

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 insufficient 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 difficulties 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, its payment does not need 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: technical debt, software repository, software estimation.

Sumário

Lista de Abreviaturas	ix
Lista de Figuras	xi
Lista de Tabelas	xiii
1 Dívida Técnica	1
1.1 Introdução	1
1.2 Definição da dívida técnica	6
1.3 A metáfora	10
1.3.1 Termos da metáfora	11
1.4 Classificações da dívida técnica	14
1.5 Tipos de dívida técnicas	15
1.5.1 Código	15
1.5.2 Design	16
1.5.3 Testes	16
1.5.4 Documentação	16
1.5.5 Defeitos	17
1.5.6 Arquitetura	17
1.5.7 Construção	18
1.5.8 Tecnologia	18
1.6 Gerenciamento da dívida técnica	19
1.6.1 Abordagens adaptadas da área financeira	19
1.6.2 Abordagens específicas	20
1.6.3 Dívida técnica como uma ferramenta estratégica	24
2 Método de pesquisa	25
2.1 Introdução	25
2.2 Pesquisa quantitativa	26
2.2.1 Mineração de repositórios	28
2.3 O estudo de caso	28
2.3.1 Etapas do estudo de caso	29
3 O modelo de estimação do juros da dívida técnica	33
3.1 Introdução	33
3.2 Estimação da produtividade em projetos de software	34

3.3	O modelo de estimação	36
3.4	O modelo abstrato de estimação dos juros da dívida técnica	37
3.5	Os modelos concretos de estimação dos juros da dívida técnica	39
3.5.1	Seleção das métricas que representam as entradas do processo	40
3.5.2	Seleção das métricas que representam as saídas do processo	42
3.5.3	Definição do método de agrupamento dos projetos semelhantes	43
3.5.4	Definição do método de particionamento dos grupos de projetos	45
3.6	O modelo de estimação dos juros em projetos de software livre	46
3.7	Entradas	46
3.7.1	Qualidade da colaboração	47
3.7.2	Assiduidade dos colaboradores	53
3.7.3	Índice de colaboração	53
3.8	Saídas	53
3.8.1	Linhas de código	54
3.8.2	<i>Pull Requests</i>	55
3.8.3	Popularidade	56
3.9	Método de agrupamento dos projetos semelhantes	57
3.10	Método de particionamento dos grupos de projetos	57
4	Estudo de caso	59
4.1	Introdução	59
4.2	O repositório de software	59
4.2.1	Controle de versões utilizando o Git	60
4.2.2	GitHub	63
4.2.3	GHTorrent	65
4.3	Ferramenta de apoio: GitResearch	66
4.3.1	Arquitetura	67
4.4	Etapas do estudo de caso	70
4.4.1	Seleção dos projetos	72
4.5	Agrupamento dos projetos	76
4.5.1	LDA	76
4.5.2	A aplicação do LDA	77
4.5.3	Resultado do LDA	78
4.6	Extração dos dados	82
4.6.1	O SonarQube	82
4.6.2	Medição da dívida técnica	85
4.6.3	Limitações na medição da dívida técnica	88
4.6.4	Divisão temporal do código	89
4.7	Cálculo das variáveis de entrada do modelo	91
4.7.1	Qualidade de colaboração	92
4.7.2	Assiduidade	94
4.8	Cálculo das variáveis de saída do modelo	94
4.9	Dados obtidos	94
4.9.1	Análise exploratória dos dados	94

4.10 Conclusão	94
5 Resultados e discussão	95
5.1 Contribuições	95
5.2 Respostas as questões de pesquisa	95
6 Ameaças à validade	97
6.1 Escala	98
6.2 Avaliação empírica das métricas utilizadas	98
6.3 Coisas a serem feitas caso haja tempo disponível	98
6.4 Imprecisão da popularidade	98
7 Conclusão	101
7.1 Trabalhos futuros	101
7.1.1 Popularidade vs Dívida Técnica	101
7.1.2 Artigo CBI	101
Referências Bibliográficas	103

Lista de Abreviaturas

TD Dívida Técnica (*Technical Debt*)

Lista de Figuras

1.1	Exemplo de dívida técnica em código fonte.	3
1.2	Exemplo de dívida técnica em banco de dados.	5
1.3	Exemplo de refatoração para viabilizar a criação de um teste unitário.	6
1.4	Panorama da dívida técnica. Adaptado de [KNOF13].	8
1.5	Localização da dívida técnica na hierarquia dos problemas de qualidade de software.	9
1.6	Representação dos juros como o esforço adicional causado pela dívida técnica.	13
1.7	Framework para gerenciamento da dívida técnica. Adaptado de [SG11].	22
2.1	Níveis de abstração do modelo de estimação dos juros da dívida técnica.	26
2.2	Resumo das etapas do estudo de caso.	29
3.1	Níveis de abstração do modelo de estimação dos juros da dívida técnica.	37
3.2	As relações de seguir e poder ser seguido no GitHub. Adaptado de [MSHZ15].	50
4.1	Exemplo da estrutura de armazenamento de versões do Git.	61
4.2	Exemplo de merge entre duas <i>branches</i>	62
4.3	Tabelas do GHTorrent.	66
4.4	Arquitetura do framework Spring Batch. Adaptada de [Min11].	67
4.5	Código responsável por configurar o <i>Job</i> no GitResearch.	69
4.6	Passos do processamento no GitResearch.	70
4.7	Resumo das etapas do estudo de caso.	71
4.8	Mapeamento entre as etapas do estudo de caso e as etapas de processamento no GitResearch.	72
4.9	Código responsável por criar a configuração para a execução do <i>scanner</i> do SonarQube.	83
4.10	Código responsável por contar a quantidade de <i>commits</i> de cada projeto.	90
4.11	Código responsável por executar o <i>scanner</i> do SonarQube em cada projeto.	91
4.12	Código responsável por calcular o pagerank.	93

Lista de Tabelas

1.1	Sinónimos do termo dívida técnica. Adaptado de [P ⁺ 15]	10
3.1	Atividades necessárias para a criação de um modelo concreto de estimação dos juros da dívida técnica específico.	40
4.1	Comandos básicos do GIT.	63
4.2	Regras para a exclusão de projetos do estudo de caso	76
4.3	Transformações realizadas no texto das descrições dos projetos	78
4.4	Tópicos e suas palavras.	81
4.5	Métricas extraídas dos projetos.	85
4.6	Exemplo de cálculo da dívida técnica com o SonarQube. A variável m representa o tempo necessário para escrever uma linha de código. O valor padrão no SonarQube é de 30 minutos. Adaptado de [dJdM17].	87
4.7	Exemplos de regras para identificação de dívidas técnicas no SonarQube.	88

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 principal 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		Curso	
ALUNO_ID	Nome	CURSO_ID	Curso
101	João	1	Matemática
102	Henrique	2	História
103	Matheus	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 funcionamento 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.

V1

```
function calculaImposto(funcionario,salario,aliquota)
{
  /* Acessa a variável global com a conexão do banco de dados */
  global bancoDeDados;
  imposto = salario * aliquota;
  funcionario.salario = salario - imposto;
  bancoDeDados.atualizar(funcionario);
}
```

V2

```
function calculaImposto(funcionario,salario,aliquota,bancoDeDados)
{
  imposto = salario * aliquota;
  funcionario.salario = salario - imposto;
  bancoDeDados.atualizar(funcionario);
}
```

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.

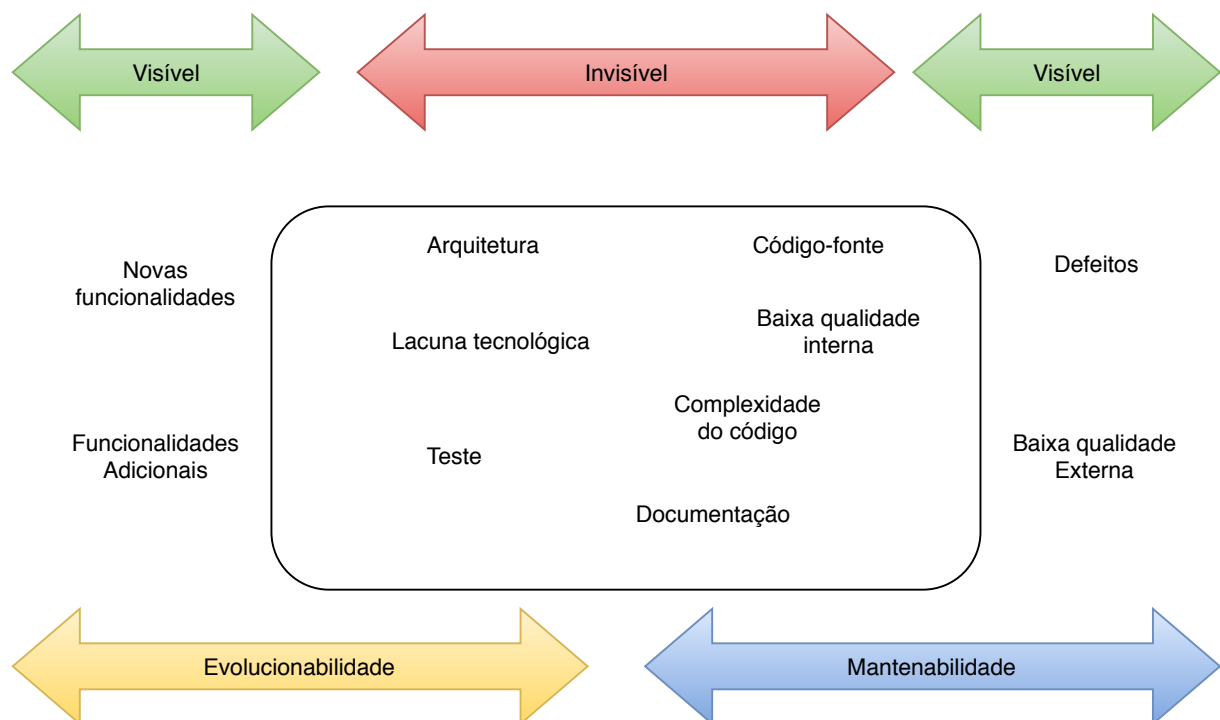


Figura 1.4: Panorama da dívida técnica. Adaptado de [KNOF13].

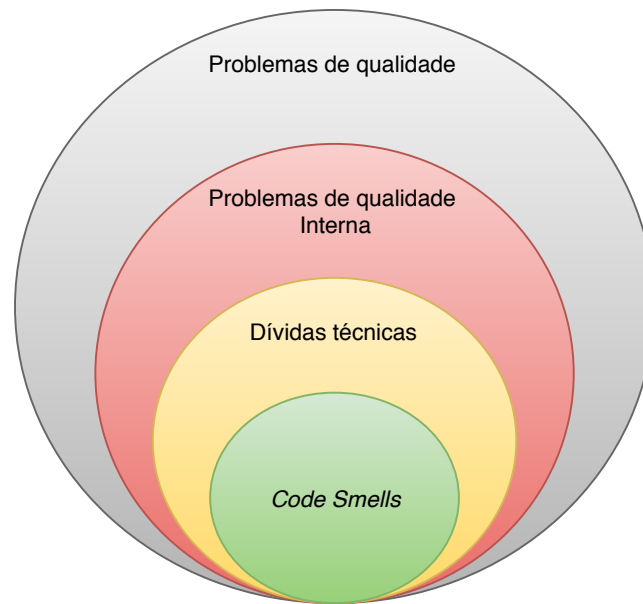


Figura 1.5: *Localização da dívida técnica na hierarquia dos problemas de qualidade de software.*

Com o aumento da popularidade dessa metáfora, novos tipos de dívida técnica surgiram. Alguns desses novos 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, conseqüentemente, 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.

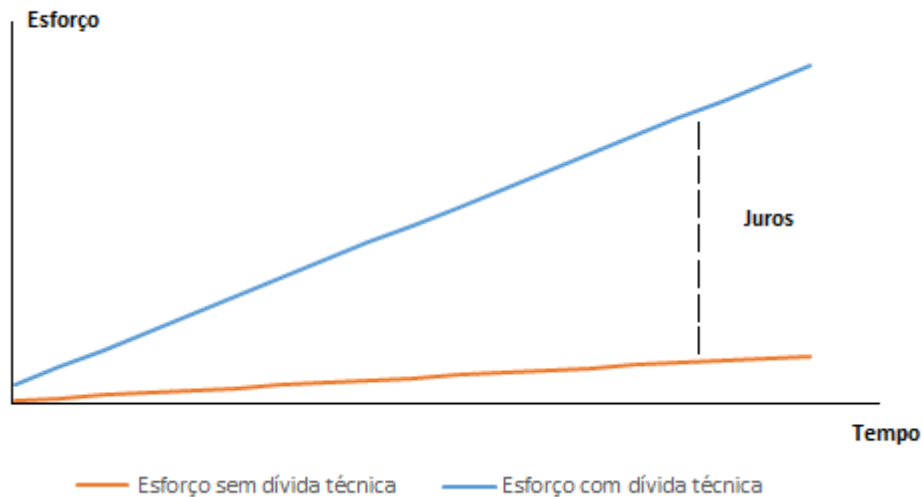


Figura 1.6: Representação dos juros como o esforço adicional causado pela dívida técnica.

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, KTW11]. 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.

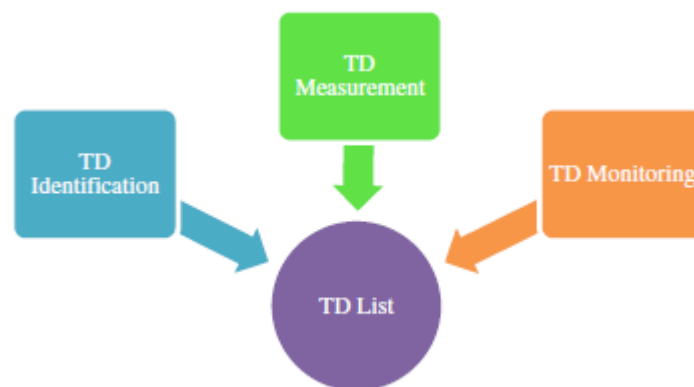


Figura 1.7: *Framework para gerenciamento da dívida técnica. Adaptado de [SG11].*

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 desenvolvido 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 3.1, 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 3 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.

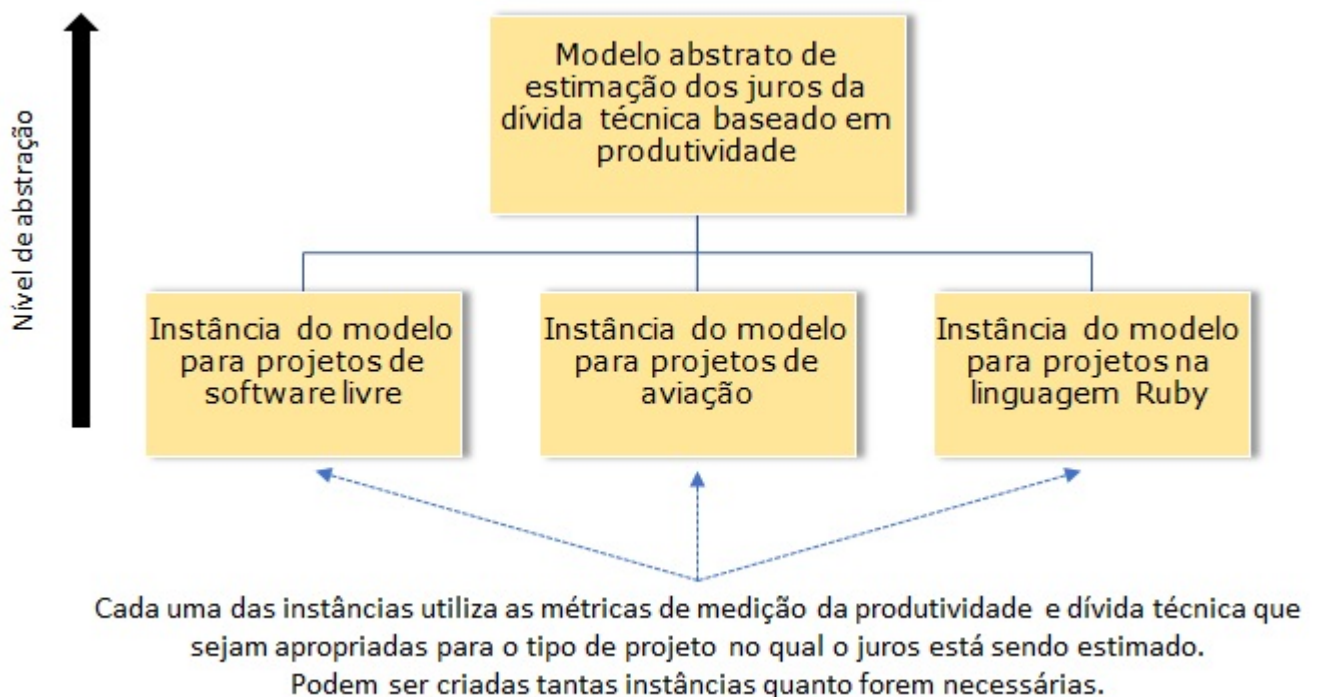


Figura 2.1: Níveis de abstração do modelo de estimação dos juros da dívida técnica.

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 et 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. [ABS^N02] 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 subseqüentes verificações.

São muitas as atividades necessárias para a realização de uma pesquisa quantitativa. Essas atividades podem ser ditas em duas fases. A primeira fase é constituída 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 diferentes unidades de interesse. Ou seja, são consideradas 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 3, 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 4.7. 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 4.3. 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 [?].

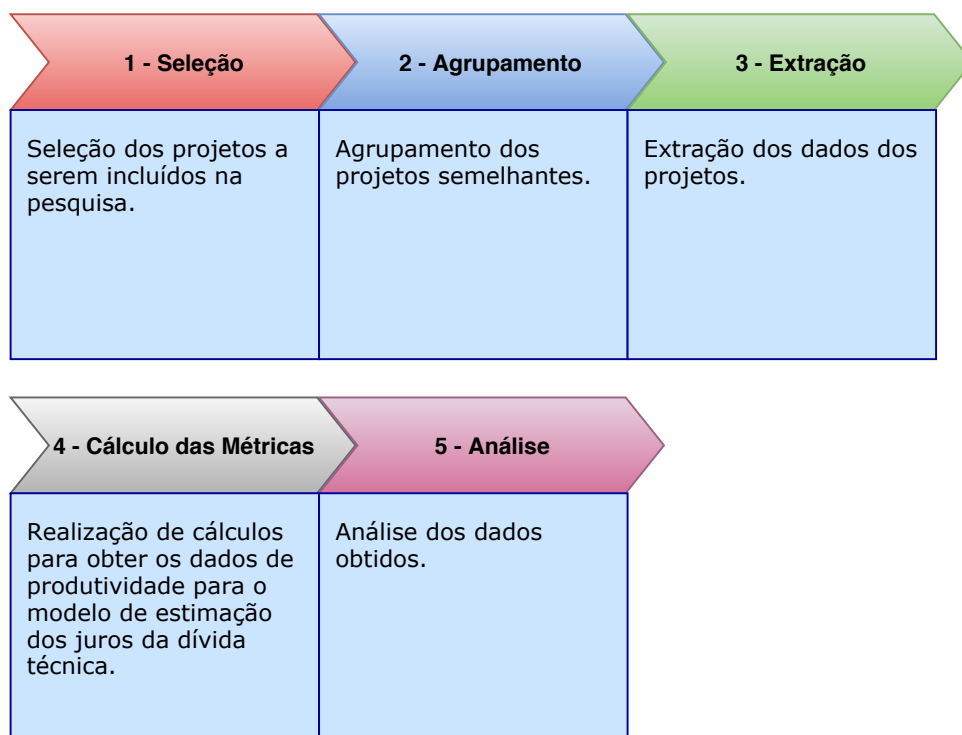


Figura 2.2: Resumo das etapas do estudo de caso.

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 domínio 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 estudadas 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.[KGM106]. 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.

Etapas 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 faturações para diminuí-la substancialmente. Ao extrairmos as métricas em diferentes momentos da evolução do software, estamos considerando essa variação temporal.

Etapas 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 dessas 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.

Etapas 5 -Análise dos resultados

Nesta etapa, analisamos estatisticamente todos os 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.

Capítulo 3

O modelo de estimação do juros da dívida técnica

Neste capítulo definiremos um modelo para estimação dos juros da dívida técnica baseado na variação da produtividade nos projetos de desenvolvimento de software. Inicialmente, iremos descrever a versão de alto nível desse modelo e em quais ideias ele está baseado. Após isso, forneceremos uma descrição de uma aplicação desse modelo na estimação dos juros da dívida técnica em projetos de software livre.

3.1 Introdução

Tradicionalmente, os juros da dívida técnica são definidos como o conjunto das dificuldades adicionais, causadas pela existência da dívida, para realizar as atividades de desenvolvimento de software. Caso a dívida técnica não estivesse presente, essas dificuldades não existiriam. Nesta pesquisa, sugerimos uma nova perspectiva para analisar os juros da dívida técnica e assim permitir seu gerenciamento: considerá-los como uma variação negativa na produtividade dos projetos. Quanto mais juros um projeto tem, mais a sua produtividade real é menor do que a produtividade que seria obtida em um cenário onde não houvesse dívida técnica. Isso acontece, pois, mais recursos terão de ser utilizados para atingir os mesmos objetivos. Vamos ilustrar essa degradação na produtividade com um exemplo: uma empresa de desenvolvimento de software resolve iniciar um projeto para o desenvolvimento de algumas novas funcionalidades para um sistema de vendas existente. Ao realizar uma análise de viabilidade, o time responsável descobre que o código atual do sistema apresenta uma documentação insuficiente, problemas de arquitetura e design e quantidade de testes unitários não compatível com o nível de qualidade esperado para o projeto. O time conclui,

corretamente, que haverá uma série de dificuldades adicionais para a conclusão do projeto. Nesse exemplo, fica claro que a produtividade do time seria melhor caso esses problemas identificados não existissem. Ou seja, menos recursos teriam de ser gastos para alcançar os mesmos objetivos. Os problemas encontrados são as dívidas técnicas do software que foram sendo adquiridas com o passar do tempo. As dificuldades para realizar o projeto de adição das funcionalidades adicionais são os juros causados por essas dívidas. Nossa abordagem para estimar esses juros será calcular a diferença entre a produtividade em um cenário onde não exista dívida técnica e a produtividade em um cenário onde exista a dívida técnica. Como falaremos mais à frente, na verdade o cenário onde não existe dívida técnica não existe. O que usaremos será uma aproximação. Para calcular os juros, usaremos cenários onde a dívida técnica seja muito baixa. Logo, a produtividade do projeto estará muito próxima da melhor possível. Então compararemos essa produtividade com outros projetos semelhantes. A diferença entre a produtividade dos projetos nesses dois cenários será uma estimativa dos juros da dívida técnica.

