EVOLUÇÃO DA DISTRIBUIÇÃO DE CONHECIMENTO DE SOFTWARE EM PROJETOS *OPEN SOURCE*

TALITA SANTANA ORFANÓ

EVOLUÇÃO DA DISTRIBUIÇÃO DE CONHECIMENTO DE SOFTWARE EM PROJETOS *OPEN SOURCE*

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientadora: Mariza Andrade da Silva Bigonha
Coorientadora: Kecia Aline Marques Ferreira

Belo Horizonte

Agosto de 2020

Orfanó, Talita Santana

D1234p Evolução da distribuição de conhecimento de software em projetos *open source* / Talita Santana Orfanó. — Belo Horizonte, 2020
xxiii, 105 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais
Orientador: Mariza Andrade da Silva Bigonha

1. Computação — Dissertação. 2. Engenharia de Software — Teses. I. Orientador. II. Título.

CDU 519.6*82.10

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Evolução da Distribuição de Conhecimento de Software em Projetos Open Source

TALITA SANTANA ORFANO

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROFA. MARIZA ANDRADE DA SILVA BIGONHA - Orientadora
Departamento de Ciência da Computação - UFMG

PROFA. KÉCIA ALINE MARQUES FERREIRA - Coorientadora
Departamento de Computação - CEFET-MG

PROF. MARCO TÚLIO DE OLIVEIRA VALENTE
Departamento de Ciência da Computação - UFMG

PROF. RICARDO TERRA NUNES BUENO VILLELA
Departamento de Ciência da Computação - UFLA

Belo Horizonte, 14 de Agosto de 2020.

A Deus pelas mãos da Santíssima Virgem Maria.

Agradecimentos

A Deus, pois tudo que tenho, tudo que sou e serei, provém de seu infinito e misericordioso Amor. À Nossa Senhora, que cuidou de cada detalhe e me sustentou nos momentos de sofrimento e dificuldade. À São José, que me ensina a perseverar com paciência e determinação no cumprimento dos meus deveres cotidianos.

Aos meus pais pelo amor incondicional, carinho, cuidado e incentivo durante toda a vida. À minha irmã pelo apoio, paciência e por me fortalecer nos momentos difíceis.

Às minhas orientadoras Mariza e Kecia, pela dedicação constante e suporte na realização deste trabalho, pela compreensão, orientação imprescindível e carinho durante todo o curso.

Aos familiares próximos, pela força e encorajamento. Às minhas madrinhas, por todo apoio, incentivo e atenção desde a infância.

Aos meus amigos e amigas, agradeço pela compreensão nos períodos de ausência, pelo sustento na oração e por tornarem meu cotidiano mais leve, sendo sinais de amor e alegria em todas as etapas da minha vida.

Aos professores e funcionários do DCC e a UFMG, pela estrutura provida para a realização das atividades, pelo apoio e suporte ofertados e, pela excelência do curso e de seu corpo técnico, contribuindo de forma extraordinária para a minha formação acadêmica e profissional.

Ao Departamento de Computação do CEFET-MG que, durante o curso técnico de Informática Industrial e a graduação em Engenharia de Computação, contribuíram imensamente para a minha formação pessoal, acadêmica e profissional.

A Symplicity e a SYDLE, pela compreensão e todo apoio fornecido para que eu pudesse conciliar as atividades e responsabilidades do trabalho com o Mestrado. A contribuição e o incentivo do time Contratanet foi fundamental nessa conquista.

"Fazei tudo por Amor.
Assim não há coisas pequenas: tudo é grande.
A perseverança nas pequenas coisas, por Amor, é heroísmo."
(São José Maria Escrivá)

Resumo

Propriedade de código se refere ao conhecimento e responsabilidade que um desenvolvedor possui sobre um determinado trecho de código. Estudos apontam uma significativa relação entre a qualidade de software e fatores humanos, relatando que uma das principais causas da degradação da qualidade do software é a falta de domínio dos desenvolvedores sobre o código fonte. Este trabalho busca compreender como o conhecimento do software se distribui entre os desenvolvedores ao longo do ciclo evolutivo de projetos *open source*. Há dois tipos de desenvolvedores, os chamados *heroes* e os pequenos ou periféricos. Os resultados deste estudo mostram que grande parcela do conhecimento do projeto concentra-se em um conjunto restrito de autores, os *heroes*. Contudo, aqueles que são os principais contribuidores no início do projeto não permaneceram nessa posição ao longo de sua evolução. O conhecimento torna-se mais distribuído durante o ciclo de vida do software. Apesar disso, os desenvolvedores periféricos não se convertem em autores frequentes por limitação de tempo e interesse. As principais formas de propagação do conhecimento entre os desenvolvedores é por meio das contribuições no código fonte (*bug/features*), documentações, e-mails e conversas informais. Os resultados deste trabalho contribuem para aprofundar o conhecimento que se tem até o momento sobre autoria de software, propriedade de código e a distribuição do conhecimento no processo evolutivo do software.

Palavras-chave: Propriedade de código; Autoria de software; Projetos *open source*; Desenvolvimento de software.

Abstract

Code ownership refers to the knowledge and responsibility a developer has about a given software code. Previous studies point out a significant relationship between software quality and human factors, reporting that one of the leading causes of software quality degradation is the lack of developers' knowledge of the source code. This study aims to understand how the code's knowledge is distributed among developers throughout the life cycle of open source projects. There are two types of developers, the so-called heroes and the small or peripheral ones. Results show that a large part of the knowledge is concentrated in a restricted set of authors, the heroes. However, those who are the main contributors at the project's beginning did not retain their position throughout its evolution. Knowledge becomes more distributed during the software life cycle, although peripherals developers do not become frequent authors due to limited time and interest. The main ways of spreading knowledge among developers is through contributions in the source code (bug/features), documentation, emails, and informal conversations. This study's results contribute to deepening the knowledge about software authorship, code ownership, and the distribution of knowledge in the evolutionary process of the software.

Palavras-chave: Code ownership; Authorship; *Open source* Projects; Software Development.

Lista de Figuras

2.1 Distribuição do esforço de manutenção de software (Imagem adaptada de [Bodhuin, 1995]) 8

4.1 Representação da metodologia adotada no trabalho 27

5.1 Dados de um *commit* no Django disponibilizados pela API GitHub 41

6.1 Total de contribuições efetuadas por autor no projeto *Bootstrap*, no qual 68% dos autores efetuaram apenas uma contribuição 48
6.2 Total de contribuições efetuadas por autor no projeto *Django*, no qual 65% dos autores efetuaram apenas uma contribuição 48
6.3 Total de contribuições efetuadas por autor no projeto *Elasticsearch*, no qual 62% dos autores efetuaram apenas uma contribuição 49
6.4 Total de contribuições efetuadas por autor no projeto *Pandas*, no qual 64% dos autores efetuaram apenas uma contribuição 50
6.5 Total de contribuições efetuadas por autor no projeto *Spring Boot*, no qual 69% dos autores efetuaram apenas uma contribuição 50
6.6 Contribuições dos autores *heroes* durante o ciclo de vida do sistema *Bootstrap* 53
6.7 Contribuições dos autores *heroes* durante o ciclo de vida do sistema *Django* 53
6.8 Contribuições dos autores *heroes* durante o ciclo de vida do sistema *Elasticsearch* 54
6.9 Contribuições dos autores *heroes* durante o ciclo de vida do sistema *Pandas* 54
6.10 Contribuições dos autores *heroes* durante o ciclo de vida do sistema *Spring Boot* 55
6.11 Propriedade de código no decorrer do ciclo de vida do *Bootstrap* 58
6.12 Propriedade de código no decorrer do ciclo de vida do *Django* 58
6.13 Propriedade de código no decorrer do ciclo de vida do *Elasticsearch* 59
6.14 Propriedade de código no decorrer do ciclo de vida do *Pandas* 59
6.15 Propriedade de código no decorrer do ciclo de vida do *Spring Boot* 60

6.16 Totais de contribuições durante o ciclo de vida do *Bootstrap* 61
6.17 Totais de contribuições durante o ciclo de vida do *Django* 62
6.18 Totais de contribuições durante o ciclo de vida do *Elasticsearch* 63
6.19 Totais de contribuições durante o ciclo de vida do *Pandas* 63
6.20 Totais de contribuições durante o ciclo de vida do *Spring Boot* 64
6.21 Gráfico de dispersão das amostras - Eixo horizontal corresponde ao valor da propriedade de código e o eixo vertical ao total de contribuições efetuadas 65

7.1 Autores periféricos - Motivação principal para contribuições no projeto . . 76
7.2 Autores periféricos - Razões para não ter se tornado um autor frequente no projeto 77
7.3 Autores médios e *heroes* - Tipos de contribuições efetuadas 78
7.4 Autores periféricos - Tipos de contribuições efetuadas 78
7.5 Transferência de conhecimento do software: em azul é como o conhecimento foi compartilhado com o autor; em vermelho é como ele transferiu o conhecimento 80

Lista de Tabelas

5.1 Repositórios selecionados no GitHub 34
5.2 Repositórios selecionados e as janelas de tempo T 39
5.3 Total de autores identificados como ambíguos entre si e agrupados 45

6.1 Categorização de autores e como as contribuições totais do projeto estão distribuídas entre eles 52
6.2 Propriedade de código dos repositórios do *data set* 57
6.3 Correlação entre a propriedade de código e o total de contribuições nos projetos. 66
6.4 Correlação entre as contribuições do proprietário do período e o total de contribuições do período nos projetos. 67

7.1 Total de autores convidados a responder o *Survey*, segmentado por projeto e categoria 75

Sumário

Agradecimentos **ix**

Resumo **xiii**

Abstract **xv**

Lista de Figuras **xvii**

Lista de Tabelas **xix**

1 Introdução **1**
- 1.1 Objetivos . . . 3
- 1.2 Contribuições . . . 4
- 1.3 Estrutura da Dissertação . . . 5

2 Referencial Teórico **7**
- 2.1 Manutenção de Software . . . 7
- 2.2 Evolução de Software . . . 9
- 2.3 Autoria de Software e Propriedade de Código . . . 10
- 2.4 Considerações Finais . . . 13

3 Trabalhos Relacionados **15**
- 3.1 Fatores Humanos e Organizacionais . . . 15
- 3.2 Autoria e Propriedade de Código . . . 17
- 3.3 Desenvolvedores *Heroes* vs Desenvolvedores Periféricos . . . 21
- 3.4 Considerações Finais . . . 24

4 Projeto do Estudo **27**
- 4.1 Questões de Pesquisa . . . 28
- 4.2 Coleta de Dados . . . 30

4.3 Caracterização dos Dados e Análise Quantitativa 30
4.4 *Survey* e Análise Qualitativa 31
4.5 Considerações Finais 31

5 Coleta de Dados **33**
5.1 Critérios de Escolha 33
5.2 Definição do *Data Set* 34
5.3 Coleta de Dados 36
5.4 Definição de Janela de Tempo 38
5.5 Coleta de Informações 39
5.5.1 Contribuições por Autor 39
5.5.2 Contribuições por Período de Tempo T 40
5.6 Tratamento de Ambiguidade de Autores 40
5.6.1 Por *Login* 41
5.6.2 Por Nome/E-mail 42
5.7 Considerações Finais 45

6 Evolução da Distribuição do Conhecimento entre os Desenvolvedores 47
6.1 Distribuição da População de Autores 48
6.2 Caracterização da População de Autores 51
6.3 Distribuição dos *Heroes* no Ciclo Evolutivo 52
6.4 Propriedade de Código no Ciclo Evolutivo do Software 56
6.5 Frequência de Contribuição no Ciclo de Vida do Software 61
6.6 Relação entre Concentração de Conhecimento e Quantidade de Contribuições 64
6.7 Respostas das Questões de Pesquisa 67
6.8 Considerações Finais 70

7 Distribuição do Conhecimento de Software - Análise Qualitativa 73
7.1 *Survey* 73
7.2 Público Alvo 74
7.3 Resultados 74
7.3.1 Motivações 75
7.3.2 Disseminação do Conhecimento 77
7.4 Análise dos Resultados 80
7.5 Considerações Finais 82

8 Discussão dos Resultados **85**

9 Ameaças à Validade **89**

10 Conclusão **91**

Referências Bibliográficas **95**

Apêndice A **101**

A.1 Questões Enviadas aos Autores *Heroes* 101

A.2 Questões Enviadas aos Autores Médios 103

A.3 Questões Enviadas aos Autores Periféricos 104

Capítulo 1

Introdução

Dentre as etapas do ciclo de vida de um software, a que despende o maior custo é a manutenção. Segundo Meyer [1988], a manutenção pode demandar até dois terços do esforço total de desenvolvimento do projeto. Correção de defeitos, alteração de funcionalidades, evolução do sistema com o acréscimo de novos requisitos e até mudanças de ambiente são atividades que caracterizam essa fase.

Lehman [1996] afirma que, à medida que o software evolui, há uma tendência de ocorrer uma maior degradação de sua qualidade interna. Alguns estudos apontam uma significativa relação entre fatores humanos e a qualidade do software [Meneely & Williams, 2009; Bird et al., 2011; Rahman & Devanbu, 2011; Nagappan et al., 2008]. Um desses fatores é a falta de conhecimento que os desenvolvedores têm sobre o código fonte em que fazem manutenção [Greiler et al., 2015]. Rahman & Devanbu [2011] sugerem que, para a manutenção de software, a experiência especializada do desenvolvedor em um determinado arquivo é mais importante do que sua experiência geral do sistema, uma vez que a falta de experiência específica tem relação direta com as falhas a ele associadas. Os estudos de Meneely & Williams [2009] indicam que muitos desenvolvedores alterando o código fonte o torna mais propenso a falhas. Adicionalmente, Palomba et al. [2018] mostram que a estrutura organizacional dos desenvolvedores e a fragilidade dos padrões adotados têm relação direta com a existência de *code smells*[1] nos projetos.

O conceito de *propriedade* de código se refere ao conhecimento e à responsabilidade que um desenvolvedor possui sobre um determinado trecho de código [Fritz et al., 2014]. Crowston et al. [2006] definem o conceito de *core developers* para referenciar os desenvolvedores que concentram o conhecimento do software. Em projetos *open source*, os desenvolvedores que efetuam a maior parte das contribuições são nomeados

[1] *Code Smells:* `https://martinfowler.com/bliki/CodeSmell.html`

na literatura como *core developer*, *hero*, *major* ou *main developer*. Neste trabalho, adotamos *heroes* como nomenclatura para referenciar os contribuidores principais. Na mesma linha, Ricca & Marchetto [2010], Foucault et al. [2014] e Agrawal et al. [2018] definem, por meio de estudos em sistemas *open source*, dois tipos de desenvolvedores: os chamados *heroes*, que são aqueles poucos que contribuem em uma grande parcela de módulos do projeto por um longo período de tempo, e os desenvolvedores *pequenos* ou *periféricos*, que são aqueles que se envolvem pouco com o projeto, por exemplo, desenvolvendo uma pequena funcionalidade ou resolvendo um *bug* simples. Essa divisão na carga de trabalho é a principal diferença entre os desenvolvedores que contribuem em projetos *open source* para os que trabalham em empresas com projetos privados. Em geral, os desenvolvedores que trabalham em projetos privados dedicam 100% de seu tempo e contribuem com o software por diversos meses.

Foucault et al. [2014] afirmam que em softwares médios e grandes, o conhecimento tende a se concentrar em único desenvolvedor, o *hero*. A concentração de conhecimento em poucos desenvolvedores pode representar um risco para o projeto. Essa ideia está representada no conceito de *Truck Factor* (TF)[2], uma métrica de concentração de conhecimento do software proposta pela comunidade Agile. Essa métrica é dada pelo número de desenvolvedores cuja saída do projeto poderia inviabilizá-lo [Torchiano et al., 2011; Ferreira et al., 2019]. Avelino et al. [2016] concluíram que a maioria dos sistemas *open source* avaliados possuíam o valor da métrica *Truck Factor* menor ou igual a dois, o que resulta em uma forte dependência do projeto em poucas pessoas. Avelino et al. [2019a] também investigaram a saída dos desenvolvedores *Truck Factor* em repositórios GitHub e de que modo ocorreu a entrada de novos contribuidores para a sobrevivência desses projetos.

Há poucos trabalhos que estudam como a distribuição do conhecimento dos desenvolvedores evolui no decorrer do ciclo de vida dos projetos. Dentre os trabalhos que analisam a evolução da atuação dos desenvolvedores *heroes* e periféricos, Joblin et al. [2017] indicam que há uma forte relação de contribuição entre os desenvolvedores principais, os quais possuem posições estáveis e importantes na hierárquicas do projeto em relação aos desenvolvedores periféricos. Avelino et al. [2019b] estudaram a autoria de software no *kernel* do Linux[3] e constataram, a partir da avaliação de 66 *releases*, que apenas 2% dos autores são responsáveis pela maioria das alterações do projeto. Robles et al. [2009] avaliaram a evolução e o comportamento dos principais desenvolvedores em sistemas *open source* hospedados no SourceForge[4] e constataram dois

[2] *Truck Factor:* `http://www.agileadvice.com/2005/05/15/agilemanagement/truck-factor/`
[3] Kernel Linux: `https://github.com/torvalds/linux`
[4] SourceForge: `https://sourceforge.net/`

padrões de comportamento entre os desenvolvedores principais do projeto: i) grupo de desenvolvedores principais estável, durante todos os períodos analisados; ii) grupo de desenvolvedores principais que sofria alterações no decorrer dos períodos, isto é, alguns autores deixavam de contribuir e outros tornavam-se mais frequentes. Em ambos cenários, esse grupo era composto por um número pequeno de contribuidores. O conhecimento sobre a evolução da atuação dos desenvolvedores ao longo da existência de projetos *open source* ainda não é amplo. Este trabalho visa contribuir para avançar nesse conhecimento, realizando estudos de caso para analisar detalhadamente como a evolução da distribuição do conhecimento do software evoluiu nos casos estudados.

1.1 Objetivos

Este trabalho investiga, por meio de estudos de caso, como a distribuição do conhecimento de código evolui ao longo do ciclo de vida do projeto. O escopo do trabalho se delimita a softwares *open source* presentes nos repositórios GitHub.[5] Especificamente, este estudo visa responder as seguintes questões de pesquisa (QPn):

- **QP1:** O conhecimento do software se difunde ao longo de seu ciclo de vida?
- **QP2:** Os desenvolvedores considerados *heroes* nas primeiras versões permanecem *heroes* durante a evolução do projeto?
- **QP3:** A concentração do conhecimento é impactada pela quantidade total de contribuições do projeto?
- **QP4:** Como se dá a atuação dos desenvolvedores periféricos nos projetos?
- **QP5:** Como ocorre a disseminação de conhecimento do software entre os desenvolvedores?

Para realização deste trabalho foram analisados cinco softwares *open source*, desenvolvidos em JavaScript, Python e Java. Foram construídos *scripts* em Python para coleta de dados das contribuições, a partir da API REST GitHub[6]. Os dados coletados foram armazenados no MongoDB Cloud[7] e em arquivos *CSV*. Os dados foram analisados quantitativamente, a fim de responder as questões de pesquisa QP1, QP2 e QP3. Posteriormente, foi conduzido um *survey* com os desenvolvedores dos projetos analisados para responder as questões de pesquisa QP4 e QP5.

[5] https://github.com/
[6] API REST GitHub: `https://developer.github.com/v3/`
[7] MongoDB Cloud: `https://www.mongodb.com/cloud`

1.2 Contribuições

Estudar a distribuição de conhecimento entre os autores possibilita compreender as características intrínsecas dos sistemas *open source* e propor ações de incentivo e auxílio aos colaboradores, tanto aos *heroes*, indispensáveis para o projeto, quanto aos desenvolvedores periféricos, para que possam aumentar o domínio do código e a frequência de contribuições e adquirirem um conhecimento maduro do sistema. Os resultados obtidos neste trabalho contribuem para aprofundar o conhecimento que se tem até o momento sobre autoria de software, propriedade de código e a distribuição do conhecimento no processo evolutivo do software, temas vinculados ao comportamento humano e que estão se tornando essenciais na área de manutenção e evolução de software.

As principais contribuições deste trabalho são:

- Análise da evolução da distribuição do conhecimento do software entre os desenvolvedores no decorrer do ciclo de vida do projeto. Segundo Rahman & Devanbu [2011], o conhecimento sobre a propriedade do código auxilia os gestores a encontrar especialistas para o qual serão designadas as tarefas, bem como compreender os desenvolvedores essenciais em cada projeto. Portanto, este trabalho gera contribuições relevantes para a área científica, tanto quanto para as empresas, especialmente aquelas que utilizam softwares *open source* e envolvem a gerência de pessoas em projeto de desenvolvimento de software

- *Scripts* para coleta de dados de autoria de software a partir do GitHub. Esses *scripts* estão disponíveis no repositório pessoal da autora desta dissertação no GitHub `https://github.com/taliorfano/knowledgedistribution_master/tree/master/Coleta_Dados/Scripts`

- Conjunto de dados de cada um dos projeto e os arquivos CSV gerados nas coletas. Esses dados estão disponíveis em `https://github.com/taliorfano/knowledgedistribution_master/tree/master/Coleta_Dados/Dados`, permitindo a replicação do trabalho e a utilização deles em estudos futuros.

1.3 Estrutura da Dissertação

O trabalho está disposto da seguinte forma:

- **Capítulo 2** apresenta conceitos importantes para compreensão do trabalho, como a manutenção de software, leis de evolução de software, repositórios *open source*, autoria e propriedade de código.
- **Capítulo 3** descreve um conjunto de trabalhos relacionados ao tema. São apresentados trabalhos que comprovam a influência de fatores humanos sobre a qualidade do código desenvolvido. Também são abordados trabalhos associados a autoria de software e propriedade de código.
- **Capítulo 4** detalha as questões de pesquisa propostas. Também é retratada a metodologia adotada na execução do estudo, envolvendo a coleta de dados, a caracterização dos dados, a criação de *surveys*, bem como a análise quantitativa e qualitativa dos resultados.
- **Capítulo 5** descreve sobre os softwares escolhidos e os critérios usados na definição do *data set*. Também são destacados os métodos existentes para a coleta de dados no GitHub e a forma adotada para medição da janela de tempo durante o ciclo evolutivo dos projetos. O total de contribuições por autor e por período de tempo são considerados, assim como o tratamento da ambiguidade de autores.
- **Capítulo 6** apresenta os estudos de caso com as caracterizações de autores e de suas contribuições efetuadas no decorrer do ciclo de vida do projeto. A partir dos dados coletados, avalia-se o cálculo da métrica de propriedade de código e a distribuição do conhecimento. Por meio de análises quantitativas, os resultados são examinados, possibilitando que as três primeiras questões sejam respondidas.
- **Capítulo 7** descreve a criação do *survey*, a motivação de suas questões e o público alvo escolhido. Também são explanados os resultados, obtidos por meio de pesquisas enviada aos contribuidores. Por fim, efetua-se uma análise qualitativa dos resultados, proporcionando a discussão e respostas para as últimas duas questões de pesquisa.
- **Capítulo 8** discute os resultados do trabalho.
- **Capítulo 9** aborda as ameaças à validade do estudo realizado.
- **Capítulo 10** apresenta a conclusão da dissertação, sumarizando os resultados e as principais contribuições, e aponta possíveis trabalhos futuros.

Capítulo 2

Referencial Teórico

Este capítulo aborda conceitos empregados no trabalho. A primeira seção relata sobre manutenção e evolução de software, a segunda, sobre evolução de software e a terceira descreve os conceitos de propriedade de código e autoria de software.

2.1 Manutenção de Software

Sommerville et al. [2007] afirmam que a manutenção de software é o processo geral de alterações de um sistema após a sua entrega ao cliente. As alterações feitas no software variam desde pequenas correções de falhas, alterações mais extensas para correção de erros de projeto, modificações de requisitos já entregues, até o acréscimo de novas funcionalidades e componentes.

As atividades de manutenção de software podem ser classificadas em quatro categorias: manutenção corretiva, adaptativa, perfectiva e preventiva [Pressman, 2005].

Manutenção corretiva: visa corrigir os problemas oriundos de defeitos do sistema e de suas funcionalidades. Geralmente, as correções iniciais são emergenciais e temporárias com o objetivo de atender a necessidade do usuário e manter o sistema funcionando. Essas falhas são geralmente decorrentes de erro de codificação, projeto e erros lógicos.

Manutenção adaptativa: refere-se à adequação do código devido a mudanças externas a ele. Ela também pode se realizada quando há mudanças no hardware, no banco de dados, no sistema operacional ou no ambiente [Pfleeger, 2004].

Manutenção perfectiva: consiste em efetuar melhorias no sistema, seja por meio do acréscimo de novas funcionalidades ou da melhoria dos requisitos já existentes. A melhoria do código fonte para facilitar a leitura e aprimorar futuras manutenções, também são exemplos de manutenção perfectiva [Pfleeger, 2004].

Manutenção preventiva: trata da modificação de alguns aspectos do sistema com o objetivo de prevenir futuras falhas. Ela também ocorre quando o desenvolvedor observa um defeito em potencial, porém que ainda não gerou nenhuma falha perceptível ao usuário final.

Bodhuin [1995] sumariza, com o gráfico mostrado na Figura 2.1, a distribuição do esforço entre as diferentes categorias da manutenção. Segundo Meyer [1988], o custo gasto nessa etapa é cerca de dois terços do orçamento total do projeto.

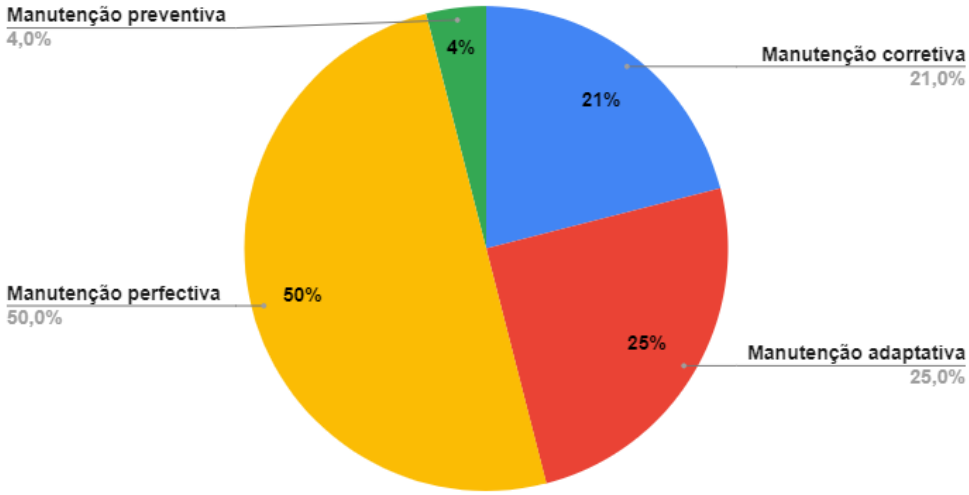

Figura 2.1. Distribuição do esforço de manutenção de software (Imagem adaptada de [Bodhuin, 1995])

Geralmente, o esforço para adicionar uma funcionalidade depois que o sistema está em operação é maior do que implementar a mesma funcionalidade durante a fase de desenvolvimento. Sommerville et al. [2007] citam algumas razões para isso, dentre elas, a falta estabilidade e habilidade da equipe. A falta de estabilidade na equipe é causada pela rotatividade de pessoal. Novos desenvolvedores vinculados ao projeto podem ser responsáveis por sua manutenção, mas não conhecem o sistema de modo profundo, nem mesmo as decisões tomadas para a construção de sua arquitetura.

