas. A seleção dos projetos foi realizada utilizando uma combinação dessas heurísticas juntamente com outras regras apropriadas para este estudo. Algumas das regras utilizadas foram:

- Projetos realizados na linguagem Java. Essa regra foi definida por dois motivos. O primeiro motivo é a diferença que existe entre as dívidas técnicas de uma linguagem e outras. Por exemplo, existem dívidas específicas para linguagens orientadas a objetos que não são possível de serem encontradas em linguagens procedurais. O segundo motivo é técnico e está relacionado com as limitações da ferramenta utilizada para a extração de métricas. Ela possui uma maior compatibilidade com a linguagem Java.

- Projetos no qual a documentação estive escrita em inglês. O motivo dessa restrição foi a necessidade de separar os projetos por domínio de aplicação. Essa separação foi feita aplicando técnicas de machine learning na documentação dos projetos. A inclusão de múltiplas linguagens iria trazer uma complexidade substancial a esse processo. Além disso, a técnica de classificação utilizada não era compatível com textos em múltiplas linguagens.

Etapa 2 - Agrupamentos dos projetos semelhantes

De acordo com Kitchenham et. al.[KM04], a comparação de produtividade entre projetos de software deve ser feita considerando o domínio de cada um deles. Ou seja, não faz sentido comparar a produtividade de um projeto da área da aviação, que possui padrões extremamente rígidos de qualidade, com um projeto de uma aplicação para a internet. Por isso, os projetos utilizados no estudo de caso foram divididos em domínios de aplicação como sistemas gerenciadores de banco de dados, jogos, frameworks, linguagens de programação e etc.

Inicialmente foram consideradas algumas estratégias para estimação do domíno do projeto. Uma delas foi a proposta de Idri. et. al.[IA01]. Nela, os autores utilizam um modelo baseado em lógica Fuzzy para estimar o domínio de um projeto de software. Além disso, foram estudas outras abordagens baseadas no código fonte da aplicação. Entre elas estão o trabalho de Yamamoto et. al.[YMKI05] e a ferramenta MudaBlue, proposta por Kawaguchi et. al.[KGMI06]. Essas abordagens não foram utilizadas já que dependiam de informações que não tínhamos acesso ou da construção do projeto. Como utilizamos uma abordagem automática, muitas vezes não era possível compilar os projetos devido a algum erro no código, incompatibilidade com o ambiente ou falta de alguma dependência.

Para estimar o domínio de cada projeto, utilizamos uma estratégia baseada em *Latent Dirichlet allocation* (LDA)[BNJ03, HBB10, BNJ02]. O LDA é uma técnica de aprendizado de máquina que basicamente consegue classificar documentos em assuntos. Essa técnica foi aplicada na documentação dos projetos a fim de agrupá-los de acordo com o domínio estimado e comparar a produtividade apenas entre projetos de um mesmo domínio.

Etapa 3 - Extração dos dados

Foi utilizada a ferramenta SonarQube[CP13] para realizar a extração das métricas dos projetos. Os dados obtidos podem ser divididos dois grupos de métricas: um geral com informações diversas a respeito do projeto e o outro com as informações a respeito da dívida técnica. Todas as métricas foram obtidas observando a evolução temporal dos projetos. Isso foi feito ordenando as atualizações nos códigos fonte de forma sequencial e as dividindo em 5 pontos. Então as métricas foram obtidas para cada um desses pontos. Isso foi feito para viabilizar a análise temporal da evolução da dívida técnica do projeto e melhorar as estimativas. Essa estratégia foi necessária porque a dívida técnica de um projeto pode variar muito com o tempo. Um projeto pode começar com muita dívida e depois realizar fatorações para diminuí-la substancialmente.