3.2 Estimação da produtividade em projetos de software

O termo produtividade é usado para descrever a proporção entre o valor dos recursos utilizados em um processo e o valor do que é efetivamente produzido. Os recursos aplicados são chamados de entradas enquanto que os resultados do processo são chamados de saídas. O problema de medir a produtividade de um processo pode ser resumido em duas partes. A primeira é identificar quais são as entradas e as saídas. A segunda é quantificar as entradas e saídas de tal forma que a relação entre elas possa ser calculada. O processo mais eficiente é aquele no qual mais valor é produzido com menos valor gasto nas entradas.

Existe uma série de desafios para identificar e quantificar as entradas e saídas do processo de desenvolvimento de software. Devido a inerente complexidade desse processo, existem diversas possibilidades para quais serão as entradas e saídas a serem incluídas na análise de produtividade. A quantidade de homens/hora e a quantidade de linhas de código produzidas são métricas normalmente utilizadas como entrada e saída respectivamente. Entretanto essas métricas, apesar de poderem ser consistentemente medidas, são demasiadamente imprecisas já que existem diversos outros fatores relevantes conforme mostrado no estudo de MacCormack et al. [MKCC03]. No caso da quantidade de homens/hora, por exemplo, um outro fator importante é o nível de experiência das pessoas envolvidas. A hora de um colaborador inexperiente naturalmente será menos valiosa do que a hora de um com mais experiência. Semelhantemente, a quantidade de linhas de código

produzidas, apesar de muito utilizada, também é uma métrica imprecisa já que não inclui uma quantificação do valor dessas linhas de código. É possível que uma grande quantidade de linhas de código seja criada para realizar uma atividade, porém, que essa mesma atividade possa ser desenvolvida com uma quantidade bem menor. Além destes dois exemplos, existem outras entradas e saídas que podem ser utilizadas em modelos de medição de produtividade conforme mostrado por Hernández-López et al.,[HLCPSAL15]. Podemos observar que não existe uma forma totalmente precisa para se avaliar a produtividade dos processos de desenvolvimento de software. Isso se dá tanto pela existência de diversas medidas possíveis como também devido aos aspectos subjetivos dessas medidas. Apesar disso, é possível que seja adotado um modelo impreciso, mas que seja o mais adequado para um determinado contexto.

Nesta pesquisa iremos sugerir um modelo de avaliação de produtividade baseado no trabalho de Kitchenham e Mendes [KM04]. No modelo sugerido por esses autores, a produtividade é avaliada comparando o esforço estimado para a realização de um determinado projeto de software com o esforço efetivamente gasto conforme mostrado na equação 3.1. O esforço estimado, representado pela variável *AdjustedSize* no modelo da equação 3.1 é calculado utilizando dados históricos de projetos similares. Ou seja, a partir de alguns parâmetros previamente selecionados é realizada uma estimativa de quanto teria de ser gasto de recursos para produzir o projeto. Essa estimativa é feita utilizando um modelo de regressão linear. A variável *Effort* é o esforço efetivamente gasto para a realização do projeto. Se a variável *Productivity* for maior que 1 quer dizer que o projeto foi realizado com menos esforço do que a média dos outros projetos semelhantes.

$$Productivity = AdjustedSize / Effort \quad (3.1)$$

O modelo descrito na equação 3.1 não determina quais medidas serão utilizadas para quantificar as variáveis *AdjustedSize* e *Effort*. Isso é esperado já que essas medidas são diferentes para cada domínio da aplicação sendo desenvolvida. Os autores fornecem um exemplo da aplicação do modelo em um projeto de desenvolvimento web. Nesse exemplo, é utilizado o número de páginas, o número de imagens e o número de funcionalidades da aplicação como medidas para estimar o esforço. Essas medidas são extraídas de um conjunto de projetos anteriores. Depois é criado um modelo de regressão linear com base nesses dados. A avaliação de um projeto consiste em calcular a variável *AdjustedSize* utilizando esse modelo de regressão. A produtividade então é avaliada utilizando a equação 3.1.

Apesar da viabilidade dessa estratégia, os autores chegam a conclusão de que calcular a produ-

tividade esperada, utilizando os fatores associados, é uma atividade complexa e que também não pode ser feita com precisão absoluta [Pet11].

3.3 O modelo de estimação

Nosso modelo de estimação dos juros foi dividido em dois níveis de abstração. O primeiro traz uma definição abstrata da ideia de estimação dos juros da dívida técnica por meio da estimação da variação de produtividade. Nesse nível de abstração, não há uma definição de quais métricas serão utilizadas tanto para a estimação da dívida quanto para a estimação da produtividade. Ao invés disso, há uma descrição de como a estimação dos juros pode ser feita por meio da avaliação das variações da produtividade dos projetos. Já no segundo nível de abstração, existe uma definição precisa a respeito de quais métricas serão utilizadas e qual o processo será realizado para estimar os juros da dívida técnica. Nesse nível de abstração há, na verdade, um modelo de estimação para cada tipo de projeto conforme ilustrado na Figura 3.1. Essa divisão em níveis de abstração foi necessária devido às diferenças na forma de avaliar a produtividade dos projetos. Por exemplo, um projeto de um sistema web não está sujeito as mesmas restrições de um projeto de software na área da aviação. Logo, é difícil criar um método de avaliação de produtividade geral o suficiente para que seja utilizado nesses dois contextos. Por isso, cada tipo de projeto terá de ter um modelo de avaliação dos juros compatível com suas características, especialmente em relação às métricas de produtividade.

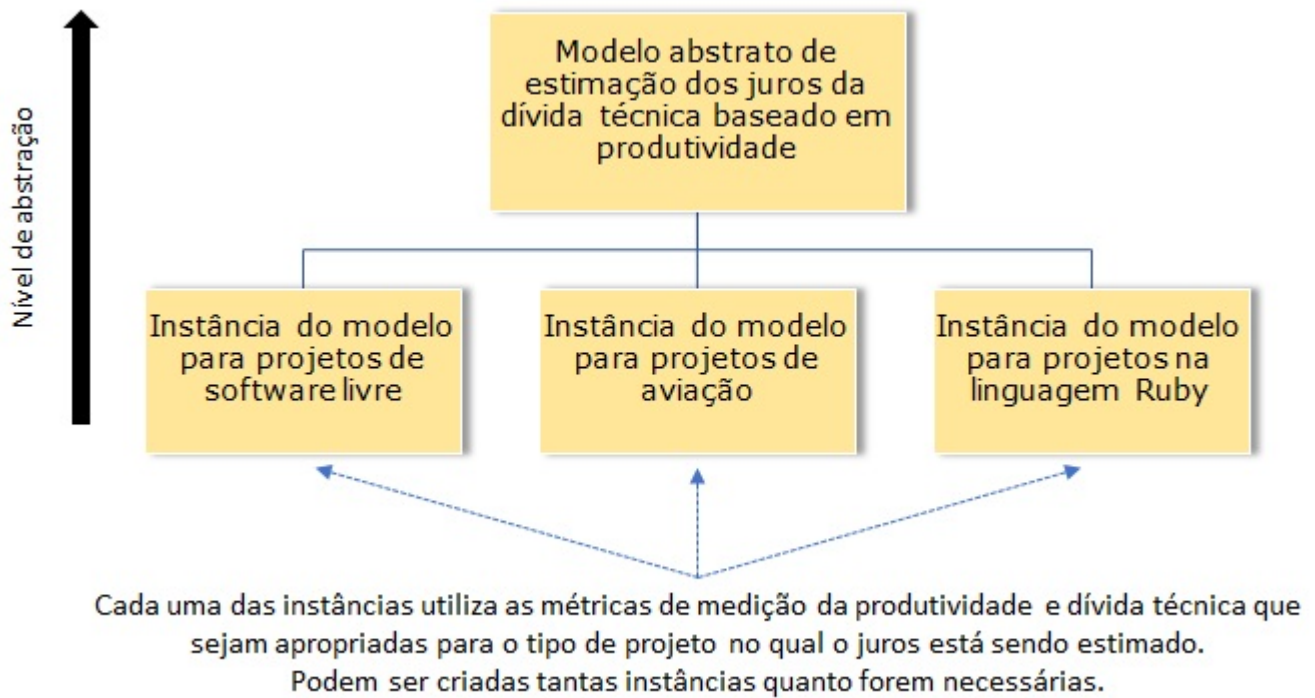


Figura 3.1: Níveis de abstração do modelo de estimativa dos juros da dívida técnica.

3.4 O modelo abstrato de estimativa dos juros da dívida técnica

Para explicar a sustentação lógica do modelo abstrato de estimativa dos juros da dívida técnica definiremos dois cenários fictícios. Chamaremos de **produtividade ótima** o cenário onde a produtividade de um projeto de software é a melhor possível com os recursos disponíveis. Ou seja, levando em consideração todo o contexto no qual o projeto é desenvolvido, a produtividade obtida é a melhor que poderia ser alcançada. Esse contexto é formado por uma série de variáveis relacionadas. Algumas dessas variáveis estão relacionadas ao time de desenvolvimento como a competência, quantidade e experiência dos participantes. Outras variáveis estão associadas às características do projeto como complexidade, área e prazos. Como dissemos anteriormente, a dívida técnica é um fator que influencia negativamente a produtividade de um projeto. Entretanto, no cenário de **produtividade ótima** isso não acontece pois nesse cenário não há dívida técnica. Chamaremos de **produtividade afetada** o cenário onde existe um déficit de produtividade no projeto causado pela existência de uma quantidade significativa de dívida técnica. Neste cenário, a produtividade não é a melhor que poderia ser obtida considerando o contexto e os recursos disponíveis. E isso é causado pela existência da dívida técnica. Ou seja, a causa da produtividade dos projetos no cenário de **produtividade afetada** não ser tão boa quanto os do cenário de **produtividade ótima** é

exclusivamente a existência de dívida técnica.

É importante notar que o cenário de **produtividade ótima** nunca existirá já que não é possível a existência de um software sem nenhuma dívida técnica. Temos algumas razões para afirmar a inexistência de projetos sem nenhuma dívida técnica. A primeira delas é o fato de que a própria identificação do que é ou não de uma dívida técnica é subjetiva e muitas vezes intangível. A segunda razão é associada às dívidas técnicas de tecnologia. Com o passar do tempo uma tecnologia vai se tornando obsoleta e continuar a utilizá-la pode trazer esforços adicionais. Com isso, devida a contínua criação de novas tecnologias, é impossível garantir que não haja algum tipo de obsolescência. No entanto, ignoraremos o impacto que essa dívida técnica mínima terá na produtividade e ainda sim chamamos esse cenário de **produtividade ótima**.

A base lógica da nossa metodologia para calcular essa estimativa pode ser explicada por meio da seguinte situação fictícia: imaginemos que um determinado projeto de desenvolvimento que consistia em adicionar um conjunto de novas funcionalidades a um sistema já existente, foi realizado em um cenário de **produtividade afetada**. Ou seja, esse sistema existente possuía um número D de dívidas técnicas. Esse projeto trouxe um número S que representa uma estimativa do que deveria ter sido produzido pelo projeto levando em consideração o que foi gasto. Essa estimativa será calculada utilizando a estratégia descrita na seção 3.2. Para obter o resultado S foi utilizada uma equipe de desenvolvimento de software de tamanho E obtendo uma produtividade Y . Com isso, a produtividade Y do projeto pode ser calculada de acordo com a Equação 3.2.

$$Y = \frac{S}{D * E} \quad (3.2)$$

Agora imaginemos que esse mesmo projeto foi realizado pela mesma equipe e em um contexto exatamente igual. Entretanto, neste cenário a dívida técnica do projeto não existe. Ou seja, o projeto dessa vez foi realizado em um cenário de **produtividade ótima**. Logo, a produtividade do time de desenvolvimento será outra que chamaremos de \bar{Y} . A produtividade \bar{Y} pode ser calculada utilizando a Equação 3.3. É evidente que a produtividade \bar{Y} será melhor do que a produtividade Y já que a única diferença entre os dois cenários é a existência ou não de dívida técnica. Sem a dívida técnica, o time terá mais facilidade para desenvolver as funcionalidades do projeto e com isso elas serão desenvolvidas em um menor tempo aumentando assim a produtividade. Observando esse exemplo fictício podemos inferir a Equação 3.4 onde J é o juros da dívida técnica que foi pago durante a execução do projeto de desenvolvimento no cenário de **produtividade afetada**. *Note que não estamos dizendo que J é uma estimativa dos juros e sim uma medição exata.*

Entretanto, esse cenário fictício obviamente não pode ser reproduzido exatamente como descrito. Um motivo é a impossibilidade de executar o mesmo projeto duas vezes com o mesmo time sem que a segunda execução seja facilitada pelas experiências obtidas pela primeira. Outro motivo é a impossibilidade de remover totalmente a dívida técnica D do sistema existente. Contudo, esse exemplo apresenta-se útil como uma argumentação lógica para a metodologia que utilizamos para estimar J em situações reais.

$$\bar{Y} = \frac{S}{E} \quad (3.3)$$

$$J = \bar{Y} - Y \quad (3.4)$$

Conforme descrito anteriormente, não podemos calcular precisamente os juros da dívida técnica utilizando a Equação 3.4. Porém, utilizaremos essa fórmula e a situação fictícia que descrevemos como uma base lógica para o cálculo de uma estimativa dos juros da dívida técnica. Para isso, precisamos de valores estimados para Y e \bar{Y} . Nossa estratégia será a de estimar o valor de Y utilizando os projetos no cenário de **produtividade afetada** e o valor de \bar{Y} utilizando os projetos no cenário de **produtividade ótima**. Essa estratégia consiste em realizar uma aproximação da situação fictícia que descrevemos já que não podemos reproduzi-la. Como não podemos executar um mesmo projeto duas vezes sem que a segunda vez seja facilitada pelas experiências da primeira vez, vamos utilizar dois ou mais projetos semelhantes. Ou seja, só iremos calcular $\bar{Y} - Y$ utilizando projetos nos cenários de **produtividade ótima** e **produtividade afetada** que sejam semelhantes. Com isso, vamos simular a remoção da dívida D de um projeto. Essa simulação será feita ao usarmos dois projetos. Um projeto que possua um nível de dívida técnica D e outro projeto que possua um nível de dívida técnica \bar{D} . Sendo que \bar{D} é substancialmente menor do que D .

3.5 Os modelos concretos de estimação dos juros da dívida técnica

Para aplicar em situações reais as ideias apresentadas no modelo de alto nível é necessário definir diversos aspectos específicos para o tipo de projetos que serão avaliados. Ou seja, o modelo abstrato é utilizado como base para a criação de modelos concretos e específicos. Conforme ilustrado na Figura 3.1, é possível criar diversos modelos concretos, um para cada tipo de projeto de software. Para a criação dos modelos concretos será necessário definir certas métricas que serão utilizadas para avaliar a produtividade dos projetos. Além disso, será necessário definir como agrupar os projetos

semelhantes como também particionar cada um desses grupos de forma a separar os projetos nos cenários de produtividade ótima e produtividade afetada. A tabela 3.1 resume todas as definições que precisam ser feitas para a criação de um modelo concreto para a estimação dos juros da dívida técnica. A seguir descreveremos cada uma dessas atividades.

Tabela 3.1: *Atividades necessárias para a criação de um modelo concreto de estimação dos juros da dívida técnica específico.*

#	Atividade
1	Seleção das métricas que representam as entradas do processo
2	Seleção das métricas que representam as saídas do processo
3	Definição do método de agrupamento dos projetos semelhantes
4	Definição do método de particionamento dos grupos de projetos

3.5.1 Seleção das métricas que representam as entradas do processo

Em um modelo de avaliação de produtividade de um processo, as entradas representem aquilo que é gasto para que o resultado do processo seja alcançado. Nesta etapa, são definidas quais entradas serão utilizadas. Na avaliação da produtividade em projetos de software, normalmente são usadas como entradas métricas relacionadas ao esforço necessário para a produção do software. Esse esforço é medido tradicionalmente por meio da quantidade de pessoas que foram alocadas para o projeto e pela quantidade de tempo no qual essas pessoas atuaram. Entretanto, existe uma série de aspectos que podem afetar a medida de esforço em uma análise de produtividade e que podem ser incluídas nos modelos específicos para torná-los mais precisos. A literatura nos fornece alguns desses aspectos conforme listaremos a seguir:

- O nível de experiência da equipe tem uma influência direta na avaliação de produtividade de um projeto de software. Um exemplo onde isso pôde ser verificado empiricamente pode ser encontrado no trabalho de Kitchenham et al. [KM04]. Um dos resultados dessa pesquisa mostrou uma divergência significativa entre o nível de produtividade medido pelo modelo sugerido pelos autores e a produtividade medida pela empresa para um projeto real. A produtividade medida pelo modelo foi muito baixa enquanto que a empresa considerava que o projeto foi extremamente bem sucedido. Ao analisar com detalhes os dados obtidos, os pesquisadores chegaram a conclusão que o projeto foi realizado por uma equipe júnior e inexperiente. Contudo, ele foi realizado em um tempo satisfatório pela empresa. Como o modelo da pesquisa não utilizava informações a respeito da experiência dos profissionais, o modelo não foi capaz de

identificar que se trata sim de um projeto produtivo pois foi realizado com uma equipe menos experiente e consequentemente mais barata. Além do tempo, o fator conhecimento também pode ser incluído em um modelo de análise de esforço. Um profissional pode trabalhar por muito tempo em uma determinada atividade sem que ele realmente adquira novos conhecimentos e melhore a execução das suas responsabilidades. Por outro lado, um profissional com menos tempo pode se dedicar ao seu crescimento individual e alcançar resultados melhores do que o de profissionais com mais anos de experiência. Isso se torna especialmente importante no contexto da análise de produtividade quando a organização investe no aperfeiçoamento individual. Os custos desse desenvolvimento podem ser incorporados como esforço no modelo de análise de produtividade.

- O tempo gasto pela equipe de desenvolvimento não é exclusivamente gasto produzindo código. De acordo com Wagner et al. [WR18], até um terço do tempo de um desenvolvedor é usado com reuniões, apresentações, gestão de projetos e realização de cursos para o aprimoramento individual. Se no contexto no qual o software está sendo desenvolvimento há uma priorização para atividades não técnicas como essas, é possível que o resultado da análise de produtividade seja prejudicado. Isso torna-se especialmente preocupante quando o objetivo é avaliar a produtividade de projetos em contextos diferentes. Um projeto em um ambiente muito burocrático, por exemplo, no qual sejam realizadas muitas reuniões que não estejam relacionadas ao projeto, pode ser taxado como improdutivo caso o tempo gasto com essas reuniões seja contabilizado como tempo gasto no projeto.
- Alguns fatores não técnicos podem influenciar a produtividade de uma equipe. De acordo com uma revisão sistemática realizada por Wagner et al. [WR18], esses fatores são relacionados à aspectos como o quão amigável é o ambiente no qual o software é desenvolvido, a diferença de temperamentos entre os membros do time de desenvolvimento e a adequação do local de trabalho para a realização de atividades que exijam criatividade.

Apesar dessas informações aparentemente adicionarem maior precisão aos modelos de produtividade, nem sempre elas poderão ser utilizadas. O primeiro motivo é a possível inexistência de dados que tornem possível a sua medição. Por exemplo, muitos repositórios de software não possuem informações detalhadas a respeito do histórico dos seus membros. Isso inviabiliza medir o nível de experiência de um determinado contribuidor. Além disso, algumas das informações que poderiam incrementar a precisão das métricas de esforço são difíceis de serem medidas. Um exemplo é o nível adequação do local de trabalho para a realização de trabalho criativo. Para obter dados

para medir esses aspectos podem ser realizadas entrevistas ou utilizados questionários. Entretanto essa medição pode se tornar cara quando o número de colaboradores e projetos é muito grande. Além disso, novamente, dificilmente essas informações são encontradas em repositórios públicos com informações sobre projetos de software.

3.5.2 Seleção das métricas que representam as saídas do processo

Há uma dificuldade adicional na escolha das métricas de saída em um modelo de avaliação da produtividade em projetos de software. Essa dificuldade é causada pelo fato de que normalmente as reais saídas de um projeto são subjetivas. Apesar de o número de linhas de código serem utilizadas extensivamente, essa métrica pode dizer muito pouco a respeito do valor que realmente foi produzido durante o projeto. Um projeto pode ter uma quantidade alta de linhas de código e ainda assim não atingir seus objetivos. Com isso, uma métrica que considere os objetivos do projeto de software pode ser mais adequada para medir o que foi produzido. Uma das alternativas é a utilização dos pontos por função. Nessa técnica de medição é estabelecida, pelo usuário do software, uma pontuação para cada funcionalidade. Essa pontuação é independente da tecnologia ou linguagem que será utilizada para implementar a funcionalidade. O tamanho do software é então medido pela quantidade total de pontos de todas as suas funcionalidades[JL97]. Apesar da quantidade de linhas de código e os pontos por função serem utilizados largamente como métricas para representar o tamanho do software, elas sozinhas podem não ser suficientes para capturar todos os aspectos necessários para realmente calcular o tamanho de um projeto.