Aspectos organizacionais e pessoais estão associados aos principais problemas existentes na fase de manutenção. O entendimento limitado da equipe responsável pelo software é um obstáculo frequentemente enfrentado durante sua manutenção [Pfleeger, 2004]. Parikh & Zvegintzov [1983] mostram que 47% do esforço gasto na manutenção do software implica em compreender o código que será modificado. O presente trabalho analisa o processo de manutenção de software em projetos *open source*, estudando a distribuição das contribuições (*commits*) efetuados pelos desenvolvedores do projeto.

Acredita-se que as principais atividades de manutenção realizadas nesse contexto são manutenções corretiva e perfectiva.

2.2 Evolução de Software

Evolução de software é o estudo das mudanças e adaptações que um sistema sofre ao longo de sua vida útil, isto é, desde o início do ciclo de desenvolvimento até o fim de suas manutenções. Para acompanhar as variações das demandas do contexto em que estão inseridos, os softwares devem possuir uma natureza dinâmica, sendo adaptados para atender à novas requisições que possam surgir ao longo do tempo [Sommerville et al., 2007].

Lehman & Belady [1985] realizaram vários estudos empíricos sobre as mudanças de sistemas, com o intuito de compreender as características intrínsecas associadas a evolução do software. A partir desses estudos, foram propostas as chamadas "Leis de Lehman" [Lehman, 1996]. Essas leis possivelmente são válidas para todos os tipos de grandes sistemas organizacionais, chamados de sistemas Tipo-E. Os sistemas do Tipo-E estão fortemente ligados ao mundo real e são afetados por mudanças no ambiente, por isso devem ser adaptáveis [Pfleeger, 2004]. As oito leis da evolução de software, segundo Lehman [1996] são listadas a seguir.

1. **Mudança contínua**: um programa que é usado deve passar por contínuas mudanças senão se tornará cada vez menos útil.

2. **Aumento da complexidade**. Quando um programa é continuamente modificado, sua estrutura se deteriora, causando um aumento em sua complexidade.

3. **Auto-regulação**. A evolução do programa é um processo de auto-regulação em que as medidas de produto e processo se aproximam de uma distribuição normal.

4. **Conservação da estabilidade organizacional**. Durante o ciclo de vida de um programa, a taxa de atividade global tende a ser invariável, isto é, a taxa de trabalho tende a ser constante.

5. **Conservação da familiaridade**. Durante o ciclo de vida de um programa, a taxa de mudanças incrementais entre versões tende a ser constante.

6. **Crescimento contínuo**. As funcionalidades do sistema tendem a aumentar ao longo do tempo para satisfazer as necessidades dos usuários.

7. **Qualidade decrescente**. A qualidade do programa diminuirá ao longo do tempo, a menos que o projeto sofra adaptações e manutenções a fim de mantê-la.

8. **Sistema de retorno**. Os processos de evolução incorporam sistemas de *feedback* com vários agentes que devem ser utilizados para conseguir melhorias significativas do produto.

Este trabalho de dissertação analisa dados do ciclo evolutivo de cinco projetos, hospedados no GitHub. As principais leis relacionadas a esse estudo são a mudança contínua e a conservação da estabilidade organizacional. A partir das coletas realizadas, será possível constatar se os sistemas *open source* selecionados possuem características que corroboram com essas leis.

2.3 Autoria de Software e Propriedade de Código

Autoria de software e *propriedade de código* são temas que têm se tornado cada vez mais frequentes na literatura. No entanto, ainda não existem conceitos precisamente definidos para esses termos, especialmente para propriedade de código. É comum observar nos artigos científicos que os autores ao utilizarem esses termos, os conceituam conforme a realidade em que irão trabalhar. Isso acarreta algumas divergências na definição dos conceitos. A seguir, são apresentados os principais estudos que definem de alguma forma esses conceitos.

LaToza et al. [2006] afirmam que a propriedade do código é uma sensação frequentemente implícita e a prática incidente de correção de *bugs* ou desenvolvimento de novos recursos em um módulo por parte de um mesmo desenvolvedor ou equipe.

O termo *autoria de software* também é utilizado para referenciar o autor responsável por um trecho de código ou módulo. Fritz et al. [2014] destacam que, em um dia de trabalho em ambiente corporativo, o desenvolvedor está envolvido em diversas autorias. Segundo os autores, a criação de um novo método, a modificação realizada em um método já existente e o ato de aceitar as alterações feitas por um outro membro da equipe são todos considerados eventos de autoria de software. Assim, Fritz et al. [2014] propuseram um modelo chamado DOK (*Degree-of-Knowledge*), esse modelo calcula o grau de conhecimento e é composto por dois componentes: *DOA* e *DOI*. O primeiro indica o conhecimento a longo prazo de um desenvolvedor em relação a um elemento do código fonte, calculado pelo valor do grau de autoria. O segundo indica o conhecimento a curto prazo de um desenvolvedor, representado pelo grau de interesse que é calculado pela quantidade de interação que o desenvolvedor teve com o elemento, como por exemplo, seleções e edições.

Rahman & Devanbu [2011] definem como autoria, o valor resultante da razão entre o número de linhas alteradas para corrigir um *bug* e o número de linhas com as quais o autor contribuiu para corrigir o *bug*. Esse valor resultante equivale à atuação efetiva do autor na correção do *bug*. Assim, para esses autores, o termo propriedade de código se refere à autoria do desenvolvedor que mais contribuiu com o fragmento de código analisado.

Existem várias definições para *propriedade de código* na literatura. Como relata Müller et al. [2015], o termo propriedade de código descreve quem possui autoria sobre certa parte do código, sendo o proprietário aquele que contribuiu com mais linhas para o arquivo ou módulo. Assim, o conceito de propriedade de código pode ser utilizado para distinguir dentre os autores do código aquele que possui uma maior responsabilidade, ou seja, aquele desenvolvedor que criou ou alterou a maior parte do código existente no módulo analisado.

Greiler et al. [2015] definem propriedade de código como o número de contribuições totais para o código fonte em relação à proporção de contribuições de um desenvolvedor. Assim, da mesma forma que Rahman & Devanbu [2011] propuseram, esses autores sugerem que o desenvolvedor responsável pela maior contribuição, ou seja, pelo maior número de mudanças, seja escolhido como sendo o responsável e proprietário por aquele fragmento.

Segundo Bird et al. [2011], o termo propriedade de código é usado para descrever a responsabilidade que se tem por um componente de software ou, até mesmo, para indicar que não se tem um claro responsável para um componente. Eles definem a métrica de propriedade de código de duas formas.

- Uma medida de propriedade é definida pela quantidade de atividade de desenvolvimento de um componente realizada por determinado desenvolvedor. Se, por exemplo, 80% das mudanças de um componente foram realizadas por um desenvolvedor, então pode-se dizer que o componente tem alta propriedade.

- A outra maneira relatada em seu estudo para medir a propriedade de código é analisar se existem muitos desenvolvedores de baixa especialização trabalhando em um componente. Se muitos desenvolvedores estão realizando algumas poucas mudanças em um componente, então isso indica que há muitos não-especialistas trabalhando no componente e, por isso, esse componente possui baixa propriedade.

Bird et al. [2011] consideram que ter um claro proprietário para o componente pode levar esse componente a ter menos falhas. Em contrapartida, quando existem

muitos não-especialistas realizando alterações, o componente é propenso a ter mais falhas.

Foucault et al. [2014] replicaram o estudo de Bird et al. [2011] para softwares *open source* e detalham como medir a propriedade de código. Ambos assumem que o projeto de software é composto por um conjunto finito de módulos, desenvolvidos por um conjunto finito de desenvolvedores que, por sua vez, submeteram os códigos modificados via *commits* no repositório. Assim, o modelo formal proposto por Bird et al. [2011] utiliza as seguintes notações:

- M é conjunto de módulos de um determinado projeto de software,
- D é conjunto de desenvolvedores e
- C é o conjunto de *commits* enviados pelos desenvolvedores.

Sendo,

- m_i é o módulo do software em questão,
- d_j é o desenvolvedor que modificou o módulo e
- c_k é o *commit* feito pelo desenvolvedor que modificou o módulo.

Para simplificar a compreensão, Foucault et al. [2014] estenderam o modelo formal proposto por Bird et al. [2011], acrescentando as seguintes notações:

- $\omega\ (m)$ é a soma de todas contribuições dos desenvolvedores realizadas em um módulo m,
- $\omega\ (d)$ é a soma de todas contribuições realizadas por um desenvolvedor d e
- $\omega\ (m,d)$ é a soma de todas contribuições realizadas por um desenvolvedor d em um módulo m.

Baseado na definição de contribuição de desenvolvedor, a métrica da propriedade de código mede a razão das contribuições feitas por um desenvolvedor pelo restante dos desenvolvedores. Em outras palavras, para cada módulo m e para cada desenvolvedor d, a métrica de propriedade é:

$$own_{m,d} = \frac{\omega(m,d)}{\omega(m)}$$

Segundo Bird et al. [2011], o valor mais elevado da razão entre as contribuições realizadas por todos os desenvolvedores em um módulo de software m é o valor da

propriedade e o desenvolvedor responsável por essa contribuição é considerado o proprietário do módulo. Desse modo, tem-se que:

$$\max(\{\ own_{m,d} \mid d \in D\})$$

A partir das definições apresentadas sobre propriedade de código, foram adotados neste trabalho os conceitos definidos por Bird et al. [2011] e por Foucault et al. [2014]. A motivação para essas escolhas justifica-se pelo fato de que esses estudos detalham e explicam de forma mais abrangente a definição dos conceitos de autoria e propriedade de código, além de serem utilizados como principal referência em outros estudos [Greiler et al., 2015; Thongtanunam et al., 2015; Tufano et al., 2015; McIntosh et al., 2016; Thongtanunam et al., 2016; Müller et al., 2015; Canfora et al., 2012].

Também constatou-se que a maioria dos trabalhos medem a propriedade de código por *commits*, módulos, arquivos ou linhas de código. Ainda não há consenso na literatura sobre a melhor maneira de avaliar autoria e contribuição, especialmente no que se refere a projetos *open source*. Para realização desse trabalho de mestrado, a granularidade adotada foi a análise de *commits*. A forma de coleta é amplamente abordada no Capítulo 5.3.

2.4 Considerações Finais

A qualidade do software durante a fase de manutenção e evolução do sistema é diretamente influenciada por fatores humanos. A falta de familiaridade com o projeto, o conhecimento escasso do código fonte, das regras de negócio e do padrão de codificação pelos desenvolvedores podem acarretar a degradação da qualidade do software e contribuir para a redução de sua manutenibilidade. Este trabalho dedica-se a estudar a distribuição do conhecimento entre os autores de projetos *open source* durante seu processo evolutivo. Para isso, serão utilizados os conceitos de propriedade de código definidos por Bird et al. [2011] e Foucault et al. [2014].

Capítulo 3

Trabalhos Relacionados

O estudo e a aplicação sobre o conceito de autoria perpassam diferentes áreas da computação. No contexto da Engenharia de Software, estudos sobre autoria e propriedade de código têm sido realizados na literatura contemporânea, especialmente pela significativa relação encontrada entre fatores humanos e a qualidade do código desenvolvido [Meneely & Williams, 2009; Greiler et al., 2015]. Os pesquisadores também buscam compreender como o conhecimento do código é adquirido, transmitido e distribuído entre os desenvolvedores. Este capítulo descreve os principais trabalhos que têm sido realizados nesse contexto. O estudos foram agrupados em três categorias: i) fatores humanos e organizacionais, ii) autoria e propriedade de código; e iii) desenvolvedores *heroes* vs desenvolvedores periféricos.

3.1 Fatores Humanos e Organizacionais

Estudos têm analisado a influência que fatores humanos e organizacionais possuem sobre a qualidade do código e a forma de disseminação do conhecimento do código. Nesse sentido, para compreender a causa da retenção de conhecimentos implícitos e entender as práticas adotadas pelos desenvolvedores, LaToza et al. [2006] realizaram pesquisas e entrevistas com desenvolvedores da Microsoft, tendo como base o conceito de propriedade de código. Os resultados da pesquisa apontaram que a comunicação face a face era a mais utilizada pelos desenvolvedores naquele contexto, visto ser uma comunicação mais rápida e eficiente. No entanto, isso acarretava a baixa retenção de conhecimento implícito, pois as informações trocadas não eram documentadas e acabavam por se perder com o tempo.

A interseção de processos, organização e desenvolvedores têm sido amplamente estudada. Assim, Nagappan et al. [2008] apresentam um estudo empírico, com da-

dos do Windows Vista, que investiga a relação entre a estrutura organizacional e a qualidade de software. Eles identificaram métricas que quantificam a complexidade organizacional, como o número de engenheiros que editaram o componente e ainda trabalham na empresa, o número de ex-engenheiros que trabalharam em um componente e a frequência de edição do código-fonte. Essas métricas foram comparadas com as métricas tradicionais, tais como complexidade de código, cobertura e dependências, buscando identificar uma relação entre elas e a predição de falhas. Os resultados encontrados evidenciam que as métricas organizacionais são preditores efetivos de propensão a falhas, apresentando melhores resultados do que as métricas tradicionais.

Na mesma linha, os estudos de Palomba et al. [2018] demonstram que estruturas organizacionais e as técnicas de software estão profundamente interconectadas. Eles estudaram a relação entre *community smells* e *code smells*. *Code smells* são características ou sintomas no código fonte que podem indicar a existência de problemas profundos em sua qualidade. Por sua vez, *community smells* refletem as vulnerabilidades nos padrões organizacionais da comunidade de software. Os resultados indicaram que a presença de *community smells* dificulta a refatoração de *code smells*. A partir de pesquisas realizadas com 162 desenvolvedores de nove projetos *open source*, concluiu-se que os fatores relacionados a organização da comunidade de software são percebidos, intuitivamente, pelos desenvolvedores como sendo a causa da existência de *code smells*.

O desenvolvimento distribuído, suas dificuldades e potencialidades também é um ponto estudado pela comunidade científica. Acredita-se que o desenvolvimento de software distribuído é mais arriscado do que aquele alocado em um mesmo espaço. A partir desse fato, Bird et al. [2009] levantaram uma hipótese: as equipes de desenvolvimento que são distribuídas têm mais falhas *pós-releases* que aqueles alocados em um mesmo espaço. Na realização desse estudo, a base de coleta dos dados foi realizada a partir do Windows Vista, que possui milhares de arquivos desenvolvidos por alguns milhares de desenvolvedores distribuídos em 59 prédios, 21 campus e três continentes. O sistema operacional possui dez milhões de linhas de código (LOC). Foram analisados a sua qualidade, a localização geográfica dos *commits* e o responsável pela propriedade de código em cada arquivo. Os resultados indicam que o desenvolvimento distribuído não contribui de forma efetiva com o número de falhas *pós-release*, portanto a hipótese não é verdadeira.

Dando continuidade à pesquisa de Bird et al. [2009], Kocaguneli et al. [2013] estudaram os dados do Microsoft Office 2010 e buscaram validar os efeitos do desenvolvimento distribuído a partir da avaliação de diferentes métricas, como a frequência de edição do código, total de linhas adicionadas e removidas por arquivo, total de classes e métodos, o número de proprietários de código, dentre outras. Os autores também

concluíram que o desenvolvimento distribuído não é prejudicial à qualidade do software. Meneely & Williams [2009] buscaram compreender se um projeto que possui muitos desenvolvedores torna-se inseguro. Com o propósito de analisar a segurança em projetos *open source* no contexto de colaboração de desenvolvedores, os autores utilizaram o *kernel* do Red Hat Enterprise Linux.[1] Os resultados apontaram que arquivos que possuíam alterações feitas por mais de nove desenvolvedores foram 16 vezes mais propensos a ter vulnerabilidade do que os arquivos modificados por menos desenvolvedores, indicando que muitos desenvolvedores alterando o código pode gerar um efeito prejudicial.

Esses trabalhos enfatizam a importância de considerar fatores humanos nos estudos da Engenharia de Software. O enfoque deste trabalho de mestrado também está relacionado a fatores humanos uma vez que considera os autores responsável pela contribuição, a fim de analisar de forma detalhada os desenvolvedores que possuem maior propriedade do código, a variação no número de suas contribuições ao longo do tempo e o modo como o desenvolvedor transmite o conhecimento adquirido com o restante da equipe do projeto. Dentre os trabalhos prévios realizados nessa linha, os mais próximos do presente trabalho são os de Bird et al. [2009] e Nagappan et al. [2008], que também consideram dados de autoria de software. Na mesma linha do estudo de Bird et al. [2009], neste trabalho os *commits* são utilizados como indicadores de autoria de software.

3.2 Autoria e Propriedade de Código

Esta seção descreve trabalhos que abordam a temática da autoria e propriedade de código, sob diferentes perspectivas.

Nordberg [2003] realizou um estudo que propõe quatro modelos de propriedade de código em softwares:

Especialista de produto: é um único indivíduo gerindo todo o código com ajuda ocasional de outras pessoas.

Propriedade de subsistemas: é o modelo em que existe um proprietário para cada subsistema.

[1] Red Hat Enterprise Linux: `www.redhat.com/pt-br/technologies/linux-platforms/enterprise-linux`

Chefe de arquitetura: ocorre quando existe um desenvolvedor que é o proprietário primário, que conhece todo o código fonte, enquanto os outros membros da equipe assumem papéis que visam auxiliá-lo na construção do produto.

Propriedade coletiva: ocorre quando todos possuem a propriedade efetiva sobre o código, no qual cada membro da equipe tem a capacidade e a possibilidade de contribuir em qualquer parte do código e em qualquer subsistema.

Nordberg [2003] constatou que para projetos médios e grandes, a melhor opção a ser adotada deve ser a composta pelos quatro modelos, solução que ele nomeou de propriedade de código dinâmica. Essa solução sugere que em cada fase do projeto um modelo de propriedade de código distinto deve ser adotado. Supondo as fases do Processo Unificado, na primeira fase, a concepção, deve-se utilizar o especialista de produto; na elaboração do projeto recomenda-se o modelo de chefe de arquitetura; na fase de construção e desenvolvimento do sistema, deve ser adotado a propriedade coletiva e finalmente na fase de transição, isto é, os testes e a implantação, o modelo de propriedade de subsistemas. Portanto, segundo Nordberg [2003], durante a fase de desenvolvimento do software, o modelo de propriedade de código coletiva é o mais indicado. O autor afirma que uma equipe que segue esse modelo incentiva uma maior interação dos membros, lida facilmente com o aumento de novas funcionalidades e com o crescimento da própria equipe. Para garantia da qualidade do código, práticas e metodologias como o *XP* (Extreme Programming)[2] devem ser adotadas.

Rahman & Devanbu [2011] discutem a relação entre a propriedade do código, as falhas, a autoria e a experiência do desenvolvedor no nível do código fonte do projeto. Os estudos realizados utilizaram quatro softwares *open source*, desenvolvidos na linguagem C e que usavam o Git como sistema de controle de versão. Foram definidos, no estudo, dois conceitos de avaliação de experiência do desenvolvedor e da equipe, a experiência especializada e a experiência geral. A experiência especializada corresponde à experiência em um único arquivo, enquanto a experiência geral é correspondente a todo o projeto. Os resultados encontrados mostram que as experiências especializadas do desenvolvedor são mais relevantes do que a experiência geral e que a falta dessa experiência individual em um determinado arquivo tem relação direta as falhas a ele associadas.

Bird et al. [2011] apresentam o primeiro estudo empírico quantificado sobre os efeitos que a propriedade de código tem sobre sua qualidade. O trabalho desenvolveu-se a partir da análise de dois grandes softwares comerciais de código fonte fechado, o

[2]XP: `http://www.extremeprogramming.org`

Windows Vista e o Windows 7. O objetivo principal é compreender a relação entre a propriedade de código e a qualidade do mesmo, e como tal relação se estabelece durante o processo de desenvolvimento. Para isso, foram definidas métricas de propriedade, considerando o número de vezes que um desenvolvedor trabalhou em um determinado componente e sua experiência de aprendizagem. Bird et al. [2011] constataram que, em componentes com um único e claro proprietário, são encontradas menos falhas e, consequentemente, maior é a qualidade do código, do que quando há muitos desenvolvedores com pouca experiência trabalhando em um mesmo componente.

Fritz et al. [2014] desenvolveram um modelo conhecido como DOK (*degree-of-knowledge*), que mede o grau de conhecimento do desenvolvedor acerca do código, baseado na autoria de software e nos dados de interação. A partir dessa informação, eles afirmam que é possivel compreender quem conhece mais sobre uma determinada região do código e quem deveria ser o responsável por comunicar sobre as mudanças realizadas em um trecho de código ao restante da equipe. O modelo desenvolvido calcula o grau de conhecimento do código de cada desenvolvedor individualmente, utilizando informações tanto da autoria de software quanto de uma simples interação com o código fonte, isto é, consultas realizadas ao código durante o trabalho. Fritz et al. [2014] consideram que quanto mais frequente e recente for a interação de um desenvolvedor com um elemento particular do código fonte, maior o conhecimento do desenvolvedor para aquele elemento. O estudo envolveu 19 desenvolvedores monitorados durante cinco semanas.

Foucault et al. [2014] replicaram os estudos de Bird et al. [2011] utilizando como objeto de estudos software de código livre e aberto (*FLOSS - Free/Libre/Open Source Software*[3]) desenvolvidos na linguagem Java. O objetivo da pesquisa foi generalizar a "lei de propriedade" que afirma que desenvolvedores pequenos devem ser evitados. As métricas definidas por Bird et al. [2011] classificam como desenvolvedores pequenos, os responsáveis por menos de 5% da soma total de *commits* no módulo e na periodicidade analisada, os desenvolvedores considerados grandes são aqueles responsáveis por mais de 5% da soma total de *commits*. O estudo analisa a relação entre as métricas de propriedade e a propensão a falhas em sete projetos FLOSS. No entanto, diferentemente do esperado, os resultados encontrados mostraram uma fraca relação entre as métricas de propriedades e as falhas de módulos. Assim, Foucault et al. [2014] concluíram que a lei de propriedade não pode ser generalizada para projetos FLOSS e as métricas de código clássicas ainda se mostram mais eficientes nos indicadores de propensão a falhas em relação às métricas de propriedade. Esse resultado diverge dos achados de Rahman

[3]FLOSS: `https://fsfe.org/freesoftware/comparison.en.html`

& Devanbu [2011], que concluíram que a falta de conhecimento especializado em um arquivo tem relação à ocorrência de falhas nele.

Greiler et al. [2015] replicaram e estenderam os estudos de Bird et al. [2011] e Foucault et al. [2014]. Para estabelecer a relação entre o conceito de propriedade de código e qualidade, os autores definem novas métricas de propriedade de código e as organizam em quatro grupos de acordo com sua atuação. O primeiro grupo é composto pelas métricas de propriedade individual para arquivos, o segundo grupo pelas métricas de propriedade individual para diretórios, o terceiro pelas métricas de propriedade organizacional para arquivos e o quarto grupo trata as métricas de propriedade organizacional para diretórios. Durante os experimentos, foram analisados quatro produtos comerciais da Microsoft, sendo eles: Office, Office365, Exchange e Windows. Todos os dados coletados são provenientes da ferramenta CodeMine.[4] Os resultados obtidos apontaram que as métricas de propriedade analisadas não têm relação com o número de falhas. No entanto, foi constatado que o número de autores tem relação direta com o número de falhas, confirmando o resultado levantado por Bird et al. [2011].

Estudos relacionados a *Truck Factor*[5] também aplicam o conceito de autoria e propriedade de código. A base da ideia dessa métrica é a mesma dos estudos de Foucault et al. [2014], a concentração de conhecimento em um grupo restrito de desenvolvedores. A métrica *Truck Factor* visa detectar como o conhecimento sobre o código está distribuído entre os membros do projeto. Essa métrica indica o número de desenvolvedores que, se forem desligados da equipe, acarretariam sérios problemas para o projeto. Torchiano et al. [2011] apresentam um modelo teórico sobre *Truck Factor* e, em seguida, é investigado o valor máximo alcançável para o *Truck Factor* (TF) em um projeto. A análise envolvendo autoria de software é realizada no nível de arquivos de linguagem Java. O modelo teórico desenvolvido por Torchiano et al. [2011] visa maximizar o valor do TF e utiliza variáveis diretamente relacionadas ao termo de autoria de software. Os dados para esse estudo são provenientes de 20 projetos *open source* oriundos de diferentes repositórios.

Estudos apontam que projetos *open source* não estão preparados para lidar com a rotatividade de desenvolvedores *heroes*. Avelino et al. [2016] calculam, a partir de um método proposto por eles, o valor do *Truck Factor* para 133 projetos *open source*. Os resultados indicaram que 65% dos sistemas avaliados possuem o valor de TF <=2. Dentre os estudos conduzidos por Ferreira et al. [2019], foi investigada a relação existente entre os desenvolvedores *Truck Factor* e os *core developers*. Foram analisados

[4]CodeMine: `http://www.codemine.com/`

[5]Truck Factor: https://medium.com/@aserg.ufmg/what-is-the-truck-factor-of-github-projects-bb0d5f019a6f

dados de 35 projetos *open source*, hospedados no GitHub. Os resultados indicaram que os desenvolvedores chave do *Truck Factor*, cuja saída acarretaria problemas sérios para sobrevivência do projeto, são um subconjunto do grupo dentro do grupo de *core developers* do projeto. Ferreira et al. [2016] analisaram como o conhecimento dos desenvolvedores que compõe o *Truck Factor* de um sistema está distribuído na arquitetura do software.

O presente trabalho também é baseado no conceito de autoria de software. A diferença essencial deste trabalho para os apresentados nesta seção é que ele analisa como o valor da métrica de propriedade de código varia no decorrer do ciclo evolutivo do software, baseado nas definições de Foucault et al. [2014] e Bird et al. [2011]. Dessa forma, ao considerar intervalos de tempo no cálculo da métrica, torna-se possível avaliar a maneira como a distribuição de conhecimento do software evolui durante o seu ciclo de vida. Assim, poder-se-á afirmar que o conhecimento do software tornou-se mais distribuído quando o valor da métrica nos últimos anos avaliados for menor do que nos primeiros anos de vida desse projeto.