Etapa 4 - Cálculo das métricas de produtividade

Nessa etapa os dados obtidos dos projetos foram usados para calcular as métricas de produtividade do projeto. O cálculo de alguns dos componentes desses métricas de produtividade foi feito utilizando uma versão adaptada do algoritmo PageRank[PBMW99]. Nesta versão a qualidade das contribuições dos colaboradores foi medida usando, entre outros fatores, dados a respeito da popularidade do colaborador. Isso envolveu medir a reputação de cada colaborador que contribuiu com o projeto. Como são milhares de projetos e colaboradores, houve uma alta complexidade em criar algoritmos que pudessem realizar esses cálculos em um tempo viável. Por isso, esse cálculo das métricas de produtividade foi separado em uma etapa exclusiva ao invés de considerado como um passo auxiliar da extração de dados.

Etapa 5 -Análise dos resultados

Nesta etapa, iremos realizar uma análise estatística dos dados obtidos nas etapas anteriores. Para tal, utilizaremos técnicas da estatística inferencial e métodos da inteligência artificial. Os objetivos nessa etapa é avaliar a aplicabilidade do modelo de estimação dos juros da dívida técnica em projetos reais.

Referências Bibliográficas

[AB13] Esra Alzaghoul e Rami Bahsoon. Cloudmtd: Using real options to manage technical debt in cloud-based service selection. Em *Managing Technical Debt (MTD), 2013 4th International Workshop on*, páginas 55–62. IEEE, 2013. 20

[ABSN02] Dilanthi Amaratunga, David Baldry, Marjan Sarshar e Rita Newton. Quantitative and qualitative research in the built environment: application of mixed research approach. *Work study*, 51(1):17–31, 2002. 27

[Ach14] Sujoy Acharya. *Mastering Unit Testing Using Mockito and JUnit*. Packt Publishing Ltd, 2014. 6

[AQ10] Rafa E Al-Qutaish. Quality models in software engineering literature: an analytical and comparative study. *Journal of American Science*, 6(3):166–175, 2010. 8

[AR15] Zahra Shakeri Hossein Abad e Guenther Ruhe. Using real options to manage technical debt in requirements engineering. Em *2015 IEEE 23rd International Requirements Engineering Conference (RE)*, páginas 230–235. IEEE, 2015. 20

[BCG+10] Nanette Brown, Yuanfang Cai, Yuepu Guo, Rick Kazman, Miryung Kim, Philippe Kruchten, Erin Lim, Alan MacCormack, Robert Nord, Ipek Ozkaya et al. Managing technical debt in software-reliant systems. Em *Proceedings of the FSE/SDP workshop on Future of software engineering research*, páginas 47–52. ACM, 2010. 10, 14, 27

[BK99] Michel Benaroch e Robert J Kauffman. A case for using real options pricing analysis to evaluate information technology project investments. *Information Systems Research*, 10(1):70–86, 1999. 20

[BL08] Jie BAI e Chun-ping LI. Mining software repository: a survey. *Application Research of Computers*, 1:006, 2008. 28

[BNJ02] David M Blei, Andrew Y Ng e Michael I Jordan. Latent dirichlet allocation. Em *Advances in neural information processing systems*, páginas 601–608, 2002. 30

[BNJ03] David M Blei, Andrew Y Ng e Michael I Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003. 30

[CAAA15] Alexander Chatzigeorgiou, Apostolos Ampatzoglou, Areti Ampatzoglou e Theodoros Amanatidis. Estimating the breaking point for technical debt. Em *Managing Technical Debt (MTD), 2015 IEEE 7th International Workshop on*, páginas 53–56. IEEE, 2015. 13

[Chr96] Neil Chriss. *Black Scholes and Beyond: Option Pricing Models*. McGraw-Hill, 1996. 20

[CL02] Yoonsik Cheon e Gary T Leavens. A simple and practical approach to unit testing: The jml and junit way. Em *European Conference on Object-Oriented Programming*, páginas 231–255. Springer, 2002. 4

[CP13] G Campbell e Patroklos P Papapetrou. *SonarQube in Action*. Manning Publications Co., 2013. 15, 31