Cada contexto pode exigir métricas diversas para capturar os objetivos do projeto e consequentemente seu tamanho. Um exemplo fornecido por Kitchenham et al.[KM04] é de projetos de websites. A quantidade de linhas de código ou os pontos por função podem ser adequados para medir o tamanho das funcionalidades dinâmicas do website tais como comércio eletrônico e interação dos usuários com o conteúdo. Entretanto, algumas características estáticas como as imagens e o número de páginas com conteúdo estão diretamente ligadas ao esforço necessário para realizar o projeto e ainda assim não podem ser capturadas por essas duas métricas. Outro exemplo são os projetos de software livre. Nesses projetos, os objetivos a serem alcançados também são as funcionalidades que serão disponibilizadas aos seus usuários. Porém, no contexto desses projetos, pode ser adequado incluir outras métricas como popularidade e facilidade de colaboração para representar o tamanho do software.

De acordo com Kitchenham et al.[KM04], em um modelo de análise de produtividade, é necessário que as métricas utilizadas para representar as saídas do processo de desenvolvimento

estejam relacionadas com as métricas utilizadas para representar o esforço. Isso quer dizer que, estatisticamente, deve haver uma correlação não nula entre o esforço e cada uma dessas métricas de tamanho. Ou seja, a medida que mais ou menos esforço seja realizado, deverá haver um impacto no valor das variáveis de tamanho. A razão disso é que não faria sentido incluir em uma análise de produtividade informações que não são afetadas pelo valor gasto para produzir as saídas. Isso acontece, pois, uma medida de produtividade é, por definição, a relação entre o quanto se gasta e o quanto se produz.

3.5.3 Definição do método de agrupamento dos projetos semelhantes

Conforme descrito na seção 3.4, nosso modelo abstrato de estimação dos juros da dívida técnica é baseado na ideia de que os juros são as dificuldades adicionais para desenvolver o software e que essas dificuldades não seriam encontradas caso não houvesse a dívida técnica. Os juros podem ser calculados como a diferença de produtividade em um cenário com a dívida técnica e um cenário sem a dívida técnica. Conforme explicado na seção 3.4, a criação desses cenários é inviável. Contudo, propomos, ao invés do cálculo exato, uma estimação dos juros por meio de uma aproximação. Essa estimação é obtida ao compararmos a produtividade de projetos semelhantes. Sendo que uma parte desses projetos possui um nível pequeno de dívida técnica enquanto que a outra parte possui um nível normal ou grande. Para realizar essa comparação precisamos identificar se um projeto é semelhante a outro. O modelo abstrato descrito na seção 3.4 não define como essa análise de similaridade deve ser realizada. Seria difícil descrever uma estratégia de análise de similaridade que possa ser utilizada em todas as situações. Por isso, a estratégia que será utilizada deve ser definida em cada um dos modelos concretos de estimação dos juros da dívida técnica.

Na literatura podem ser encontradas algumas abordagens para a análise de similaridade entre projetos de software. Em [BR10] Barreto et al., realizam um *survey* para verificar a opinião de especialistas sobre quais são as características que devem ser levadas em consideração para calcular a similaridade entre projetos. As 5 características mais relevantes de acordo com os especialistas foram objetivo do projeto, medida usada para indicar os objetivos do projeto, cliente, experiência do time de desenvolvimento e experiência do gerente de projetos. Para calcular uma medida numérica de similaridade entre os projetos é fornecido um modelo matemático. Neste modelo, a similaridade entre dois projetos é calculada pelo somatório do inverso da diferença entre cada uma das características do projeto. Cada uma das características tem um peso calculado de acordo com a relevância indicada pelos especialistas. A similaridade das características categóricas é calculada como 1 quando os dois projetos têm o mesmo valor e 0 quando não tem. Uma das limitações

desse trabalho é a de que ele considera apenas características numéricas dos projetos ou converte as características não numéricas em numéricas. Porém, algumas características de um projeto de software podem ser medidas por meio de variáveis categóricas. As variáveis categóricas representam informações qualitativas do projeto. Um exemplo seria a variável complexidade. Ela pode possuir valores como baixa, média ou alta. Uma abordagem que resolve esse problema é proposta por Idri et al.[IA01]. Nela os autores propõem uma técnica baseada em lógica fuzzy para a avaliação de similaridade entre projetos. Essa abordagem pode ser utilizada para avaliar a similaridade de projetos com variáveis categóricas. Entretanto, ela, assim como a anterior, não leva em consideração o domínio da aplicação. Em ambas as abordagens é realizada uma análise quantitativa sobre as características do projeto como complexidade, número de funções, quantidade de arquivos e assim por diante. Sendo assim, por meio dessas abordagens não é possível distinguir se um determinado projeto se trata de um sistema complexo como um gerenciador de banco de dados ou apenas uma aplicação web. Desde que tenha as mesmas características, dois projetos tão diferentes como esses serão classificados como similares.

Na literatura, pudemos encontrar algumas estratégias que levam em consideração o domínio ou tipo de aplicação. Um exemplo é o sistema criado por Shinji et al. chamado MUDABlue[KGMI06]. Esse sistema analisa apenas o código fonte das aplicações e sugere quais projetos são similares. Essa sugestão é feita com base em diversas informações do código fonte. Uma delas é a lista de dependências do projeto. Por exemplo, se um projeto tem como uma de suas dependências uma biblioteca de tratamento de imagens, é provável que esse software seja um editor de imagens ou alguma ferramenta relacionada com o tratamento de imagens. Outra estratégia utilizada pelo MUDABlue para a categorização dos projetos é aplicar técnicas de aprendizado de máquina usando como entrada o nome das variáveis utilizadas no código fonte. Os autores sugerem que os nomes das variáveis dizem muito a respeito do domínio no qual o software pertence e que eles podem ser usados para categorização. Outra abordagem muito semelhante ao MUDABlue é proposta por MacMillan et al.[MLVPG11]. Uma diferença significativa é a de que ao invés de analisar o código fonte, os autores analisam o programa compilado. Além disso, é utilizado um algoritmo de treinamento supervisionado onde as categorias são previamente definidas.

Outra categoria de estratégias para a análise de similaridade é aquela em que é utilizada a descrição textual ou documentação dos projetos. Por meio de uma análise textual desses documentos é determinado um conjunto de tópicos e o quão provável é que um determinado documento seja sobre um desses tópicos. Uma das técnicas que seguem essa estratégia é o LDA (*Latent dirichlet allocation*)[BNJ02]. Por meio dessa técnica é possível categorizar automaticamente documentos

com textos em qualquer linguagem. Basicamente, o LDA funciona da seguinte forma:

1. É definido o número de categorias que serão utilizadas. É importante notar que o LDA exige apenas a quantidade e não quais são essas categorias.
2. O algoritmo descobre quais palavras pertencem a cada categoria. Isso é feito ao identificar quais palavras normalmente aparecem juntas nos documentos.
3. Cada documento é analisado tendo como base as categorias encontradas. A quantidade de palavras de cada categoria em cada documento irá definir o quanto aquele documento é sobre aquela categoria. Ou seja, o LDA consegue estimar que um documento é X% sobre um assunto e Y% sobre outro.

Uma das desvantagens do LDA em relação aos outros métodos é a de que ele exige a existência de uma descrição textual para conseguir realizar a categorização. Apesar disso, essa estratégia tem sido utilizada consistentemente para a categorização de software[CTNH12, TRP09, MSH08, KAAH11]

3.5.4 Definição do método de particionamento dos grupos de projetos

O último passo para a definição de um modelo específico de estimação dos juros da dívida técnica é determinar quais projetos serão uma aproximação de um projeto desenvolvido em um cenário de produtividade ótima e quais serão projetos que foram desenvolvidos em um cenário de produtividade afetada. Essa separação deve ser baseada em critérios numéricos e dependerá dos dados disponíveis a respeito dos projetos analisados. Não é possível determinar um limiar geral entre esses dois cenários por dois motivos. O primeiro motivo é o fato de que o modelo abstrato de estimação não define como o nível de dívida técnica será calculado. Isso acontece por diversas razões. Entre elas estão a inexistência de uma forma padronizada de calcular a dívida técnica e as diferenças significativas entre as dívidas técnicas em diferentes linguagens de programação. Outro impeditivo para a criação do limiar entre os dois cenários é a diferença que pode existir entre os níveis de dívida técnica dos projetos analisados. Por exemplo, pode ser que os projetos tenham todos um nível de dívida técnica muito baixo e com isso o nível de transição entre os dois cenários também será pequeno. Semelhantemente, os projetos podem ter todos um nível alto de dívida técnica. Nesse caso, até mesmo os projetos que serão utilizados como uma aproximação da produtividade ótima também terão um alto nível de dívidas técnicas.

3.6 O modelo de estimação dos juros em projetos de software livre

A seguir descreveremos um modelo concreto para estimar os juros da dívida técnica em projetos de software livre. Os softwares livres foram escolhidos pela abundância de projetos disponíveis publicamente e pela facilidade para a obtenção dos dados desses projetos. Além disso, é imprescindível que o código fonte dos projetos analisados esteja disponível para que as métricas de produtividade possam ser calculadas. Isso torna os softwares livres uma opção vantajosa já que os repositórios públicos permitem que qualquer pessoa tenha acesso ao código dos projetos armazenados.

A seguir realizaremos, conforme resumido na tabela 3.1, todas as definições necessárias para a criação do modelo concreto voltado para projetos de software livre. Adicionalmente, no capítulo 4 descreveremos um estudo de caso no qual utilizamos o modelo concreto para a estimação dos juros da dívida técnica em projetos de software livre hospedados na plataforma GitHub.

3.7 Entradas

Consideraremos a contribuição dos colaboradores dos projetos como a entrada do nosso modelo de avaliação da produtividade. Essa característica foi a escolhida para representar a entrada do modelo de análise de produtividade já que a colaboração é o que efetivamente faz com que o projeto de software livre possa evoluir. O esperado é que quanto mais colaboração um projeto tiver, mais ele estará apto a atingir seus objetivos. Existe uma particularidade nos projetos de software livre em relação ao que é gasto para que o software seja construído. Isso acontece porque normalmente não há uma relação profissional entre os colaboradores e a empresa ou organização interessada no desenvolvimento do software. Ao invés disso, essa colaboração é feita voluntariamente. Seja porque o colaborador tem interesse nas funcionalidades fornecidas pelo software sendo construído, porque ele deseja aprender mais a respeito das tecnologias utilizadas, ou seja por qualquer outro motivo. Não há, nesse caso, uma relação que envolva um pagador e um recebedor. Logo, não podemos identificar os gastos que uma organização teve para manter a equipe de funcionários que realizou o projeto já que não existe uma organização e também não existem funcionários. Ainda assim, podemos atribuir esse dois papéis (organização e funcionário), respectivamente, para a comunidade de colaboradores e os colaboradores em si. Podemos considerar que a comunidade de colaboradores está investindo recursos ao alocar colaboradores para contribuir com um projeto. Mesmo sabendo que essa alocação não parte de uma unidade específica, mas ao invés disso é realizada por meio da vontade individual de cada colaborador. Com isso, podemos estimar a produtividade de um projeto ao medir o quanto de colaboração esse projeto obteve e o quanto de retorno essa colaboração

gerou. Projetos que obtiveram muita colaboração e pouco retorno serão considerados improdutivos enquanto que projetos com pouca colaboração e muito retorno serão considerados produtivos. A forma de medir qual o retorno que um projeto forneceu à comunidade será descrita na seção 3.8.

A colaboração nos projetos de software livre será medida por meio de dois aspectos: qualidade e assiduidade. Essa divisão foi realizada com o objetivo de capturar mais características a respeito do custo da colaboração na evolução dos projetos. Claramente uma abordagem baseada apenas na quantidade de colaboradores não iria ser capaz de representar as diferenças, entre os colaboradores, que afetam significativamente a contribuição realizada. É evidente, por exemplo, que utilizando menos tempo, programadores mais experientes darão contribuições mais significativas para projetos complexos do que programadores iniciantes. Dessa forma, o tempo gasto por programadores experientes será mais caro do que o de programadores iniciantes. Esse conceito de custo deve ser observado levando em consideração nossa abordagem de pensar no tempo gasto pelos colaboradores como um investimento da comunidade no projeto. Descreveremos qual característica cada um desses aspectos irá representar e como eles serão calculados. Por fim, descreveremos a variável índice de colaboração, ela irá agregar esses 3 aspectos em uma única métrica.

3.7.1 Qualidade da colaboração

Para estimar a qualidade colaboração que um projeto recebeu iremos estimar o nível de proficiência do colaborador na linguagem utilizada no projeto. Acreditamos que colaboradores com maior conhecimento nas tecnologias que o projeto utiliza normalmente darão uma contribuição de maior qualidade do que colaboradores com menos conhecimento. O nível de conhecimento de cada colaborador será calculado por meio de uma técnica para classificar colaboradores de acordo com o seu nível de expertise.

Na literatura, pudemos encontrar algumas abordagens para estimar o nível de expertise de um indivíduo a respeito de um assunto. Hupa et al.[HRWD10] sugerem uma abordagem, baseada em três dimensões, para avaliar a habilidades de um colaborador. Essas dimensões são o conhecimento, a confiança e a rede de relacionamento. Os autores então propõem um modelo que utiliza dados a respeito dessas três dimensões. Esse modelo é utilizado para calcular uma estimativa para o desempenho de um time formado pelos colaboradores avaliados. Outra abordagem, dessa vez focada no contexto acadêmico, é o trabalho de Kalaiselvi et al. [KB13]. Nele os autores criaram uma ontologia para identificar a principal área de conhecimento dos membros de uma universidade e estimar o nível de conhecimento desses membros a respeito dessa área. Outras contribuições relevantes para esse problema são os trabalhos de Mockus et al.[MH02] e Shira et al.[SL11]. Nesses dois trabalhos,

os autores criam um modelo e uma ferramenta para encontrar especialistas em repositórios de software. Além de permitir a procura por assunto, essas ferramentas também possibilitam que os usuários possam encontrar colaboradores que sejam especialistas até mesmo em um bloco específico do código de um projeto de software. No contexto das plataformas de desenvolvimento que também possuem funcionalidades de interação social, como é o caso do GitHub e StackOverflow, existem os trabalhos de Huang et al. [HYW⁺17], Munger et al.[MZ14], Pedro et al.[SPK13] e Robber et al.[RR13]. Nesses trabalhos são sugeridas abordagens que consideram as especificidades de cada uma dessas plataformas.

Nesta pesquisa utilizaremos, para estimar o nível de expertise dos colaboradores de um projeto, a abordagem *GEMiner* que foi sugerida por Mo et al.[MSHZ15]. Essa abordagem foi escolhida, dentre as outras disponíveis, pelas seguintes razões:

- **É uma abordagem focada na classificação de programadores.** Algumas das outras abordagens encontradas na literatura são gerais e podem ser utilizadas para encontrar especialistas em diferentes domínios de atuação. Entretanto, elas não foram criadas e avaliadas levando em consideração as particularidades da área de desenvolvimento de software. Com isso, pode ser que determinadas características dessa área possam fazer com que essas abordagens produzam resultados incorretos.
- **Apresenta uma análise comparativa com trabalhos anteriores.** Na pesquisa desenvolvida por Mo et al.[MSHZ15] foi realizado uma série de experimentos para avaliar a eficácia do modelo proposto. Em uma dessas avaliações, foi realizada uma comparação entre o resultado gerado pela aplicação do modelo e uma lista oficial com os programadores mais influentes da linguagem Javascript. A comparação mostrou que houve um alto número de interseções entre a lista oficial e o resultado gerado pelo modelo. Com isso, os autores puderam concluir que a abordagem sugerida por eles apresenta um nível satisfatório de precisão.
- **A estratégia foi criada e validada considerando as particularidades da plataforma GitHub.** Isso nos garantiu que a aplicação do GEMiner seria possível já que nosso estudo de caso também iria utilizar dados provenientes do GitHub. Isso foi importante na escolha dessa proposta pois algumas das outras exigiam dados que não estavam disponíveis no GitHub.
- **Abordagem simples.** Por ser baseada em um algoritmo muito conhecido, o PageRank [PBMW99], pudemos compreender e aplicar essa abordagem de forma rápida. Além disso, os modelos matemáticos utilizados pelos autores são intuitivos e bem documentados.

- **Uma descrição detalhada a respeito dos modelos matemáticos utilizados.** Em algumas das outras abordagens possíveis para a estimação da expertise, o texto da pesquisa não era claro a respeito dos passos necessários para aplicar o modelo. Em algumas abordagens alguns cálculos não foram descritos com um nível de detalhamento suficiente ou até mesmo alguns passos para a aplicação não foram devidamente descritos. Entretanto, isso não ocorreu com o GEMiner. Os autores descreveram com detalhes cada um dos cálculos necessários. Inclusive fornecendo as fórmulas matemáticas e exemplos de como aplicá-las. Além disso, houve um nível de detalhamento satisfatório durante a apresentação dos experimentos realizados pelos autores. Isso nos permitiu adaptar e implementar a estratégia de uma forma muito próxima da original. Isso é importante pois caso tivéssemos utilizado uma abordagem muito abstrata, não poderíamos nos respaldar nas validações realizadas pelos autores já que nossa implementação poderia ter sido significativamente diferente da implementação que foi realizada e validada pelos autores.
- **Resolve problemas encontrados em abordagens anteriores.** Os autores tiveram o cuidado de analisar os trabalhos anteriores e identificar seus problemas. Uma das preocupações foi a de não considerar um especialista apenas alguém que possua prestígio na rede de colaboradores. Isso pode acontecer porque o indivíduo possui algum atributo que o faça ser bem visto pelos outros colaboradores. Entretanto, esse atributo não tem necessariamente alguma relação com suas habilidades técnicas. Esse é o caso do perfil do ex-presidente americano Barak Obama na plataforma GitHub. Esse perfil é seguido e bem avaliado por uma grande quantidade de pessoas. Entretanto, o número de contribuições feitas por ele é muito pequeno. O GEMiner consegue evitar que perfis como esse sejam incorretamente classificados como especialistas.

A seguir descreveremos os principais conceitos da abordagem GEMiner. No capítulo 4 iremos dar detalhes a respeito da nossa implementação e quais ajustes foram realizados.

Conforme dito anteriormente, o GEMiner é baseado no algoritmo PageRank, popularmente conhecido como o algoritmo do Google. O objetivo desse algoritmo é criar uma classificação baseada em uma rede de relacionamentos. No contexto das pesquisas de páginas web, esse relacionamento é formado por links. Uma página A está relacionada com uma página B , se existe em A um link que aponte para B . Assim, quanto mais links apontarem para uma determinada página, maior a chance de que essa página seja relevante e deva ser incluída nos resultados da pesquisa.

Uma das funcionalidades de interação social presentes no GitHub é a possibilidade de um

colaborador poder seguir outros colaboradores como também ser seguido. Podemos verificar por meio da Figura 3.2 que as relações entre os colaboradores podem ser modeladas como um grafo. A relevância de um determinado colaborador na comunidade é medida pela quantidade de pessoas que o seguem. Se um colaborador tem muitos seguidores, isso é um indício de que ele seja um especialista. Entretanto, o PageRank não realiza apenas uma contagem das relações individuais entre os elementos do grafo. Ao invés disso, todo o contexto do grafo é considerado. Por exemplo, na Figura 3.2 tanto o colaborador *D* quanto o colaborador *F* possuem apenas um seguidor. Entretanto, o colaborador *F* é seguido pelo colaborador *C* e esse colaborador é seguido por outros quatro colaboradores. Logo, apesar do número de seguidores de *D* e *F* ser igual, o colaborador *F* será considerado mais relevante do que o colaborador *D* pois ele é seguido pelo colaborador *C* que é o colaborador mais relevante desse conjunto. Outra característica importante do PageRank é a de que ele leva em consideração a quantidade de relações de saída de cada elemento. Por exemplo, o colaborador *C* da Figura 3.2 segue apenas o colaborador *F*. Enquanto isso, o colaborador *E* segue os colaboradores *C* e *D*. No momento de calcular a relevância dos colaboradores *C* e *D*, adicionando a eles a relevância de *E*, o algoritmo dividirá por dois a relevância que do colaborador *E* possui. Por fim, para evitar problemas durante o cálculo do grafo ao encontrar ciclos como o que vemos entre *D* e *E*, foi utilizada uma adaptação sugerida por Richardson et al.[RD02]. Nela, ao invés de escolher um nó inicial, realizar os cálculos necessários e depois navegar para os vizinhos até que haja uma convergência, o algoritmo aleatoriamente pode, a cada passo, ir para um vizinho ou escolher qualquer outro nó do grafo. Além de evitar ciclos infinitos como aconteceriam entre os nós *D* e *E*, essa versão do algoritmo também pode ser utilizada em grafos desconexos.

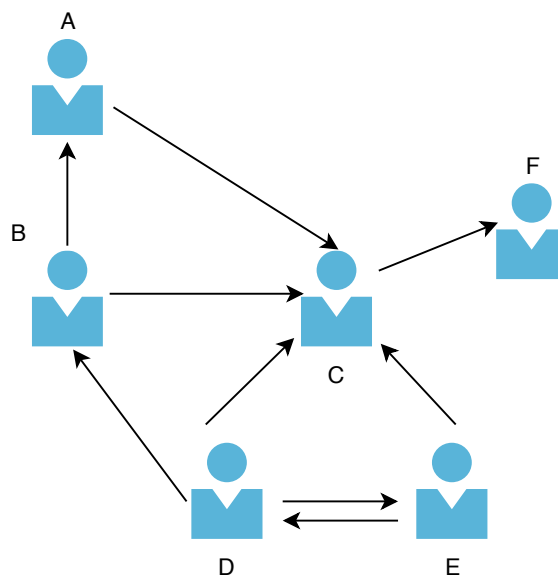


Figura 3.2: As relações de seguir e poder ser seguido no GitHub. Adaptado de [MSHZ15].