3.3 Desenvolvedores *Heroes* vs Desenvolvedores Periféricos

Uma característica recorrente entre os projetos *open source* é a concentração do conhecimento em alguns poucos desenvolvedores, chamados *heroes*, e a presença de vários desenvolvedores que realizam poucas e esporádicas contribuições, nomeados por periféricos. Esta seção apresenta alguns dos principais trabalhos realizados para avaliar o comportamento dos heroes, dos periféricos e as dificuldades por eles enfrentadas.

Crowston et al. [2006] apresentaram trabalhos iniciais em relação ao modo de classificação de *core developers*. Foi realizado um estudo comparativo entre três métodos de identificação de desenvolvedores principais, considerando dados de 116 projetos *open source*, hospedados no SourceForge. Com a popularização dos repositórios *open source*, Ricca & Marchetto [2010] buscaram compreender se era comum o surgimento de *heroes* nos projetos e se a presença deles gerava benefícios para o processo de manutenção do software. Foram analisados 37 projetos hospedados no SourceForge e no *GoogleCode*. Os resultados demonstraram que em 95% dos projetos avaliados existia ao menos um desenvolvedor *hero*. Além disso, os *heroes* atendiam os chamados de mudança (*change requests*) em um tempo 63% menor que os outros contribuidores.

Joblin et al. [2017] estudaram a relação entre os desenvolvedores principais e os periféricos, a partir de dez projetos *open source*, com pesquisas envolvendo 166 de-

senvolvedores. O conceito de rede de desenvolvedores (*developer network*) foi utilizado nesse trabalho, considerando os desenvolvedores como os nós e a relação entre eles como sendo as arestas. Nessa perspectiva, os resultados demonstraram que os contribuidores principais tendem a colaborar com outros desenvolvedores principais, enquanto os desenvolvedores periféricos preferem se relacionar com os desenvolvedores principais, e raramente há vínculos entre dois desenvolvedores periféricos.

Agrawal et al. [2018] avaliaram a distribuição dos desenvolvedores em 661 softwares *open source* e 171 projetos do GitHub Enterprise,[6] englobando projetos grandes, médios e pequenos. Os resultados mostraram que a presença de *heroes* ocorreu em apenas 19% dos projetos públicos pequenos, enquanto nos projetos privados pequenos, foi de 63%. Além disso, a presença de *heroes* em projetos médios e grandes é bastante expressiva, tanto nos projetos públicos, quanto naqueles privados.

Avelino et al. [2019a] analisaram 1.932 projetos populares presentes no GitHub com o intuito de compreender como a saída de desenvolvedores *truck factor* impactava na sobrevivência do projeto. Eles concluíram que, dentre os repositórios avaliados, 16% foram abandonados e 41% sobreviveram e que a sobrevivência desses projetos deve-se à atuação de novos desenvolvedores principais. Eles realizaram uma pesquisa com esses novos responsáveis para compreender as motivações que os levaram a contribuir para a sobrevivência do projeto. Dentre as principais respostas, destacam-se: i) a necessidade de evoluir o projeto para uso pessoal; ii) o desejo de contribuir com projetos *open source*; e iii) evitar que o projeto fosse descontinuado. Também constatou-se que a falta de tempo e a dificuldade em obter permissão para contribuição foram as principais barreiras encontradas por esses autores. Na mesma linha, Coelho & Valente [2017] apontam que os cinco principais motivos para o fracasso de projetos *open source* são i) projeto apropriado pelo concorrente; ii) projeto tornou-se obsoleto; iii) falta de tempo dos colaboradores principais; iv) falta de interesse dos colaboradores principais do repositório; v) projeto baseado em tecnologias ultrapassadas.

Coelho et al. [2018] estudaram as motivações que levaram desenvolvedores principais a contribuir com frequência em projetos *open source* e as dificuldades enfrentadas por eles. Os resultados apontaram que as principais motivações dos colaboradores foi a necessidade de melhorar o projeto, porque estava utilizando-os para outros fins. A cordialidade da comunidade, a disponibilidade dos líderes do projeto, a existência de testes de unidade e documentação foram pontos importantes e que tornaram viável a contribuição dos desenvolvedores principais nesses projetos. As principais barreiras enfrentadas foram a falta de tempo dos próprios colaboradores e dos líderes, a comple-

[6]GitHub Enterprise: `https://github.com/enterprise`

xidade e o tamanho do projeto e a baixa qualidade do código fonte.

O estudo de Steinmacher et al. [2015] analisou qualitativamente as barreiras que os desenvolvedores periféricos enfrentaram ao iniciar sua colaboração em projetos *open source*. Para isso, foram entrevistados 36 desenvolvedores pertencentes a 14 projetos distintos. Os resultados indicaram a existência de 58 barreiras, com destaque para algumas barreiras vinculadas a fatores sociais, como a necessidade de um mentor para orientação sobre o projeto, o atraso nas respostas, as dificuldades de comunicação e as diferenças culturais. Lee et al. [2017] também estudaram as principais dificuldades enfrentadas pelos colaboradores periféricos. Entre os principais resultados, estão a falta de tempo do próprio contribuidor e a complexidade no processo de envio da contribuição, bem como a complexidade do projeto. Os resultados desse estudo também mostram que a maioria dos entrevistados não tinham interesse em tornar-se colaboradores frequentes no projeto.

Pinto et al. [2016] conduziram um estudo envolvendo 275 projetos *open source*. O trabalho envolveu a elaboração de um *survey*, com o objetivo de compreender as características dos desenvolvedores casuais, também conhecidos como periféricos. Os resultados apontam que os colaboradores casuais representam 48,98% dos contribuidores dos projetos analisados, contudo eles são responsáveis apenas por 1,73% do total de *commits*. Os dados também indicaram que a principal forma de atuação desses desenvolvedores é por meio da correção de *bugs*, acréscimo de novas funcionalidades e melhorias na documentação.

Zhou & Mockus [2014] estudaram o envolvimento de novos desenvolvedores em projetos *FLOSS* e a probabilidade de eles tornarem colaboradores por um longo período, a partir da análise das atuações em ambientes de rastreamento de problemas, conhecidos como *issue tracking system*. Os resultados sugerem que as experiências iniciais dos novos colaboradores são importantes para se tornarem contribuintes de longo prazo nos projetos.

Robles et al. [2009] estudaram a evolução dos desenvolvedores principais (*core developers*) em 19 projetos *open source*. Os contribuidores que efetuaram mais que 10% do total de *commits* do projeto foram classificados como *heroes*. A duração de cada período no processo de avaliação do ciclo evolutivo, foi definida pelo resultado da divisão do número de meses de existência do projeto por 10. Por essa razão, o total de períodos analisado variou entre os projetos. Assim, os desenvolvedores mais ativos em diferentes períodos foram identificados e suas atividades calculadas ao longo do tempo, procurando padrões de evolução entre os *core developers*. Foram identificados dois padrões dentre os projetos analisados: o cenário dos "deuses do código" (*"code gods"*) e o cenário "série de gerações" (*"series of generations"*). O primeiro caso ocorre quando

existe um grupo de desenvolvedores principais altamente estável durante o ciclo de vida do sistema, o que acarretaria um grande risco a sustentabilidade do projeto, caso uma parcela significativa deles saísse. O cenário "série de gerações" ocorre quando vários grupos de desenvolvedores principais temporários se sucedem no decorrer da vida do software.

Avelino et al. [2019b] analisaram a autoria de código em 66 *releases* do *kernel* do *Linux*, correspondendo a mais de 13 anos do histórico de *commits*. As análises utilizaram como base a métrica *DOA*, definida por Fritz et al. [2014]. Os resultados indicaram que 75% dos contribuidores do projeto são responsáveis, no máximo, por 10 arquivos, enquanto uma pequena parcela de autores, cerca de 2% do total, é responsável por centenas de arquivos. Posteriormente, o estudo foi replicado em um conjunto de 118 projetos *open source* e apresentou padrões de autoria similares aos observados no estudo do *kernel* do Linux.

Dentre os estudos recentes, há poucos que se dedicam a avaliar a distribuição das contribuições dos *heroes* no decorrer do ciclo de vida do software. Apesar de Robles et al. [2009] e Avelino et al. [2019b] realizarem análises nesse âmbito, tais estudos dedicam-se na compreensão da atuação do grupo de autores, classificados como contribuidores principais analisando de forma global o conjunto de dados de autoria de um grupo de projetos. O presente trabalho de mestrado realiza estudos de casos de cinco projetos *open source*, com a finalidade de analisar individualmente a atuação de cada *hero* do projeto. Além disso, este trabalho compara a relação da atuação dos *heroes* com o crescimento do número de contribuições totais do projeto e a distribuição do conhecimento. A análise segmentada por períodos de tempo também permite analisar o processo de crescimento e redução do total de contribuições de cada um dos *heroes*. O estudo sobre autoria de software está em discussão na literatura recente e os resultados deste trabalho contribuem com essa discussão, investigando a forma como os *heroes* atuam ao longo da evolução do sistema.

3.4 Considerações Finais

Foram apresentados, neste capítulo, os principais trabalhos relacionados ao tema desta dissertação de mestrado. Os trabalhos prévios encontrados na literatura indicam que a evolução da distribuição do conhecimento durante o ciclo de vida do software é uma área que possui poucos estudos recentes. Constatou-se também que existem diferentes formas de medir a autoria e propriedade de código, porém ainda não há um consenso na literatura sobre isso. Do mesmo modo, há diferentes granularidades adotadas para

coleta e análise de dados de autoria. Neste trabalho, a propriedade de código é medida seguindo a linha utilizada por Foucault et al. [2014] e Bird et al. [2011], com a granularidade de *commits*.

O presente trabalho dedica-se a avaliar, por meio do estudo de caso de cinco projetos *open source*, a distribuição do conhecimento de software entre os desenvolvedores e a variação da métrica de propriedade de código, analisados sob perspectiva do ciclo evolutivo dos projetos. Pretende-se por meio deste trabalho, aprofundar e contribuir com os estudos na área de autoria e propriedade de código. No próximo capitulo é apresentado a metodologia aplicada no trabalho, contendo os passos adotados para a sua realização.

Capítulo 4

Projeto do Estudo

Para compreender como se dá a distribuição do conhecimento do software entre os desenvolvedores de um projeto e como esse conhecimento evolui com o tempo de vida do software, foi realizado um estudo empírico com projetos *open source* hospedados no GitHub. Este capítulo apresenta a metodologia aplicada no estudo, detalhando as questões de pesquisa e descrevendo como foram realizados a coleta de dados, a análise quantitativa dos dados e a análise qualitativa dos resultados, conforme retratado na Figura 4.1. Essas descrições são feitas de forma geral neste capítulo. Nos capítulos subsequentes, os detalhes de execução dessas etapas são apresentados.

Figura 4.1. Representação da metodologia adotada no trabalho

4.1 Questões de Pesquisa

As questões de pesquisa investigadas neste estudo são descritas a seguir.

QP1: O conhecimento do software se difunde ao longo de seu ciclo de vida?

Esta questão de pesquisa busca compreender como o conhecimento do software é propagado durante o ciclo evolutivo do software. A partir da análise das contribuições dos desenvolvedores, busca-se identificar se o conhecimento do software permanece majoritariamente concentrado em alguns poucos desenvolvedores, como afirma Foucault et al. [2014], ou, se pelo contrário, em determinada etapa do desenvolvimento as contribuições atingem uma distribuição uniforme e o projeto passa a ter os chamados "não-especialistas".

A métrica da propriedade de código, calculada com base nas definições de Bird et al. [2011] e Foucault et al. [2014], foram utilizadas neste trabalho para a análise da distribuição do conhecimento no decorrer do processo evolutivo do software. Conforme descrito na Seção 2.3, essa métrica é dada em função dos *commits* realizados pelos desenvolvedores.

QP2: Os desenvolvedores considerados *heroes* nas primeiras versões permanecem *heroes* durante a evolução do projeto?

A partir desta questão de pesquisa, pretende-se compreender como as contribuições dos *heroes* se comportam no decorrer do ciclo de vida do software. Deseja-se investigar se os principais desenvolvedores do início do projeto permanecem com o conhecimento concentrado neles até os últimos períodos analisados, ou se surgiram outros grandes contribuidores no decorrer do processo evolutivo.

Além disso, esta questão também auxiliará no entendimento acerca de como os contribuidores deixam de ser *heroes* em um projeto *open source*. Por ventura, ocorre uma natural diminuição da sua porcentagem total de contribuições, em vista do crescimento da quantidade de contribuições de outros desenvolvedores? Ou aqueles desenvolvedores deixam de ser *heroes*, de modo abrupto, ou seja, apenas quando abandonam o projeto?

QP3: A concentração do conhecimento é impactada pela quantidade total de contribuições do projeto?

Esta questão de pesquisa tem por finalidade estudar a variação da quantidade total de contribuições realizadas em um projeto no decorrer de sua evolução e relacioná-la com o valor da métrica propriedade de software [Bird et al., 2011;

Foucault et al., 2014], no mesmo período. Pretende-se identificar se existe uma correlação inversa entre esses fatores. Intuitivamente, poder-se-ia pensar que a medida que a quantidade de contribuições no repositório aumenta, ocorre uma maior disseminação do conhecimento, e por essa razão, a propriedade de código seria menor. Isso ocorreria pelo fato de que quanto maior a quantidade de *commits*, mais autores estão trabalhando no repositório e mais o conhecimento se propaga. Portanto, essa questão investigará a existência dessa relação e se há um comportamento padrão entre os projetos.

QP4: Como se dá a atuação dos desenvolvedores periféricos nos projetos?

Por meio dessa questão de pesquisa, deseja-se compreender se os desenvolvedores periféricos enfrentam dificuldades que os impeçam de realizar contribuições frequentes no repositório *open source*. Pretende-se, também, entender a forma de atuação desses autores, o tipo de contribuição que realizam, o modo como interagem com outros desenvolvedores do projeto e as razões que o levaram a contribuir naquele software.

QP5: Como ocorre a disseminação de conhecimento do software entre os desenvolvedores?

Esta questão de pesquisa visa compreender como o conhecimento do software é difundido entre as diferentes categorias de autores que compõe um projeto *open source*.

A propagação do conhecimento pode ocorrer por meio das contribuições efetuadas no código fonte, pois, com isso, o desenvolvedor adquire uma *expertise* importante em relação à regra de negócio e ao padrão de codificação utilizados naquele contexto. Contudo, o conhecimento também é disseminado entre os autores do projeto por meio da comunicação formal ou informal entre eles. Isso pode ocorrer, por exemplo, por meio de documentações, reuniões periódicas ou conferências, e-mails, *code-review* ou mesmo em conversas informais pessoalmente ou por mensagens de texto.

Busca-se detectar também se existem dificuldades entre os contribuidores para adquirir informações imprescindíveis para o desenvolvimento de suas atividades ou para transmitir conteúdos importantes, que podem ajudar ou impactar outros desenvolvedores. Essas dificuldades podem surgir com mais frequência em sistemas *open source*, uma vez que uma considerável parcela dos autores trabalha remotamente, possivelmente não se conhecem pessoalmente e não estão vinculados formalmente à empresa responsável pelo projeto.

4.2 Coleta de Dados

Nesta etapa do trabalho, foram definidos os softwares objetos de estudo para compor o *data set*, considerando a sua relevância no GitHub. As principais características consideradas para a escolha dos projetos foram: o número de estrelas, a quantidade de *commits*, o total de contribuidores, o número de *forks* e *pull requests*. Os projetos escolhidos foram: *Bootstrap*[1], *Django*[2], *Elasticsearch*[3], *Panda*[4] e *Spring Boot*[5].

Além disso, foram desenvolvidos *scripts* para realizar a coleta dos dados, bem como quais dados deveriam ser obtidos. O critério considerado para a medição da janela evolutiva de tempo dos projetos também foi estabelecido, o que possibilita análises comparativas entre os autores e os projetos, a partir de seu ciclo de vida.

Após a coleta, os dados passaram por um procedimento de tratamento, no qual são identificados os desenvolvedores com identificação ambíguos. Isso ocorre quando um mesmo desenvolvedor possui dois ou mais e-mails associados aos *commits* daquele projeto. Esse processo é importante para a confiabilidade da análise e a validade dos resultados obtidos.

Os detalhes dessa etapa estão descritos no Capítulo 5. Todos os *scripts* desenvolvidos, bem como os resultados obtidos a partir da coleta de dados, estão disponíveis *online*[6], permitindo a replicação do trabalho e auxiliando a coleta em estudos futuros.

4.3 Caracterização dos Dados e Análise Quantitativa

A partir da coleta efetuada nos repositórios GitHub, foram realizadas caracterizações e análises dos dados, a fim de responder as questões de pesquisa QP1, QP2 e QP3. Os estudos quantitativos consideraram algumas informações primordiais dentre os dados coletados, sendo eles: os autores que contribuíram no projeto, a quantidade de *commits* associados a cada um deles e a data de sua atuação.

A partir disso foram realizadas análises como:

- a distribuição da população de autores

[1]Bootstrap: `https://github.com/twbs/bootstrap`
[2]Django: `https://github.com/django/django`
[3]ElasticSearch: `https://github.com/elastic/elasticsearch`
[4]Panda: `https://github.com/pandas-dev/pandas`
[5]Spring Boot: `https://github.com/spring-projects/spring-boot`
[6]Coleta de Dados: `https://github.com/taliorfano/knowledgedistribution_master/tree/master/Coleta_Dados`

- a categorização da população de autores em função da quantidade de contribuições deles
- a distribuição dos *heroes* no ciclo evolutivo do software
- a variação da métrica de propriedade de código no decorrer do ciclo de vida e a frequência de contribuições efetuadas em cada projeto considerando as transições existentes durante a vida do software.

O Capítulo 6 expõe essas análise e seus resultados.

4.4 *Survey* e Análise Qualitativa

Nesta etapa, foram criados *surveys*, com a finalidade de compreender de forma qualitativa o comportamento dos desenvolvedores dentro de um projeto *open source*. Pretende-se, com o *survey*, entender as motivações que levaram os desenvolvedores a contribuirem naquele projeto, o tipo de contribuição realizada, a forma de atuação no projeto, os meios que ele utiliza para solucionar as dúvidas associadas às suas atividades, bem como o modo que o conhecimento adquirido é propagado para os outros autores. Essa análise visa responder as questões de pesquisa QP4 e QP5.

Em seguida, as análises das respostas obtidas com as pesquisas são evidenciadas, observando se existe algum padrão entre elas. Essa etapa do trabalho é tratada no Capítulo 7.

4.5 Considerações Finais

Este capítulo retratou a metodologia aplicada nesta dissertação, detalhando as questões de pesquisa e apresentando como foram realizadas a coleta de dados, a análise quantitativa e a análise qualitativa dos dados. A partir das etapas descritas nesse capítulo, pretende-se atingir os objetivos propostos, bem como prover com os resultados um auxílio tanto para a comunidade científica, por meio do aprofundamento da discussão sobre propriedade de código, distribuição do conhecimento e evolução de software, quanto para os responsáveis e membros de projetos *open source*, evidenciando a importância da gestão e acompanhamento dos seus contribuidores, pois são eles os detentores do conhecimento do software e as peças fundamentais para a sua evolução e manutenção.

Capítulo 5

Coleta de Dados

Este capítulo apresenta o *data set* contendo os repositórios escolhidos para o presente estudo. Também são explanados os critérios utilizados na seleção, bem como os métodos estudados para realização das coletas de dados no GitHub. O tratamento dos dados e as informações geradas por eles também são abordados neste capítulo.

5.1 Critérios de Escolha

O escopo deste trabalho delimita-se a estudar repositórios *open source*. A plataforma de hospedagem de código para controle de versão e colaboração mais popular entre os desenvolvedores *open source*[1] é o GitHub e, por essa razão, foi o repositório de softwares utilizado neste trabalho. O GitHub abrange desde repositórios públicos *open source* a projetos privados particulares ou empresariais. Ele permite hospedar e revisar códigos, gerenciar projetos e agrega mais de 40 milhões de desenvolvedores. Além disso, ele possui uma interface de fácil uso e uma vasta e detalhada documentação[2].

O estudo foi realizado em cinco projetos populares hospedados no GitHub. Para seleção dos projetos foram considerados a relevância quanto ao número de estrelas, quanto ao total de autores e *commits*, bem como em relação ao número de *releases* publicadas e *forks* efetuados. Os cinco repositórios foram selecionados dentre as três linguagens de programação mais populares no GitHub: JavaScript, Python e Java[3].

[1] A maior comunidade *open source* no mundo: `https://github.com/open-source`
[2] Sobre a plataforma GitHub: `https://github.com/`
[3] Linguagens mais populares no GitHub: `https://octoverse.github.com/`

5.2 Definição do *Data Set*

Neste estudo, foi selecionado um conjunto de cinco projetos com diferentes características e funcionalidades: *Bootstrap*[4], *Django*[5], *Elasticsearch*[6], *Panda*[7] e *Spring Boot*[8]. Para a seleção desses repositórios foram consideradas as restrições estabelecidas na Seção 5.1. A Tabela 5.1 destaca as principais características consideradas na escolha desses repositórios.

Tabela 5.1. Repositórios selecionados no GitHub

Repositório	# estrelas	# contribuidores	# commits	# releases	# forks
Bootstrap	139k	1.285	18.359	57	68.3k
Django	47.5k	2.084	26.491	275	20.5k
Elasticsearch	47.2k	1.223	36.185	267	16.1k
Panda	23.8k	1.702	18.581	112	9.5k
Spring Boot	45.7k	671	19.953	148	28.8k

Bootstrap

É um *framework* para desenvolvimento de componentes *front-end* utilizando HTML, CSS e JavaScript. Ele possibilita a criação de aplicações *web* com interface amigável e responsiva para dispositivos móveis. Bootstrap foi inicialmente desenvolvido por Mark Otto e Jacó Thornton para o site Twitter, como um instrumento para incentivar a consistência por meio de ferramentas internas. Atualmente é o *framework* mais popular do segmento.[9]

Django

É um *framework* desenvolvido em Python que torna mais rápido e fácil a construção de aplicações Web. Utiliza o conceito DRY (*Don't Repeat Yourself*), levando o desenvolvedor a reaproveitar o código já feito. Ele também cuida da autenticação do usuário, administração de conteúdo, mapas do *site*, *feeds* RSS, além de ser seguro e altamente escalável, capaz de atender às demandas de tráfego mais pesadas.[10]

[4]Bootstrap: `https://github.com/twbs/bootstrap`
[5]Django: `https://github.com/django/django`
[6]ElasticSearch: `https://github.com/elastic/elasticsearch`
[7]Panda: `https://github.com/pandas-dev/pandas`
[8]Spring Boot: `https://github.com/spring-projects/spring-boot`
[9]GetBootstrap: `https://getbootstrap.com/`
[10]Django Project: `https://www.djangoproject.com/`

Elasticsearch

É uma ferramenta *open source* desenvolvida em Java sobre o Apache Lucene que permite busca e análise de um grande conjunto de dados em tempo real, sejam dados textuais, numéricos, geoespaciais, estruturados ou não estruturados. A velocidade e escalabilidade oferecidas pelo *Elasticsearch*, bem como sua capacidade de indexar muitos tipos de conteúdo, permitem com que ele seja utilizado em diversos contextos e circunstâncias como por exemplo: buscas em aplicação, buscas em *website*, buscas empresarial, análise de dados de *log*, análise e visualização de dados geoespaciais, análise de segurança, análise de dados empresarial, dentre tantas outras aplicações.[11]

Pandas

É uma ferramenta *open source* desenvolvida em Python, sob licença BSD, que permite a análise e manipulação de dados de forma fácil e flexível ao usuário[12]. Pandas foi desenvolvida para ser um sistema de uso geral e linguagem de computação científica, reunindo em um único lugar várias plataformas de computação estatística e linguagens de banco de dados específicas do domínio, e também fornecer novos recursos como o alinhamento automático de dados e a indexação hierárquica. Foi inicialmente desenvolvida para aplicações de análise de dados financeiros, mas tem por objetivo ser fácil e flexível a todos usuários acadêmicos e da indústria na manipulação de dados estatísticos.[13]

Spring Boot

Spring[14] é um *framework open source*, considerado atualmente como o projeto mais popular entre os desenvolvedores Java. Ele busca prover alta produtividade, simplicidade e rapidez na execução das atividades cotidianas do desenvolvedor. Fornece módulos como persistência de dados, com mecanismos de segurança e controle de transações; desenvolvimento Web; acesso remoto e programação orientada por aspectos. Sua arquitetura é baseada na inversão de controle (IoC) e injeção de dependência (DI), assim proporciona sempre um baixo acoplamento entre as classes da aplicação orientada por objetos. O *Spring Boot*[15] é um projeto da Spring que visa facilitar o processo de configuração e publicação de novas

[11] What is Elasticsearch: `https://www.elastic.co/pt/what-is/elasticsearch`

[12] Pandas Pydata: `https://pandas.pydata.org/`

[13] Foundational Python Library for Data Analysis and Statistics: `www.dlr.de/sc/Portaldata/15/Resources/dokumente/pyhpc2011/submissions/pyhpc2011_submission_9.pdf`

[14] Why Spring: `https://spring.io/why-spring`

[15] Spring Boot Project: `https://spring.io/projects/spring-boot`

aplicações, aumentando com isso a produtividade e possibilitando ao desenvolvedor gastar menos tempo com atividades de configuração e mais tempo com as regras de negócio próprias da aplicação. Ele cria aplicações *Spring* independentes, incorpora diretamente o *Tomcat*, *Jetty* ou *Undertow*, fornece dependências para simplificar as configurações de compilação, configura automaticamente bibliotecas *Spring* e até de terceiros, fornece recursos prontos para produção, como métricas e verificações de integridade; sem nenhuma geração de código nem pedido de configuração em arquivos XML.

5.3 Coleta de Dados

Uma vez que os projetos escolhidos estão presentes no GitHub, fez-se necessária a definição de quais dados deveriam ser coletados e também qual ferramenta adotada para realizar a coleta dos dados de autoria de software. Essa seção descreve esses aspectos.

Métodos de coleta

A análise de conhecimento e autoria em um repositório pode ser efetuada utilizando diferentes granularidades: por *commits*, por arquivos ou por linhas de um arquivo. Observando os trabalhos relacionados presentes no Capítulo 3, bem como o custo para processar, armazenar e analisar a grande gama de informações geradas por coletas de granularidade menor, optou-se por avaliar o conhecimento de software no nível de contribuições, ou seja, a partir do total de *commits* que cada autor efetuou no projeto.

Ferramentas para coleta

Algumas soluções para mineração de dados no GitHub foram avaliadas, dentre elas pode-se destacar: GitHub API[16], GHTorrent[17] e GitHub Archive[18] com o Google BitQuery[19]. Uma vez considerada a facilidade de utilização, a ampla documentação e a completude das informações coletadas diretamente dos repositórios originais, a GitHub API foi a opção escolhida. A desvantagem dessa solução é a limitação de 5.000 requisições por hora pela API, contudo, isso não impactou significativamente o procedimento.