[CSS12] Bill Curtis, Jay Sappidi e Alexandra Szynkarski. Estimating the size, cost, and types of technical debt. Em *Proceedings of the Third International Workshop on Managing Technical Debt*, páginas 49–53. IEEE Press, 2012. 11

[Cun93] Ward Cunningham. The wycash portfolio management system. *ACM SIGPLAN OOPS Messenger*, 4(2):29–30, 1993. 6, 10

[Dav13] Noopur Davis. Driving quality improvement and reducing technical debt with the definition of done. Em *2013 Agile Conference (AGILE)*, páginas 164–168. IEEE, 2013. 17

[dF15] Paulo André Faria de Freitas. Software repository mining analytics to estimate software component reliability. 2015. 28

[dOBdAFT15] Márcio de Oliveira Barros, Fábio de Almeida Farzat e Guilherme Horta Travassos. Learning from optimization: A case study with apache ant. *Information and Software Technology*, 57:684–704, 2015. 18

[EN16] Ramez Elmasri e Sham Navathe. *Fundamentals of database systems*. Pearson London, 2016. 4

[FKNO14] Davide Falessi, Philippe Kruchten, Robert L Nord e Ipek Ozkaya. Technical debt at the crossroads of research and practice: report on the fifth international workshop on managing technical debt. *ACM SIGSOFT Software Engineering Notes*, 39(2):31–33, 2014. 10

[Fow09] Martin Fowler. *Refactoring: improving the design of existing code*. Pearson Education India, 2009. 9

[Fow18] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018. 10

[FSGVY15] Carlos Fernández-Sánchez, Juan Garbajosa, Carlos Vidal e Agustin Yague. An analysis of techniques and methods for technical debt management: a reflection from the architecture perspective. Em *Software Architecture and Metrics (SAM), 2015 IEEE/ACM 2nd International Workshop on*, páginas 22–28. IEEE, 2015. 24

[FSGY15] Carlos Fernández-Sánchez, Juan Garbajosa e Agustin Yague. A framework to aid in decision making for technical debt management. Em *Managing Technical Debt (MTD), 2015 IEEE 7th International Workshop on*, páginas 69–76. IEEE, 2015. 20

[GK07] Max Goldman e Shmuel Katz. Maven: Modular aspect verification. Em *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, páginas 308–322. Springer, 2007. 18

[GS11] Yuepu Guo e Carolyn Seaman. A portfolio approach to technical debt management. Em *Proceedings of the 2nd Workshop on Managing Technical Debt*, páginas 31–34. ACM, 2011. 17, 19, 24

[GSS16] Yuepu Guo, Rodrigo Oliveira Spínola e Carolyn Seaman. Exploring the costs of technical debt management–a case study. *Empirical Software Engineering*, 21(1):159–182, 2016. 22

[HBB10] Matthew Hoffman, Francis R Bach e David M Blei. Online learning for latent dirichlet allocation. Em *advances in neural information processing systems*, páginas 856–864, 2010. 30

[IA01] Ali Idri e Alain Abran. A fuzzy logic based set of measures for software project similarity: validation and possible improvements. Em *Software Metrics Symposium, 2001. METRICS 2001. Proceedings. Seventh International*, páginas 85–96. IEEE, 2001. 30

[Kac12] Tomek Kaczanowski. *Practical Unit Testing with TestNG and Mockito*. Tomasz Kaczanowski, 2012. 6

[KGB+14] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German e Daniela Damian. The promises and perils of mining github. Em *Proceedings of the 11th working conference on mining software repositories*, páginas 92–101. ACM, 2014. 29

[KGMI06] Shinji Kawaguchi, Pankaj K Garg, Makoto Matsushita e Katsuro Inoue. Mudablue: An automatic categorization system for open source repositories. *Journal of Systems and Software*, 79(7):939–953, 2006. 30

[KM04] Barbara Kitchenham e Emilia Mendes. Software productivity measurement using multiple size measures. *IEEE Transactions on Software Engineering*, 30(12):1023–1035, 2004. 30