A equação 3.5 é a base do algoritmo PageRank. Como pode ser visto, trata-se de uma equação recursiva. Isso acontece pois para se calcular a relevância de um colaborador é necessário calcular a relevância de todos os colaboradores que o seguem. A primeira parte da equação inicializa o pagerank de cada elemento ao dividir o número $1 - d$ pela quantidade de colaboradores em todo o grafo sendo analisado. Sendo que d é o chamado *dumping factor* e indica a probabilidade de que no próximo passo o algoritmo calcule a relevância de um nó adjacente ao atual ou de um nó qualquer aleatório do grafo. O valor sugerido para d pelos autores é de 0.85. Os vértices do grafo analisado são representados pelas letras u e v . Sendo assim, a equação representa o cálculo da relevância do vértice v e para isso, realiza uma soma da relevância de todos os outros vértices u de tal forma que u seja v . O elemento $L(u)$ indica quantos outros colaboradores o colaborador u também segue.

$$PR(v) = \frac{1 - d}{|V|} + d * \sum_{(u,v) \in E} \frac{PR(u)}{L(u)} \quad (3.5)$$

Além de considerar o relacionamento seguir e ser seguido, os autores consideram outros aspectos para avaliar a expertise de um colaborador. Eles nomeiam essa abordagem de *Multi-Source PageRank* já que a classificação é realizada por meio de múltiplos atributos. O outro aspecto utilizado para avaliar a expertise de um colaborador é o nível de popularidade dos projetos que ele contribuiu. Isso é feito de duas formas, a primeira é analisando quais colaboradores estão observando (*watching*) o projeto. Quanto mais colaboradores relevantes observam um determinado projeto, maior é a relevância dele. A segunda forma de avaliar a relevância de um projeto é feita medindo a relevância dos colaboradores que contribuíram com o projeto. Essas duas avaliações são realizadas por meio das equações 3.6 e 3.7 respectivamente. Na equação 3.6, o conjunto E_W representa as relações de observar entre colaboradores e projetos. Logo, nessa equação, são somados os pageranks de todos os colaboradores u que observam o projeto r . Além disso, $L_W(u)$ representa a quantidade de projetos que o colaborador u observa. Lembrando que essa divisão por $L_W(u)$ é realizada para que um colaborador que observa muitos projetos não afete incorretamente o resultado final. Na equação 3.7 o conjunto $r(U)$ representa todos os colaboradores que contribuíram com o projeto r . Além disso, $L_C(u)$ representa a quantidade de projetos em que o colaborador u já contribuiu.

$$PR(r) = \sum_{(u,r) \in E_W} \frac{PR(u)}{L_W(u)} \quad (3.6)$$

$$PR(r) = PR(r) + \sum_{u \in r(U)} \frac{PR(u)}{L_C(u)} \quad (3.7)$$

Além dos já apresentados, os autores acrescentam mais dois atributos para calcular o pagerank de um colaborador: o pagerank dos colaboradores no qual ele já trabalhou em conjunto e o pagerank dos projetos no qual o colaborador trabalhou. Os autores defendem que, de acordo com suas observações, colaboradores que contribuem frequentemente em projetos com outros colaboradores relevantes, provavelmente sejam também relevantes. Além disso, colaboradores que contribuem frequentemente em projetos relevantes, também provavelmente sejam relevantes para a comunidade. Para incluir esses dois aspectos no cálculo do pagerank dos colaboradores foram utilizadas as equações 3.8 e 3.9. Na equação 3.8, o grupo E_C contém os pares de colaboradores que já contribuíram no mesmo projeto. O número $L_C(u_2)$ é igual à quantidade de projetos em que o colaborador u_2 contribuiu. Os grupos $u_1(R)$ e $u_2(R)$ representam, respectivamente, todos os projetos em que o colaborador u_1 contribuiu e todos os projetos em que o colaborador u_2 contribuiu. Logo, a equação 3.8 é utilizada para adicionar ao pagerank do colaborador u_1 o pagerank de todos os colaboradores no qual o colaborador u_1 tenha atuado em conjunto no mesmo projeto. Já na equação 3.9, o pagerank do colaborador u é somado ao pagerank de todos os projetos no qual o colaborador u tenha atuado.

$$PR(u_1) = PR(u_1) + \sum_{(u_1, u_2) \in E_C} PR(u_2) * \frac{|u_1(R) \cap u_2(R)|}{L_C(u_2)} \quad (3.8)$$

$$PR(u) = PR(u) + \sum_{r \in u(R)} \frac{PR(r)}{L_C(r)} \quad (3.9)$$

Após a aplicação de todas essas fórmulas matemáticas, é calculado o pagerank de cada colaborador. Quanto maior o pagerank, maior o nível de relevância do colaborador dentro da comunidade de desenvolvimento de software. Porém, conforme dito anteriormente, isso não garante que colaboradores com um alto pagerank sejam necessariamente especialistas em alguma tecnologia. Além disso, é preciso avaliar o volume de contribuição real que um colaborador realizou. No GEMiner isso é realizado observando a quantidade de linhas de código alteradas ou adicionadas pelo colaborador. Porém, em nossa pesquisa, utilizaremos uma medida alternativa. Ao invés da quantidade de linhas de código alteradas, utilizaremos a quantidade média diária de *commits* que cada colaborador realizou. A essa medida, demos o nome de assiduidade. Essa pequena alteração em relação a abordagem original do GEMiner foi realizada devido à dificuldade de calcular a quantidade de linhas alteradas ou modificadas em um conjunto grande projetos. Mais detalhes a respeito dessa alteração e de como implementamos o GEMiner serão fornecidas no estudo de caso apresentado no

Capítulo 4.

3.7.2 Assiduidade dos colaboradores

Para medir a assiduidade de um colaborador vamos calcular qual a média diária de contribuições e compará-la com a média geral de todos os colaboradores. Essas contribuições serão calculadas por meio de uma contagem dos *commits* que esse colaborador realizou. De acordo com Loeliger et al [LM12], um *commit* é uma alteração nos arquivos de um projeto. Essa alteração pode ser composta pela inclusão e remoção de dados ou arquivos. Além disso, um *commit* contém informações a respeito do seu autor. Isso permite rastrear o histórico de um projeto e identificar quem foi responsável por cada uma das mudanças. Utilizaremos a frequência média de *commits* que um colaborador realiza em um projeto como base para estimar a assiduidade de um colaborador em relação a um projeto. A assiduidade de um colaborador será calculada pela distância entre a sua média de *commits* e a média geral de todos os colaboradores daquele projeto.

3.7.3 Índice de colaboração

O índice de colaboração($I_c(r)$) será o valor numérico final a ser utilizado como entrada para o modelo de análise de produtividade dos projetos. Esse índice será calculado para cada projeto r de acordo com a equação 4.1. A função $A(u)$ representa a assiduidade do colaborador u enquanto que a função $Q(u)$ representa a qualidade do colaborador u . O conjunto $u(R)$ representa todos os colaboradores que realizaram alguma contribuição no projeto r .

$$I_c(r) = \sum_{u \in u(R)} A(u) * Q(u) \quad (3.10)$$

3.8 Saídas

Conforme descrito na seção 3.5.2 existe uma restrição quanto a escolha das saídas que serão utilizadas no modelo de produtividade: elas precisam estar estatisticamente correlacionadas com as entradas. A única forma de garantir isso, é verificando a significância dessa correlação. Isso só pode ser feito utilizando os dados reais dos projetos que serão analisados. Com isso, a definição definitiva das saídas não pode ser feita antes que os dados tenham sido obtidos. Entretanto, listaremos algumas das métricas candidatas a serem utilizadas. Essas métricas foram escolhidas por intuitivamente terem relação com as entradas. Porém, essa relação só será devidamente verificada no estudo de caso realizado no Capítulo 4.

3.8.1 Linhas de código

A primeira saída a ser utilizada no modelo de produtividade será a quantidade de linhas de código. Essa métrica é uma das mais utilizadas, tanto no âmbito teórico quanto no prático, para representar o tamanho do software. No contexto da pesquisa em engenharia de software, ela tem sido utilizada extensivamente como uma preditora do esforço que foi ou será gasto para a criação do software. Essa métrica foi uma das escolhidas para representar o tamanho no contexto dos softwares livres por algumas razões. A primeira delas é o aspecto prático: ela pode ser calculada utilizando métodos automáticos de contagem. Essa característica é especialmente relevante no contexto desta pesquisa já que iremos analisar uma grande quantidade de projetos. Uma métrica que exigisse algum tipo de intervenção manual seria naturalmente inviável. Esse seria o caso se usássemos uma métrica como os pontos por função. Não teríamos como medi-la automaticamente. Ao invés disso, teríamos de usar alguma abordagem qualitativa e isso inviabilizaria a análise de uma grande quantidade de projetos.

Como pode se imaginar, a contagem de linhas de código é uma métrica longe de ser perfeita para avaliar o tamanho do software. Baht et al. [BTP⁺12] descreve algumas das desvantagens dessa métrica:

- Elas não conseguem captar plenamente o esforço realizado pelos desenvolvimento. De acordo com Hoffman[Hof00], apenas por volta de 35% do esforço realizado pelos desenvolvedores é convertido diretamente em linhas de código.
- Elas não estão necessariamente relacionadas com as funcionalidades oferecidas pelo software. Por exemplo, muito código pode ser desenvolvido para a realização de apenas uma funcionalidade. Logo, isso pode influenciar negativamente a utilização das linhas de código em uma avaliação de produtividade.
- Entre os desenvolvedores podem existir costumes sintáticos distintos fazendo com que haja uma divergência na quantidade de linhas de código utilizadas para escrever uma mesma parte do software. Algumas dessas divergências podem ser amenizadas pela estratégia utilizada para contar as linhas de código. Podem ser realizadas alterações no código original antes de contar a quantidade de linhas. Essas alterações obviamente não devem alterar a lógica do programa, mas podem servir para remover essas diferenças sintáticas que prejudiquem a contagem. Ainda assim, boa parte da literatura indica que a quantidade de linhas de código nunca deve ser a única métrica utilizadas para avaliar a produtividade individual de algum

desenvolvedor.

Apesar dos problemas apresentados em utilizar as linhas de código como uma métrica de tamanho, também podemos encontrar na literatura pesquisa que utilizaram essa métrica de forma bem sucedida[[MKCC03](#), [FBHK05](#), [SFP12](#), [TLB⁺09](#)]. Analisando essas pesquisas, podemos encontrar a indicação de cuidados importantes que devem ser considerados ao utilizar as linhas de código como uma métrica de tamanho:

- Essa métrica não deve ser utilizada para comparar linguagens diferentes. [[Ros97](#), [Par92](#)].
- Não existe uma padronização amplamente aceita sobre como essa métrica deve ser calculada. Apesar da existência de esforços como o padrão criado pelo instituto de software da universidade Carnegie-Mellon[[Par92](#)]. Essa falta de um padrão definitivo acontece pelas diferenças significativas entre as linguagens de programação existente e o fato de que novas linguagens, com sintaxes diferentes, são criadas constantemente.

Uma outra alternativa seria medir o tamanho do software utilizando a quantidade de arquivos. Inclusive realizaremos a medição desse dado durante o nosso estudo de caso. Porém, estudos recentes mostram que não há diferenças significativas entre essas duas medidas. Um exemplo é a pesquisa realizada por Herraiz et al.[[HRGB⁺06](#)]. Nelas os autores mostram, por meio da análise empírica de projetos de software livre, que o padrão de crescimento desses projetos é o mesmo, independente da métrica utilizada. Ou seja, medindo o tamanho dos projetos em linhas de código ou número de arquivo leva a resultados muito semelhantes. Por isso, elencaremos apenas a quantidade de linhas de código ao invés de utilizar uma agregação das duas medidas.

3.8.2 *Pull Requests*

A base para a filosofia de software livre é a contribuição voluntária. Essa contribuição, no contexto dos softwares livre, tem sido realizada com o auxílio de repositórios de software como o GitHub, BitBucket, GitLab, SourceForge, e Launchpad. Esses repositórios normalmente utilizam um modelo de controle de versão distribuído. Nesse modelo, o software não fica armazenado em um único e exclusivo repositório central. Ao invés disso, cada colaborador possui uma versão dos arquivos para o controle de versão. Esses arquivos locais podem, inclusive, ser a base para um novo repositório que evolua o projeto de uma forma diferente da realizada no repositório original. Porém, para inserir, no repositório original, as mudanças realizadas em um repositório paralelo, é utilizado um recurso chamado de *pull request*. Nesse modelo as mudanças são primeiro avaliadas

antes de serem incluídas definitivamente no projeto. Essa avaliação pode consistir na verificação automática de conformidade do novo código com regras previamente definidas como também a execução automática de testes de software. Além disso, dentro de um *pull request*, normalmente é possível discutir essas mudanças. Essa discussão pode envolver uma avaliação funcional que verifique a efetiva necessidade de a mudança ser feita como também pode envolver discussões técnicas como o impacto da mudança no desenho e arquitetura do software. Ao final dessa discussão, a mudança pode ser aprovada ou rejeitada. Quanto mais *pull request* um projeto tem, maior o nível de interesse da comunidade em contribuir com o projeto. Acreditamos que com isso o projeto está atingindo um dos objetivos do software livre: maior interesse da comunidade em colaborar. Ou seja, elencamos a quantidade de *pull requests* como um possível medida de saída por que faz sentido considerar a constância em que colaboradores externos conseguem contribuir para um projeto como um aspecto da evolução do projeto. Quanto mais pessoas querendo contribuir, maior o projeto.

3.8.3 Popularidade

Conforme dito anteriormente, há o que se questionar na relação entre a quantidade de linhas de código de um software e o número de funcionalidades que ele disponibiliza para seus usuários e outros interessados. Pode haver uma diferença significativa entre esses dois aspectos. Logo, faz sentido incluir em nosso modelo de avaliação de produtividade, alguma métrica que seja utilizada na tentativa de capturar a quantidade ou valor das funcionalidades que um software fornece aos seus usuários. Por isso, para nosso modelo específico, inserimos a popularidade do projeto como uma possível métrica de saída para o modelo de estimação de produtividade. Essa popularidade será avaliada utilizando as funcionalidades presentes no repositório de software sendo utilizado. No caso do GiiHub, que é o repositório que utilizaremos no nosso estudo de caso do Capítulo 4, são fornecidas duas ferramentas para que um usuário possa indicar seus interesses ou sua aprovação em relação a um projeto: estrelas e *watch*. Ao dar uma estrela para um projeto o usuário indica que acha aquele projeto é relevante e gostaria de alguma forma marcá-lo para tê-lo associado a sua conta. Todos os projetos marcados com estrelas podem ser consultados em uma página da conta do usuário. Ao dar um *watch* em um projeto, o usuário irá receber notificações a respeito do projeto. Essas notificações incluem dentre outras, a liberação de uma nova release, a criação de *issues* e seus comentários e a criação de *pull requests*.

3.9 Método de agrupamento dos projetos semelhantes

No modelo concreto de estimação dos juros da dívida técnica em projetos de software livre utilizamos o LDA (*Latent Dirichlet Allocation*) para agrupar os projetos semelhantes. Utilizaremos essa técnica por dois motivos. O primeiro é o fato de que grande parte dos software livres possuem algum documento que descreve as suas funcionalidades. Esse documento pode ser usado pelo LDA para estimar qual o assunto do texto e consequentemente qual o domínio o software pertence. O segundo motivo é a existência de experimentos na literatura que utilizaram o LDA de forma satisfatória para a categorização de projetos de software livre. Um exemplo que foi utilizado como uma das bases para a implementação realizada no Capítulo 4 é o trabalho de Ray. et al. [RPFD14]. Nessa pesquisa os autores realizam uma extensa mineração de dados para analisar as relações entre as linguagens de programação e o número de defeitos nos projetos de software livre. Assim como nesta pesquisa, os autores tiveram a necessidade de apenas comparar projetos que fossem de um mesmo domínio de aplicação já que o domínio poderia influenciar o número de defeitos dos projetos. Por isso, eles aplicaram o LDA para categorizar os projetos de acordo com o domínio de aplicação e assim eliminar essa possibilidade de interferência nos resultados.

3.10 Método de particionamento dos grupos de projetos

Conforme dito anteriormente, só podemos definir qual o nível de dívida técnica será o limiar entre um projeto considerado de produtividade ótima ou produtividade afetada, após termos obtidos os dados dos projetos. Com essa etapa de definição do método de particionamento dos grupos para o modelo concreto será realizada apenas no estudo de caso do capítulo Capítulo 4.

Capítulo 4

Estudo de caso

4.1 Introdução

Realizaremos um estudo de caso múltiplo para avaliarmos o nosso modelo concreto de estimação dos juros da dívida técnica em projetos de software livre descrito no Capítulo 3. Com esse estudo de caso observaremos a adequação do modelo na avaliação dos juros em projetos reais hospedados em uma plataforma pública de versionamento de software. Adicionalmente, descreveremos a ferramenta GitResearch. Essa ferramenta foi criada para automatizar partes das atividades realizadas durante o estudo de caso.

4.2 O repositório de software

No contexto do *open source*, tornar público o código-fonte de um projeto de software não se trata apenas de uma característica positiva. Isso é uma necessidade que caracteriza os princípios dessa filosofia de desenvolvimento de software. A comunidade interessada no software deve ter livre acesso a ele para que possa usá-lo, alterá-lo e distribuí-lo. Por conta dessas necessidades, houve uma popularização dos repositórios públicos de software. Esse tipo de plataforma fornece aos usuários meios para acessar o código de softwares de terceiros como também permite que eles mesmos disponibilizem seus projetos de software. A popularização dessas plataformas fez com que elas fossem a fonte de uma grande quantidade de dados. Essa diversificada e imensa quantidade de dados é produzida de diversas formas. Seja ela obtida por meio do próprio código-fonte desses projetos como também pela interação realizada entre os usuários. Em todos os casos, a facilidade de acesso e a pluralidade de dados faz com que os repositórios de software sejam uma opção interessante para a obtenção de dados para pesquisas. Em nosso estudo de caso, obteremos dados

do GitHub: Um repositório de projetos de software baseado na tecnologia versionamento Git.

4.2.1 Controle de versões utilizando o Git

Um sistema de controle de versões - ou configurações - é um sistema que fornece uma série de funcionalidades para controlar a evolução de um conjunto de arquivos. Entre essas funcionalidades está ter acesso às versões anteriores de arquivos, controlar quem realizou uma determinada alteração e comparar versões com o intuito de analisar as diferenças. Além disso, conforme Otte. [Ott09], esses sistemas têm sido utilizados como uma ferramenta de backup, pois permitem voltar a uma versão anterior caso algo esteja errado com a versão atual. O Git é um sistema moderno de controle de versão desenvolvido pelo também criador do Linux Linus Torvalds. Conforme Loeliger et al. [LM12], ele pode ser visto como uma evolução de sistemas mais antigos como o CVS [Ves06] e o Subversion [PCSF08].

O principal diferencial do Git em relação aos outros sistemas de versionamento é a sua característica distribuída. O Git foi projetado para funcionar muito bem em contextos onde exista uma grande quantidade de pessoas interagindo com o mesmo projeto e ainda assim, haja a necessidade que essa interação ocorra de uma forma organizada e rastreável. Essa característica distribuída é alcançada por meio de alguns recursos:

- **Facilidade para a realização de cópias de um repositório.** Qualquer pessoa no mundo pode realizar uma cópia local de um repositório de arquivos público gerenciado por uma instância do sistema Git. Para isso, basta instalar uma versão do Git no computador local e utilizar o comando *clone* seguido pela URL do repositório. A cópia obtida contém todo o histórico do projeto como também todos os arquivos necessários para que o versionamento possa ser realizado. Isso faz com que não exista obrigatoriamente um único repositório central de um determinado projeto. Cada cópia pode se comportar como uma nova cópia principal caso a cópia original seja perdida.
- **É extremamente eficiente comparado às alternativas anteriores.** Uma das razões para essa eficiência está na estratégia utilizado para realizar o controle das versões dos arquivos. No Git existe uma pasta chamada *.git* na pasta onde estão os arquivos sendo gerenciados. Essa pasta contém todos os dados referentes ao histórico dos arquivos como também todas as informações necessárias para realizar o controle de versão. Dessa forma, o Git não precisa acessar a rede para buscar alguma informação necessária para o controle de versão. Todos os dados estão no próprio sistema de arquivos usado para armazenar os arquivos gerenciados.

Outra razão da eficiência do Git é a inexistência de um processo que necessariamente precisa estar em execução para que o controle de versão funcione. Isso era necessário nos sistemas anteriores como é o caso do Subversion. Nele havia um *daemon* responsável por realizar o versionamento de arquivos. No Git isso não ocorre. Ao invés disso, o Git fornece um conjunto de programas que ao serem executados conseguem realizar as alterações necessárias nos arquivos de controle de versionamento.

O funcionamento do Git é baseado em um grafo onde cada nó representa uma versão dos arquivos. Um exemplo dessa organização pode ser visto na Figura 4.1. Nela, um repositório foi iniciado pelo *commit* C1 e depois houve uma alteração por meio do *commit* C2. Nesse ponto, foram criadas duas novas *branches*. Uma *branch* é um fluxo independente de versões. Normalmente, a *branch* principal de um repositório Git é chamada de master. No exemplo apresentado, os *commit* C4 e C6 foram criados a partir do *commit* C2 da *branch* principal. Porém, tanto eles quanto o *commit* C2 são totalmente independentes e podem apresentar conteúdos diferentes. Ainda assim, essas linhas independentes de evolução, podem novamente serem combinadas por meio do que é chamado de *merge*. Na Figura 4.2 temos o resultado do *merge* realizado entre as *branches* master e a *branch* 1. A versão obtida após o *commit* C8 contém tanto as alterações que foram realizadas nos *commits* C4 e C5 quanto as alterações realizadas no *commit* C3. Entretanto, essa operação não traz nenhum impacto à *branch* 2. A criação de *commits*, *branches*, *merges* e todas as outras ações necessárias para o versionamento dos arquivos em um repositório Git são realizado por meio de comandos que esse sistema disponibiliza. Para extrairmos as informações dos projetos analisados nesta pesquisa, utilizaremos alguns desses comandos. Na Tabela 4.1 há uma lista com os principais deles e quais as suas funções.