[16]GitHub API: `https://developer.github.com/v3`
[17]GH Torrent: `https://ghtorrent.org`
[18]GH Archive: `https://www.gharchive.org`
[19]BigQuery: `https://cloud.google.com/bigquery/docs`

A API REST escolhida é a API disponibilizada oficialmente pelo GitHub e com ela é possível consultar informações do projeto, detalhes dos *commits*, informações dos contribuidores, número de estrelas, *forks*, *issues*, dentre outras informações. Os resultados são obtidos no formato *JSON*. As requisições são feitas diretamente em `api.github.com`. Para conquistar o limite estendido de cinco mil requisições, faz-se necessário uma autenticação com os dados de acesso do GitHub, como por exemplo, o nome de usuário e um *token* de acesso.

Solução construída

O processo de coleta de dados foi realizado por uma aplicação de autoria própria desenvolvida em Python, com o auxílio da API REST GitHub. Os *scripts* utilizados para as coletas estão disponíveis no repositório pessoal da autora desta dissertação no GitHub.[20]

Os dados coletados foram armazenados no banco de dados MongoDB[21] e em arquivos com extensão *CSV* para consultas posteriores. A escolha de manter os dados em uma base de dados não relacional, como é o caso do MongoDB, ocorreu pelos seguintes motivos: alto desempenho de leitura e escrita de uma grande carga de dados; gratuidade da hospedagem dos dados na MongoDB Cloud[22]; a facilidade de manipulação e análise dos dados ali presentes; além da simplicidade de conexão do banco a partir da linguagem de programação adotada.

O procedimento obteve os seguintes dados dos cinco repositórios em estudo:

- código identificador do *commit (sha)*
- informações essenciais do contribuidor (nome, e-mail e *login*)
- data do *commit* e nome do repositório no GitHub.

Todos os *commits* recuperados são da *branch default* (*master*) de cada projeto e abrange todo o seu ciclo de vida, isto é, desde o primeiro *commit* até a data no qual as coletas foram efetuadas.[23]

[20] `https://github.com/taliorfano/knowledgedistribution_master/tree/master/Coleta_Dados/Scripts`

[21] MongoDB: `https://www.mongodb.com`

[22] MongoDB Cloud Database Solution: `https://www.mongodb.com/cloud`

[23] As coletas foram realizadas no primeiro semestre de 2019.

5.4 Definição de Janela de Tempo

Para obter-se um padrão de comparação em comum entre os diferentes repositórios analisados, estabeleceu-se que a janela evolutiva deveria ser medida em função de um intervalo T. Esse intervalo é utilizado nas análises quantitativas e qualitativas, apresentadas nos Capítulos 6 e 7. As possibilidades avaliadas para mensuração do intervalo T abrangeram desde o número de *releases* do projeto até os períodos mensais, trimestrais ou anuais de desenvolvimento do projeto.

A utilização do número de *releases* para mensuração do intervalo foi a primeira opção avaliada, mas, visto que cada projeto possui suas próprias definições quanto a frequência e modo de criação de *releases* - seja *major*, *minor* ou *patch* - foi descartada. Poderia ocorrer uma comparação errônea, por exemplo, entre um projeto na quarta *release* que realizou várias *releases minors* em um longo espaço de tempo, e um outro projeto que em um curto período migrou da *release* 4 para 5. Claramente a evolução dos desenvolvedores no conhecimento do projeto, suas chances de abandono ou mesmo a mudança de papel ou responsabilidade dentro do repositório são distintas nesses dois exemplos. Portanto, concluiu-se que a forma de medir a janela T deveria ser a partir de um período de tempo que pode ser igualmente medido, independente das particularidades do projeto.

A segunda alternativa avaliada para medida de T foi o período trimestral. Assim, a cada três meses decorridos do projeto, uma nova janela de tempo era analisada. Desse modo, um projeto que iniciou-se há três anos, teria um total de doze intervalos de tempos para análise no ciclo evolutivo. Essa abordagem foi inicialmente considerada adequada para medição de T. Contudo, após as coletas do primeiro projeto, *Django*, percebeu-se a dificuldade de trabalhar com tamanha quantidade de intervalos T, especialmente na análise quantitativa. O projeto *Django* foi iniciado em 2005, assim, só nesse projeto haveria mais de 50 intervalos de tempo para análise. Além disso, uma análise global dos dados evidenciou que as varições entre os dados de janelas subsequentes eram pequenas, quando não inexistentes. Por essa razão, a solução que contemplava as coletas a partir de períodos de tempo trimestrais foi abandonada e também, consequentemente, a alternativa que contemplava períodos T mensais.

A opção adotada para definição da janela de tempo T foi a solução anual. Dessa maneira, a cada novo ano decorrido do projeto, uma nova janela T é iniciada. Para melhor entendimento, segue um exemplo ilustrativo: um projeto A iniciou-se em Julho/2009 e outro projeto B em Abril/2012, o T1 do projeto A ocorre durante o ano de 2009 e do projeto B no ano de 2012; quando ocorre a mudança de ano, ou seja, Jan/2010, o projeto A inicia a janela T2 e em Jan/2013 o projeto B também inicia a

janela T2. Essa solução permite que o primeiro ano de cada projeto chamado T1 tenha meses diferentes, mas a partir de T2 a janela de tempo é sempre coincidente entre todos os projetos. Além disso, essa abordagem anual torna mais simples e compreensível as análises evolutivas do ciclo de vida dos projetos. A última janela de tempo considerada em todos os projetos coincide com o ano de 2018, uma vez que as coletas foram efetuadas no primeiro semestre de 2019. A Tabela 5.2 expõe as informações referentes ao período de avaliação e a quantidade de janelas de tempo T de cada repositório estudado nesse trabalho.

Tabela 5.2. Repositórios selecionados e as janelas de tempo T

Repositório	Ano inicial	Total de janelas T
Bootstrap	2011	8
Django	2005	14
Elasticsearch	2013	6
Panda	2009	10
Spring Boot	2012	7

5.5 Coleta de Informações

Nessa seção são apresentadas as informações geradas a partir dos dados inseridos no banco MongoDB. Para cada repositório, foram sumarizados o total de contribuições por autor, bem como o total de contribuições por autor em uma dada janela de tempo T.

5.5.1 Contribuições por Autor

Este trabalho visa compreender como se dá a distribuição de conhecimento do software entre os desenvolvedores. Considerando que esse conhecimento é medido pelo número de *commits* realizados, é primordial identificar quais autores mais contribuíram no repositório desde sua criação, da mesma forma é fundamental entender como é distribuído o total de *commits* entre os contribuidores daquele projeto.

Nesta primeira etapa, um *script Python* de autoria própria foi utilizado para realização de consultas na base MongoDB, com o objetivo de agrupar o total de *commits* efetuados por cada autor. Como resultado desse procedimento, foi concebido um arquivo CSV para cada um dos repositórios do *data set*, contendo os dados do autor (nome e e-mail) e o total de contribuições efetuadas por ele.

Os *scripts* utilizados para a busca dos dados de cada um dos projeto e os arquivos CSV gerados nas coletas, estão disponíveis para consulta.[24]

5.5.2 Contribuições por Período de Tempo *T*

Um dos objetivos do trabalho é compreender como ocorre a distribuição de conhecimento entre os autores durante o ciclo de vida do software. Portanto, não bastaria apenas conhecer a quantidade de contribuições de cada autor, mas também é importante compreender como essas contribuições cresceram ou diminuíram no decorrer do ciclo de vida do projeto. Também interessa saber como foi o processo de evolução daqueles que são considerados *heroes* do projeto.

Sendo assim, na segunda etapa da coleta de informações, foi calculado o total de *commits* feitos por cada autor segmentado pela janela de tempo T, definida na Seção 5.4. Para isso também foi desenvolvido um *script* na linguagem *Python* para manipulação dos dados armazenados no MongoDB. O resultado desse programa é a geração de um arquivo CSV para cada intervalo T, isto é, para cada ano de vida do repositório em análise. Dessa forma, um projeto como o ElasticSearch que possui 6 janelas de tempo, terá como resultado do programa seis arquivos com extensão CSV, cada um contendo os dados dos autores (nome e e-mail) e o total de suas respectivas contribuições naquele período. Todos os arquivos com as informações coletadas, assim como o *script* construído podem ser encontrados no GitHub.[25]

5.6 Tratamento de Ambiguidade de Autores

Dentre os principais dados coletados, estão as informações do autor responsável pelas contribuições realizadas no projeto. Contudo, frequentemente são encontrados nos repositórios contribuidores ambíguos, isto é, usuários que no mesmo projeto utilizam nomes ou e-mails diferentes e, por isso, são identificados erroneamente como sendo duas ou mais pessoas distintas. Essas particularidades, se não forem tratadas podem impactar os resultados dos estudos. Portanto, essa seção aborda algumas metodologias avaliadas para esse procedimento, bem como as decisões adotadas para o tratamento da base coletada.

[24] `https://github.com/taliorfano/knowledgedistribution_master/tree/master/Coleta_Dados`

[25] `https://github.com/taliorfano/knowledgedistribution_master/tree/master/Coleta_Dados/Dados`

5.6.1 Por *Login*

Assumindo que o *login* no GitHub é único e intransferível, considerou-se inicialmente essa a melhor opção para tratamento de ambiguidade. Desse modo, ao agrupar o total de contribuições realizadas por cada autor, obtivemos um arquivo com duas colunas: *login* e quantidade de *commits* do repositório analisado.

No entanto, em uma etapa posterior ao analisar cuidadosamente os dados e os elementos da API, percebeu-se que o login retornado corresponde ao usuário *committer*, ao invés do usuário autor. Em muitos casos esses usuários coincidem, contudo existem algumas situações em que eles são diferentes afetando, com isso, os resultados da coleta e a respectiva análise. Isso pode ocorrer, por exemplo, em um *commit* de *merge* entre *branches*, no qual os arquivos *commitados* não foram desenvolvidos pelo usuário responsável pelo *commit*. Um outro exemplo, pode acontecer do *login* estar vinculado ao *committer* porque o autor não tem conta no GitHub. Portanto, devido esse comportamento, concluiu-se que a desambiguação por meio do *login* não é adequada. A Figura 5.1 exemplifica uma das situações descritas, encontrada em um *commit* coletado do repositório Django.


```
"sha": "55ffcf8e7b414a39e2dfc9c9eb4c5d3fa548e78e",
"node_id": "MDY6Q29tbWl0NDE2NDQ4Mjo1NWZmY2Y4ZTdiNDE0YTM5ZTJkZmM5YzllYjRjNWQzZmE
"commit": {
  "author": {
    "name": "Raúl Cumplido",
    "email": "raulcd@tid.es",
    "date": "2012-06-07T11:46:06Z"
  },
  "committer": {
    "name": "Tim Graham",
    "email": "timograham@gmail.com",
    "date": "2012-06-30T21:16:40Z"
  },
"url": "https://api.github.com/repos/django/django/commits/55ffcf8e7b414a39e2df
"html_url": "https://github.com/django/django/commit/55ffcf8e7b414a39e2dfc9c9eb
"comments_url": "https://api.github.com/repos/django/django/commits/55ffcf8e7b4
"author": null,
"committer": {
  "login": "timgraham",
  "id": 411869,
  "node_id": "MDQ6VXNlcjQxMTg2OQ==",
  "avatar_url": "https://avatars3.githubusercontent.com/u/411869?v=4",
  "gravatar_id": "",
  "url": "https://api.github.com/users/timgraham",
```


Figura 5.1. Dados de um *commit* no Django disponibilizados pela API GitHub

5.6.2 Por Nome/E-mail

Um modo de avaliar a ambiguidade dos responsáveis pelos *commits* em um projeto é por meio da análise comparativa dos nomes e e-mails dos contribuidores. Existem duas formas popularmente conhecidas para efetuar essa desambiguação: usando algoritmos de desambiguação ou análise manual. Esta seção aborda a metodologia, a vantagem e a desvantagem de cada uma delas, bem como as razões que motivaram a decisão tomada neste trabalho.

Orfanó et al. [2018] estudaram e compararam cinco heurísticas de desambiguação de autores em projetos *open source*: *Bird* [Bird et al., 2006], *Canfora* [Canfora et al., 2011], *Simple* [Kouters et al., 2012], *Robles* [Robles & Gonzalez-Barahona, 2005] e *Goeminne* [Goeminne & Mens, 2011]. Esses algoritmos comparam a similaridade entre os nomes e os e-mails dos autores. Os melhores resultados foram obtidos por aquelas heurísticas que apresentaram 100% do coeficientes de revocação (*recall*) em todas as categorias de projeto. No entanto, elas apresentaram baixa precisão quando utilizadas em repositórios que possuem mais de 500 autores. Mesmo sob essas circunstâncias, optou-se por avaliar os resultados gerados por uma dessas heurísticas nas coletas efetuadas em nosso trabalho.

O algoritmo proposto por Bird et al. [2006] foi o selecionado devido à simplicidade de sua implementação, a facilidade de compreensão e por ter sido uma das heurísticas mais bem avaliadas nos resultados do estudo comparativo supracitado.

O Algoritmo 1 exibe as condições consideradas na heurística de *Bird*, responsável por determinar a ambiguidade entre dois autores. Esse algoritmo compara a similaridade entre o primeiro e último nome, o nome completo e o prefixo de e-mail. As duas identidades <nome, e-mail> são consideradas como pertencentes a um mesmo autor caso a similaridade de *Levenshtein* seja maior ou igual ao valor de referência *t (threshold)*, definido por meio de experimentos como sendo 0,93. Além disso, também é verificado se o prefixo de e-mail de um desenvolvedor contém o nome, sobrenome ou derivações desses, semelhantes ao outro desenvolvedor. Antes de verificar a semelhança e detectar a ambiguidade, os dados dos autores foram normalizados e tratados pela retirada de acento, letras maiúsculas, caracteres especiais e outras particularidades [Orfanó et al., 2018]. A implementação do algoritmo foi realizada em Java e encontra-se disponível no GitHub.[26]

[26]Heurísticas de desambiguação: `https://github.com/taliorfano/comparescontributorsheuristics`

Algoritmo 1: Algoritmo de desambiguação proposto por Bird et al. (2006)

Entrada: Autor autor1, Autor autor2 *(composto por <nome, email>)*

Saída: variável de ambiguidade booleana

```
1  início
2  |  ambiguidade ⟵ falso;
3  |  t ⟵ 0,93;
4  |  Normalização dos dados (autor1, autor2);
5  |  se Similaridade do nome completo(autor1.nome, autor2.nome) >= t OU
6  |  (Similaridade primeiro nome(autor1.nome, autor2.nome) >= t E
   |   Similaridade último nome(autor1.nome, autor2.nome) >= t ) OU
7  |  Similaridade entre os prefixos de email(autor1.email, autor2.email) >= t
   |   OU
8  |  Prefixo contêm primeiro e último nome(autor1.email, autor2.nome) OU
9  |  Prefixo contêm primeiro e último nome(autor2.email, autor1.nome) OU
10 |  Prefixo contêm primeiro nome e inicial do último nome(autor1.email,
   |   autor2.nome) OU
11 |  Prefixo contêm primeiro nome e inicial do último nome(autor2.email,
   |   autor1.nome) OU
12 |  Prefixo contêm inicial do primeiro nome e o último nome(autor1.email,
   |   autor2.nome) OU
13 |  Prefixo contêm inicial do primeiro nome e o último nome(autor2.email,
   |   autor1.nome) então
14 |  |  ambiguidade ⟵ verdadeiro;
15 |  fim
16 |  retorna ambiguidade
17 fim
```

A heurística foi executada para o conjunto <nome, e-mail> dos autores coletados pertencentes a cada um dos cinco projetos em estudo. Os resultados foram avaliados manualmente e percebeu-se que existiam erros consideráveis na identificação dos autores. Assim, optou-se por desconsiderar os resultados realizados pela desambiguação automática. Os resultados completos gerados pela heurística para cada um dos projetos do *data set* encontram-se disponíveis no GitHub.[27] Alguns exemplos dos erros encontrados são mostrados a seguir.

[27]Desambiguação com heurística *Bird*: `https://github.com/taliorfano/knowledgedistribution_master/tree/master/Coleta_Dados/Desambiguacao`

Bootstrap:
- contact@helmutgranda.com, contact@weirdog.com e contact@dominicbarnes.us
- cesidio.dilanda@gmail.com, sidp@knights.ucf.edu e sid@sidroberts.co.uk

Django:
- bphillips (bphillips@caktusgroup.com) e Joshua Philips (jphillips@imap.cc)
- Thomas Orozco (thomas@orozco.fr) e Thomas Sorrel (thomas@pandeiro.fr)

Elasticsearch:
- ludo.helder@gmail.com e ludovic@xwiki.com
- Tim B (tim@uncontended.net) e Tim Vernum (tim@adjective.org)

Panda:
- joejev@gmail.com e joe@quantopian.com
- John Freeman (jfreeman08@gmail.com) e Jesse Farnham (jfarnham20@gmail.com)

Spring boot:
- alexander.abramov.pub@gmail.com e alexander.constantin@gmail.com
- Jay Anderson (jaanderson@shutterfly.com) e Jayaram Pradhan (jayaramimca@gmail.com)

A desambiguação manual é um processo mais custoso, porém com uma precisão e confiabilidade maior. Para esse procedimento, foram considerados o nome e o e-mail dos autores. Inicialmente, os nomes foram ordenados alfabeticamente e comparados entre si, aqueles que possuíam nomes iguais eram unificados. Posteriormente, os e-mails foram ordenados e o processo se repetiu. Aqueles que tinham nomes e/ou e-mails muito similares entre si, mas distintos, foram avaliados individualmente com o auxílio de pesquisas externas e na própria conta do GitHub, a fim de compreender se deveriam ser unificados.

Devido ao grande número de autores e o procedimento manual ser lento, decidiu-se por tratar apenas as ambiguidades que envolviam desenvolvedores *heroes*, médios e também os pequenos autores que possuíam um grande número de contribuições, isto é, aqueles que poderiam tornar-se autores da categoria médio, caso tivessem seus *commits* divididos em duas contas ambíguas. Esses casos foram priorizados porque se houvesse ambiguidade entre grandes e médios autores, o impacto nos resultados do trabalho seria significativo. Por outro lado, as ambiguidades existentes nos casos dos demais autores não impactariam nos resultados deste estudo. Esse tratamento foi feito após a etapa de Contribuições por Autor, apresentada na Seção 5.5.1. A Tabela 5.2 expõe o total de ambiguidades identificadas no tratamento manual para cada um dos projetos em análise. Os detalhes com os autores identificados como ambíguos estão disponíveis

para consulta.[28]

Tabela 5.3. Total de autores identificados como ambíguos entre si e agrupados

Repositório	Ambiguidades identificadas
Bootstrap	1
Django	14
Elasticsearch	10
Panda	10
Spring Boot	4

5.7 Considerações Finais

Foram apresentados nesse capítulo, os critérios de escolha adotados para seleção dos cinco repositórios que compõe o *data set* da pesquisa, sendo eles: *Bootstrap*, *Django*, *Elasticsearch*, *Panda* e *Spring Boot*.

A API REST GitHub foi escolhida para a coleta dos dados e foram desenvolvidos *scripts* em Python para auxiliar o processo de coleta e tratamento dos dados. Também foi discorrido sobre alguns métodos para o tratamento de desambiguação dos autores.

Por meio dos dados coletados, foram segmentados o total de contribuições realizadas por cada autor e também por cada período de tempo T, com o propósito de auxiliar nas discussões das questões de pesquisa propostas.

A análise quantitativa dos dados coletados e seus resultados são apresentados no próximo capítulo.

[28]Heurísticas de desambiguação: `https://github.com/taliorfano/knowledgedistribution_master/tree/master/Coleta_Dados/Desambiguacao`

Capítulo 6

Evolução da Distribuição do Conhecimento entre os Desenvolvedores

Este capítulo apresenta a análise quantitativa das informações coletadas nos repositórios GitHub. O objetivo da análise é compreender a atuação dos contribuidores e a distribuição de seu conhecimento no decorrer do ciclo de vida do projeto, com o intuito de responder as seguintes questões de pesquisa:

- QP1: O conhecimento do software se difunde ao longo de seu ciclo de vida?
- QP2: Os desenvolvedores considerados *heroes* nas primeiras versões permanecem *heroes* durante a evolução do projeto?
- QP3: A concentração do conhecimento é impactada pela quantidade total de contribuições do projeto?

Foram realizadas quatro análises. Seção 6.1 mostra a distribuição da população de autores nos projetos; Seção 6.2 aborda a categorização da população de autores; a Seção 6.3 apresenta a distribuição dos *heroes* no ciclo evolutivo e a Seção 6.4 descreve a expressividade da propriedade de código em cada projeto. Seção 6.5 apresenta a variação da quantidade de contribuições durante o ciclo de vida dos softwares. Por fim, Seção 6.7 discute os resultados apresentados visando responder as questões de pesquisa.

Figura 6.1. Total de contribuições efetuadas por autor no projeto *Bootstrap*, no qual 68% dos autores efetuaram apenas uma contribuição

Figura 6.2. Total de contribuições efetuadas por autor no projeto *Django*, no qual 65% dos autores efetuaram apenas uma contribuição

6.1 Distribuição da População de Autores

Considerando a grande quantidade de *commits* em cada repositório, apresentado na Tabela 5.1 (Capítulo 5), foi realizada uma análise para compreender como os *commits* se distribuem entre os autores do projeto. O objetivo desta análise foi observar se há

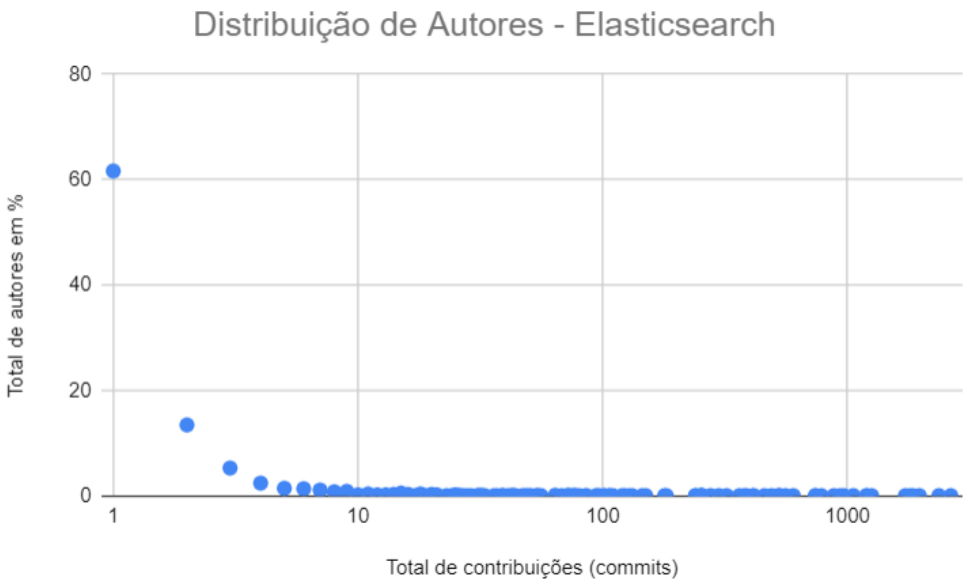

Figura 6.3. Total de contribuições efetuadas por autor no projeto *Elasticsearch*, no qual 62% dos autores efetuaram apenas uma contribuição

algum tipo de concentração no número de contribuições ou se elas são uniformemente distribuídas entre os autores.

As Figuras 6.1 a 6.5 mostram como o número de contribuições está distribuído entre os autores. O eixo horizontal, que está em escala logarítmica, corresponde ao total de *commits* que um autor fez em todo histórico do projeto analisado. O eixo vertical corresponde à porcentagem de desenvolvedores existentes no projeto que efetuou aquela quantidade de contribuições.

Observa-se, examinando as figuras supracitadas, que a maior parcela de autores realizaram poucas contribuições nos projetos. Os resultados dos cinco repositórios apontam que aproximadamente 65% dos autores realizaram somente uma contribuição no projeto e 15% efetuaram apenas duas contribuições. Portanto, pode-se considerar que ao menos 80% dos autores vinculados a um desses projetos possuem autoria de uma quantidade muito pequena sobre o código daquele software. Esse comportamento foi homogêneo em todos os projetos do *data set*.

Também é possível observar que a distribuição da população dos autores possui uma cauda longa, uma vez que o número de autores diminui exponencialmente e torna-se mais esparso à medida que o número de contribuições no repositório cresce. Os resultados evidenciam que são poucos os desenvolvedores que efetivaram mais de mil contribuições em seus respectivos projetos.

Portanto, essa análise indica que nos cinco repositórios *open source* estudados o

Figura 6.4. Total de contribuições efetuadas por autor no projeto *Pandas*, no qual 64% dos autores efetuaram apenas uma contribuição

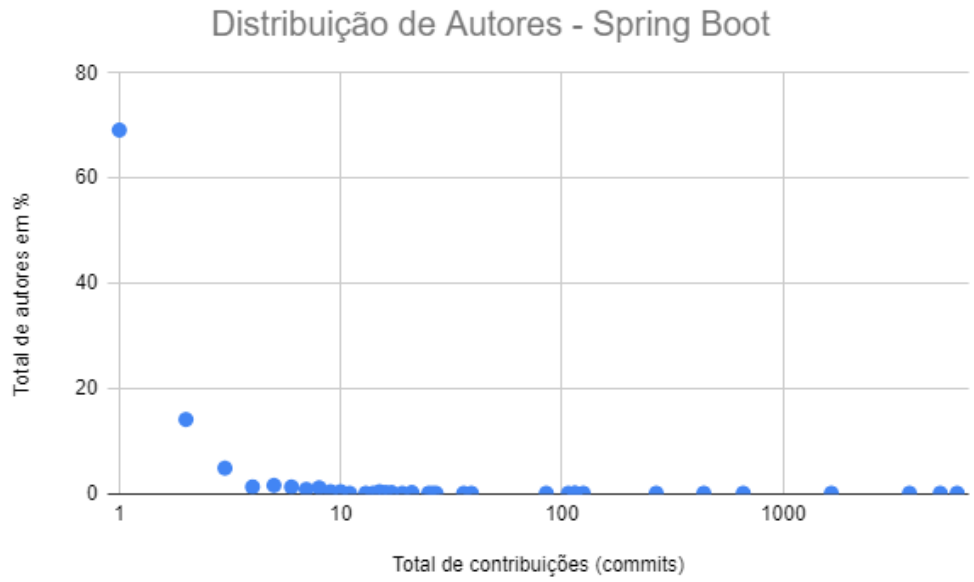

Figura 6.5. Total de contribuições efetuadas por autor no projeto *Spring Boot*, no qual 69% dos autores efetuaram apenas uma contribuição

conhecimento é concentrado em poucos contribuidores. Esse resultado está em concordância com achados de trabalhos prévios que também apontam para essa concentração de conhecimento [Avelino et al., 2019b; Greiler et al., 2015; Foucault et al., 2014; Ricca & Marchetto, 2010].

6.2 Caracterização da População de Autores

A diferente atuação do total de autores em um projeto assinala a necessidade de estudá-los de forma segmentada. Esta seção aborda como esses desenvolvedores foram segmentados.