[KNOF13] Philippe Kruchten, Robert L Nord, Ipek Ozkaya e Davide Falessi. Technical debt: towards a crisper definition report on the 4th international workshop on managing technical debt. *ACM SIGSOFT Software Engineering Notes*, 38(5):51–54, 2013. ix, 8, 9, 17

[KTWW11] Tim Klinger, Peri Tarr, Patrick Wagstrom e Clay Williams. An enterprise perspective on technical debt. Em *Proceedings of the 2nd Workshop on managing technical debt*, páginas 35–38. ACM, 2011. 14

[LAL15] Zengyang Li, Paris Avgeriou e Peng Liang. A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, 101:193–220, 2015. 11, 19

[Leh80] Meir M Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980. 7

[Leh96] Manny M Lehman. Laws of software evolution revisited. Em *European Workshop on Software Process Technology*, páginas 108–124. Springer, 1996. 7

[Lin12] Markus Lindgren. Bridging the software quality gap, 2012. 10

[LK94] David L Lanning e Taghi M Khoshgoftaar. Modeling the relationship between source code complexity and maintenance difficulty. *Computer*, (9):35–40, 1994. 10

[LTS12] Erin Lim, Nitin Taksande e Carolyn Seaman. A balancing act: What software practitioners have to say about technical debt. *IEEE software*, 29(6):22–27, 2012. 10

[MBC14] Antonio Martini, Jan Bosch e Michel Chaudron. Architecture technical debt: Understanding causes and a qualitative model. Em *Software Engineering and Advanced Applications (SEAA), 2014 40th EUROMICRO Conference on*, páginas 85–92. IEEE, 2014. 17

[MFC00] Tim Mackinnon, Steve Freeman e Philip Craig. Endo-testing: unit testing with mock objects. *Extreme programming examined*, páginas 287–301, 2000. 6

[MGSB12] J David Morgenthaler, Misha Gridnev, Raluca Sauciuc e Sanjay Bhansali. Searching for build debt: Experiences managing technical debt at google. Em *Proceedings of the Third International Workshop on Managing Technical Debt*, páginas 1–6. IEEE Press, 2012. 18

[MJ06] Kane Mar e Michael James. Technical debt and design death, 2006. 7

[MKAH14] Shane McIntosh, Yasutaka Kamei, Bram Adams e Ahmed E Hassan. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. Em *Proceedings of the 11th Working Conference on Mining Software Repositories*, páginas 192–201. ACM, 2014. 5

[MS15] Everton da S Maldonado e Emad Shihab. Detecting and quantifying different types of self-admitted technical debt. Em *Managing Technical Debt (MTD), 2015 IEEE 7th International Workshop on*, páginas 9–15. IEEE, 2015. 14

[MSM+16] Rodrigo Morales, Aminata Sabane, Pooya Musavi, Foutse Khomh, Francisco Chicano e Giuliano Antoniol. Finding the best compromise between design quality and testing effort during refactoring. Em *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, páginas 24–35. IEEE, 2016. 5

[Mus14] Benjamin Muschko. *Gradle in action*. Manning, 2014. 18

[OCBZ09] Steffen Olbrich, Daniela S Cruzes, Victor Basili e Nico Zazworka. The evolution and impact of code smells: A case study of two open source systems. Em *Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on*, páginas 390–400. IEEE, 2009. 9

[OGS15] Frederico Oliveira, Alfredo Goldman e Viviane Santos. Managing technical debt in software projects using scrum: An action research. Em *Agile Conference (AGILE), 2015*, páginas 50–59. IEEE, 2015. 23

[P+15] Dmitrii Poliakov et al. A systematic mapping study on technical debt definition. 2015. xi, 9, 10

[PBMW99] Lawrence Page, Sergey Brin, Rajeev Motwani e Terry Winograd. The pagerank citation ranking: Bringing order to the web. Relatório técnico, Stanford InfoLab, 1999. 31