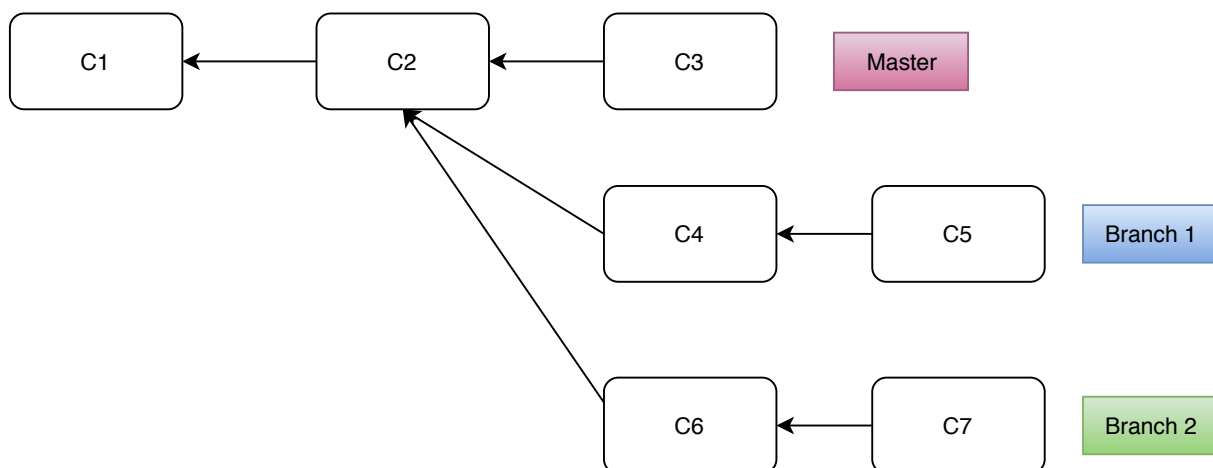


Figura 4.1: Exemplo da estrutura de armazenamento de versões do Git.

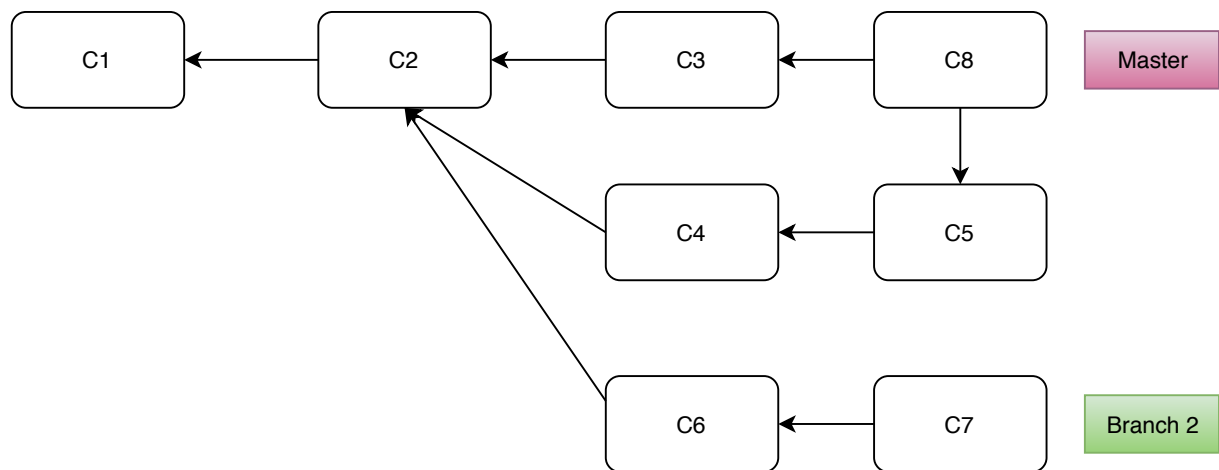


Figura 4.2: *Exemplo de merge entre duas branches.*

Comando	Descrição	Exemplo
clone	Realiza uma cópia local de um repositório remoto	<code>clone https://github.com/torvalds/linux.git</code>
add	Inclui um ou mais arquivos no gerenciamento de versão	<code>add file.txt</code>
commit	Efetiva no histórico as mudanças realizadas	<code>commit -m "History message"</code>
push	Envia as alterações locais para um repositório remoto	<code>git push -u origin branch</code>
pull	Atualiza os arquivos locais com base em um repositório remoto	<code>git pull</code>
merge	Realiza o merge entre duas <i>branches</i>	<code>merge branch1</code>
branch	Cria uma nova branch	<code>branch branch3</code>
checkout	Muda o diretório de trabalho atual para uma determinada branch	<code>checkout branch3</code>

Tabela 4.1: *Comandos básicos do GIT.*

4.2.2 GitHub

O GitHub é uma plataforma online para o armazenamento de repositórios Git. O que diferencia o GitHub de um servidor Git convencional é o conjunto de funcionalidades adicionais que a plataforma disponibiliza aos seus usuários. Essas funcionalidades são adições aos recursos de versionamento que o Git já disponibiliza como também recursos sociais que permitem a interação entre os usuários. De acordo com Dabbish et al. [DSTH12] essas funcionalidades sociais facilitam a colaboração entre os usuários pois viabilizam que eles monitorem as atividades de uma grande

quantidade de projetos além de gerar uma grande quantidade de dados. Esses dados, publicamente disponíveis, tem sido usados ativamente como uma fonte de dados para pesquisas científicas. Entretanto, algumas particularidades dessa plataforma precisam ser devidamente consideradas para que elas não prejudiquem os resultados dessas pesquisas. Na literatura podemos encontrar algumas recomendações práticas de como evitar problemas durante a utilização de dados do GitHub em pesquisas científicas [KGB⁺14, BRB⁺09, TDH14]. Algumas das recomendações que foram consideradas nesta pesquisa durante a fase de extração de dados do GitHub são:

- **Nem sempre um repositório no GitHub é utilizado para armazenar um projeto de software.** Apesar de ser voltado para a hospedagem de projetos de software, o GitHub não impõe nenhum tipo de restrição ao conteúdo dos repositórios que ele armazena. Por ser uma plataforma gratuita, isso faz com que muitos usuários o utilizem para armazenagem de arquivos diversos. Para evitar a inclusão de repositórios que não armazenam projetos de software, utilizamos uma série de heurísticas. Algumas delas sugeridas por Killiamvakou et al.[KGB⁺14] e outras criadas especificamente para esta pesquisa.
- **Muitos projetos não são totalmente desenvolvidos no GitHub.** Esses desenvolvimento fora do GitHub pode ser de duas formas: prévio e paralelo. Um projeto pode ser desenvolvido previamente em uma outra plataforma e depois ser migrado para o GitHub. Essa migração pode ser feita importando todo o histórico do projeto como também, esse histórico pode ser ignorado. Isso precisa ser considerado já que nesses casos esse histórico anterior pode ser relevante para a pesquisa sendo realizada. A outra forma de desenvolvimento fora do GitHub é quando algumas das atividades necessárias para o desenvolvimento do projeto não são realizadas no GitHub ou são realizadas parcialmente. Um exemplo seria o gerenciamento de *issues* que pode ser feito no GitHub ou em outro serviço como o FindBugs[AMP⁺07]. Assim, uma pesquisa poderia, equivocadamente, realizar conclusões com base em dados incompletos. O caso em que poderia haver algum impacto nos resultados desta pesquisa é o de desenvolvimento prévio. Caso um projeto fosse criado no GitHub sem que o histórico fosse importando, nossas análises de produtividade não poderiam ser feitas utilizando essa contribuição prévia. Para evitar esse problema, utilizamos, novamente, um conjunto de heurísticas baseadas nas datas de criação dos arquivos do repositório e nas datas dos primeiros *commits* realizados no GitHub. Se um repositório possuía um arquivo com uma data muito anterior ao primeiro *commit*, esse repositório provavelmente teve um desenvolvimento prévio fora do GitHub e, portanto, deveria ser excluído da pesquisa.

4.2.3 GHTorrent

O GitHub disponibiliza seus dados por meio de uma API. Nessa API é possível pesquisar informações a respeito de um repositório específico como também buscar repositórios utilizando critérios como linguagem, data de criação e descrição. Porém, devido a questões de segurança, essa API impõe severas restrições de acesso aos seus usuários. Existe uma quantidade pequena de requisições que podem ser realizadas por hora. Além disso, é necessário um esforço de desenvolvimento para que os dados obtidos possam ser armazenados e principalmente relacionados entre si. Devido a essas dificuldades, surgiram projetos como o GHTorrent, um banco de dados que espelha as informações contidas no GitHub. Esse projeto foi criado por Gousious et al. [GS12] para distribuir de uma forma simples e rápida as informações disponibilizadas pela API do GitHub. Ou seja, pesquisadores podem utilizar os dados no GitHub sem ter de desenvolver um sistema que faça a extração desses dados.

O GHTorrent disponibiliza os dados por meio de arquivos de backup do banco de dados MySQL, Big Tables do Google e por meio de um banco de dados online onde os usuários podem se cadastrar e realizar consultas. Escolhemos obter os dados por meio do download de um arquivo de backup do MySQL. Optamos por essa forma de acesso, pois queremos incluir uma grande quantidade de projetos em nosso estudo de caso. Por isso, teremos a necessidade de realizar a extração de dados com rapidez. Ao utilizar um backup completo do GHTorrent pudemos importá-lo em um ambiente no qual o poder de processamento fosse compatível com essas nossas expectativas de velocidade. O arquivo obtido foi o *mysql-2017-12-01*. Esse arquivo contém os dados obtidos até o último dia de dezembro de 2017. A Figura 4.3 contém uma lista com as principais tabelas encontrados no arquivo do GHTorrent e que foram utilizadas nesta pesquisa.

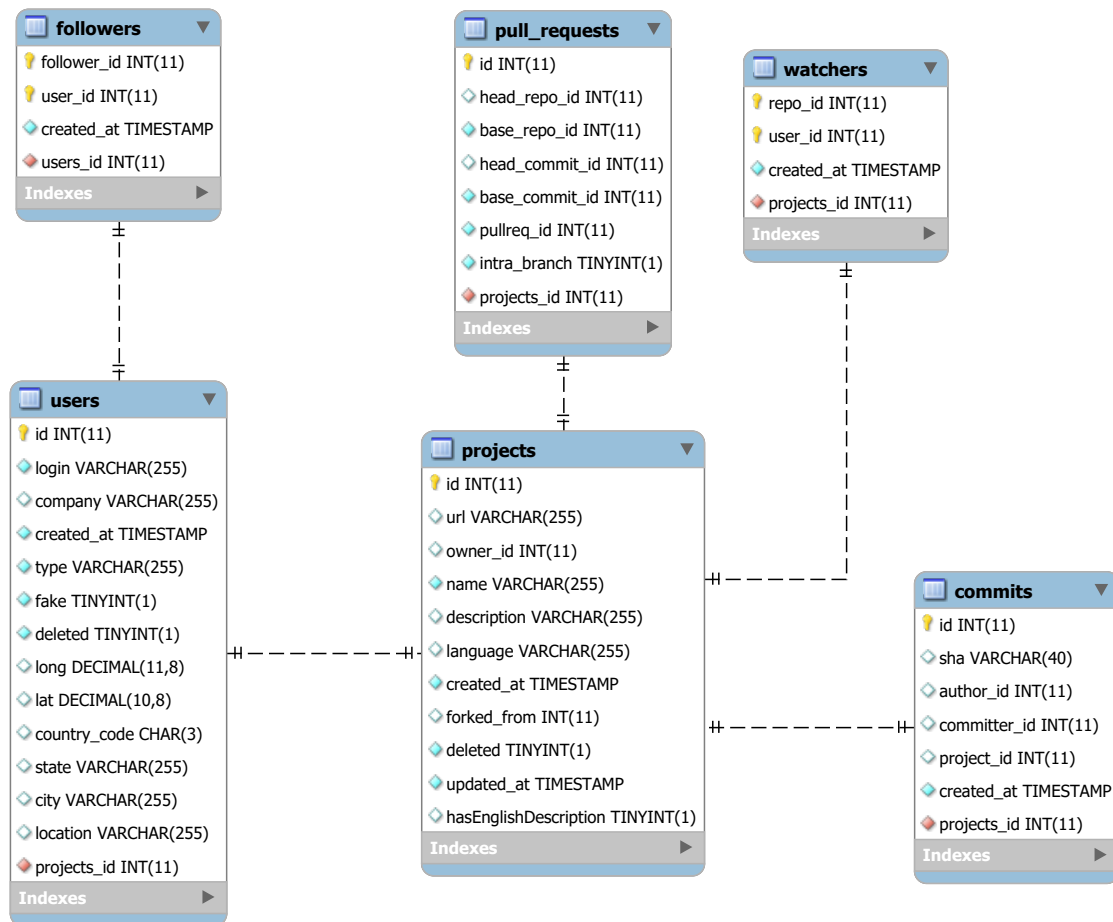


Figura 4.3: Tabelas do GHTorrent.

4.3 Ferramenta de apoio: GitResearch

Construímos uma ferramenta chamada GitResearch para automatizar uma parte das atividades realizadas neste estudo de caso. Essa ferramenta foi desenvolvida na linguagem Java e está disponível em <https://github.com/Jandisson/git-research>. Uma das razões para a construção dessa ferramenta é a de que ela permitirá que os resultados obtidos nesta pesquisa possam ser reproduzidos. Além disso, apesar de ter sido desenvolvida especificamente para esta pesquisa, ela foi planejada de forma que possa ser utilizada, com as devidas alterações, em outras pesquisas que envolvam a extração e o processamento de dados de repositórios de software como o GitHub. Durante a concepção dessa ferramenta, um dos requisitos almejados foi a possibilidade de processar uma grande quantidade de dados utilizando de forma satisfatória o hardware disponível. Isso foi alcançado ao realizar esse processamento de uma forma paralela. Ou seja, a ferramenta foi planejada de forma que as atividades em cada etapa da pesquisa pudessem ser realizadas em diversas linhas

de execução. Além disso, houve a preocupação em criar uma arquitetura flexível que facilitasse a alteração das funcionalidades existentes e adição de novas funcionalidades. Para alcançar todos esses objetivos, escolhemos, como a base para o GitResearch, o framework Spring Batch[CTGB11]. Utilizando esse framework vamos executar a tarefa de extração e processamento dos dados como um processo batch. De acordo com Martin. et al. [MSD⁺15], um processamento batch acontece quando casos semelhantes são processados simultaneamente sem a intervenção de um usuário. Escolhemos esse modelo de processamento pois ele se adequa as nossas necessidades de desempenho e condiz com as tarefas que serão realizadas no estudo de caso.

4.3.1 Arquitetura

O GitResearch é baseado na arquitetura do Spring Batch. Um resumo dessa arquitetura é apresentado na Figura 4.4. Um *Job* representa um processo batch. Cada *Job* possui um conjunto, que pode ser ou não ordenado, de *Steps* que por sua vez são as atividades que devem ser realizadas durante o processamento. Tanto um *Job* quanto um *Step* utilizam um repositório de dados chamado de *JobRepository*. Esse repositório armazena, principalmente, dados de controle a respeito de cada execução de um *Job*. Além disso, o repositório pode ser usado pelos *Steps* para armazenar quaisquer informações que sejam necessárias para realizar o processamento. Cada *Step* é formado por três objetos: *ItemReader*, *ItemProcessor*, *ItemWriter*. Esses três objetos são responsáveis respectivamente por ler, processar e armazenar cada item do processo batch. Cada item será um dos projetos a serem processados neste estudo de caso.

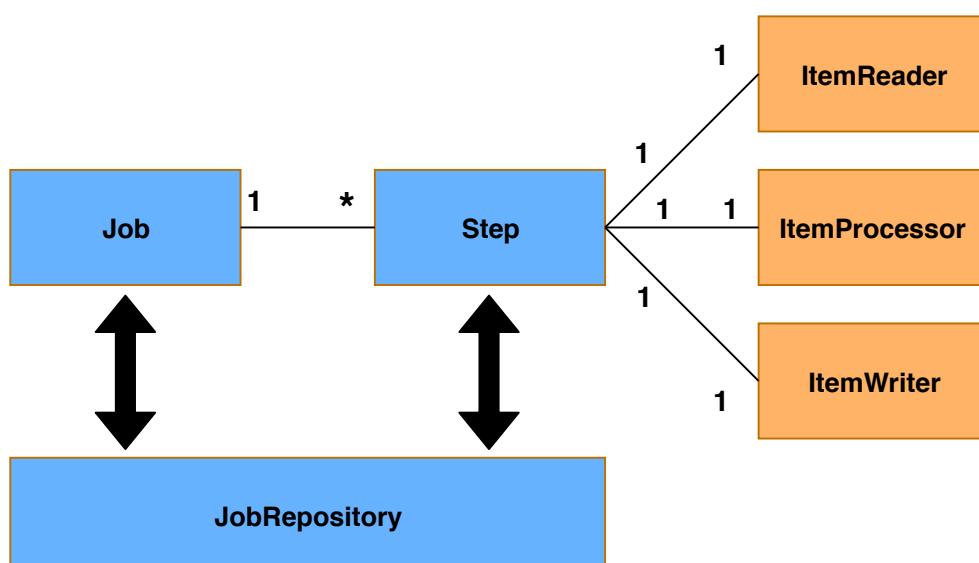


Figura 4.4: Arquitetura do framework Spring Batch. Adaptada de [Min11].

Para a utilização do framework Spring Batch, basicamente, são necessários os seguintes passos:

1. **Configuração do JobRepository.** Pode ser usado qualquer banco de dados que tenha suporte ao JDBC[Ree00]. Em nosso estudo de caso, utilizamos um banco de dados de código livre chamado Derby[Mei10]. Ele foi escolhido por ser de fácil configuração e não precisar ser instalado. O banco de dados e gerenciador do banco de dados são armazenados em um único arquivo. Isso facilitará a reprodução do estudo de caso e, conseqüentemente, a validação dos resultados obtidos.
2. **Criação e configuração do Job.** A configuração de um *Job* consiste em indicar quais *Steps* serão executadas e como essa execução deve ser feita. Nesse ponto é possível determinar qual será o fluxo de *Steps*, se haverá ou não paralelismos e a quantidade de recursos, como linhas de execução, que deverá ser utilizada. A Figura 4.5 apresenta o código de configuração do *Job* no GitResearch. Em nosso caso, foi necessária apenas a configuração de um *Job*.
3. **Implementação dos Steps necessários.** Essa implementação consiste na criação dos *ItemReader*, *ItemProcessor* e *ItemWriter*. Para um determinado *Step*, é possível que não seja necessário criar novos *ItemReader* ou *ItemWriter*. Ao invés disso, é possível utilizar algum objeto previamente criado e disponibilizado pelo framework. Foi isso que foi realizado no GitResearch. A maioria dos *Steps* utiliza algum *ItemReader* ou *ItemWriter* padrão do framework. Entretanto, no caso do *ItemProcessor* não há como utilizar um objeto padrão já que esse elemento é o responsável pela lógica de processamento e isso é particular para cada aplicação. Com isso, a implementação dos *Steps* já existentes no GitResearch como a implementação de novos *Steps*, irá sempre exigir a escrita de novos *ItemProcessor*. No GitResearch foi necessária a implementação de oito *Steps* conforme ilustrado na Figura 4.6. Conforme mostrado na Figura, a entrada para o processamento realizado pelo GitResearch é uma lista com todos os projetos que serão analisados. Ou seja, essa seleção dos projetos é uma atividade que não é realizada pelo GitResearch. Essa etapa será descrita no item 4.4.1.

```
import ...

@Configuration
@EnableBatchProcessing
public class JobConfiguration {

    @Autowired
    private JobBuilderFactory jobs;

    @Bean
    public Job extractDataJob(
        @Qualifier("downloadProjectRepository") Step download,
        @Qualifier("normalizeProjectDescriptionText") Step normalize,
        @Qualifier("prepareCorpus") Step prepareCorpus,
        @Qualifier("runLda") Step runLda,
        @Qualifier("runSonar") Step runSonar,
        @Qualifier("readSonar") Step readSonar,
        @Qualifier("calculateCollab") Step calculateCollab,
        @Qualifier("consolidateResults") Step consolidateResults

    ) {
        return this.jobs.get("extractDataJob")
            .start(download)
            .next(normalize)
            .next(prepareCorpus)
            .next(runLda)
            .start(readSonar)
            .start(calculateCollab)
            .start(consolidateResults)
            .build();
    }
}
```

Figura 4.5: Código responsável por configurar o Job no GitResearch.

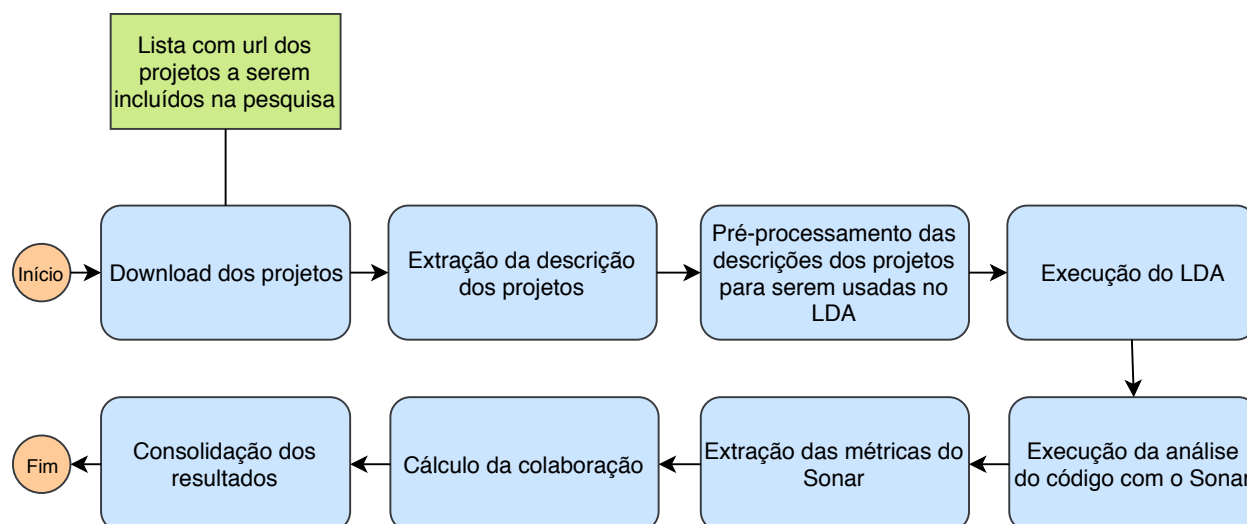


Figura 4.6: *Passos do processamento no GitResearch.*

4.4 Etapas do estudo de caso

Nosso estudo de caso foi realizado em seis etapas conforme ilustrado na Figura 4.7. Conforme dito anteriormente, todas essas etapas serão automatizadas ou semiautomatizadas por meio da ferramenta GitResearch. A única exceção é a etapa de seleção dos projetos a serem analisados. Comparando a Figura 4.6 e a Figura 4.7, podemos ver que há diferenças entre as etapas do GitResearch e as etapas do estudo de caso. Isso aconteceu porque algumas etapas do Estudo de caso tiveram de ser divididas no GitResearch em blocos menores. Essa divisão foi realizada tanto por uma questão de coesão quanto de desempenho. Um exemplo é a etapa 2 do estudo de caso. Ela consiste no agrupamento dos projetos que são semelhantes. Esse agrupamento foi realizado aplicando uma técnica de processamento de linguagem natural chamada LDA. Para aplicarmos essa técnica tivemos, antes, que encontrar a descrição do projeto e realizar alguns pré-processamentos. Com isso, a etapa 2 do estudo de caso gerou 3 etapas no GitResearch: extração da descrição, pré-processamento e execução do LDA. Na Figura 4.8 apresentamos um mapeamento entre as etapas do estudo de caso e as etapas de processamento no GitResearch. A seguir descreveremos a execução de cada uma das etapas do estudo de caso.