Considerou-se dividir os contribuidores em duas categorias, como sugerido por Foucault et al. [2014]: os chamados *"heroes"*, que são aqueles poucos que contribuem em diversos módulos do projeto em um grande período de tempo, e os pequenos ou periféricos, que são aqueles que efetuaram uma pequena contribuição ao sistema. Segundo o mesmo estudo, os desenvolvedores *heroes* são aqueles que a soma das contribuições realizadas é maior ou igual a 5% do total do projeto. Consequentemente, o autor periférico é aquele que contribuiu com menos de 5% da soma total de *commits* do repositório.

No entanto, diante dos resultados obtidos e previamente analisados das coletas, observou-se uma discrepância no total de contribuições entre aqueles que seriam considerados dois autores periféricos de um mesmo projeto, como por exemplo um contribuidor que realizou um *commit* e outro que efetuou 300.

Sendo assim, optou-se por agrupar os desenvolvedores em três categorias: autores *heroes*, autores médios e autores pequenos ou periféricos. Assumiu-se, ainda, que os autores *heroes* são aqueles que contribuíram em pelo menos 5% do total dos *commits* existentes no projeto, seguindo o mesmo limiar utilizado por Foucault et al. [2014]. Foi considerado como autor médio aqueles cuja soma da contribuição foi maior ou igual a 1% e menor que 5% do total. O autor periférico é aquele que efetivou menos que 1% de contribuições do projeto.

Dessa forma, os autores pequenos ou periféricos são aqueles que possuem um conhecimento pequeno ou insuficiente em relação ao código do projeto, devido ao número limitado de contribuições. Os autores médios são aqueles que realizaram uma contribuição maior, mas não central, podendo-se enquadrar nessa categoria. Por exemplo, os que permaneceram no projeto por um razoável espaço de tempo e detiveram um significativo conhecimento do código nesse período, mas abandonaram o projeto antes de tornarem-se *heroes*. Uma outra possibilidade são aqueles desenvolvedores que ainda estão ativos no projeto e o número de suas contribuições está aumentando a cada iteração, indicando que em breve poderão se tornar autores *heroes*. Consideram-se autores *heroes* aqueles desenvolvedores que contribuíram amplamente no projeto durante seu ciclo de vida, possuíram ou ainda possuem uma grande parcela de conhecimento do sistema.

A Tabela 6.1 apresenta a segmentação das categorias de autores nos cinco pro-

Tabela 6.1. Categorização de autores e como as contribuições totais do projeto estão distribuídas entre eles

Projeto	Categoria	∑ autores	∑ *commits*	Percentual de contribuição (%)
Bootstrap	Autor periférico	1.278	3.579	19,49
	Autor médio	3	811	4,42
	Autor *hero*	4	13.969	76,09
Django	Autor periférico	2.065	7.869	29,70
	Autor médio	13	5.928	22,38
	Autor *hero*	6	12.694	47,92
Elasticsearch	Autor periférico	1.196	8.963	24,77
	Autor médio	21	14.870	41,09
	Autor *hero*	6	12.352	34,14
Pandas	Autor periférico	1.690	5.624	30,27
	Autor médio	9	4.187	22,53
	Autor *hero*	3	8.770	47,20
Spring Boot	Autor periférico	664	1.970	9,86
	Autor médio	3	1.365	6,84
	Autor *hero*	4	16.618	83,3

jetos em estudo. Assim como observado nas figuras da Seção 6.1, pode-se constatar um grande número de autores periféricos que efetuaram uma quantidade pequena de *commits* e um pequeno número de autores *heroes* que concentram juntos uma grande parcela das contribuições do projeto. A maior concentração de conhecimento é observada no *Spring Boot*, no qual 83,3% dos *commits* estão concentrados em apenas 0,59% dos autores. No *Elasticsearch*, a presença de 21 autores médios mostram que aparentemente foi o projeto com as contribuições mais balanceadas entre as três categorias, contudo, ao considerar o total de autores *heroes* e médios, constata-se que apenas 2,21% do total de desenvolvedores concentram mais de 75% do conhecimento daquele projeto. O resultado encontrado mostra uma concentração expressiva do conhecimento do código. A próxima seção caracteriza com mais detalhes os autores *heroes* e suas contribuições.

6.3 Distribuição dos *Heroes* no Ciclo Evolutivo

Nesta seção, são relatados os resultados das análises para compreender como as contribuições dos *heroes* evolui, por exemplo, se estão em constante crescimento ou não, se ocorre um rápido pico ou se o aumento das contribuições é gradual.

As Figuras 6.6 a 6.10 exibem a evolução das contribuições de cada um dos *heroes*

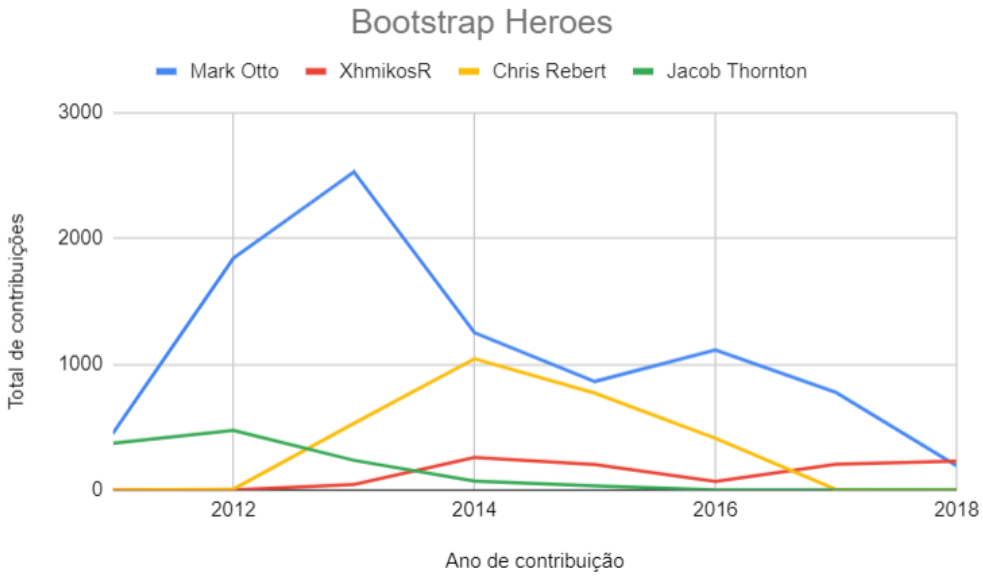

Figura 6.6. Contribuições dos autores *heroes* durante o ciclo de vida do sistema *Bootstrap*

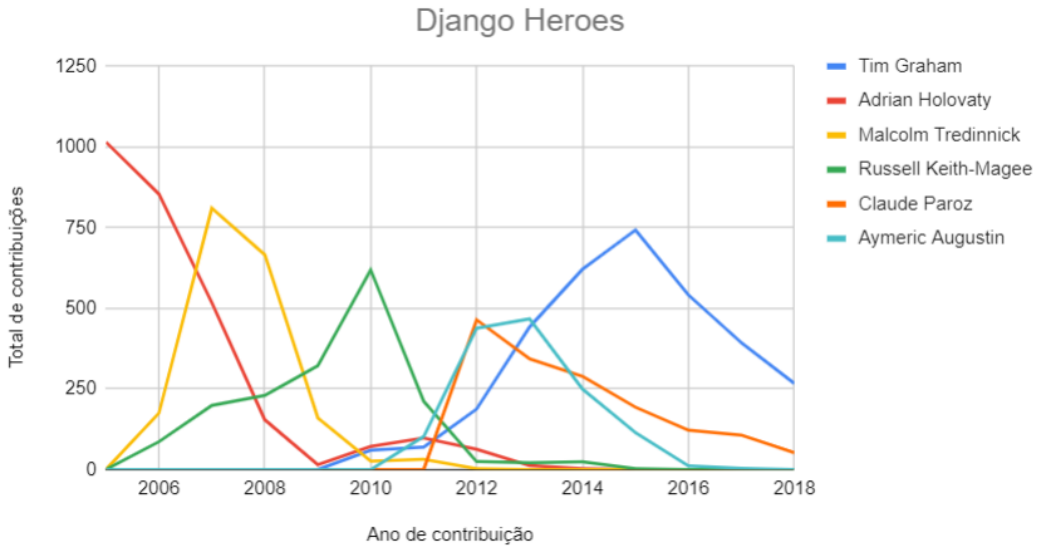

Figura 6.7. Contribuições dos autores *heroes* durante o ciclo de vida do sistema *Django*

dos projetos analisados neste trabalho. O eixo horizontal apresenta o ano de contribuição, variando do primeiro ao último ano analisado em cada projeto. O eixo vertical corresponde à quantidade total de contribuições feitas pelo desenvolvedor naquele intervalo de tempo.

Percebe-se que, na maioria dos casos, os autores não são *heroes* do projeto de

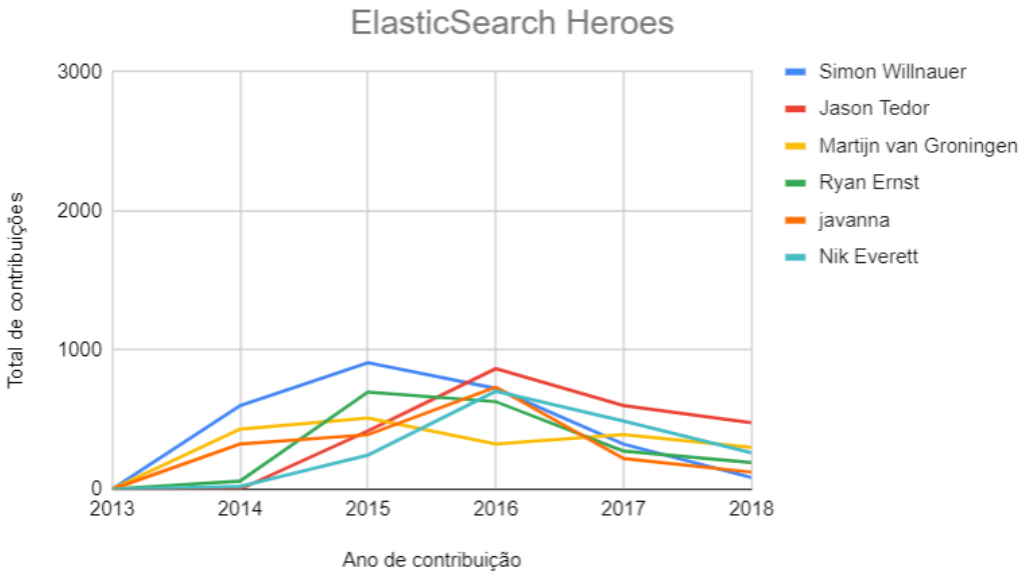

Figura 6.8. Contribuições dos autores *heroes* durante o ciclo de vida do sistema *Elasticsearch*

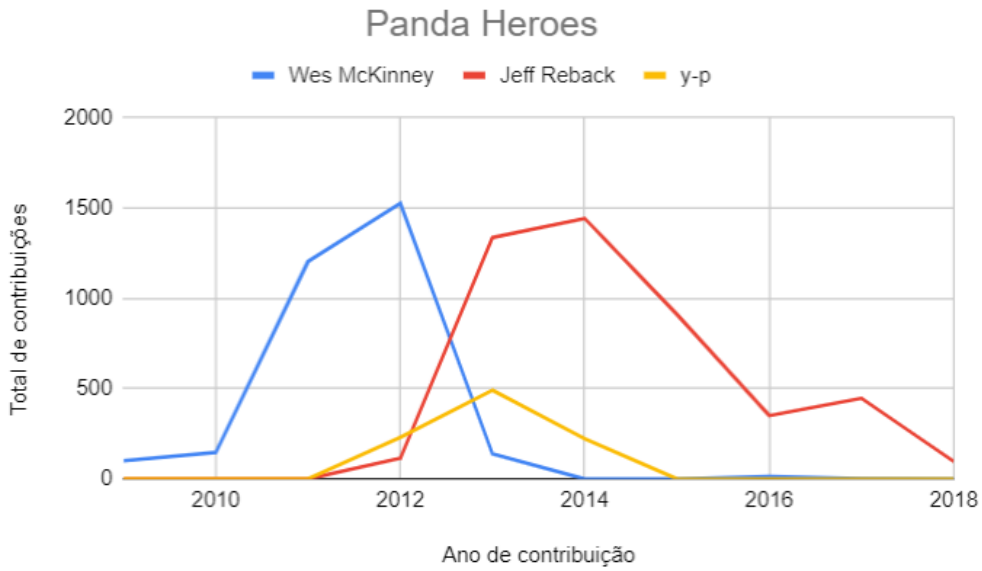

Figura 6.9. Contribuições dos autores *heroes* durante o ciclo de vida do sistema *Pandas*

modo simultâneo, mas que em cada período de tempo um ou dois deles se destacaram. A Figura 6.7 torna mais evidente essa realidade, pois os seis autores do projeto *Django* alternam no protagonismo de contribuições. No momento em que *Russel Keith-Magee* (verde) estava no auge de sua parcela de colaboração ao projeto, os autores *Aymeric Augustin*, *Claude Paroz* e *Tim Graham* estavam começando a crescer no número de

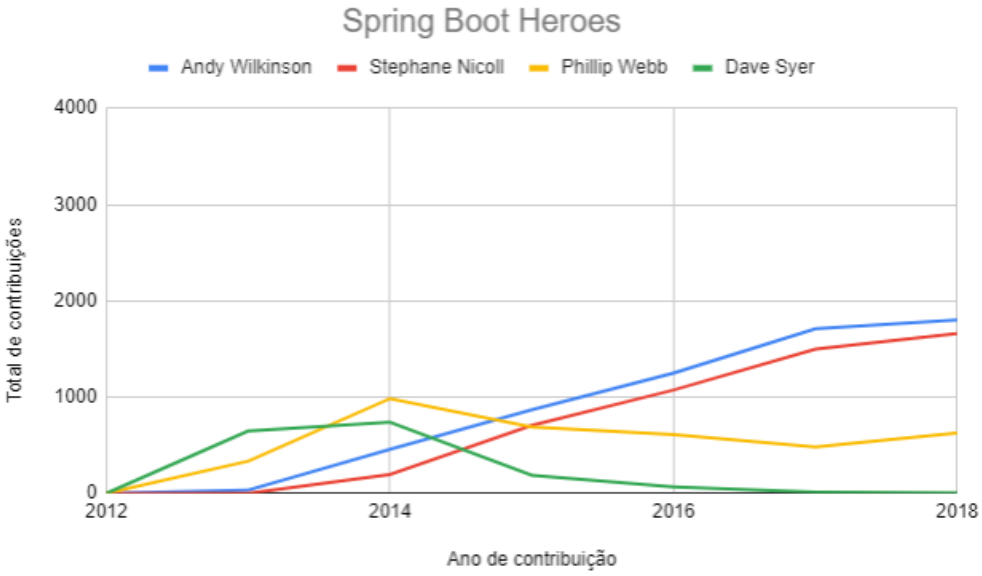

Figura 6.10. Contribuições dos autores *heroes* durante o ciclo de vida do sistema *Spring Boot*

commits. Nos anos de 2016 a 2018, o desenvolvedor *Tim Graham* (azul) permanece como principal *hero* do projeto, porém é notável a redução no número de suas contribuições, podendo sugerir de acordo com o padrão observado, que nos próximos intervalos de tempo um autor médio, se tornará um *hero* do sistema. Os projetos *Bootstrap* e *Pandas* seguiram um padrão similar ao observado em *Django*.

A partir da Figura 6.10, a distribuição de autores *Spring Boot* mostra que no período inicial dois desenvolvedores dividiram o protagonismo do projeto e nos últimos anos eles reduziram suas contribuições. Concomitantemente a isso, outros dois desenvolvedores tornaram-se os principais *heroes* do projeto nos anos finais da análise. Esse comportamento reforça o resultado observado em relação a intercalação da responsabilidade principal do projeto.

Ao observar a atuação dos *heroes* no *Elasticsearch* representada na Figura 6.8, percebe-se que o comportamento foi distinto dos anteriores. Os seis autores tiveram a curva de crescimento, seus picos de contribuições e redução, de modo quase concomitante. Esse fato revela que os *heroes* somaram esforços conjuntamente para o crescimento do projeto, especialmente entre os anos de 2015 a 2017.

Essa análise possibilitou compreender como se dá a distribuição das contribuições dos *heroes* durante o ciclo de vida do software. Os resultados indicam que há uma alternância do principal proprietário na maioria dos casos e que o tempo gasto para um autor pequeno se tornar um *hero* é, em geral, de um a dois anos.

6.4 Propriedade de Código no Ciclo Evolutivo do Software

Esta seção apresenta as análises sobre como a métrica de propriedade de código se comporta durante o ciclo evolutivo em projetos *open source*. Para a realização deste estudo, as definições e notações sugeridas por Foucault et al. [2014] e detalhadas no Capítulo 2 foram utilizadas como base.

Ao observar o comportamento dos autores *heroes* apresentado na Seção 6.3, nota-se que existe uma alternância entre eles em relação ao protagonismo das contribuições em cada período, com exceção do projeto *Elasticsearch*. Isso sugere que (i) a propriedade de código permanece durante todos os períodos do projeto com elevados valores, sempre vinculados ao *hero* dominante naquele período; (ii) o aumento e a diminuição no valor da métrica acompanha o crescimento e a queda das contribuições dos *heroes*, seguindo o mesmo comportamento dos gráficos da Seção 6.3.

Adaptando as notações desenvolvidas por Foucault et al. [2014] para o contexto do presente estudo, pode-se considerar que:

- $\omega\ (p)$ é a soma de todas contribuições dos desenvolvedores realizadas em um projeto p;
- $\omega\ (d)$ é a soma de todas contribuições realizadas por um desenvolvedor d; e
- $\omega\ (p,d)$ é a soma de todas contribuições realizadas por um desenvolvedor d em um projeto p.

Assim, a propriedade de código é dada pela razão das contribuições feitas por um desenvolvedor pelo total das contribuições do projeto. Tem-se que o valor da métrica por desenvolvedor é calculada por:

$$own_{p,d} = \frac{\omega(p,d)}{\omega(p)}$$

Bird et al. (2011) salientam que o valor mais elevado dessa razão representa a propriedade de código do projeto. E o desenvolvedor responsável por essa contribuição é considerado o *proprietário*. Desse modo, sendo D o conjunto de contribuidores do projeto, tem-se que que o proprietário é dado por:

$$\max(\{\ own_{p,d} \mid d \in D\})$$

Dessa forma, foram realizados os cálculos para a propriedade de código dos cinco projetos do *data set*, a partir das contribuições totais realizadas por cada um dos autores. A Tabela 6.2 mostra os resultados encontrados para max($own_{p,d}$). Contudo, o

valor total da propriedade de código não traz informações suficientes para se compreender a distribuição do conhecimento do código, uma vez que um autor considerado como o proprietário pode ter efetuado um grande número de contribuições apenas no início do projeto. Sendo assim, o cálculo explanado na Tabela 6.2 pode não representar como o conhecimento está distribuído atualmente ou a forma como esse conhecimento foi distribuído no decorrer do tempo.

Tabela 6.2. Propriedade de código dos repositórios do *data set*

Projeto	Proprietário	Propriedade de código
Bootstrap	Mark Otto	0,490
Django	Tim Graham	0,126
Elasticsearch	Simon Willnauer	0,073
Pandas	Jeff Reback	0,253
Spring Boot	Andy Wilkinson	0,306

Dessa forma, para se compreender a evolução do conhecimento no decorrer do ciclo de vida do software, a métrica foi calculada para cada período de tempo T. Considerando que:

- ω (p,t) é a soma de todas contribuições realizadas no projeto p em uma janela de tempo t; e
- ω (d,t) é a soma de todas contribuições realizadas pelo desenvolvedor d no projeto em uma janela de tempo t.

A propriedade de código no período de tempo T é dada pela razão das contribuições feitas por um desenvolvedor pelo total de contribuições no projeto no período. Em outras palavras, para um determinado intervalo de tempo T e para cada desenvolvedor d a métrica de propriedade no tempo é dado por:

$$own_{t,d} = \frac{\omega(d,t)}{\omega(p,t)}$$

O valor mais elevado dessa razão em um espaço de tempo T é o valor da métrica de propriedade de código do projeto naquele intervalo. O desenvolvedor responsável por essa contribuição é considerado o proprietário no período é identificado por:

$$\max(\{\ own_{t,d} \mid d \in D\})$$

Para efetuar esse cálculo, as contribuições de cada desenvolvedor foram agrupadas no período de tempo correspondente e a propriedade de código do projeto em T é

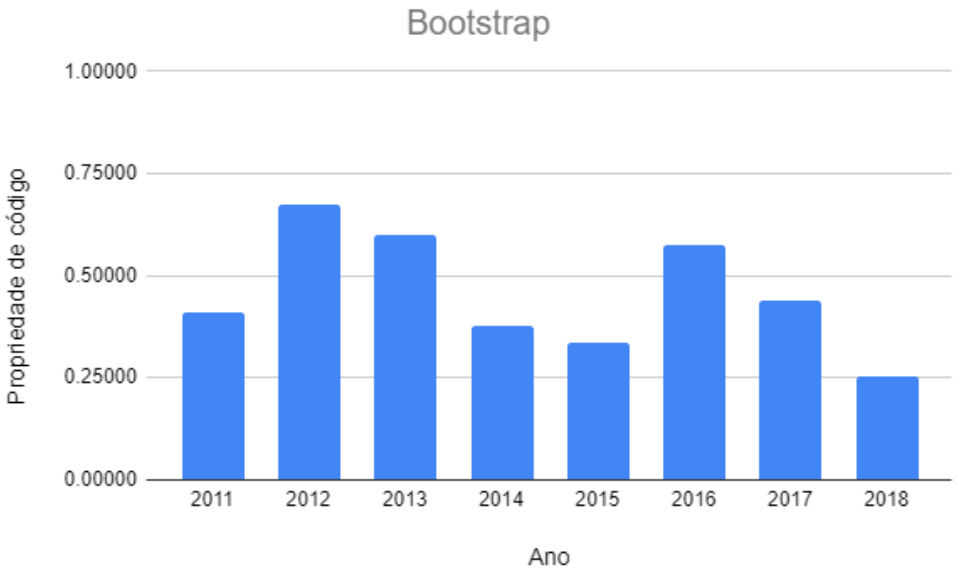

Figura 6.11. Propriedade de código no decorrer do ciclo de vida do *Bootstrap*

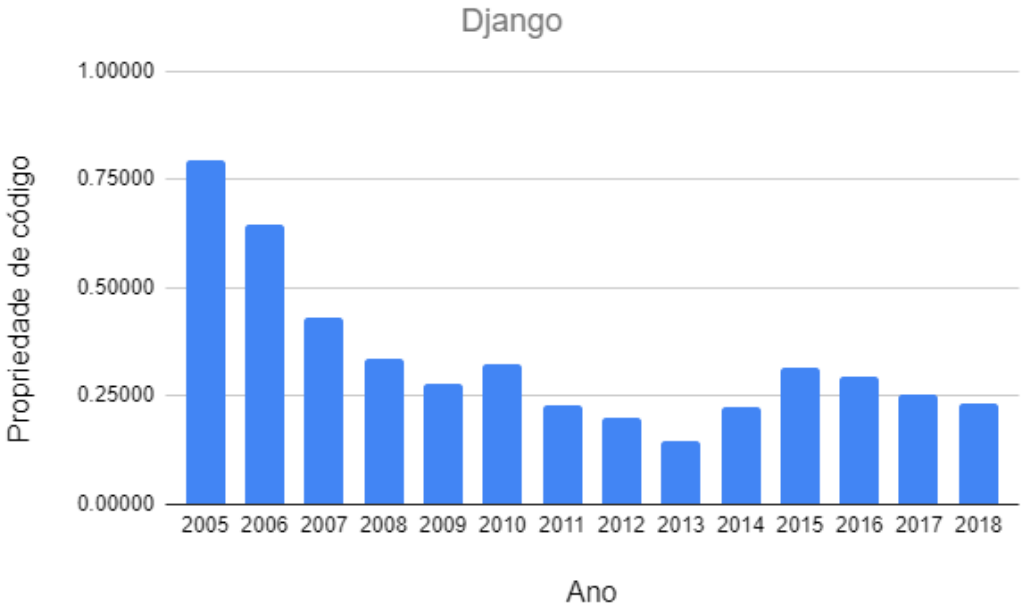

Figura 6.12. Propriedade de código no decorrer do ciclo de vida do *Django*

o valor de $\max(own_{t,d})$. Além disso, o proprietário é aquele que efetuou essas contribuições. Quanto maior for esse valor, mais o conhecimento do sistema naquele período ficou concentrado e, de forma oposta, quanto menor o for, mais distribuído esteve o conhecimento. É importante salientar que essa análise considera todos os desenvolvedores do período, não apenas os *heroes*, como feito na Seção 6.3. As Figuras 6.11 a

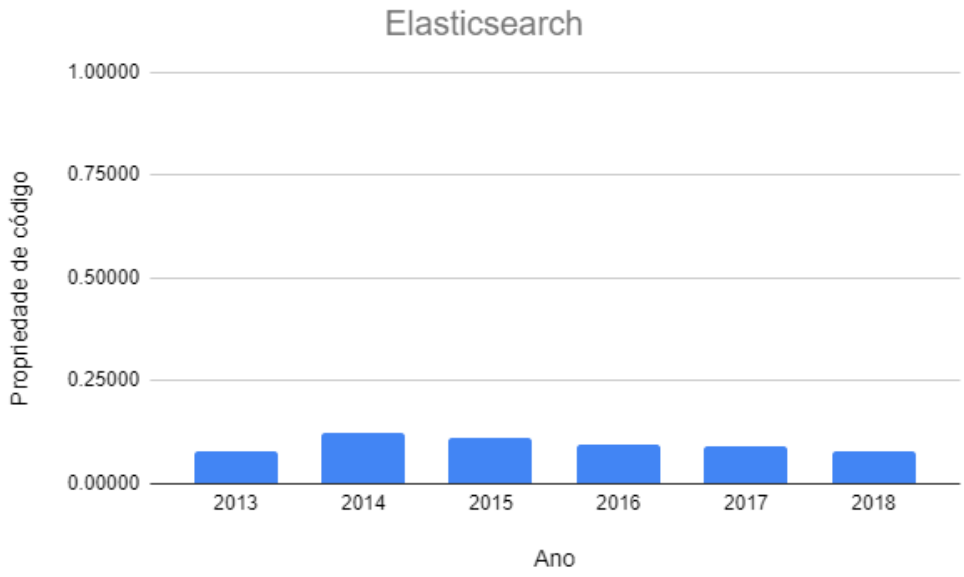

Figura 6.13. Propriedade de código no decorrer do ciclo de vida do *Elasticsearch*

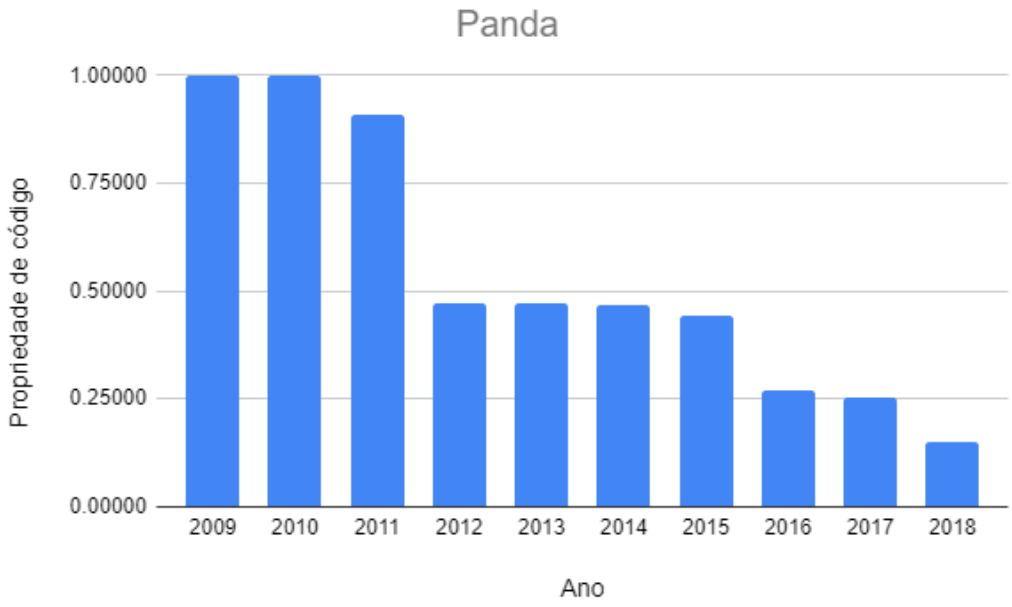

Figura 6.14. Propriedade de código no decorrer do ciclo de vida do *Pandas*

6.15 exibem como o valor da métrica oscilou durante o ciclo de vida dos projetos.