[Pow13] Ken Power. Understanding the impact of technical debt on the capacity and velocity of teams and organizations: viewing team and organization capacity as a portfolio of real options. Em *Managing Technical Debt (MTD), 2013 4th International Workshop on*, páginas 28–31. IEEE, 2013. 19

[Run06] Per Runeson. A survey of unit testing practices. *IEEE software*, 23(4):22–29, 2006. 4

[Rus13] Matthew A Russell. *Mining the Social Web: Data Mining Facebook, Twitter, LinkedIn, Google+, GitHub, and More*. "O'Reilly Media, Inc.", 2013. 29

[Sch13] Klaus Schmid. On the limits of the technical debt metaphor: Some guidance on going beyond. Em *Proceedings of the 4th International Workshop on Managing Technical Debt*, páginas 63–66. IEEE Press, 2013. 11

[SG11] Carolyn Seaman e Yuepu Guo. Measuring and monitoring technical debt. *Advances in Computers*, 82:25–46, 2011. ix, 11, 21, 22, 23, 24

[SGHS11] Michael Smit, Barry Gergel, H James Hoover e Eleni Stroulia. Code convention adherence in evolving software. Em *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, páginas 504–507. IEEE, 2011. 10

[SSFM07] José Luiz dos SANTOS, Paulo SCHMIDT, Luciane Alves FERNANDES e Nilson Perinazzo MACHADO. Teoria da contabilidade: introdutória, intermediária e avançada. *São Paulo: Atlas*, 2007. 12

[SSK14] Vallary Singh, Will Snipes e Nicholas A Kraft. A framework for estimating interest on technical debt by monitoring developer activity related to code comprehension. Em *Managing Technical Debt (MTD), 2014 Sixth International Workshop on*, páginas 27–30. IEEE, 2014. 11

[Ste10] Chris Sterling. *Managing software debt: building for inevitable change*. Addison-Wesley Professional, 2010. 13, 14

[VEM02] Eva Van Emden e Leon Moonen. Java quality assurance by detecting code smells. Em *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, páginas 97–106. IEEE, 2002. 9

[Wan14] Hui Wang. Software defects classification prediction based on mining software repository. 2014. 28

[WC16] John W Creswell. *Research Design.: Qualitative, Quantitative, Mixed Methods Approaches*. University Of Nebraska-Lincoln, 2016. 26

[WESL12] Kristian Wiklund, Sigrid Eldh, Daniel Sundmark e Kristina Lundqvist. Technical debt in test automation. Em *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, páginas 887–892. IEEE, 2012. 16

[WHH03] Claes Wohlin, Martin Höst e Kennet Henningsson. Empirical research methods in software engineering. Em *Empirical methods and studies in software engineering*, páginas 7–23. Springer, 2003. 26, 28

[WRS+15] Igor Scaliante Wiese, Reginaldo Ré, Igor Steinmacher, Rodrigo Takashi Kuroda, Gustavo Ansaldi Oliva e Marco Aurélio Gerosa. Predicting change propagation from repository information. Em *Software Engineering (SBES), 2015 29th Brazilian Symposium on*, páginas 100–109. IEEE, 2015. 28

[WWAL13] Lihshing Leigh Wang, Amber S Watts, Rawni A Anderson e Todd D Little. 31 common fallacies in quantitative research methodology. *The Oxford handbook of quantitative methods*, página 718, 2013. 26

[XHJ12] Jifeng Xuan, Yan Hu e He Jiang. Debt-prone bugs: technical debt in software maintenance. *International Journal of Advancements in Computing Technology 2012a*, 4(19):453–461, 2012. 17

[Yin11] Robert K Yin. *Applications of case study research*. Sage, 2011. 28

[YMKI05] Tetsuo Yamamoto, Makoto Matsushita, Toshihiro Kamiya e Katsuro Inoue. Measuring similarity of large software systems based on source code correspondence. Em *International Conference on Product Focused Software Process Improvement*, páginas 530–544. Springer, 2005. 30