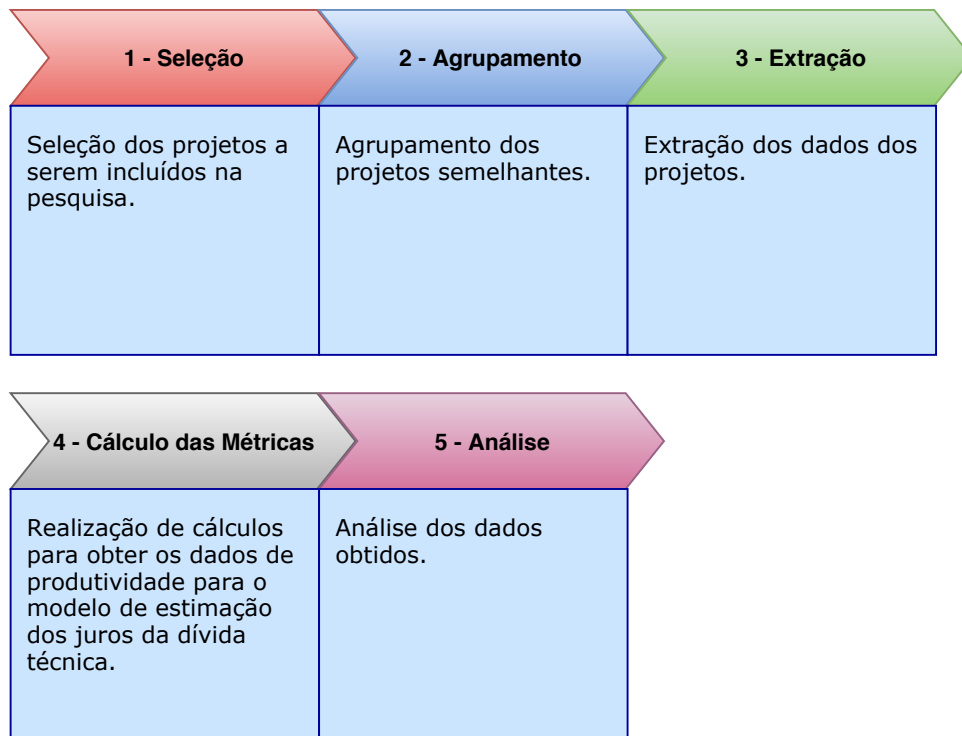


Figura 4.7: *Resumo das etapas do estudo de caso.*

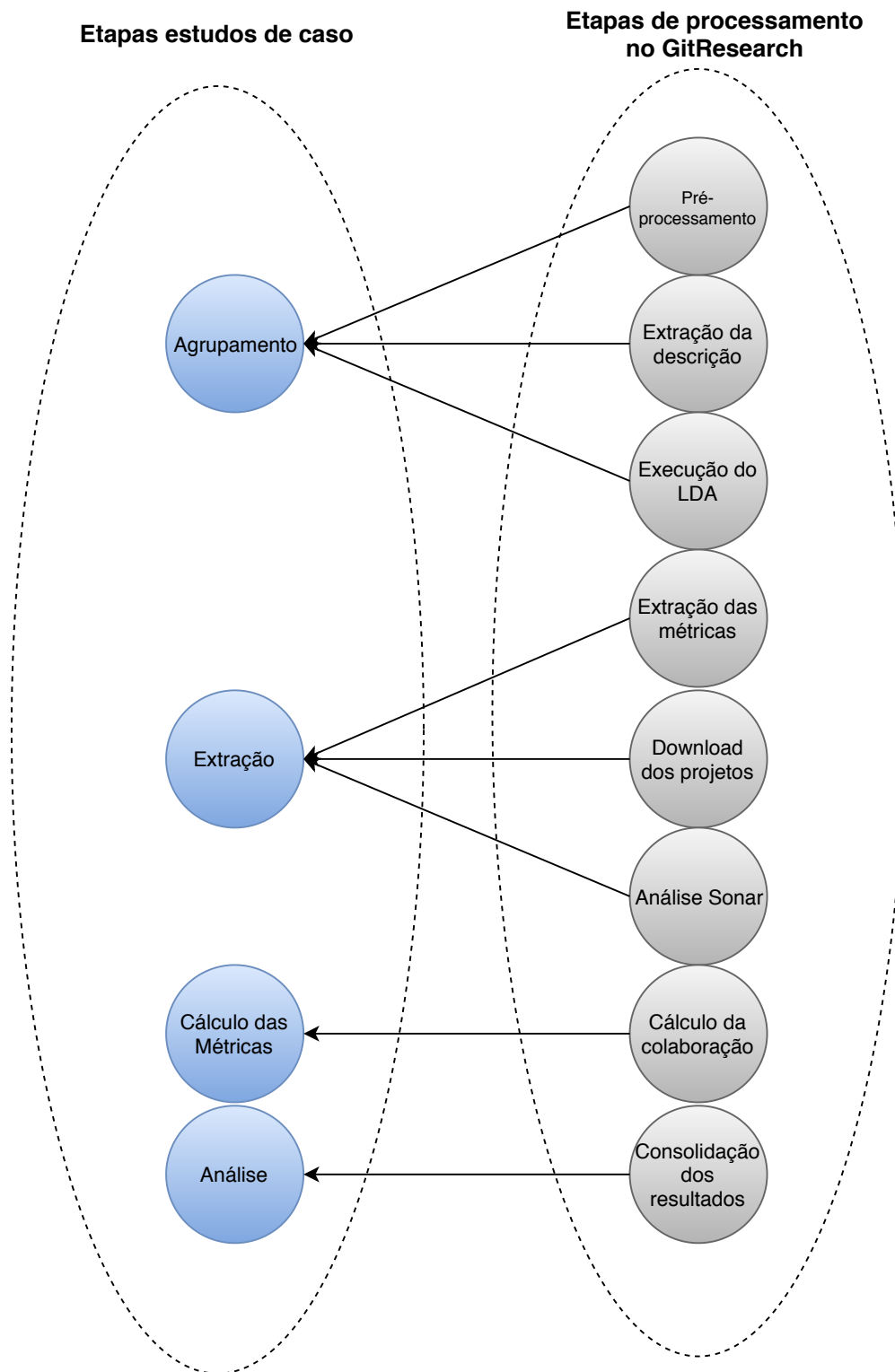


Figura 4.8: Mapeamento entre as etapas do estudo de caso e as etapas de processamento no GitResearch.

4.4.1 Seleção dos projetos

A seleção dos projetos a serem incluídos neste estudo de caso foi realizada utilizando um conjunto de regras de exclusão. Ou seja, inicialmente selecionamos todos os projetos armazenados no GitHub e depois essas regras de exclusão foram aplicadas fazendo com que o número de projetos

diminuísse. Essa estratégia foi usada porque nosso objetivo era incluir a maior quantidade de projetos possível e com isso diminuir a possibilidade de criação de algum viés caso fossem selecionados projetos específicos. Acreditamos que com isso, a amostra de projetos selecionados é uma representação estatisticamente relevante da população de projetos existentes no GitHub. As regras de exclusão foram definidas observando três grupos de aspectos da pesquisa:

1. **Restrições metodológicas.** Esse grupo de regras inclui aquelas criadas pela necessidade de excluir alguns projetos por eles não possuírem alguma característica que era obrigatória para a aplicação de algum aspecto da metodologia definida para a pesquisa. Um exemplo é a exclusão de projetos no qual a descrição não estava na língua inglesa. Essa regra foi criada porque a análise de projetos em múltiplas línguas não é compatível com a atividade de agrupamento de projetos semelhantes. Para essa atividade utilizamos uma técnica de processamento de linguagem natural chamada LDA. Essa técnica agrupa as descrições dos projetos utilizando como base o número de ocorrência de palavras. Se por exemplo, uma descrição contém diversas vezes as palavras *management* e *database*, o LDA irá criar um tópico com essas duas palavras e inserir nele todos os projetos que possuem também muitas ocorrências dessas duas palavras. Provavelmente, os projetos incluídos serão gerenciadores de banco de dados ou projetos relacionados. Entretanto, caso sejam incluídos projetos de gerenciadores de banco de dados no qual a descrição esteja em outro idioma, e as palavras *management* e *database* fossem escritas nesse idioma, esses projetos poderiam ser colocados em outro tópico. Ou seja, projetos de um mesmo domínio seriam agrupados em grupos diferentes apenas por causa da diferença no idioma da descrição. Por essa razão, foi criada uma regra que removeu todos os projetos que não estavam com a descrição em inglês. Foi escolhido o idioma inglês, pois a aproximadamente 70% dos projetos no GitHub estavam documentados em inglês.
2. **Restrições operacionais.** Nesse grupo de regras estão aquelas que foram criadas devido a limitações operacionais durante a realização do estudo de caso. Essas restrições, na maior parte das vezes, foram encontradas nas ferramentas utilizadas em algumas etapas da pesquisa. Um exemplo é a ocorrência de falhas durante a extração das métricas do código-fonte utilizando a ferramenta SonarQube. Alguns projetos não puderam ter seu código-fonte analisado por essa ferramenta mesmo após diversas tentativas e o acionamento do suporte da ferramenta. Com isso, esses projetos tiveram de ser excluídos já que uma extração manual dessa métricas era inviável devido ao tamanho dos projetos. Ainda nesse grupo de regras de exclusão estão aquelas que foram criadas devido a incompatibilidade com o sistema operacional utilizado.

Alguns arquivos tinham nomes muito extensos ou utilizavam caracteres que não eram aceitos pelo sistema operacional que foi utilizado para a execução do estudo de caso. Isso levou a exclusão dos projetos que continham os arquivos que geraram esses problemas.

3. Inclusão apenas de repositórios de software. O último grupo de regras é formado por aquelas que foram criadas para excluir repositórios que não sejam projetos de desenvolvimento de software. Conforme já dito no item 4.2.2, muitos dos repositórios criados no GitHub são usados apenas para armazenar arquivos. Para excluir esses repositórios nós utilizamos regras sugeridas pela literatura. Além disso, após a realização de análises preliminares da lista de projetos, também criamos nossas regras de exclusão. Com isso, um número substancial de repositórios foi removido da pesquisa.

A Tabela 4.2 contém uma lista com todas as regras de exclusão utilizadas para a seleção dos projetos. As regras estão ordenadas na tabela na ordem em que foram aplicadas. Nessa tabela fornecemos uma descrição da regra, indicamos se ela se trata de uma regra metodológica, operacional ou para verificar se o repositório é realmente um projeto de software. Além disso, informamos se a regra foi aplicada utilizando como base os dados no GHTorrent ou houve a necessidade de acessar o código-fonte do projeto. Por fim, informamos a quantidade de projetos que foram removidos por cada regra de exclusão.

É necessário detalhar a motivação de algumas das regras de exclusão criadas. Uma delas é a regra número 4 foi criada para evitar que projetos duplicados fossem incluídos na pesquisa. No GitHub, um *Fork* é um procedimento onde um usuário realiza uma cópia, para a sua conta, de um repositório [TBLJ13]. Esse procedimento é realizado quando esse usuário deseja contribuir com esse repositório original ou quando ele tem interesse em iniciar um novo projeto tendo como base esse repositório inicial. Em ambos os casos, esses dois repositórios serão muito parecidos. Logo, resolvemos remover esses repositórios que foram gerados por meio de um *Fork*. Outra regra que precisa ser esclarecida é a de número 7. Removemos os projetos que tinham menos de 6 colaboradores e um número de *commits* menor do que 516. Essa exclusão foi realizada com o objetivo de remover da pesquisa os repositórios que não são projetos de software. De acordo com Bird et al.[BRB⁺09], o número de colaboradores e o número de *commits* são uma forma eficaz de identificar repositórios que tenham sido criados apenas para a realização de testes, que sejam exemplos de código ou que sejam projetos particulares. De acordo com os autores, se um repositório teve uma quantidade mínima de *commits* e colaboradores, há uma probabilidade maior de que seja um projeto de software real. Os números mínimos utilizados nesta pesquisa foram obtidos por meio de análises dos projetos existentes no

GitHub. Foi encontrada uma média desses números em todo o conjunto de projetos do GitHub e a essa média foi adicionada uma margem de dois desvios padrão. Por fim, a regra 11 permitiu a exclusão de projetos com uma quantidade muito grande de arquivos para serem analisados. Essa regra foi criada por dois motivos. O primeiro foi a impossibilidade de o SonarQube processar alguns desses projetos. Mesmo após diversas tentativas e o acionamento do suporte da ferramenta, não pudemos processá-los. O segundo motivo é a observação de que repositórios com muitos arquivos normalmente não são projetos de software. Ao invés disso, eles são criados para armazenar logs e outros tipos de arquivos que não tem relação com o desenvolvimento de software.

No banco de dados do GHTorrent há um total de 72.433.097 repositórios. Após a aplicação das onze regras de exclusão listadas na Tabela 4.2, sobrou um total de 1.867 repositórios. Ou seja, neste estudo de caso analisaremos 1.867 repositórios de software.

#	Descrição	Tipo	Local	Projetos Excluídos
1	Exclusão de projetos abandonados, exemplos, exercícios, tutoriais e projetos android.	Software	GHTorrent	20.512.394
2	Exclusão de projetos que não utilizem a linguagem de programação Java.	Metodológica	GHTorrent	48.385.193
3	Exclusão de projetos deletados.	Software	GHTorrent	367.570
4	Exclusão de projetos que são <i>Forks</i> de outros projetos.	Metodológica	GHTorrent	1.637.704
5	Exclusão de projetos sem descrição no GHTorrent.	Metodológica	GHTorrent	595.557
6	Exclusão de projetos no qual a descrição não esteja em inglês.	Metodológica	Código-fonte	342.925
7	Exclusão de projetos que tenham poucos <i>commits</i> e poucos colaboradores.	Software	GHTorrent	588.909

8	Exclusão de projetos com nomes de arquivos incompatíveis.	Operacional	Código-fonte	126
9	Exclusão de projetos no qual o SonarQube não foi capaz de analisar o código.	Operacional	Código-fonte	54
10	Exclusão de projetos com arquivo de descrição inexistente ou muito pequeno .	Metodológica	Código-fonte	726
11	Exclusão de projetos muito grandes .	Operacional	Código-fonte	72

Tabela 4.2: Regras para a exclusão de projetos do estudo de caso

4.5 Agrupamento dos projetos

Nosso modelo de estimação dos juros da Dívida técnica é baseado na comparação das produtividades de projetos que possuam pouca dívida técnica e projetos que possuem um nível médio ou alto da dívida técnica. Entretanto, essa comparação será feita apenas entre projetos de um mesmo domínio de aplicação. Por exemplo, arcabouços de desenvolvimento como o Spring Batch[CTGB11], que foi usado para desenvolver a ferramenta GitResearch, só terá sua produtividade comparada com outros projetos que também sejam arcabouços de desenvolvimento ou estejam relacionados a arcabouços de desenvolvimento. Neste estudo de caso, realizaremos esse agrupamento dos projetos por domínio utilizando uma técnica de processamento de linguagem natural chamada *Latent Dirichlet Allocation* (LDA). Extrairemos de cada projeto a sua descrição e aplicaremos o LDA para identificar o seu domínio.

4.5.1 LDA

O LDA considera que um documento é formado por palavras que por sua vez estão relacionadas a um tópico. As palavras são associadas a um tópico automaticamente durante a análise do conjunto de documentos. Se uma palavra w_1 aparece em muitos documentos junto com uma palavra w_2 , o LDA assume que há uma relação semântica entre essas palavras e por isso elas devem ficar em um mesmo tópico. Um exemplo seriam as palavras *player*, *game* e *joystick*. Essas palavras normalmente

seriam muito encontradas nas descrições dos projetos relacionados aos jogos eletrônicos. Isso faria com o LDA criasse um tópico com essas e outras palavras relacionadas aos jogos e associasse esse tópico a todos os documentos no qual elas aparecem muitas vezes. O LDA baseia-se na ideia de que cada documento analisado foi gerado por uma combinação de $X\%$ de palavras de um tópico A , $Y\%$ de um tópico B e assim por diante. Sendo assim, dado um documento D , o LDA realiza o caminho inverso para identificar o quanto do tópico A foi usado para gerar o documento D , o quanto do tópico B foi usado para gerar o documento D e assim por diante. Em nosso estudo de caso esses documentos serão os textos com a descrição dos projetos. Consideraremos que dois projetos são similares se os tópicos mais presentes em cada um deles são iguais.

4.5.2 A aplicação do LDA

A primeiro item a ser definido antes da aplicação do LDA em um conjunto de documentos é a definição de quantos tópicos serão utilizados. Neste estudo de caso, utilizamos o número 30. Esse número foi escolhido após a realização de experimentos onde vários números foram testados. Com uma quantidade muito grande de tópicos percebemos que alguns deles ficaram com apenas um ou nenhum projeto. Quando um número muito pequeno de tópicos foi escolhido, percebemos que alguns dos tópicos gerados claramente misturavam mais de um assunto. Além disso, o número 30 foi escolhido por ser um número próximo do utilizados em outros trabalhos semelhantes como o de Ray et al. [RPFD14].

Para a execução do LDA foi necessário extrair a descrição de cada projeto. Essa descrição foi obtida do próprio repositório onde o projeto é armazenado. Isso pôde ser feita já que há uma padronização em relação ao nome do arquivo onde a descrição do projeto deve ser inserida. Esse arquivo fica localizado no diretório raiz do repositório e normalmente recebe o nome de *readme.md*. Entretanto, é possível que sejam usadas variações desse nome como *readme.txt* e *readme.html*. Quando nem o arquivo padrão nem alguma das variações foi encontrada, foi utilizado qualquer arquivo no qual o nome inicia com a palavra *readme*. Se mesmo assim, não foi encontrado nenhum arquivo, então o projeto foi excluído da pesquisa conforme a regra 10 da Tabela 4.2.

Após a extração da descrição de cada projeto, foi necessário realizar uma série de manipulações textuais nas descrições de tal forma que o elas pudesse ser utilizadas com o LDA. Essas manipulações incluem a remoção de palavras irrelevantes como preposições e artigos, remoção de marcações HTML e XML, a remoção de caracteres especiais e a redução de palavras para sua forma comum (*stemming* [J⁺11]). Essas modificações foram realizada no GitResearch por meio da classe *DescriptionNormalizer*. Cada uma das transformações foi escrita em uma classe individual e

inseridas no grupo padrão de transformações. Em outras pesquisas, será possível criar novos grupos de transformações. A Tabela 4.3 apresenta uma lista com todas as classes de transformação que foram utilizadas no pré-processamento das descrições.

Classe	Transformação
RemoveUrlTransformation	Remove todas as URL do texto
RemoveXmlTransformation	Remove todas as marcações HTML e XML
RemoveEspecialCharactersTransformation	Remove caracteres especiais
RemoveSpacesTransformation	Remove espaços múltiplos
RemoveNumbersTransformation	Remove todos os números
LowerCaseTransformation	Transforma todo o texto em minúsculo
RemoveStopWordsTransformation	Remove palavras como "the" e "and" que são irrelevantes para a categorização dos textos
RemoveLicenseWordsTransformation	Remove texto a respeito das licenças dos softwares

Tabela 4.3: Transformações realizadas no texto das descrições dos projetos

4.5.3 Resultado do LDA

A Tabela 4.4 apresenta todos os 30 tópicos e suas respectivas palavras. Conforme pode ser visto, as palavras estão em sua forma reduzida devido à aplicação do *stemming*. Um exemplo pode ser visto no Tópico 12. A primeira palavra desse tópico é a palavra "test". Entretanto, na verdade,

essa palavra representa todas as palavras que contém o prefixo “test”, como “testing”, “tests”, “testability” e assim por diante.

É importante destacar que as palavras estão sendo exibidas na tabela 4.4 ordenadas pelo seu nível de relevância dentro do tópico. Por isso, para identificar qual domínio cada tópico representa, a primeira palavra deve ser a mais decisiva. Existem alguns tópicos no qual as suas palavras permitem-nos claramente identificar o domínio que ele representa. Esse é o caso do tópico 22 que agrupará os projetos relacionados aos jogos eletrônicos. Outro exemplo é o Tópico 23 que agrupará projetos relacionados ao processamento distribuído de dados. Enquanto isso, alguns tópicos não nos permitem identificar qual domínio eles representam.