É possível observar uma semelhança entre o comportamento nos sistemas *Django*, *Pandas* e *Spring Boot* apresentados nas Figuras 6.12, 6.14 e 6.15, respectivamente. Nesses três projetos, a propriedade de código na primeira versão analisada é muito alta. Contudo, no decorrer do ciclo evolutivo do software, a propriedade de código reduziu

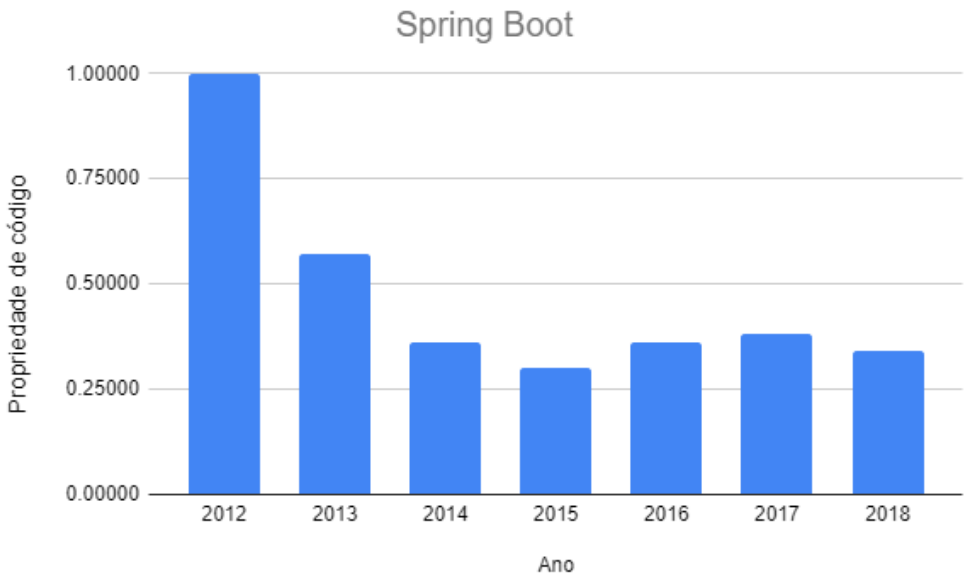

Figura 6.15. Propriedade de código no decorrer do ciclo de vida do *Spring Boot*

gradativamente atingindo a marca de 0,3 em *Spring Boot* e 0,15 em *Pandas* e *Django*. O resultado mostra que os três sistemas tiveram o conhecimento altamente concentrado em um desenvolvedor, porém, no decorrer da evolução do projeto, o conhecimento vai se tornando mais distribuído.

A Figura 6.13 expõe os resultados encontrados no projeto *Elasticsearch*. O comportamento da métrica de propriedade de código nesse caso apresentou, desde o primeiro intervalo de tempo, um valor menor que 0,15 e permaneceu assim até o último. Esse resultado mostra um sistema que em todas as etapas de seu ciclo de vida teve o conhecimento distribuído entre os autores.

Por fim, observa-se o comportamento do *Bootstrap* presente na Figura 6.11. Os resultados foram distintos dos demais, pois apresentou comportamento variável no decorrer dos anos. Observa-se, no entanto, que o maior valor da métrica de propriedade foi de 0,67, ocorrido no segundo período do ciclo evolutivo, enquanto o menor valor foi encontrado no último período analisado, como o valor de 0,25. O comportamento desse gráfico assemelha-se a uma onda, no qual há momentos de pico e outros de leve queda.

Pode-se concluir, a partir dos resultados encontrados nessa análise, que a propriedade de código não permaneceu com elevados valores durante todos os períodos do projeto em nenhum dos sistemas analisados. O comportamento observado foi o contrário disso, pois constatou-se que os sistemas tendem a iniciar com o conhecimento mais concentrado, porém, no decorrer da evolução do software existe uma forte tendência

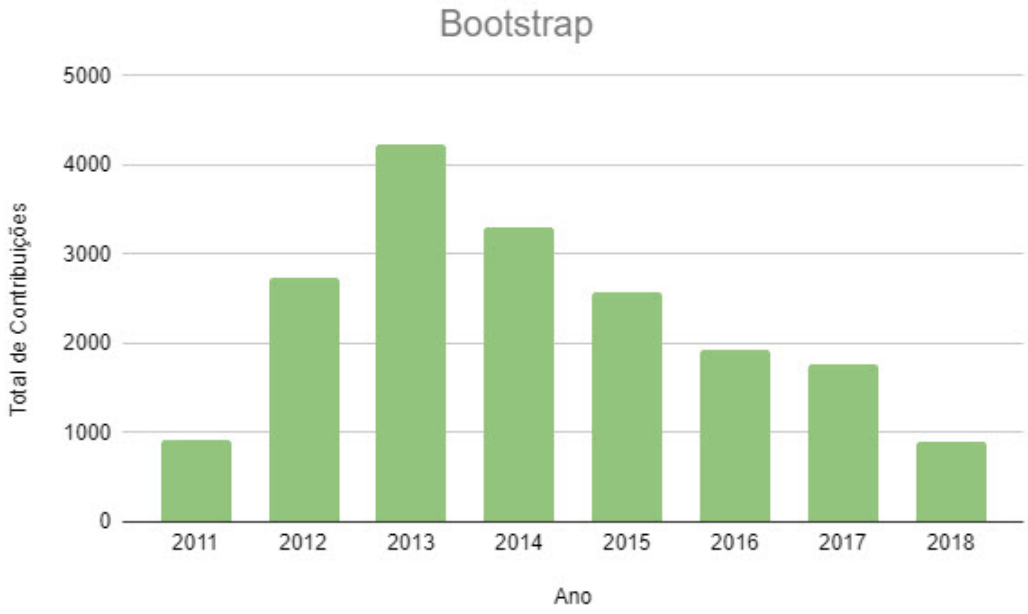

Figura 6.16. Totais de contribuições durante o ciclo de vida do *Bootstrap*

do conhecimento se tornar mais distribuído entre os desenvolvedores.

Além disso, na maioria dos casos, o aumento e a diminuição no valor da propriedade não acompanha os resultados encontrados nos gráficos das contribuições dos *heroes* da Seção 6.3. No entanto, identifica-se uma razoável semelhança entre os gráficos do projeto *Bootstrap*, no qual os picos do *hero Mark Otto* nos anos de 2013 e 2016, coincidem com dois picos da métrica na Figura 6.11.

6.5 Frequência de Contribuição no Ciclo de Vida do Software

Nesta seção, é analisada a forma como as contribuições realizadas pelos autores é distribuída no decorrer do ciclo de vida desses softwares.

As Figuras 6.16 a 6.20 retratam a variação no total de *commits* efetuados nos repositórios, durante seu ciclo evolutivo. O eixo horizontal mostra o ano da contribuição, enquanto o eixo vertical corresponde à quantidade total de contribuições feitas no projeto naquele período de tempo.

A distribuição do total de contribuições realizadas no projeto *Bootstrap* durante seu ciclo de vida é apresentada na Figura 6.16. É possível observar que houve um período inicial de crescimento constante no número de *commits* submetidos, atingindo seu pico no ano de 2013. É importante notar que a maior contribuição efetuada por

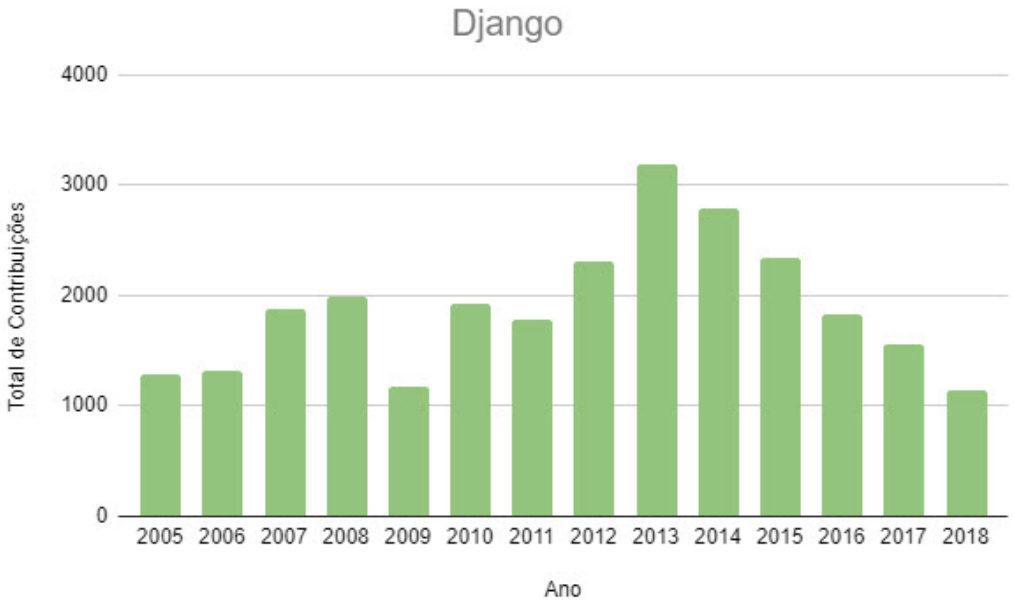

Figura 6.17. Totais de contribuições durante o ciclo de vida do *Django*

heroes nesse projeto ocorreu nesse mesmo ano, conforme mostra a Figura 6.6. Contudo, nos anos posteriores o total de contribuições permaneceu em queda contínua até atingir o seu menor valor no ano de 2018, último período analisado.

A Figura 6.17 representa as variações nas contribuições do repositório *Django*. Pode-se salientar que durante os 14 períodos analisados, houve uma constância em relação à quantidade de contribuições, uma vez que durante 10 anos, o número de *commits* permaneceu dentro do intervalo de 1.000 a 2.000 contribuições. A exceção a esse comportamento encontra-se no pico de contribuições verificado entre os anos de 2012 a 2015. Esse comportamento elucida a forma de atuação dos *heroes* desse repositório, apresentada anteriormente pela Figura 6.7. Nele, apesar da alternância de protagonismo entre os *heroes*, há uma constância quanto à permanência de um grande desenvolvedor em cada intervalo de tempo do projeto.

As contribuições no repositório *Elasticsearch*, apresentadas na Figura 6.18, começaram muito baixas em 2013. Contudo, houve um grande crescimento no número de *commits* em 2014, atingindo seus maiores valores nos anos de 2015 e 2016, com um total de 9.014 e 9.295 *commits*, respectivamente. Apesar de uma relativa queda nos últimos dois anos da análise, a atuação dos desenvolvedores continua alta no projeto. Constata-se que o total de contribuições de todos os desenvolvedores do projeto segue o mesmo padrão das contribuições dos *heroes* mostradas na Figura 6.8, ainda que esse tenha sido o repositório com a maior distribuição de conhecimento entre os autores.

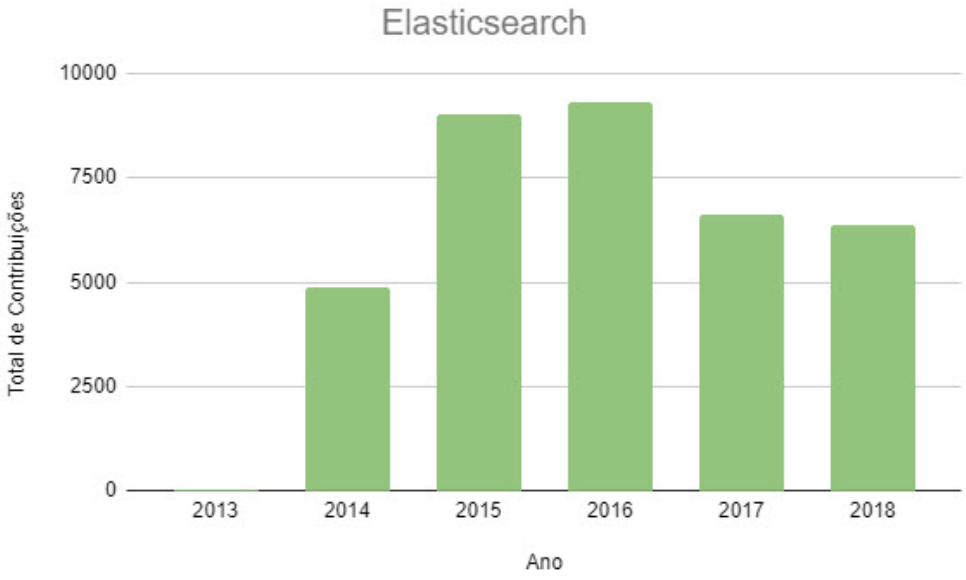

Figura 6.18. Totais de contribuições durante o ciclo de vida do *Elasticsearch*

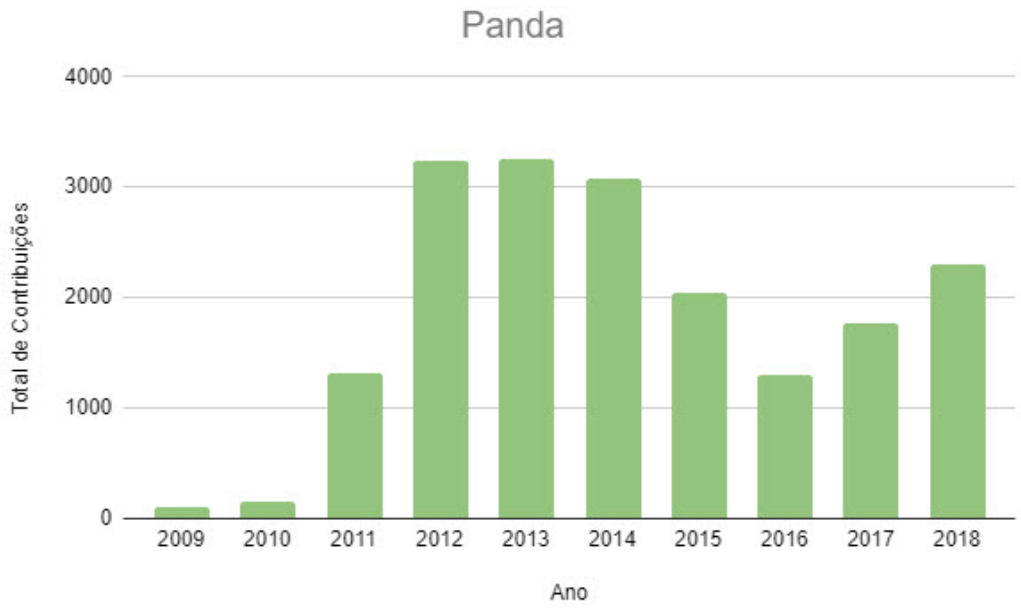

Figura 6.19. Totais de contribuições durante o ciclo de vida do *Pandas*

O projeto *Pandas* teve uma baixa contribuição em seus dois primeiros anos e um significativo aumento no ano de 2011, conforme indica a Figura 6.19. Nos anos de 2012 a 2014, ele atingiu o pico de *commits*, totalizando mais de três mil contribuições anuais. Nos anos seguintes, o repositório continuou com uma grande atuação, apesar de uma ligeira queda na quantidade efetiva de contribuições recebidas. Assim como verificado

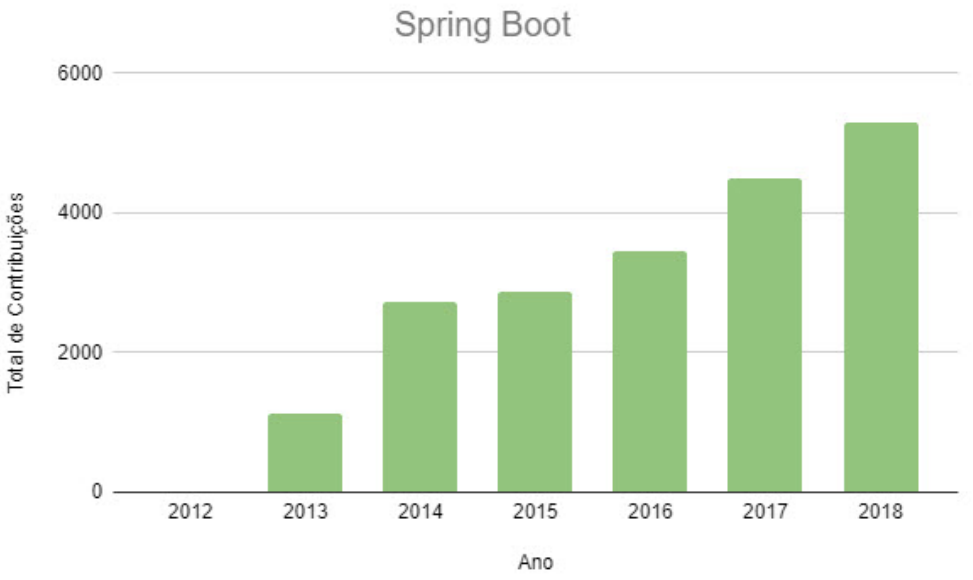

Figura 6.20. Totais de contribuições durante o ciclo de vida do *Spring Boot*

nos outros projetos, o pico de contribuições do *Pandas* aconteceu na mesma janela de tempo em que houve o auge da atuação dos três *heroes* existentes no projeto.

Constata-se a partir da Figura 6.20, que o projeto *Spring Boot* está em constante crescimento desde sua criação no ano de 2012. O pico de contribuições ocorreu no último período de tempo analisado e se seguir o padrão apresentado, os anos seguintes terão uma quantidade de *commits* ainda maior. A relação entre o total das contribuições envolvendo a totalidade de autores e a quantidade de contribuições apenas dos *heroes* também subsiste nesse repositório. A Figura 6.10 apresentada anteriormente mostra que o pico dos *heroes* também ocorreu no ano de 2018.

A análise apresentada nesta seção possibilita a compreensão da forma de atuação temporal dos desenvolvedores nos cinco projetos. Assim, é possível identificar em quais períodos de tempo T houve uma maior dedicação no repositório ou uma redução no investimento, bem como identificar que ocorreu uma considerável relação entre a variação da atuação dos desenvolvedores do projeto e dos *heroes*.

6.6 Relação entre Concentração de Conhecimento e Quantidade de Contribuições

Ao longo da evolução do software, a quantidade de contribuições varia e a concentração de conhecimento pode variar também. Essa análise visa identificar se existe correlação

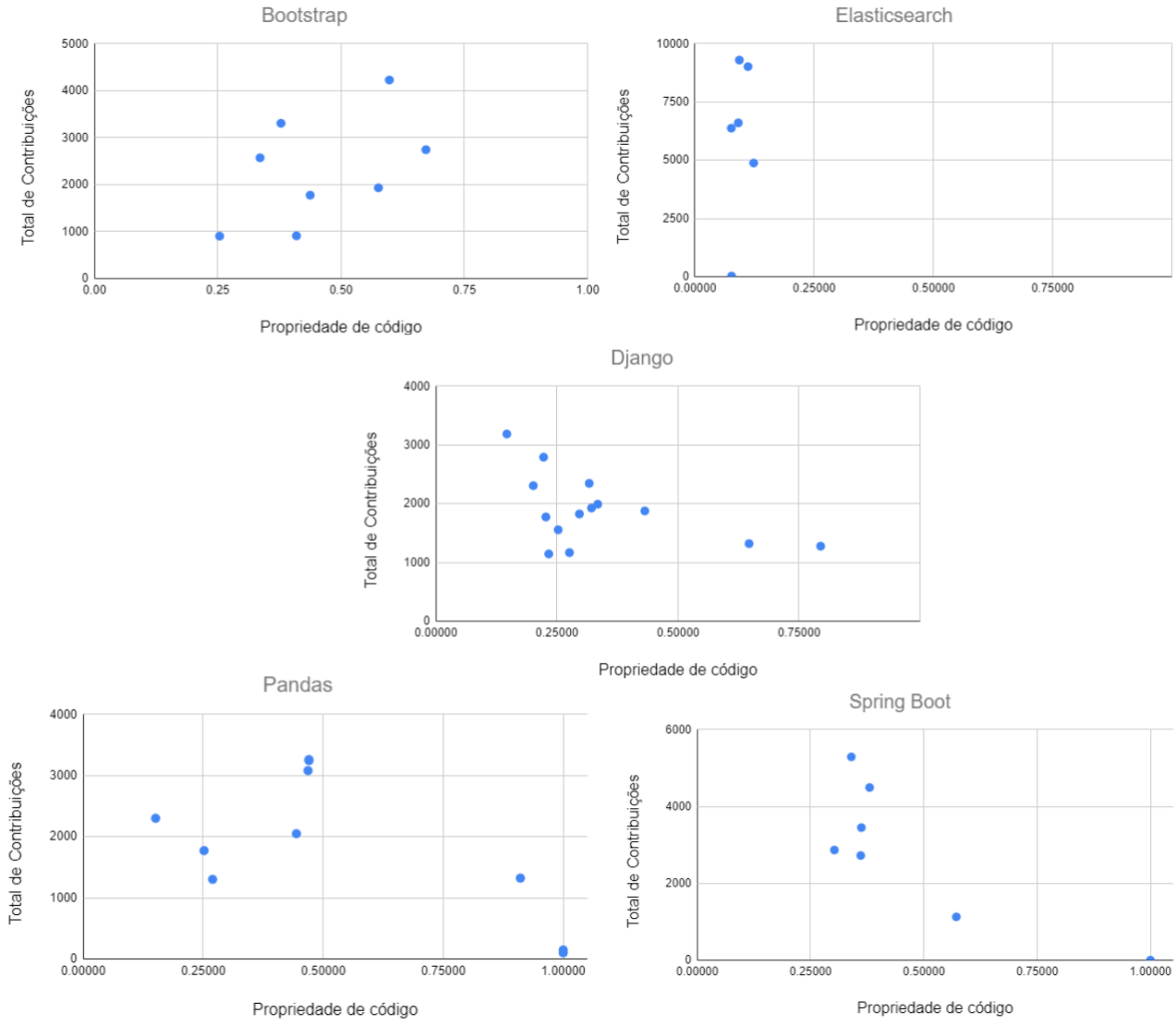

Figura 6.21. Gráfico de dispersão das amostras - Eixo horizontal corresponde ao valor da propriedade de código e o eixo vertical ao total de contribuições efetuadas

entre a quantidade total de contribuições realizadas em um período e a propriedade de código. Para definir o método de correlação a ser usado, foram gerados os gráficos de dispersão das amostras. O eixo horizontal representa o valor da propriedade de código, calculada na Seção 6.4 como $\max(own_{t,d})$. O eixo vertical refere-se à quantidade total de contribuições no período, apresentado na Seção 6.5. A Figura 6.21 apresenta os gráficos de dispersão para cada um dos projetos em estudo. Ao examinar os dados, a partir dos gráficos de dispersão, observa-se que eles não seguem uma distribuição normal e que a amostra é pequena (tamanho da amostra é menor do que 30). Como os dados não se apresentam conforme a distribuição normal, os coeficientes não poderiam ser calculados

por meio da correlação de *Pearson*[1], que medem somente relações lineares. Por isso, utilizou-se a correlação de *Spearman*[2] para o calculo dos coeficientes de correlação deste trabalho.

A Tabela 6.3 apresenta os resultados encontrados no cálculo da correlação. Sabe-se que um coeficiente de correlação de *Spearman* maior que 0,9 (positivo ou negativo) indica uma correlação muito forte; entre 0,7 a 0,9 indica uma correlação forte; entre 0,5 a 0,7 uma correlação moderada; entre 0,3 a 0,5 uma correlação fraca e de 0 a 0,3 resulta em uma correlação desprezível. O valor do coeficiente varia entre -1 (correlação inversa) e +1 (correlação direta).[3] É possível observar que não há uniformidade entre os coeficientes calculados. Portanto, constata-se que para os projetos analisados não existe uma correlação entre o valor da propriedade de código e a variação da quantidade de contribuições efetuadas no repositório.

Tabela 6.3. Correlação entre a propriedade de código e o total de contribuições nos projetos.

Projeto	Correlação de *Spearman*	Classificação
Bootstrap	0,5	correlação direta moderada
Django	-0,376	correlação inversa fraca
Elasticsearch	0,257	correlação direta fraca
Pandas	-0,339	correlação inversa fraca
Spring Boot	-0,571	correlação inversa moderada

Contudo, os resultados da Seção 6.5 mostram indícios de que há uma correlação entre o aumento de contribuições nos repositórios e o crescimento da atuação dos *heroes*. Por isso, a partir desses dois fatores o coeficiente de correlação de *Spearman* foi calculado. Para cada intervalo de tempo T, foi considerada a quantidade total de contribuições e o valor máximo dentre as contribuições dos *heroes*, isto é, a quantidade de *commits* efetuada pelo proprietário do período. A Tabela 6.4 mostra os resultados obtidos para os coeficientes de correlação nesse caso. Os valores comprovam que há uma forte correlação entre o crescimento das contribuições no projeto e o aumento das contribuições dos *heroes*. Apenas o sistema *Django* manifesta uma correlação fraca, possivelmente ocasionada pela troca constante de proprietários *heroes* durante o ciclo evolutivo do projeto. Os resultados encontrados mostram correlação direta forte entre as contribuições do proprietário do período e o total de contribuições do período nos projetos.

[1]Pearson: `http://www.leg.ufpr.br/~silvia/CE701/node79.html`
[2]Spearman: `http://www.leg.ufpr.br/~silvia/CE701/node80.html`
[3]Correlation coefficient: `https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3576830`

Tabela 6.4. Correlação entre as contribuições do proprietário do período e o total de contribuições do período nos projetos.

Projeto	Correlação de *Spearman*	Classificação
Bootstrap	0,952	correlação direta muito forte
Django	0,2	correlação direta fraca
Elasticsearch	0,841	correlação direta forte
Pandas	0,672	correlação direta moderada
Spring Boot	0,964	correlação direta muito forte

6.7 Respostas das Questões de Pesquisa

Nesta seção, as questões de pesquisa são respondidas e discutidas com base nos resultados encontrados nas análises.

QP1: O conhecimento do software se difunde ao longo do ciclo de vida?

O intuito dessa questão de pesquisa é compreender se o conhecimento do software permanece desbalanceado entre os contribuidores, como sugerido por Foucault et al. [2014], ou se, em uma determinada etapa da maturidade do software, as contribuições tornam-se uniformes.

Os resultados encontrados nas Seções 6.1 e 6.2 sugerem que o conhecimento do software não se difunde ao longo do seu ciclo de vida, uma vez que, em média, 80% dos desenvolvedores efetuam no máximo duas contribuições nos projetos analisados. Assim, pode-se considerar que o conhecimento permanece restrito a uma pequena parcela de autores, nomeados *heroes*.

Contudo, ao observar os resultados encontrados na Seção 6.4, é possível constatar que, apesar de existir uma parcela dos desenvolvedores que efetuam uma grande quantidade de contribuições, existe uma tendência de disseminação do conhecimento à medida que o software evolui. Essa constatação só é possível verificar quando se analisa a evolução do valor da métrica de propriedade de código. Ao comparar os primeiros e os últimos períodos de tempo T dos cinco projetos, tem-se que a métrica de propriedade é sempre menor nos últimos períodos.

O projeto *Elasticsearch*, que teve baixos valores para a métrica no decorrer de todos os anos, mas, ainda assim, também atingiu o menor valor no último período analisado. Acredita-se que os resultados desse projeto foram similar ao esperado para empresas privadas, tendo o conhecimento distribuído entre os autores durante as etapas de seu ciclo de vida.

Esses resultados mostram que há uma distribuição significativa do conhecimento no decorrer dos anos de vida do software. Isso pode facilitar a entrada de novos contribuidores e possibilitar que eles venham a se tornar contribuidores centrais para o projeto. Conforme apontam Avelino et al. [2019a], a sobrevivência de uma quantidade relevante de projetos *open source* (41%) ocorre devido a ajuda ativa de novos desenvolvedores.

QP2: Os desenvolvedores considerados *heroes* nas primeiras versões permanecem *heroes* durante a evolução do projeto?

O objetivo desta questão de pesquisa é estudar e caracterizar o comportamento dos desenvolvedores responsáveis pela maior parte das contribuições efetuadas nos projetos, os chamados *heroes*. Procura-se entender se os principais contribuidores no início do projeto permanecem contribuindo até os últimos períodos analisados. Se o pressuposto for positivo, deseja-se compreender se essas contribuições permanecem numerosas ou se houve uma diminuição em relação ao número de *commits* desses autores. Caso contrário, pretende-se entender em quais condições outros contribuidores tornam-se *heroes* e como os antigos diminuíram suas contribuições, se foi gradativamente ou se houve uma queda súbita no número de contribuições indicando um abandono do projeto.

A Seção 6.3 descreveu a análise do comportamento dos autores *heroes* no decorrer do ciclo de vida do software. Os resultados desse estudo apontaram que, apesar de permanecerem como grandes contribuidores do projeto pelo total de contribuições, ao comparar as contribuições do primeiro e do último período de tempo, observa-se que os *heroes* iniciais já não estão presentes dentre os *heroes* finais. O autor que mais contribuiu no primeiro período apresentou nenhuma ou uma quantidade pequena de contribuições no último período analisado. Esse comportamento foi unânime nos cinco projetos.

O estudo também permite concluir que os *heroes* de um projeto *open source*, em geral, não ocupam essa posição de modo simultâneo. Em cada período de tempo T, um ou dois deles concentram as contribuições do projeto. A Figura 6.7 também permite inferir que a diminuição das contribuições de cada *hero* ocorre de modo gradual no decorrer dos anos do projeto. A transformação de um novo autor, inicialmente considerado periférico até tornar-se um dos *heroes*, também ocorre de forma gradual e leva em média dois anos.

A análise por períodos foi importante para compreender o comportamento dos projetos e de seus contribuidores. Os *heroes* do repositório, como categorizado na Seção 6.2, apenas são considerados como tal devido à sua expressividade no total de

contribuições efetuadas. Contudo, apesar de ser considerado *hero*, ele não concentra o domínio e conhecimento daquele software no momento corrente. Um exemplo desse fato ocorre com um dos autores principais do *Django*, Adrian Holovaty: ele iniciou o projeto em 2005, mas após 2008 teve uma grande queda na atuação do projeto e, no ano de 2014, ele encerrou suas contribuições no desenvolvimento. Dessa forma, apesar de existirem oito *heroes* no *Django* ao longo da sua evolução, ao considerar o último período do projeto, apenas dois deles detêm o conhecimento e o domínio do software no momento presente.

Os resultados indicam que em um projeto *open source*, os desenvolvedores *heroes* não abandonam ou reduzem a contribuição repentinamente no projeto; ele permanece por algum tempo colaborando e dando suporte ao sistema. Essa diminuição gradual é uma diferença essencial em relação a projetos comerciais de código fechado, uma vez que, ao ser desligado da empresa, as contribuições do desenvolvedor diminuem de forma abrupta, fato que pode dificultar a difusão do conhecimento do software que estava concentrado nesse desenvolvedor. O surgimento de *heroes* no decorrer da evolução do software observada em todos os casos também reforça o resultado encontrado na QP1.

Esses resultados ressaltam a importância da gestão de projetos *open source* e sua atuação, de modo a identificar os desenvolvedores *heroes* e incentivá-los a permanecer no projeto. A criação de mecanismos de disseminação do conhecimento, que esses autores detêm, também é importante, para que o tempo de aprendizado dos futuros *heroes* seja otimizado.

QP3: A concentração do conhecimento é impactada pela quantidade total de contribuições do projeto?

O intuito dessa questão de pesquisa é compreender se existe uma relação entre a quantidade total de contribuições no projeto e a disseminação do conhecimento entre os desenvolvedores, isto é, se a medida que as contribuições no repositório aumentam, o conhecimento torna-se mais distribuído entre os autores.

A Seção 6.6 calculou por meio da correlação de *Spearman*, a relação entre o total de contribuições no repositório e os valores encontrados para a métrica de propriedade de código, considerando para isso todos períodos de tempo do projeto. Além disso, a relação entre a quantidade de *commits* efetuados pelos *heroes* em cada período e a quantidade de contribuições totais no mesmo período, também foi calculada utilizando o método de *Spearman*.

A partir dos dados considerados na Seção 6.6, os resultados indicam que não se pode inferir nenhuma relação, direta ou inversa, entre o valor da métrica de proprie-

dade de código e a variação no número de contribuições totais do projeto. Não houve uniformidade entre os valores dos coeficientes de correlação calculados para os cinco repositórios analisados. Esse resultado sugere que a concentração do conhecimento não é impactada exclusivamente pela quantidade de contribuições. Assim, podem ser encontrados projetos *open source* com distintos cenários: i) a concentração do conhecimento, calculada pela métrica propriedade de código, permanece constante, independente da variação no total de contribuições; ii) concentração do conhecimento diminui, quando há um aumento no número de contribuições, possibilitando maior disseminação do conhecimento entre os desenvolvedores; e iii) concentração do conhecimento aumenta quando há um crescimento no total de *commits*. Os achados dessa questão de pesquisa demonstram que apenas a avaliação da variação da quantidade de contribuições não é o suficiente para explicar a variação no valor da métrica de propriedade de código no decorrer do ciclo de vida do projeto.

Por outro lado, os resultados indicaram que há uma relação direta forte entre o aumento dos *commits* efetuados pelos desenvolvedores *heroes* com o aumento no total de contribuições realizadas no projeto no mesmo período. Acredita-se que a estabilidade e a permanência dos desenvolvedores *heroes* é de grande importância para a evolução e sustentabilidade do projeto [Robles et al., 2009]. Esses resultados corroboram com essa ideia, pois evidenciam a importância dos desenvolvedores *heroes* para o crescimento do total de contribuições do projeto.

6.8 Considerações Finais

Neste capítulo foram realizados estudos quantitativos a partir dos dados coletados dos repositórios GitHub. Foram explanadas a forma de distribuição e caracterização da população de autores. A métrica de propriedade de código foi calculada e analisada, com o intuito de compreender como ocorre a distribuição do conhecimento nos projetos do *data set* (QP1). Além disso, houve um enfoque no estudo dos autores *heroes*, buscando compreender a variação de suas contribuições durante o ciclo evolutivo do software (QP2). A correlação de *Spearman* foi calculada com o intuito de compreender se há uma relação direta entre a propriedade de código e o total de contribuições realizadas no repositório (QP3). É importante salientar que todos os dados e resultados utilizados nesse capítulo encontram-se disponíveis para consulta.[4]

As análises quantitativas não permitem responder questões como: a razão pela qual tantos autores efetuam poucas contribuições nos projetos *open source*; os moti-

[4] `https://github.com/taliorfano/knowledgedistribution_master/tree/master/Coleta_Dados/Tratamento_dados_e_an%C3%A1lises`

vos da saída dos *heroes* do projeto; se os desenvolvedores periféricos têm o desejo de efetuar mais contribuições no software; e como a disseminação do conhecimento é experimentada pelos desenvolvedores no cotidiano. Visando investigar questões como essas, neste trabalho também foi realizado um estudo com os desenvolvedores dos projetos analisados neste trabalho. O estudo é apresentado no Capítulo 7.

Capítulo 7

Distribuição do Conhecimento de Software - Análise Qualitativa

Este capítulo apresenta uma análise qualitativa realizada entre os autores dos cinco projetos em estudo - *Bootstrap*, *Django*, *Elasticsearch*, *Pandas* e *Spring Boot* - com o objetivo de responder as seguintes questões de pesquisa:

- QP4: Como se dá a atuação dos desenvolvedores periféricos nos projetos?
- QP5: Como ocorre a disseminação de conhecimento do software entre os autores?

Foi realizado um *survey* com autores dos projetos analisados neste trabalho. Seção 7.1 apresenta como o *survey* foi construído e suas questões propostas. Seção 7.2 define o público alvo que o *survey* foi direcionado. Seção 7.3 exibe os resultados encontrados nas respostas ao *survey*. Seção 7.4 discute os resultados encontrados, com o intuito de responder as questões de pesquisa QP4 e QP5.

7.1 *Survey*

O capítulo anterior apresentou uma análise quantitativa da natureza evolutiva das contribuições de autores dos projetos. Contudo, aqueles resultados não são capazes de responder as duas últimas questões de pesquisa com precisão, uma vez que elas envolvem não apenas as informações encontradas nos repositórios, mas também alguns fatores externos que são decisivos para a contribuição constante dos autores no projeto.

Sendo assim, foi realizado um *survey* que foi enviado para uma parcela de autores dos projetos em estudo. Essa análise permite uma melhor compreensão da influência

dos fatores externos nas contribuições, identificando o perfil dos autores, as suas motivações e as dificuldades encontradas durante as contribuições naquele projeto.

As questões tratadas no *survey* dividem-se em dois grupos: caracterização do autor e caracterização da disseminação do conhecimento. O primeiro grupo de perguntas busca compreender qual o perfil daquele autor, quais as motivações que o levaram a contribuir naquele repositório - trabalho, pesquisa, estudos, contribuição com a comunidade *open source*, etc -, quais as categorias das contribuições efetuadas e se existe interesse em tornar-se um autor frequente do projeto. O segundo grupo de perguntas busca compreender como ocorre a disseminação do conhecimento entre os autores, se há algum tipo de repasse de conhecimento para eles e se eles compartilham de alguma maneira o conhecimento daquilo que desenvolvem no projeto.

As perguntas do *Survey* encontram-se no Apêndice A. A próxima seção descreve como essas questões foram direcionadas para as três categorias de contribuidores.

7.2 Público Alvo

Com o intuito de compreender as motivações e características das diferentes categorias de autores, foram criadas três versões do questionário, contemplando as especificidades entre os autores periféricos, médios e *heroes*.

O questionário direcionado aos autores categorizados como *heroes* foi encaminhado para 100% desses autores nos cinco projetos. O mesmo foi feito no caso dos autores médios. Uma vez que a quantidade de autores periféricos é muito elevada, estabeleceu-se que apenas 5% desses autores receberiam o *survey* para ser respondido.

A Tabela 7.1 exibe os dados consolidados do público alvo do *survey*. Ela mostra por categoria: o número de autores selecionados para participar da pesquisa; o total de e-mails enviados com sucesso, isto é, que não retornaram; e o total de respostas obtidas.

7.3 Resultados

Foram obtidas um total de 47 respostas dos questionários enviados, o que corresponde a 11% da quantidade dos autores selecionados. Esta seção analisa as respostas fornecidas por esses autores.

Tabela 7.1. Total de autores convidados a responder o *Survey*, segmentado por projeto e categoria

Projeto	Categoria	Selecionados	E-mails enviados*	Respostas
Bootstrap	Autor periférico	64	64	2
	Autor médio	3	3	-
	Autor *hero*	4	4	-
Django	Autor periférico	104	104	22
	Autor médio	13	13	2
	Autor *hero*	6	6	1
Elasticsearch	Autor periférico	60	60	6
	Autor médio	21	18	1
	Autor *hero*	6	6	-
Pandas	Autor periférico	85	85	8
	Autor médio	9	7	1
	Autor *hero*	3	2	-
Spring Boot	Autor periférico	34	34	3
	Autor médio	3	3	1
	Autor *hero*	4	3	-

7.3.1 Motivações

O primeiro grupo de perguntas visa entender as motivações que levaram os autores a contribuir nos projetos em estudo, com enfoque na categoria de autores periféricos. Deseja-se compreender as razões pelas quais tantos autores contribuíram tão pouco no projeto.

Os resultados mostram que as principais motivações que levaram os autores periféricos a contribuir nos seus respectivos projetos foram: emprego na organização responsável pelo projeto; em razão de pesquisas acadêmicas ou estudo; desejo de aumentar os conhecimentos técnicos; ou a necessidade de melhorias no software para ser usado em projetos pessoais. O único autor *hero* que respondeu ao *survey* foi *Adrian Holovaty*, responsável pela criação do projeto *open source* do *Django*. Ele contribuiu nesse software por um longo período, antes mesmo de ele se tornar um projeto de código aberto.

A Figura 7.1 apresenta as motivações que levaram os autores periféricos a contribuírem nos projetos. Observa-se que o desejo de aumentar os conhecimentos técnicos pessoais e o emprego em uma organização responsável por aquele software foram as principais razões. O desejo de ajudar a comunidade *open source* não estava entre as opções do formulário enviado, mas não obstante isso, essa foi a terceira razão mais citada pelos autores. A atuação como *freelancer* e a necessidade de melhorar a solução

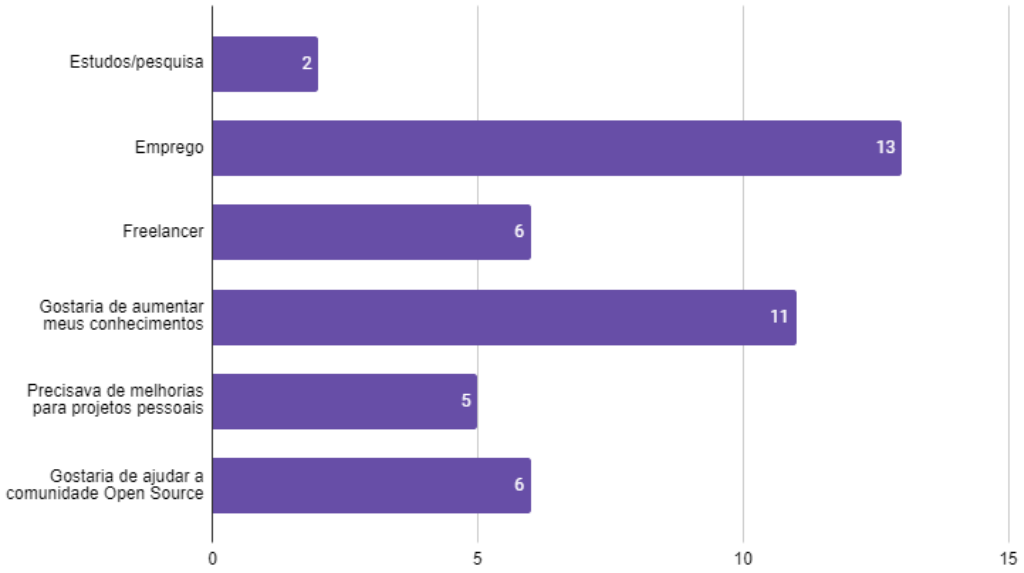

Figura 7.1. Autores periféricos - Motivação principal para contribuições no projeto

para ser usada em projetos pessoais também foram mencionadas.

O *survey* também indagou os autores periféricos acerca das razões que eles identificam para não terem se tornado autores frequentes no projeto. A Figura 7.2 mostra que a grande razão apontada por eles que justifica o número reduzido de contribuições é a falta de disponibilidade de tempo, totalizando 26 respostas. A falta de interesse em contribuir no projeto também foi mencionada por uma parcela considerável de autores. Além disso, sete contribuidores categorizados como periféricos consideraram-se como autores frequentes no projeto. Um desses autores afirmou que *"As contribuições que faço exigem muita pesquisa para cada commit, eu me consideraria um colaborador um tanto frequente"*.[1] Acredita-se que essa divergência de percepção deve ocorrer com aqueles contribuidores que efetuaram por volta 50 a 100 *commits* e, apesar de ser um número representativo, é uma quantidade considerada pequena em relação a quantidade total de *commits* daquele repositório. Três autores apontaram que não possuíam os requisitos técnicos necessários para tornar-se frequente, alguns citaram a falta de domínio na linguagem de programação e um outro, a dificuldade com a língua inglesa.

Foi questionado aos autores médios se eles pretendiam tornar-se um dos principais autores do projeto. Dois autores afirmaram que 'não', dois responderam 'não sei' e um considera que tornou-se um dos principais autores. Com relação aos autores *heroes*, foi perguntado se e por que as contribuições deles têm diminuído ou aumentado no decorrer

[1] "The contributions I make require a lot of research for each commit, I would consider myself a somewhat frequent contributor"

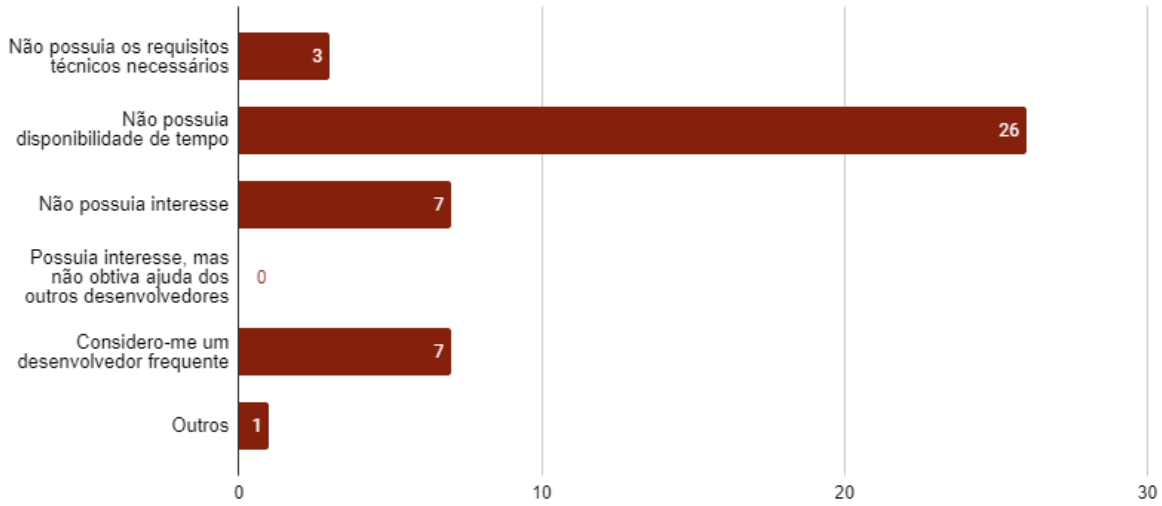

Figura 7.2. Autores periféricos - Razões para não ter se tornado um autor frequente no projeto

dos últimos anos. O único *hero* que respondeu ao *survey* teve uma forte diminuição no total dos *commits* até encerrar as contribuições no repositório, como constatado na Figura 6.7. O autor afirmou que estava disposto a fazer algo novo em sua vida, por isso deixou as contribuições.[2]

7.3.2 Disseminação do Conhecimento

O segundo grupo de perguntas busca compreender as principais formas de disseminação de conhecimento utilizada pelos autores durante o processo de desenvolvimento *open source*. Acredita-se que o conhecimento pode ser adquirido por meio da contribuição direta no código fonte e também pela comunicação entre os autores do projeto.

Uma das questões abordadas no *survey* pretende entender quais os tipos de contribuições feitas por cada uma das categorias de autores. Acredita-se que os autores periféricos não realizam tarefas complexas e que demandam um conhecimento profundo do software por parte do autor e muitas vezes exigem um grande tempo de dedicação para efetuá-las. São exemplos dessas tarefas: melhorias arquiteturais, alterações nas regras de negócio e refatorações expressivas (*refactor*).

A Figura 7.3 apresenta os tipos de contribuições realizadas pelos autores da categoria médio e o autor *hero*. Nota-se que o *hero* contribuiu com o projeto em todas as diferentes modalidades. Os autores médios também desempenharam diversos tipos de contribuição, seja por meio do desenvolvimento de novas funcionalidades, correção

[2] "I was ready to do something new with my life"

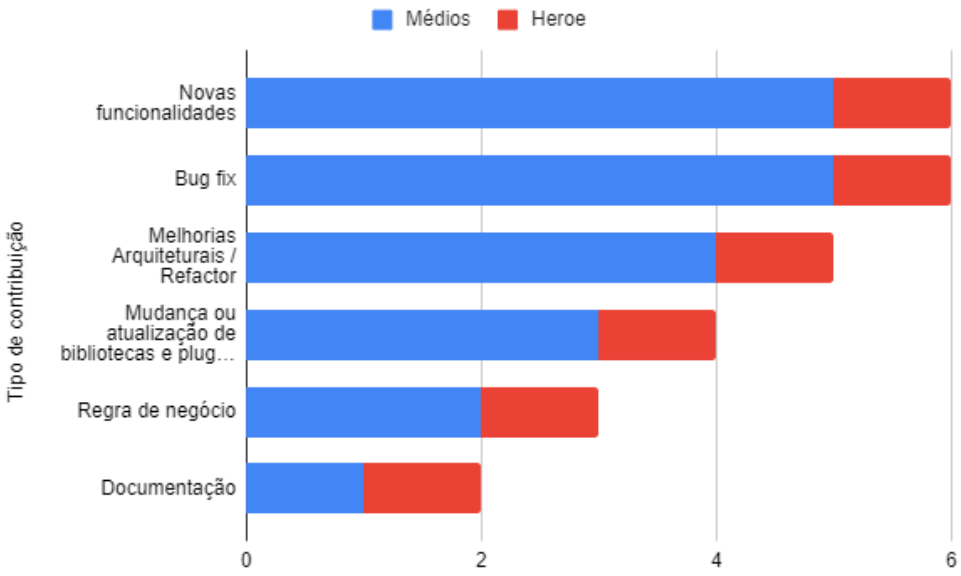

Figura 7.3. Autores médios e *heroes* - Tipos de contribuições efetuadas

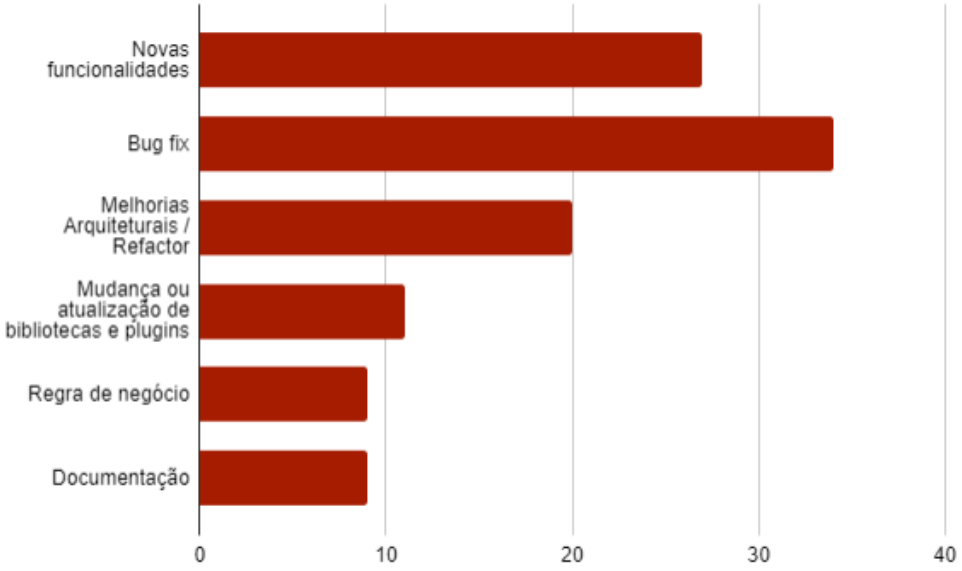

Figura 7.4. Autores periféricos - Tipos de contribuições efetuadas

de *bugs*, melhorias arquiteturais/refatoração, alteração de bibliotecas/*plug-ins* ou documentação. Entretanto, na maioria dos casos, diferentemente do *hero*, esses autores não realizam alterações nas regras de negócio do software.

A Figura 7.4 apresenta como as contribuições dos autores periféricos são tipificadas por eles. Constata-se que a correção de *bugs* é a principal dentre as contribuições efetuadas por essa categoria de autores. O acréscimo de novas funcionalidades e a exe-

cução de melhorias arquiteturais também são ações constantemente realizadas, segundo indica os resultados do *survey*. Apesar de não constar dentre as opções de resposta no formulário, o tipo de contribuição que envolve alteração e correção da documentação foi citado por nove vezes pelos autores que responderam o questionário, sendo oito respostas advindas de autores periféricos.