Nome	Palavras
Tópico 1	applic android fix xprivacy addon restrict support ad data devic forg app play improv processor return permiss mode googl map
Tópico 2	index elasticsearch cassandra data set search json type file query solr field document fs true default test map bin support
Tópico 3	build instal java run file jar directory packag ant sourc command window download compil path test linux document git bin
Tópico 4	connect client server user authent command default host session key configur ssh option password set login request winrm overther file
Tópico 5	query api custom item id storefront checkout respons field widget java chart content overrid v1 terasoluna type gfw public payment
Tópico 6	distribut export wicket includ govern inform encrypt law file h2o security requir build impli kind cryptograph applic country import condit
Tópico 7	docker servic run imag doc user contain cluster api creat meso aw compos configur karaf url command provid build jame
Tópico 8	stream sampl data event file configur messag parser log queue tnt4j activ xml defin field property string chronicl paramet entri

Tópico 9	applic develop servic manag web support data base system server framework user compon model build platform api integr document modul
Tópico 10	java report attribut set job user operationresult resourc servic row session method id column client jasperadmin organ request list server
Tópico 11	java class method public string type return annot builder final object map static xml gener interfac org api void person
Tópico 12	test driver tabl databas java run jdbc engin configur execut gwt sql default requir assert selenium junit user connect firefox
Tópico 13	issu contribut build develop sourc document code statu support list maven download badg open bug request mail doc api wiki
Tópico 14	java core imag question demo src main aima text link student answer ui search org jar library post bridgedb afc
Tópico 15	graph java data algorithm handlebar src script templat workflow input com- ment structur output languag code taverna jwetherel compil task yesworkflow
Tópico 16	file library format imag jar support data gnu gener includ sourc java public code free distribut form common term work
Tópico 17	click sdk select modul file elixir function tab run png screenshot button line true configur menu test open view raw
Tópico 18	matrix de jsf msf4j openbaton implement jersey opengl microservic chouett redisson build matrix4f baton microserv joml transform en dddlib ob1k
Tópico 19	test simul run remot worker coordin hazelcast start client command agent machin member predict file benchmark cflint java script map
Tópico 20	file set code support option creat time number work make note provid class chang type default configur user system includ

Tópico 21	run server databas instal test configur applic file build start deploy properti tomcat mysql user mvn default web db war
Tópico 22	game mod player minecraft team robot openmr block server main make item build play develop creat place piec gener asset
Tópico 23	build eclips plugin run test maven instal mvn gradl jar gradlew repository java file modul command id target clean org
Tópico 24	data model research analysi databas tool develop rif univers genom gener base public peptideshak comput health repository inform sourc design
Tópico 25	data hadoop spark cluster distribut index node oper read random algorithm process scala exampl deep learn model machin comput pipelin
Tópico 26	api modul schema swagger uri file gener json java document config client batch data library marklog corb javascript codegen option
Tópico 27	spring cloud googl java client code repository applic data servic maven io boot id build add core info pull sourc
Tópico 28	cach counter morphium neo4j info mongodb transact id xxx current data main iteratortest java object query public privat string wrong
Tópico 29	git master branch build tool repository commit pull yourkit clone server plugin push repo job merg upstream fork request instal
Tópico 30	android java studio app event weblog output gem api intern stanford read test statu connect result io ssl relex notif

Tabela 4.4: *Tópicos e suas palavras.*

É evidente que nosso agrupamento dos projetos por domínio utilizando apenas sua descrição jamais será totalmente preciso. Isso acontece tanto por imprecisões no LDA quanto por causa do conteúdo das descrições dos projetos. Em alguns casos, essa descrição não permite nem mesmo que um especialista em software consiga identificar qual o propósito do projeto e em qual domínio ele se

encaixa. Ainda assim, acreditamos que esse agrupamento aproximado possa ser utilizado de forma eficaz para aumentar a precisão do nosso modelo de estimação dos juros da dívida técnica. De qualquer forma, ao analisarmos os resultados finais do estudo de caso, iremos comparar os dados obtidos quando realizamos o agrupamento por domínio e quando não realizamos esse agrupamento.

4.6 Extração dos dados

Nesta etapa do estudo de caso, iremos extrair os dados dos projetos selecionados. Serão extraídas diversas métricas comumente utilizadas para a avaliação de projetos de software tais como quantidade de arquivos, linhas de código e complexidade. Além disso, nessa etapa iremos medir o nível de dívida técnica de cada um dos projetos. Tanto as métricas do software quanto a dívida técnica serão medidas utilizando a ferramenta SonarQube[CP13]. Algumas outras informações como popularidade e colaboração serão obtidas do GHTorrent. Todos os dados serão extraídos em cinco pontos diferentes da evolução do projeto.

4.6.1 O SonarQube

O SonarQube é uma ferramenta que permite verificar diversos aspectos relacionados à qualidade de um projeto de software. Essa verificação é feita por meio de uma análise estática do código-fonte onde são obtidas informações relevantes para a avaliação do nível de qualidade do projeto. O SonarQube é composto basicamente por dois módulos: uma aplicação para a internet e um *scanner*. O *scanner* é o responsável por realizar a análise estática do código-fonte. É possível analisar projetos escritos em diversas linguagens. A aplicação para a internet permite a visualização de relatórios a respeito dos dados obtidos pelo *scanner*. Nessa aplicação ficam armazenados os dados de todas as análises realizadas em um projeto. Neste estudo de caso foi utilizada a versão 7.1 do SonarQube.

Para que o *scanner* possa ser utilizado é necessário que um conjunto de configurações sejam definidas para cada projeto e versão sendo analisada. A Figura 4.9 mostra a função no GitResearch responsável por definir essas configurações. Dentre as configurações necessárias está o identificador único do projeto (*sonar.projectKey*), a versão (*projectVersion*), em qual pasta estão os códigos-fonte (*sonar.sources*), a url da aplicação para internet do SonarQube (*sonar.host.url*) e qual a linguagem utilizada no projeto (*sonar.language*).

```

import ...;

public class RunSonarProcessor {

    private void executeSonarScanner(
        Project project,
        String temporaryProjectFolder,
        int version){

        HashMap prop = new HashMap();

        prop.put("sonar.projectKey", String.format("Project%s",project.getId()));
        prop.put("sonar.projectName", String.format("Project%s",project.getId()));
        prop.put("sonar.projectVersion", String.valueOf(version));
        prop.put("sonar.projectBaseDir",temporaryProjectFolder);
        prop.put("sonar.sources",temporaryProjectFolder + File.separator + ".");
        prop.put("sonar.java.binaries", temporaryProjectFolder);
        prop.put("sonar.host.url", this.sonarUrl);
        prop.put("sonar.language", "java");
        this.scanner.execute(prop);
    }
}

```

Figura 4.9: Código responsável por criar a configuração para a execução do scanner do SonarQube.

A Tabela 4.5 contém as métricas que são disponibilizadas e que utilizamos nesta pesquisa. Essas métricas ficam disponíveis no SonarQube automaticamente após a execução do *scanner* em um projeto de software. Algumas dessas métricas são calculadas utilizando apenas o código-fonte da aplicação. Esse é o caso, por exemplo, da quantidade de arquivos, linhas de código, funções e classes. Entretanto, algumas das métricas disponibilizadas pelo SonarQube dependem de informações adicionais para serem calculadas. Esse é o caso da Métrica *VIOLATIONS*. Essa métrica contém a quantidade de violações a regras de qualidade que foram encontradas no código-fonte. Essas regras ficam armazenadas no SonarQube e podem ser alteradas pelo usuário. Neste estudo de caso utilizamos o perfil de regras padrão disponibilizadas pelo SonarQube para a análise de qualidade de projetos java. Esse perfil é chamado de *Sonar Way*[Ara12] e é baseado na metodologia de qualidade chamada *sqale*[Let12].

Métrica	Descrição
---------	-----------

CODE_SMELLS	Quantidade de trechos do código que contém uma violação a algum princípio fundamental da programação de sistemas[SSS14].
COGNITIVE_COMPLEXITY	Um índice que indica o quão difícil é entender o código-fonte do projeto[Cam17].
COMMENT_LINES	Quantidade de linhas de comentários
COMMENT_LINES_DENSITY	Quantidade de linhas de comentários/(Linhas de código + Quantidade de linhas de comentários) * 100
COMPLEXITY	Complexidade ciclomática do código-fonte[McC76].
DIRECTORIES	Número de diretórios.
DUPLICATED_LINES	Linhas duplicadas
DUPLICATE_LINES_DENSITY	Linhas duplicadas / total de linhas * 100.
DUPLICATED_BLOCKS	Número de blocos de códigos duplicados.
DUPLICATED_FILES	Número de arquivos duplicados.
FILES	Número de arquivos.
FUNCTIONS	Número de funções.

NLOC	Número de linhas que contenham pelo menos um caractere que não seja um espaço, tabulação ou parte de um comentário.
SQALE_DEBT_RATIO	Razão entre o tempo estimado para resolver a dívida técnica do projeto e o tempo estimado que foi gasto para desenvolver o projeto.
SQALE_RATING	Uma classificação de 1 até 5 relativa ao nível de dívida técnica do projeto. Projetos com menos dívida técnica recebem a nota 1.
STATEMENTS	Número de declarações no código-fonte.
VIOLATIONS	Número de violações das regras de qualidade definidas no SonarQube.

Tabela 4.5: *Métricas extraídas dos projetos.*

4.6.2 Medição da dívida técnica

Conforme descrevemos no Capítulo 1, existem diversos tipos de dívida técnica. Alguns desses tipos são de difícil medição devido às suas características subjetivas ou contextuais. A dívida técnica de tecnologia, por exemplo, acontece quando um projeto de software está utilizando uma tecnologia obsoleta. Porém, definir se uma tecnologia é ou não obsoleta é uma atividade normalmente subjetiva e sujeita às discordâncias. Já as dívidas técnicas de arquitetura dependem de uma análise contextual para serem medidas. Esse tipo de dívida é descrito como uma violação a algum princípio ou regra pré-definido para como deve ocorrer a interação entre os componentes de um software. Acontece que parte dessas regras são específicas para o contexto no qual o software é desenvolvido. Isso acontece seja pelas necessidades do software sendo desenvolvido ou por aspectos relacionados aos desenvolvedores. No caso das dívidas técnicas arquiteturais, existem dificuldades de medição tanto por características contextuais quanto por aspectos subjetivos já que a aderência ou não de uma determinada arquitetura às expectativas de um desenvolvedor é algo subjetivo conforme boa parte dos arcabouços de avaliação de arquiteturas como é o caso do ATAM[KKC00]. Dessa forma, medir certos tipos de dívida técnica é uma tarefa difícil e em alguns casos inviável de ser feita

automaticamente.

Neste estudo de caso tivemos que limitar os tipos de dívida técnica avaliados. Isso foi feito por causa do nosso objetivo de medir quantitativamente e automaticamente o nível de dívida técnica dos projetos. Além disso, essa limitação foi realizada considerando os tipos de dívida técnica que a ferramenta SonarQube consegue medir. Com isso, o nível de dívida técnica que consideraremos em um projeto não será igual ao nível real já que serão considerados apenas alguns tipos de dívida.

A medição da dívida técnica é feita pelo SonarQube usando as regras definidas no perfil de qualidade utilizado no projeto. O nível de dívida técnica de um projeto é equivalente à soma do tempo estimado para alterar seu código-fonte de forma que todas as violações a regras de qualidade fossem eliminadas. Ou seja, o SonarQube escaneia o código-fonte do software em busca de violações. A cada violação encontrada, ele adiciona uma quantidade de tempo ao tempo total necessário para eliminar todas as dívidas técnicas do projeto. Após todo o código-fonte ser escaneado, o SonarQube então armazena essa quantidade total de tempo. O nível de dívida técnica de um projeto é então calculado pela razão entre o tempo total estimado para desenvolver o software e o tempo total que seria gasto para eliminar todas as dívidas técnicas. Esse cálculo é armazenado na métrica `SQALE_DEBT_RATIO` conforme mostrado na Tabela 4.5. A Tabela 4.6.2 apresenta algumas das regras de qualidade definidas no perfil padrão do SonarQube para a análise de projetos Java (*Sonar Way*) e a quantidade de minutos estimados para eliminar cada uma das ocorrências de violações de cada regra.

Forneceremos um exemplo de cálculo da dívida técnica utilizando o SonarQube. Para isso, analisaremos 2 projetos conforme a Tabela 4.6.2. O primeiro projeto tem 45.600 minutos de dívida técnica. Isso significa que após o SonarQube analisar todo o projeto, ele encontrou uma quantidade de dívidas que, somadas, levariam 45.600 minutos para serem resolvidas. Além disso, o primeiro projeto possui 80.000 linhas de código-fonte. Com isso, o nível de dívida técnica do projeto 1 é calculado como $\frac{45.600}{80.000 * m}$ sendo m a quantidade de tempo que se estima necessária para escrever uma linha de código. O SonarQube usa como padrão 30 minutos para a variável m . No caso do projeto 1, o nível de dívida técnica foi calculado como 1,9%. Isso significa que seriam gastos para eliminar a dívida técnica desse projeto 1,9% do esforço necessário para construir o projeto inteiro. De acordo com Jesus et al. [dJdM17], um índice abaixo de 3% é considerado baixo. Ou seja, o projeto 1 tem um nível baixo de dívida técnica. Já o projeto 2, conforme a Tabela 4.6.2 possui a mesma quantidade de dívida técnica medida em minutos. Entretanto, medido em número de linhas de código, o projeto 2 é muito menor do que o projeto 1. Isso fez com que o nível de dívida técnica do projeto 2 fosse calculado como 11,7%. Ou seja, muito maior do que o projeto 1. Com isso,

mesmo tendo o mesmo volume de dívida técnica que o projeto 1, de acordo com a estratégia de medição adotada, o projeto 2 é considerado pior do que o projeto 1 em termos de dívida técnica. Isso acontece porque o SonarQube calcula o nível de dívida técnica de um projeto de uma forma proporcional ao seu tamanho.

Projeto	Dívida técnica em minutos	Linhas de código	Cálculo	<i>Debt Ratio</i>
1	45600	80000	$\frac{45600}{80000*m}$	1,9 %
2	45600	13000	$\frac{45600}{80000*m}$	11,7 %

Tabela 4.6: Exemplo de cálculo da dívida técnica com o SonarQube. A variável m representa o tempo necessário para escrever uma linha de código. O valor padrão no SonarQube é de 30 minutos. Adaptado de [dJdM17].

Categoria	Descrição	Minutos
Mutabilidade	Bloco de código duplicado.	60
Manutenção	Variável não usada.	10
Testabilidade	Complexidade ciclomática [McC76] maior do que 10.	11
Reusabilidade	Parâmetro usado como seleção em um método público.	15
Confiabilidade	A condição de um laço nunca será verdadeira.	10
Segurança	Comandos sendo enviados para o sistema operacional sem nenhuma validação	30
Portabilidade	Uso de métodos descontinuados.	15
Manutenção	Existência de código-fonte comentado.	5

Tabela 4.7: *Exemplos de regras para identificação de dívidas técnicas no SonarQube.*

4.6.3 Limitações na medição da dívida técnica

É possível identificar algumas limitações em relação a forma como o SonarQube realiza a medição do nível de dívida técnica de um projeto. A primeira delas é a falta de uma verificação empírica para a quantidade de minutos necessários para corrigir cada dívida. Por exemplo, de acordo com a Tabela 4.6.2, remover um bloco de código duplicado levaria 60 minutos. Porém, não é possível encontrar na documentação da ferramenta nenhuma indicação de como esse valor foi definido. Com isso o tempo médio para refatorar o código-fonte e resolver esse problema pode ser

muito maior ou muito menor do que 60 minutos. Outro ponto questionável é a lista de regras que são usadas para identificar dívidas técnicas na configuração padrão do SonarQube. Não há também na documentação da ferramenta qualquer menção a qual metodologia foi utilizada para criar essa lista. Inclusive, a capacidade de identificar dívidas técnicas por meio de algumas das regras utilizadas são claramente questionáveis.

Apesar das limitações, o SonarQube tem sido usado efetivamente para a análise da dívida técnica em projetos de software, inclusive no contexto de pesquisas científicas. Uma das evidências da eficácia dessa ferramenta foi obtida por Marek et al.[SWR12]. Nesse trabalho os autores realizaram um *survey* com especialistas para que eles analisassem e indicasse o nível de dívida técnica de alguns projetos. Os dados obtidos foram então comparados com a medição realizada pelo SonarQube. As diferenças entre os dois resultados foram muito pequenas. Isso trouxe evidências de que a dívida técnica medida pelo SonarQube é tão precisa quanto aquela medida por especialistas. Além disso, uma pesquisa realizada por Fontana et al.[FRZ16b] comparou algumas ferramentas que são capazes de calcular a dívida técnica de um projeto. Os autores chegaram à conclusão de que o SonarQube é a ferramenta mais precisa atualmente disponível. Apesar disso, conforme apontado por Fontana et al.[FRZ16a], o índice de dívida técnica calculado pelo SonarQube não é apropriado para a análise individual de projetos. Ao invés disso, esse índice deve ser usado apenas para comparar projetos diferentes.

4.6.4 Divisão temporal do código

Todas as métricas obtidas do código-fonte dos projetos foram extraídas em 5 pontos diferentes na evolução do software. Para realizar essa divisão utilizamos a quantidade de *commits* de cada projeto. Por exemplo, se um projeto tem 1.000 *commits*, fizemos a extração das métricas quando o projeto tinha apenas 200, depois quando tinha 400,600,800 e finalmente 1.000 *commits*. A Figura 4.10 exibe a porção de código no GitResearch responsável pela contagem dos *commits* de um projeto. É possível notar que não basta apenas contar a quantidade de *commits* já que, como mostrado na Figura 4.2, é possível que um *commit* esteja envolvido em algum *merge* e por isso possua mais de um pai. Para resolver esse problema e podermos navegar corretamente no grafo de *commits*, nós consideramos apenas o primeiro pai encontrado conforme pode ser visto na linha 21 da Figura 4.2. O código responsável por executar o SonarQube em cada projeto é exibido na Figura 4.11.

```
1 import ...
2
3 public class RunSonarProcessor {
4
5     private int countNumberOfCommits(
6         Repository repository,
7         String defaultBranchName
8     ){
9
10         logger.info("Counting the number of commits.");
11         int numberOfCommits = 0 ;
12         RevWalk walk = new RevWalk(repository);
13         RevCommit head = walk.parseCommit(
14             repository.findRef(defaultBranchName).getObjectId()
15         );
16
17         while (head != null) {
18             numberOfCommits++;
19             RevCommit[] parents = head.getParents();
20             if (parents != null && parents.length > 0) {
21                 head = walk.parseCommit(parents[0]);
22             } else {
23                 head = null;
24             }
25         }
26
27         return numberOfCommits;
28     }
29 }
30 }
31
```

Figura 4.10: Código responsável por contar a quantidade de commits de cada projeto.

```

1  import ...
2
3  public class RunSonarProcessor {
4
5      @Override
6      public Project process(Project project) throws Exception {
7
8
9          String defaultBranchName = getDefaultBranch(gitDirectory);
10         Git git = Git.open(gitDirectory);
11         numberOfCommits = countNumberOfCommits(
12             git.getRepository(), defaultBranchName
13         );
14         frameSize = numberOfCommits / this.frameNumber // frameNumber = 5;
15         for(int i = 0 ; i < this.frameNumber ; i++)
16         {
17             try {
18                 git.checkout().
19                     setCreateBranch(false).
20                     setName(defaultBranchName).
21                     setForce(true).call();
22
23                 ObjectId Myhead = git.getRepository().
24                     resolve("HEAD~" +
25                         (frameSize * (this.frameNumber - i - 1)));
26                 git.checkout().
27                     setForce(true).
28                     setName(Myhead.getName()).
29                     call();
30
31                 this.initializeSonar();
32                 executeSonarScanner(project, projectDirectory, i + 1);
33             } catch (Exception e)
34             {
35                 logger.error("Fail to execute Sonar: " + e.getMessage());
36             }
37         }
38
39         return project;
40     }
41 }
42

```

Figura 4.11: Código responsável por executar o scanner do SonarQube em cada projeto.

4.7 Cálculo das variáveis de entrada do modelo

Conforme descrito anteriormente, consideraremos a colaboração realizada em um projeto como a entrada em nossa estratégia de avaliação de produtividade. Se um projeto tem muita contribuição

e pouca evolução, esse projeto será considerado improdutivo. Da mesma forma, se um projeto tem pouca contribuição, mas tem muita evolução, esse projeto será considerado produtivo. Conforme o Capítulo 3, a colaboração de um projeto será avaliada por meio de um índice que será calculado conforme a equação 4.1. O valor $A(u)$ representa a assiduidade com o qual o colaborador u contribuiu para o projeto r . Já o valor $Q(u)$ representa a qualidade do colaborador u . o Conjunto $u(R)$ contém todos os colaboradores que contribuíram no projeto r . A seguir descreveremos como foi calculado o I_c de cada projeto.

$$I_c(r) = \sum_{u \in u(R)} A(u) * Q(u) \quad (4.1)$$

4.7.1 Qualidade de colaboração

Utilizamos a estratégia descrita no item 3.7.1 para calcular a qualidade de cada colaborador. Basicamente, para avaliar a qualidade realizamos uma análise no grafo com os relacionamentos entre colaboradores. Um colaborador será considerado de alta qualidade se ele possui muitos seguidores, contribuiu com projetos relevantes e colaborou com outros colaboradores de qualidade. O cálculo de colaboração foi implementado na ferramenta GitResearch. A Figura 4.12 mostra uma das principais funções implementadas. Nela é realizado o cálculo do *pagerank* tanto dos colaboradores quanto dos projetos. As outras partes da estratégia foram implementadas com o auxílio dos dados disponibilizados pelo GHTorrent. A complexidade dessa implementação está na imensa quantidade de relacionamentos que precisam ser avaliados. Por exemplo, a relação seguir e ser seguido de todos os colaboradores que contribuíram com os projetos analisados gerou um grafo de aproximadamente 2 milhões de nós. Com isso, foi necessária a utilização de um hardware potente para que esses relacionamentos fossem calculados em um tempo viável para a realização do estudo de caso.

```

1 ...
2 public void calculatePageRank(nodes)
3 {
4     double dumpFactor = 0.85;
5     // Setting default rank
6     nodes.forEach((k, v) -> v.setRank(1 - dumpFactor/nodes.size()));
7     // First node to visit
8     Node node = nodes.entrySet().iterator().next().getValue();
9     double nodeRank = 0;
10    int totalInteractions = 0 ;
11    while(true) {
12        double followersRank = 0;
13        for (Node follower : node.getIn()) {
14
15            followersRank += follower.getRank()/follower.getOut().size();
16        }
17        nodeRank = node.getRank()+(dumpFactor* followersRank);
18        node.setRank(nodeRank);
19        if(Math.random() < dumpFactor && user.getOut().size() > 0)
20        {
21            node = walk(node);
22        }else{
23            node = randomWalk(nodes);
24        }
25        totalInteractions++;
26        if(totalInteractions > 70_000_000)
27        {
28            System.out.println("Max interactions");
29            break;
30        }
31    }
32 }
33
34 private static Node randomWalk(Map<Integer,Node> nodes)
35 {
36     Random random = new Random();
37     List<Integer> keys = new ArrayList<Integer>(nodes.keySet());
38     Integer randomKey = keys.get( random.nextInt(keys.size()) );
39     Node node = nodes.get(randomKey);
40     return node;
41 }
42
43 private static Node walk(Node node)
44 {
45     Random random = new Random();
46     return node.getOut().get(random.nextInt(node.getOut().size()));
47 }
48 }
49
50 }

```

Figura 4.12: Código responsável por calcular o pagerank.

4.7.2 Assiduidade

Analisando o banco de dados de commits disponibilizado pelo projeto GHTorrent até o mês de dezembro de 2017 pudemos calcular que a média de commits que um colaborador faz por dia em um mesmo projeto. O resultado desse cálculo foi: 1,16. Com isso, a assiduidade de um colaborador será calculada utilizando a equação 4.2. Nessa equação D_r representa a quantidade de dias que um projeto possui desde o primeiro *commit* até o último. Já a variável $N(u_r)$ representa a quantidade de *commits* que o colaborador u realizou no projeto r . Se um colaborador tem uma média diária de *commits* maior do que 1,16, sua assiduidade no projeto será maior do que 1. Caso contrário, sua assiduidade será menor do que 1. Quanto mais *commits* um colaborador fez em um projeto, maior será a sua assiduidade.

$$A(u_r) = \frac{N(u_r)}{1,16 * D_r} \quad (4.2)$$

4.8 Cálculo das variáveis de saída do modelo

O cálculo das variáveis de saída foi realizado com facilidade já que foram escolhidas métricas que puderam ser obtidas diretamente dos dados no GHTorrent ou da análise feita pelo SonarQube. Foram utilizadas três métricas: linhas de código, estrelas e *pull requests*. A quantidade de linhas de código foi obtida por meio do SonarQube. Conforme a Tabela 4.5, essa métrica é uma das que a ferramenta disponibiliza após a análise de um projeto. Já a quantidade de estrelas e *pull request* foram extraídas do GHTorrent.

4.9 Dados obtidos

- Descrição do banco de dados com as informações dos projetos.

4.9.1 Análise exploratória dos dados

- Tamanho
- Colaboração
- Dívida técnica
- Popularidade

4.10 Conclusão

Capítulo 5

Resultados e discussão

5.1 Contribuições

- Modelo abstrato de estimação dos juros da dívida técnica.
- Modelo concreto de estimação dos juros da dívida técnica no contexto dos projetos de software livre.
- Estudo de caso múltiplo para avaliação dos modelos
- Ferramenta de mineração de repositórios
- Banco de dados com informações a respeito dos projetos, métricas e divisão dos mesmo por domínio de aplicação.

5.2 Respostas as questões de pesquisa

Capítulo 6

Ameaças à validade

Possíveis problemas na seleção dos projetos

- É possível que projetos de software válidos e relevantes tenham sido excluídos indevidamente. Especialmente por causa da regra que elimina projetos com poucos *commits* e colaboradores.

Podemos identificar algumas dificuldades em realizar a estimação dos juro utilizando essa estratégia. A seguir destacamos algumas delas:

- A dificuldade em si de medir a produtividade no desenvolvimento de software.
- Identificar qual a **produtividade ótima** para um projeto de software. Para isso, utilizaremos analogia. Ou seja, para um determinado projeto iremos identificar projetos semelhantes e com um baixo índice de dívida técnica.
- Identificar se um projeto é semelhante a outro. Na literatura existem algumas abordagens para realizar essa atividade. Iremos descrever essas abordagens e explicar como iremos adaptá-las nesse projeto.
- Isolar a interferência da dívida técnica nos projetos de software da interferência de outros fatores que estão prejudicando a produtividade. Ou seja, identificar e garantir que o que está fazendo um projeto não estar no cenário de **produtividade ótima** é exclusivamente a dívida técnica.
- Pode ser que o projeto tenha sido migrado de outro repositório ou outro sistema de versionamento. Isso precisa ser contornado de alguma forma. Uma alternativa seria verificar o tamanho do commit inicial. Preciso resolver os problemas com os commits. Alguns projetos como o vaadin não tem todos os commits na tabela commits. Alguns commits que vieram de pull requests por algum motivo não estão lá.

Descreveremos cada uma dessas dificuldades e como iremos resolvê-las ou diminuir sua interferência nos resultados desta pesquisa.

Preciso falar sobre o problema de empresas famosas como NetFlix possuírem um alto número de estrelas. Isso também acontece com profissionais que trabalham nessas empresas. Normalmente, eles também terão um grande número de seguidores. Isso obviamente nem sempre está relacionado à expertise dessas pessoas. Ao invés disso, eles podem ter um número alto de seguidores pois as pessoas normalmente se interessam pelas novidades dos projetos criados por essas empresas.

6.1 Escala

A literatura sugere que projetos maiores vão ter um prejuízo de produtividade devido à complexidade adicional trazida pelo seu tamanho. Realizar alterações em um projeto menor normalmente é mais fácil do que realizar alterações em projetos menores. Isso deveria ser considerado em um modelo como o nosso que tem como base uma estimação de produtividade.

6.2 Avaliação empírica das métricas utilizadas

Muitas métricas que utilizamos na pesquisa poderiam ser validas por meio de estudos empíricos. Um exemplo é o modelo de agregação utilizado para representar a colaboração nos projetos. Foram utilizados três aspectos: quantidade, assiduidade e qualidade. Entretanto, esse modelo não foi validade em nenhum momento. Não é possível garantir que esse modelo é adequado e realmente captura o nível de colaboração que um projeto teve.

6.3 Coisas a serem feitas caso haja tempo disponível

- Realizar um esforço maior para defender a utilização das linhas de código como uma métrica de tamanho do software. Posso buscar mais na literatura trabalhos que mostrem empiricamente o valor dessa métrica.

6.4 Imprecisão da popularidade

É possível que a popularidade do projeto seja influenciada significativamente quando ele é desenvolvido por uma empresa grande. Ou seja, a causa de um projeto ter um alto número de estrelas não necessariamente é devido a relevância que a comunidade dá ao projeto. Ao invés disso,

a popularidade do projeto pode ter sido obtida devido a popularidade da empresa. Um exemplo é o caso do projeto blitz4j que possui 504 estrelas. Enquanto isso, o projeto tinylog possui apenas 142. Entretanto, uma pesquisa em sites de buscas revelam que o tinylog é 4 vezes mais citado que o blitz4j. Logo, a maior quantidade de estrelas do blitz4j não reflete a sua popularidade entre os desenvolvedores.

Capítulo 7

Conclusão

7.1 Trabalhos futuros

- Necessidade de complementação da pesquisa por meio de estudos qualitativos. Os resultados obtidos precisam ser explicados.
- Falar sobre as possibilidades de utilizar o modelo para prever a produtividade futura dos software baseado na estimativa de crescimento da dívida técnica.

7.1.1 Popularidade vs Dívida Técnica

Poderia ter um capítulo falando sobre as relações entre a atratividade de um projeto (quantas pessoas procuram colaborar com ele) e a dívida técnica. A existência de muita dívida técnica afasta novos colaboradores?

Nos dados obtidos dos projetos do piloto é possível notar que os projetos com mais pull requests possuem menos dívida técnica.

Um assunto a ser abordado poderia ser a relação entre a quantidade de forks e a quantidade de pull requests. É possível nota no pilo que alguns projetos tem uma quantidade de forks muito maior do que a quantidade de pull requests. Ou seja, muitas pessoas fizeram uma cópia do projeto com a interação de realizar alterações mas, muitas delas não o fizeram.

7.1.2 Artigo CBI

Pode ser que a relação entre dívida técnica e linhas de código seja inversamente proporcional. Ou seja, quanto mais dívida técnica um projeto tem, mais linhas de código ele tem também. Falar sobre isso seria uma boa oportunidade para inserir o artigo apresentado na CBI e o Jornal.

Referências Bibliográficas

- [AB13] Esra Alzaghouli 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](#)
- [AMP⁺07] Nathaniel Ayewah, J David Morgenthaller, John Penix, William Pugh e YuQian Zhou. Using findbugs on production software. 2007. [64](#)
- [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](#)
- [Ara12] Charalampos Arapidis. *Sonar Code Quality Testing Essentials*. Packt Publishing Ltd, 2012. [83](#)
- [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. [31](#), [44](#)
- [BNJ03] David M Blei, Andrew Y Ng e Michael I Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003. [31](#)
- [BR10] Andrea Oliveira Soares Barreto e Ana Regina Rocha. Analyzing the similarity among software projects to improve software project monitoring processes. Em *Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the*, páginas 441–446. IEEE, 2010. [43](#)

- [BRB⁺09] Christian Bird, Peter C Rigby, Earl T Barr, David J Hamilton, Daniel M German e Prem Devanbu. The promises and perils of mining git. Em *2009 6th IEEE International Working Conference on Mining Software Repositories*, páginas 1–10. IEEE, 2009. [64](#), [74](#)
- [BTP⁺12] Kaushal Bhatt, Vinit Tarey, Pushpraj Patel, Kaushal Bhatt Mits e Datana Ujjain. Analysis of source lines of code (sloc) metric. *International Journal of Emerging Technology and Advanced Engineering*, 2(5):150–154, 2012. [54](#)
- [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](#)
- [Cam17] G Ann Campbell. Cognitive complexity-a new way of measuring understandability. Relatório técnico, Technical Report. SonarSource SA, Switzerland., 2017. [84](#)
- [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](#), [82](#)
- [CSS12] Bill Curtis, Jay Sappidi e Alexandra Szynekarski. 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](#)
- [CTGB11] Arnaud Cogoluegnes, Thierry Templier, Gary Gregory e Olivier Bazoud. *Spring Batch in Action*. Manning Publications Co., 2011. [67](#), [76](#)
- [CTNH12] Tse-Hsun Chen, Stephen W Thomas, Meiyappan Nagappan e Ahmed E Hassan. Explaining software defects using topic models. Em *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, páginas 189–198. IEEE, 2012. [45](#)
- [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](#)
- [dJdM17] Jandisson Soares de Jesus e Ana Cristina Vieira de Melo. Technical debt and the software project characteristics. a repository-based exploratory analysis. Em *2017 IEEE 19th Conference on Business Informatics (CBI)*, volume 1, páginas 444–453. IEEE, 2017. [xiii](#), [86](#), [87](#)
- [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](#)

- [DSTH12] Laura Dabbish, Colleen Stuart, Jason Tsay e Jim Herbsleb. Social coding in github: transparency and collaboration in an open software repository. Em *Proceedings of the ACM 2012 conference on computer supported cooperative work*, páginas 1277–1286. ACM, 2012. 63
- [EN16] Ramez Elmasri e Sham Navathe. *Fundamentals of database systems*. Pearson London, 2016. 4
- [FBHK05] Andrew Funk, Victor Basili, Lorin Hochstein e Jeremy Kepner. Application of a development time productivity metric to parallel software development. Em *Proceedings of the second international workshop on Software engineering for high performance computing system applications*, páginas 8–12. ACM, 2005. 55
- [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
- [FRZ16a] Francesca Arcelli Fontana, Riccardo Roveda e Marco Zanoni. Technical debt indexes provided by tools: a preliminary discussion. Em *2016 IEEE 8th International Workshop on Managing Technical Debt (MTD)*, páginas 28–31. IEEE, 2016. 89
- [FRZ16b] Francesca Arcelli Fontana, Riccardo Roveda e Marco Zanoni. Tool support for evaluating architectural debt of an existing system: An experience report. Em *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, páginas 1347–1349. ACM, 2016. 89
- [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
- [GS12] Georgios Gousios e Diomidis Spinellis. Ghtorrent: Github’s data from a firehose. Em *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, páginas 12–21. IEEE, 2012. 65
- [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. 31
- [HLCPSAL15] Adrián Hernández-López, Ricardo Colomo-Palacios, Pedro Soto-Acosta e Cristina Casado Lumberas. Productivity measurement in software engineering: a study of the inputs and the outputs. *International Journal of Information Technologies and Systems Approach (IJITSA)*, 8(1):46–68, 2015. 35
- [Hof00] Doug Hoffman. The darker side of metrics. Em *Pacific Northwest Software Quality Conference*, volume 17, página 2000, 2000. 54
- [HRGB⁺06] Israel Herraiz, Gregorio Robles, Jesús M González-Barahona, Andrea Capiluppi e Juan F Ramil. Comparison between slocs and number of files as size metrics for software evolution analysis. Em *Conference on Software Maintenance and Reengineering (CSMR'06)*, páginas 8–pp. IEEE, 2006. 55
- [HRWD10] Albert Hupa, Krzysztof Rzdca, Adam Wierzbicki e Anwitaman Datta. Interdisciplinary matchmaking: Choosing collaborators by skill, acquaintance and trust. Em *Computational social network analysis*, páginas 319–347. Springer, 2010. 47
- [HYW⁺17] Chaoran Huang, Lina Yao, Xianzhi Wang, Boualem Benatallah e Quan Z Sheng. Expert as a service: Software expert recommendation via knowledge domain embeddings in stack overflow. Em *2017 IEEE International Conference on Web Services (ICWS)*, páginas 317–324. IEEE, 2017. 48
- [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, 44
- [J⁺11] Anjali Ganesh Jivani et al. A comparative study of stemming algorithms. *Int. J. Comp. Tech. Appl*, 2(6):1930–1938, 2011. 77
- [JL97] R Jeffery e G Low. Function points and their use. *Australian Computer Journal*, 29(4):148–156, 1997. 42
- [KAAH11] Matthew B Kelly, Jason S Alexander, Bram Adams e Ahmed E Hassan. Recovering a balanced overview of topics in a software domain. Em *2011 IEEE 11th International Working Conference on Source Code Analysis and Manipulation*, páginas 135–144. IEEE, 2011. 45
- [Kac12] Tomek Kaczanowski. *Practical Unit Testing with TestNG and Mockito*. Tomasz Kaczanowski, 2012. 6
- [KB13] K Kalaiselvi e PS Balamurugan. An ontological approach to identify expert knowledge in academic institution. Em *2013 International Conference on Current Trends in Engineering and Technology (ICCTET)*, páginas 120–122. IEEE, 2013. 47
- [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, 64
- [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, 44

- [KKC00] Rick Kazman, Mark Klein e Paul Clements. Atam: Method for architecture evaluation. Relatório técnico, Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst, 2000. 85
- [KM04] Barbara Kitchenham e Emilia Mendes. Software productivity measurement using multiple size measures. *IEEE Transactions on Software Engineering*, 30(12):1023–1035, 2004. 30, 35, 40, 42
- [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. xi, 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
- [Let12] Jean-Louis Letouzey. The sqale method for evaluating technical debt. Em *Proceedings of the Third International Workshop on Managing Technical Debt*, páginas 31–36. IEEE Press, 2012. 83
- [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
- [LM12] Jon Loeliger e Matthew McCullough. *Version Control with Git: Powerful tools and techniques for collaborative software development*. ”O’Reilly Media, Inc.”, 2012. 53, 60
- [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
- [McC76] Thomas J McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, (4):308–320, 1976. 84, 88
- [Mei10] LIU Yu KONG Bo YANG Mei. Research on typical technologies of embedded database-berkeley db java edition and apache derby. *Microcomputer Information*, 32, 2010. 68
- [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
- [MH02] Audris Mockus e James D Herbsleb. Expertise browser: a quantitative approach to identifying expertise. Em *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, páginas 503–512. IEEE, 2002. 47
- [Min11] Michael Minella. *Pro Spring Batch*. Apress, 2011. xi, 67
- [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
- [MKCC03] Alan MacCormack, Chris F Kemerer, Michael Cusumano e Bill Crandall. Trade-offs between productivity and quality in selecting software development practices. *Ieee Software*, 20(5):78–85, 2003. 34, 55
- [MLVPG11] Collin McMillan, Mario Linares-Vasquez, Denys Poshyvanyk e Mark Grechanik. Categorizing software applications for maintenance. Em *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, páginas 343–352. IEEE, 2011. 44
- [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
- [MSD⁺15] Niels Martin, Marijke Swennen, Benoît Depaire, Mieke Jans, An Caris e Koen Vanhoof. Batch processing: definition and event log identification. RWTH Aachen University, 2015. 67
- [MSH08] Girish Maskeri, Santonu Sarkar e Kenneth Heafield. Mining business topics in source code using latent dirichlet allocation. Em *Proceedings of the 1st India software engineering conference*, páginas 113–120. ACM, 2008. 45
- [MSHZ15] Wenkai Mo, Beijun Shen, Yuming He e Hao Zhong. Geminer: Mining social and programming behaviors to identify experts in github. Em *Proceedings of the 7th Asia-Pacific Symposium on Internetware*, páginas 93–101. ACM, 2015. xi, 48, 50
- [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
- [MZ14] Tyler Munger e Jiabin Zhao. Automatically identifying experts in on-line support forums using social interactions and post content. Em *Proceedings of the 2014 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, páginas 930–935. IEEE Press, 2014. 48

- [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
- [Ott09] Stefan Otte. Version control systems. *Computer Systems and Telematics*, páginas 11–13, 2009. 60
- [P⁺15] Dmitrii Poliakov et al. A systematic mapping study on technical debt definition. 2015. xiii, 9, 10
- [Par92] Robert E Park. Software size measurement: A framework for counting source statements. Relatório técnico, Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 1992. 55
- [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, 48
- [PCSF08] C Michael Pilato, Ben Collins-Sussman e Brian W Fitzpatrick. *Version Control with Subversion: Next Generation Open Source Version Control*. "O'Reilly Media, Inc.", 2008. 60
- [Pet11] Kai Petersen. Measuring and predicting software productivity: A systematic map and review. *Information and Software Technology*, 53(4):317–343, 2011. 36
- [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
- [RD02] Matthew Richardson e Pedro Domingos. The intelligent surfer: Probabilistic combination of link and content information in pagerank. Em *Advances in neural information processing systems*, páginas 1441–1448, 2002. 50
- [Ree00] George Reese. *Database Programming with JDBC and JAVA*. "O'Reilly Media, Inc.", 2000. 68
- [Ros97] Jarrett Rosenberg. Some misconceptions about lines of code. Em *Proceedings Fourth International Software Metrics Symposium*, páginas 137–142. IEEE, 1997. 55
- [RPFD14] Baishakhi Ray, Daryl Posnett, Vladimir Filkov e Premkumar Devanbu. A large scale study of programming languages and code quality in github. Em *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, páginas 155–165. ACM, 2014. 57, 77
- [RR13] Romain Robbes e David Röthlisberger. Using developer interaction data to compare expertise metrics. Em *Proceedings of the 10th Working Conference on Mining Software Repositories*, páginas 297–300. IEEE Press, 2013. 48
- [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
- [SFP12] Goparaju Sudhakar, Ayesha Farooq e Sanghamitra Patnaik. Measuring productivity of software development teams. 2012. 55
- [SG11] Carolyn Seaman e Yuepu Guo. Measuring and monitoring technical debt. *Advances in Computers*, 82:25–46, 2011. xi, 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
- [SL11] Elben Shira e Matthew Lease. Expert search on code repositories. Relatório técnico, Technical Report TR-11-42, Department of Computer Science, University of ..., 2011. 47
- [SPK13] Jose San Pedro e Alexandros Karatzoglou. Multiple outcome supervised latent dirichlet allocation for expert discovery in online forums. Em *Workshops at the Twenty-Seventh AAAI Conference on Artificial Intelligence*, 2013. 48
- [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
- [SSS14] Girish Suryanarayana, Ganesh Samarthayam e Tushar Sharma. *Refactoring for software design smells: managing technical debt.* Morgan Kaufmann, 2014. 84
- [Ste10] Chris Sterling. *Managing software debt: building for inevitable change.* Addison-Wesley Professional, 2010. 13, 14
- [SWR12] Marek G Stochel, Mariusz R Wawrowski e Magdalena Rabiej. Value-based technical debt model and its application. Em *ICSEA 2012, The Seventh International Conference on Software Engineering Advances*, 2012. 89
- [TBLJ13] Ferdian Thung, Tegawende F Bissyande, David Lo e Lingxiao Jiang. Network structure of social coding in github. Em *2013 17th European Conference on Software Maintenance and Reengineering*, páginas 323–326. IEEE, 2013. 74
- [TDH14] Jason Tsay, Laura Dabbish e James Herbsleb. Influence of social and technical factors for evaluating contribution in github. Em *Proceedings of the 36th international conference on Software engineering*, páginas 356–366. ACM, 2014. 64
- [TLB⁺09] Thomas Tan, Qi Li, Barry Boehm, Ye Yang, Mei He e Ramin Moazeni. Productivity trends in incremental and iterative software development. Em *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, páginas 1–10. IEEE, 2009. 55

- [TRP09] Kai Tian, Meghan Revelle e Denys Poshyvanyk. Using latent dirichlet allocation for automatic categorization of software. Em *2009 6th IEEE International Working Conference on Mining Software Repositories*, páginas 163–166. IEEE, 2009. [45](#)
- [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](#)
- [Ves06] Jennifer Vesperman. *Essential CVS: Version Control and Source Code Management*. "O'Reilly Media, Inc.", 2006. [60](#)
- [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](#)
- [WR18] Stefan Wagner e Melanie Ruhe. A systematic review of productivity factors in software development. *arXiv preprint arXiv:1801.06475*, 2018. [41](#)
- [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](#)