Para compreender a importância e a forma de comunicação entre os colaboradores dos projetos *open source*, bem como as dificuldades enfrentadas, foram formuladas três questões para o *survey*, sendo uma destinada aos autores periféricos e duas direcionadas aos autores médios e *heroes*. Aos periféricos buscou-se compreender se eles receberam auxílio de outros contribuidores durante o processo de desenvolvimento do software. Conforme detalhado no Apêndice A, essa questão possui três opções:

1. não recebi ajuda e não era necessário
2. não recebi ajuda, mas era necessário
3. recebi ajuda (descreva de que forma recebeu auxílio).

Os resultados indicam que 75% dos autores periféricos receberam auxílio e 25% não recebeu ajuda e não houve necessidade. Nenhum autor relatou que não recebeu ajuda, mas que havia necessidade. Dentre as formas de auxílio e comunicação apontados pelos contribuidores pode-se citar: envio de sugestões, conselhos e *feedbacks* por parte dos autores mais experientes; *code review* e co-contribuição; conversas pelo *chat*, e-mail e *issue tracker*; reuniões e *meetups*; além de orientação sobre padrões de codificação.

Aos autores médios e *heroes*, foram enviadas duas perguntas, uma buscando compreender como ocorreu a transferência de conhecimento para eles e outra com intenção de conhecer como ele compartilhou o conhecimento adquirido no processo de desenvolvimento com os outros membros da equipe. Os resultados de ambas perguntas foram sumarizados e apresentados na Figura 7.5, contendo as respostas dos autores médios e do *hero*. Os dados em azul correspondem à forma como o conhecimento do software foi compartilhado com o autor da resposta, enquanto a representação em vermelho representa o modo como o autor disseminou o conhecimento adquirido no processo de desenvolvimento no projeto. Dentre as respostas apresentadas, apenas em um caso, o do único *hero* que respondeu o *survey*, não houve repasse de conhecimento para o autor. Isso ocorreu porque ele foi o responsável pela criação do projeto, o *Django open source*. As principais formas de compartilhamento de conhecimento pelos autores médios e *heroes* são por meio de documentação PDF/DOC, troca de e-mails diretos, listas de e-mail, *chat* interno e pessoalmente.

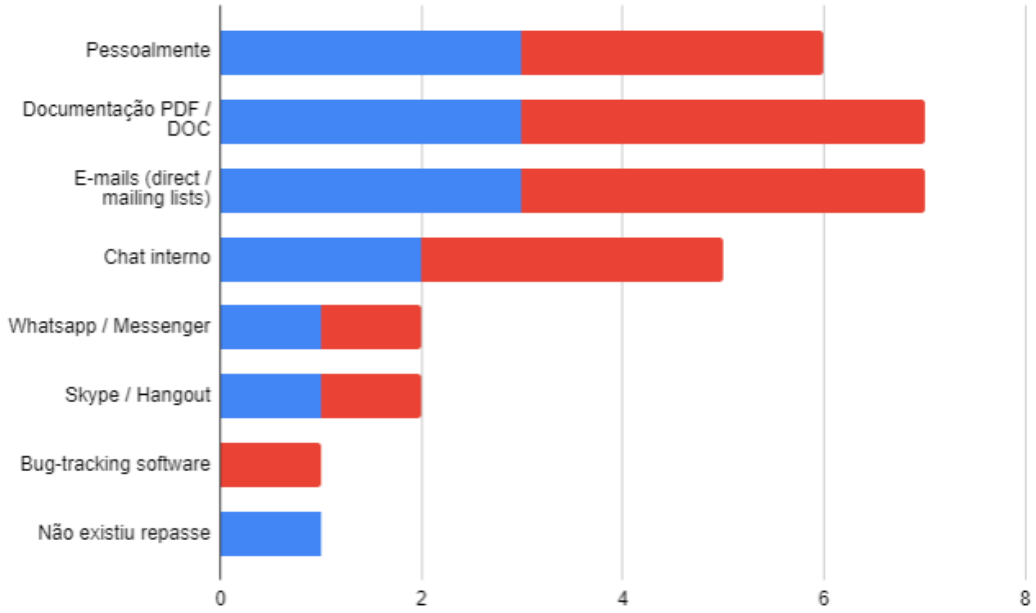

Figura 7.5. Transferência de conhecimento do software: em azul é como o conhecimento foi compartilhado com o autor; em vermelho é como ele transferiu o conhecimento

Os resultados do *survey* com todas as respostas obtidas podem ser encontrados no GitHub.[3]

7.4 Análise dos Resultados

A partir da análise dos resultados apresentados na Seção 7.3 obtidos pelas respostas do *survey*, busca-se responder as duas últimas questões de pesquisa. Esta seção apresenta essa análise.

QP4: Como se dá a atuação dos desenvolvedores periféricos nos projetos?

Observou-se no Capítulo 6 que aproximadamente 80% dos autores de um projeto efetuaram no máximo dois *commits* em todo o seu ciclo de vida. Assim, esta questão de pesquisa busca compreender melhor a atuação dos autores periféricos, suas motivações para contribuir no repositório, se existia algum desejo pessoal de tornar-se um contribuidor frequente no projeto e se houve obstáculos que o impediram de efetuar contribuições mais frequentes e que possam, por ventura, tê-los impedido de tornarem-se *heroes*.

[3] `https://github.com/taliorfano/knowledgedistribution_master/tree/master/Survey/Respostas`

Os resultados apresentados na Seção 7.3 mostram que as motivações que levam os autores periféricos a contribuir com o projeto *open source* variam entre o emprego na organização responsável por aquele projeto (30,23%), o desejo de aumentar os conhecimentos técnicos pessoais (25,58%), o desejo de contribuir com a comunidade *open source* (13,95%), *freelancer* (13,95%) e necessidade de melhorias para projetos pessoais (11,63%).

A principal razão para um autor periférico não aumentar as suas contribuições é a falta de disponibilidade de tempo, como foi majoritariamente assinalado pelos próprios autores durante a pesquisa.

As respostas obtidas também mostram que essa categoria de autores tende a efetuar, principalmente, correções de *bugs*, acréscimo de novas funcionalidades, melhorias arquiteturais e correção na documentação do projeto. O tipo de atividade menos realizada é a alteração da regra de negócio do software, uma vez que esse tipo de contribuição demanda um conhecimento mais maduro do software. Esses resultados estão em consonância com a literatura, uma vez que estudos prévios indicam que os desenvolvedores periféricos fazem contribuições significativas para a qualidade do software, especialmente por meio da correção de *bugs* e acrescentando novas funcionalidades para uso próprio [Lee et al., 2017; Setia et al., 2012; Ye & Kishida, 2003].

Os autores periféricos também relataram que frequentemente recebem ajuda dos outros colaboradores do projeto, seja por meio de conversas informais (*chat*), troca de e-mails, reuniões, *code review*, conselhos, sugestões ou mesmo *feedbacks*. Isso sugere que a dificuldade de comunicação não é um fator preponderante que os levou a contribuir pouco no projeto.

QP5: Como ocorre a disseminação de conhecimento do software entre os autores?

Esta questão de pesquisa visa compreender como ocorre a disseminação do conhecimento adquirido no projeto *open source*. Pretende-se também compreender se ocorrem fatores que dificultaram a disseminação ou aquisição do conhecimento do software.

Considerando todas as categorias de autores, a contribuição por meio do desenvolvimento de novas funcionalidades, correção de *bugs* e melhorias arquiteturais constituem os principais meios de aquisição de conhecimento sobre o projeto e o seu código fonte disponível no GitHub, segundo as respostas obtidas. Sem dúvida, pode-se salientar que, ao efetuar a correção de um *bug*, é necessário compreender o contexto no qual ele está inserido e seus possíveis impactos. O acréscimo de novas funcionalidades

também exige do autor um conhecimento razoável das funcionalidades já existentes e da arquitetura do software, visando a preservação dos padrões de codificação e o comportamento padronizado das funcionalidades do projeto.

Praticamente todos os contribuidores reportaram que houve compartilhamento de conhecimento extra-código nas duas vias, isto é, outros colaboradores compartilharam informações importantes do código com eles e eles também disseminaram o conhecimento adquirido durante o processo de desenvolvimento com outros autores do projeto. Essa troca de informações ocorre principalmente por meio de documentações do projeto, troca de e-mails, pessoalmente, por *chats* internos, *Whatsapp/Messeger*, *Skype/Hangout* e *bug-tracking*. Por meio dessas comunicações, ocorrem o envio de *feedbacks*, orientações, sugestões, auxílio no desenvolvimento da atividade, revisão de código, colaboração conjunta e reuniões de alinhamento.

Pode-se salientar a partir dessas informações que, apesar desses projetos serem *open source* e agregarem diferentes tipos de contribuidores com distintas motivações, a comunicação e disseminação do conhecimento ocorre de modo fluído e sem grandes empecilhos. Isso facilita a entrada de novos autores, mesmo que seja com o objetivo de efetuar contribuições pontuais. Nenhum autor relatou dificuldade de comunicação ou obstáculos na aquisição dos conhecimentos necessários para a execução das atividades desempenhadas.

Ye & Kishida [2003] afirmam que a participação de novos colaboradores cria oportunidade de iteração com outros desenvolvedores que possuem mais conhecimento e habilidade no projeto. Segundo os autores, a lista de e-mail era a principal maneira utilizada entre os colaboradores para disseminação do conhecimento. Por meio de e-mails eram discutidas soluções para correção de falhas, maneiras de evoluir ou melhoras funcionalidades e também como forma de ajudar os novos contribuidores a entender as particularidades do projeto. Segundo Ye & Kishida [2003], as comunidades *open source* criam redes de conhecimento, na qual compartilham seus conhecimentos e aprendem uns com os outros.

7.5 Considerações Finais

Neste capítulo, foi apresentada a análise qualitativa dos dados, a partir das respostas obtidas na pesquisa efetuada com contribuidores de cinco projetos. O *survey* foi concebido para investigar a forma de atuação e motivação dos autores periféricos (QP4), bem como o modo que o conhecimento do software é disseminado entre os autores (QP5). A pesquisa foi enviada para 100% dos contribuidores categorizados como *heroes* e médios,

e para 5% dos autores periféricos. Foram obtidas 47 respostas, resultando em 11% do total de destinatários da pesquisa, sendo a maioria contribuidores periféricos.

Os resultados mostram que os desenvolvedores periféricos, em geral, não tem pretensão de continuar colaborando com o projeto, devido limitação de tempo ou falta de interesse. As respostas também sugerem que não houveram dificuldades na comunicação entre os colaboradores e que as principais formas de compartilhamento do conhecimento entre eles é realizada por meio de documentações e listas de e-mail.

A discussão dos resultados encontrados nas análises quantitativas e qualitativas é apresentada no Capítulo 8.

Capítulo 8

Discussão dos Resultados

Foi realizado um estudo de caso com cinco grandes projetos *open source* com o objetivo de compreender como ocorre a evolução da distribuição de conhecimento entre os desenvolvedores durante o ciclo de vida do software.

Por meio dos dados coletados, pode-se constatar que, em média 80% dos desenvolvedores efetuam apenas uma ou duas contribuições no projeto. Alguns poucos desenvolvedores, nomeados *heroes*, retêm grande parcela das contribuições e, consequentemente, o conhecimento do software. Esses dados estão em consonância com outros trabalhos que se dedicaram a estudar projetos *open source* [Majumder et al., 2019; Agrawal et al., 2018; Coelho et al., 2018; Pinto et al., 2016; Avelino et al., 2016; Ricca & Marchetto, 2010; Robles et al., 2009]. Contudo, neste trabalho foi avaliada individualmente a atuação de cada *hero* durante o ciclo evolutivo do projeto.

Foi avaliada a distribuição da métrica de propriedade de código no decorrer do ciclo de vida dos projetos, a partir das definições propostas por Foucault et al. [2014] e Bird et al. [2011]. Os resultados mostram que a propriedade de código do projeto em um período de tempo, definido por $max(own_{t,d})$, tende a diminuir nos últimos períodos avaliados, o que sugere que o conhecimento torna-se mais distribuído entre os desenvolvedores no decorrer de seu processo evolutivo. Contudo, como não foram encontrados na literatura trabalhos que avaliassem a variação dessa métrica durante a evolução do software, torna-se necessário a realização de outros estudos, utilizando uma quantidade maior de projetos, para que seja possível identificar se o comportamento apresentado nesse trabalho de mestrado é uma característica recorrente entre projetos *open source*.

Os achados desse trabalho indicam que não há relação entre o número de contribuições do projeto e o valor da métrica de propriedade de código. Contudo, como esperado, há uma relação direta e forte entre o aumento das contribuições dos *heroes* e

o aumento das contribuições totais nos projetos. Esse resultado expressa a importância dos *heroes* para o crescimento dos projetos *open source*, corroborando com o trabalho de Robles et al. [2009] que indica que a permanência dos *heroes* é fundamental para a evolução do projeto.

Ao analisar a atuação dos *heroes* no decorrer do ciclo evolutivo do projeto, percebe-se que aqueles que possuíam uma atuação intensa e frequente nos primeiros períodos não permaneceram como contribuidores frequentes até os últimos períodos. Os resultados encontrados mostram que, na maioria dos casos, houve alternância entre os *heroes*. Esse cenário no qual grupos de desenvolvedores principais temporários se sucedem no decorrer do ciclo de vida do software é denominado "série de gerações"(*"series of generations"*) [Robles et al., 2009]. No entanto, é importante a realização de outros estudos como o apresentado nesta dissertação para se obter maior aprofundamento do assunto.

Os resultados evidenciam a importância em considerar o período evolutivo de contribuições dos desenvolvedores em estudos que avaliam o domínio e o conhecimento dos contribuidores em relação ao código fonte em projetos *open source*. Destaca-se que um autor pode ser classificado como *hero* pela expressividade de suas contribuições no histórico do projeto, mas não necessariamente ele possui o domínio do código no momento presente.

Também pretendia-se com esse trabalho, explorar as razões que levaram os primeiros *heroes* dos projetos analisados a reduzirem suas contribuições ou até mesmo abandonar o projeto, contudo apenas um *hero* respondeu ao *survey*. Segundo a literatura [Avelino et al., 2019a; Coelho et al., 2018; Coelho & Valente, 2017; Lee et al., 2017], as principais barreiras enfrentadas pelos *heroes* e também as razões que os levam a abandonar os projetos *open source* são a falta de tempo, a falta de interesse, o tamanho e complexidade do projeto, a falta de tempo dos líderes, código fonte com muitos erros e de difícil entendimento e, a complexidade no processo de envio das contribuições. Avelino et al. [2019a] abordam a importância do surgimento de novos *heroes* para sobrevivência dos projetos e afirmam que as principais motivações que levaram os desenvolvedores a se tornarem frequentes foram a necessidade de melhorias do software para uso pessoal, o desejo de contribuir com a comunidade *open source* e a tentativa de evitar que o projeto fosse descontinuado.

Com relação aos desenvolvedores periféricos, os resultados do *survey* realizado neste trabalho apontam que as principais motivações que os levaram a contribuir no repositório foram seus empregos, o desejo de aumentar os conhecimentos técnicos, o desejo de contribuir com a comunidade *open source*, o *freelancer* e a necessidade de melhorias para projetos pessoais. Esses resultados estão em consonância com a litera-

tura. Os resultados encontrados nos cinco estudos de caso realizados neste trabalho se enquadram em achados de Lee et al. [2017] que indicam que a maioria das contribuições realizadas pelos desenvolvedores periféricos é motivada por necessidades particulares, isto é, são realizadas correções de *bugs* ou melhorias de funcionalidades que estão impedindo a utilização do software em seus projetos pessoais ou em produtos que trabalham. Uma outra motivação citada nos resultados de Lee et al. [2017] é o desejo de contribuir e dar um retorno à comunidade *open source*.

Dentre os cinco projetos selecionados, as principais formas de atuação dos colaboradores periféricos foram por meio de correção de *bugs*, acréscimo de novas funcionalidades, refatorações no código, atualização de bibliotecas e documentação. Esses comportamentos aparecem em trabalhos prévios [Lee et al., 2017; Pinto et al., 2016; Setia et al., 2012]. As principais razões que os periféricos apontaram para não terem se tornado colaboradores frequentes são a falta de interesse e a falta de disponibilidade de tempo, esses argumentos também foram bastante mencionados nos estudos de Lee et al. [2017]. Steinmacher et al. [2015] abordam 58 barreiras, segmentadas em 6 categorias, que demonstram as dificuldades enfrentadas por desenvolvedores periféricos em projetos *open source*. Dentre essas barreiras destacam-se a falta de documentação, as dificuldades técnicas em relação ao projeto devido à sua alta complexidade e baixa qualidade do código, o transtorno no processo de submissão da contribuição, a dificuldade em encontrar um mentor, as limitações técnicas do contribuidor e barreiras culturais.

Essas dificuldades relatadas na literatura apontam principalmente para a importância da distribuição do conhecimento, em especial por meio de códigos bem escritos [Martin, 2013], documentações abrangentes e comunicação fluida entre os desenvolvedores envolvidos no projeto. Assim, o presente trabalho também dedicou-se a compreender como ocorreu a transmissão de conhecimento entre os desenvolvedores, considerando também os cenários extra-código. As respostas do *survey* sugerem que não houve grandes dificuldades no compartilhamento de informação nos cinco projetos analisados. Os principais meios utilizados na transmissão de conhecimento sobre o projeto foram por documentações, listas de e-mail e conversas informais, seja pessoalmente ou por *chat*. O compartilhamento de informações entre os contribuidores é vital para subsistência de todo projeto. Atraso nas respostas, falta de *feedbacks* nas contribuições e dificuldade em sanar dúvidas podem ser fatores decisivos para que novos colaboradores não se engajem no projeto [Zhou & Mockus, 2014; Gousios et al., 2016].

Os achados deste trabalho contribuem, principalmente, na compreensão do processo evolutivo na atuação dos desenvolvedores *heroes*, bem como a forma como a propriedade de código, conforme definida por Foucault et al. [2014] varia no decorrer

do tempo, sugerindo que o conhecimento torna-se mais distribuído ao longo do ciclo vida do projeto *open source*. No entanto, é importante a realização de estudos futuros que considerem o período evolutivo de contribuições na identificação de *heroes* para validar esses achados, especialmente em trabalhos que abrangem outras formas de análise de propriedade de código, algoritmos de *truck factor*, algoritmos que selecionam revisores de código e métodos para identificação de mentores, que possuem a finalidade de auxiliar desenvolvedores recém-chegados ao projeto.

Finalmente, os resultados também reforçam que os gestores de projetos *open source* precisam investir em ações para encontrar e reter desenvolvedores *heroes*, peças importantes para a continuidade e sucesso dos projetos. Agrawal et al. [2018] afirmam que os *heroes* são mais comuns e mais valiosos do que sugere a literatura, especialmente em projetos de médio e grande porte. O incentivo de novos desenvolvedores, mesmo os periféricos, também é importante porque por meio da correção de *bugs* e acréscimo de novas funcionalidades, eles contribuem significativamente para a qualidade do software [Setia et al., 2012]. Além disso, Ye & Kishida [2003] afirmam que, em geral, grandes contribuidores começam com participações periférica, por exemplo, relatando e corrigindo *bugs*, a partir disso, aumentam suas habilidades no projeto e gradualmente lhe são confiadas tarefas maiores e mais desafiadoras. Contudo, é importante ressaltar que a documentação sólida, a facilidade de comunicação e a qualidade do processo e do código, são fatores primordiais para retenção de novos colaboradores.

Capítulo 9

Ameaças à Validade

As ameaças à validade deste estudo, bem como as decisões tomadas para mitigá-las, são tratadas neste capítulo.

O *data set* utilizado neste trabalho é composto por cinco projetos *open source*, presentes no GitHub. Sendo assim, não é possível afirmar que os comportamentos e os resultados aqui encontrados possam ser generalizados para outros projetos, sejam eles da modalidade *open source* ou sistemas proprietários. Embora a amostra possa não ser representativa o suficiente para que os resultados possam ser generalizados, os projetos selecionados para estudo são grandes e relevantes no contexto GitHub, os quais possuem um grande número de *commits*, autores, estrelas e *forks*. A limitação da quantidade de projetos a cinco se deve à complexidade da análise realizada, que envolveu a geração e análise de cada um dos projetos, uma vez que as análises foram conduzidas como estudo de casos.

Um dos objetivos do trabalho foi ratificar o resultado do trabalho de Foucault et al. [2014] e outros estudos prévios, que afirmam que em projetos *open source* médios e grandes, o conhecimento do sistema tende a se concentrar em apenas um desenvolvedor durante o seu ciclo de vida. Por essa razão, não foram analisados projetos pequenos neste estudo. Assim, os resultados apresentados podem não ser equivalentes para repositórios de pequeno porte, especialmente os que possuem menos de cinco anos de existência no GitHub ou que tenham menos de 500 contribuidores vinculados.

Nos repositórios, é possível que um mesmo autor esteja identificado de formas distintas. Para contornar essa ameaça, foi realizado um tratamento para identificação dos casos de identificação ambígua de autores. Existem heurísticas propostas na literatura para essa desambiguação. Elas foram analisadas e concluiu-se que a aplicação delas não solucionaria a ameaça. Por essa razão, a desambiguação foi realizada manualmente. Contudo, apenas os autores *heroes*, médios e os periféricos que possuíam um grande

número de contribuições foram tratados. Esse procedimento pode ter acarretado em uma quantidade ligeiramente maior de autores periféricos do que a que de fato existe, uma vez que os ambíguos dessa categoria foram contados como sendo contribuidores distintos. De qualquer forma, essa categoria de autores não impactou nas análises realizadas no trabalho. O tratamento das ambiguidades de autores pode conter erros e influenciar no resultado. Para contornar isso, a análise foi realizada com minucioso cuidado manual.

A principal limitação das análises qualitativas resultam na quantidade de respostas fornecidas pelo público alvo. O *survey* foi encaminhado para 100% dos autores *heroes* e médios, e para 5% dos desenvolvedores periféricos. No entanto, a quantidade de respostas obtidas somam apenas 11% do total de e-mails enviados. Embora a taxa de resposta seja pequena, as respostas dos autores que responderam trazem informações importantes. O questionário possui perguntas abertas e a análise desses textos foi manual, o que insere uma ameaça, ainda que pequena, nos resultados reportados. Ao analisar as respostas do *survey* observamos que *"O desejo de ajudar a comunidade open source"* foi uma motivação citada com recorrência pelos autores, embora ela não estivesse dentre as opções do formulário enviado. Acredita-se que outros desenvolvedores não a tenham citado por não estar entre as alternativas do *survey*, mesmo sendo esta uma das razões que os levou a contribuir no projeto.

Por fim, para realização deste trabalho, foi considerado o método proposto por Bird et al. [2011] e Foucault et al. [2014] para o cálculo da propriedade de código. Existem outras formas para se tratar a análise de propriedade de código, por exemplo, considerando o total de arquivos alterados pelo desenvolvedor em vez do número de *commits*. Constata-se que a melhor forma de avaliar a propriedade de código de um software ainda é um campo aberto para pesquisas. Entretanto, para o método de cálculo da propriedade aplicado neste trabalho já existem resultados de avaliação prévios e por isso esse método foi o escolhido.

Capítulo 10

Conclusão

Muitos pesquisadores têm se dedicado a estudos mais aprofundados sobre o software, a fim de obter resultados que possam contribuir para, por exemplo, reduzir e prever falhas, otimizar e aumentar a qualidade do software. Há estudos que comprovam que fatores humanos influenciam diretamente a qualidade do software desenvolvido. No campo de software *open source*, um dos fatores humanos considerados é a propriedade de código, que pode medir o grau de contribuição e conhecimento de um desenvolvedor em relação ao projeto. Este trabalho foi conduzido nesse contexto, tendo como objetivo investigar a distribuição do conhecimento dos autores em projetos *open source* durante seu ciclo evolutivo.

Foram estudados cinco projetos *open source* presentes no *GitHub*, sendo eles *Bootstrap*, *Django*, *Elasticsearch*, *Pandas* e *Spring Boot*. Para isso, foram coletados dados vinculados aos *commits*, aos autores e data da contribuição. Os dados dos projetos foram estudados e analisados para se explorar como o conhecimento se distribuiu entre os desenvolvedores ao longo do tempo. Essas análises envolveram a distribuição e caracterização da população de autores, a distribuição dos *heroes* no ciclo evolutivo, a frequência de contribuições no projeto e o cálculo da métrica de propriedade de código. Os *scripts* criados neste trabalho para coleta de dados foram disponibilizados no GitHub, o que garante a replicabilidade do estudo e a realização de trabalhos futuros na linha deste trabalho.

Foram conduzidos três *surveys* por meio de questionários destinados aos autores *heroes*, médios e periféricos dos projetos analisados. O intuito da construção dos questionários foi compreender a forma de atuação dos desenvolvedores de projetos *open source* e as formas como o conhecimento do software é propagado entre eles, além daquelas que podem ser observadas pela análise dos dados extraídos diretamente dos repositórios.

Por meio dos resultados encontrados, pode-se salientar que a distribuição de conhecimento do software aumenta no decorrer de seu ciclo de vida, possibilitando a entrada de novos contribuidores e que eles venham a tornar-se futuros *heroes* do projeto. No entanto, devido às características intrínsecas da modalidade *open source*, todos os softwares analisados possuíam um grande número de autores periféricos que efetuaram apenas uma ou duas contribuições, enquanto um grupo pequeno de desenvolvedores são responsáveis pela maior parte dos *commits* existentes no repositório.

O estudo também permitiu concluir que os desenvolvedores *heroes* no início do projeto não permaneceram nessa posição até os últimos períodos analisados. Contudo, esses autores tiveram uma diminuição gradual das suas atividades no repositório, diferentemente do que ocorreria em um projeto privado.

Foi constatado também que não existe relação direta entre a quantidade de contribuições do projeto e a métrica de propriedade de código. Contudo, há uma forte correlação entre o crescimento das contribuições e o aumento da atuação dos desenvolvedores *heroes* no projeto.

A disseminação do conhecimento do software ocorre principalmente por meio do desenvolvimento de novas funcionalidades, pela correção de *bugs*, por disponibilização de documentações, envio de e-mails ou por meio de troca de mensagens informais. Apesar de os desenvolvedores periféricos não terem encontrado obstáculos para a realização de suas atividades nos projetos *open source*, a maioria deles não pretende tornar-se um contribuidor frequente, devido à falta de tempo e interesse pessoal.

Dentre os resultados obtidos por meio desse trabalho de mestrado, houveram duas grandes contribuições para o estado da arte. A primeira contribuição foi a análise individualizada dos *heroes* do projeto, o que resultou na constatação da alternância no comportamento dos *heroes* ao longo do ciclo evolutivo do software. A segunda nova contribuição para a literatura consiste no estudo do comportamento da métrica de propriedade de código no decorrer do ciclo de vida do projeto. Constatou-se ao avaliar o processo evolutivo de forma segmentada, que em todos os projetos ocorreu uma queda no valor da métrica nos últimos períodos analisados.

Os resultados deste trabalho contribuem para o aprofundamento do conhecimento sobre a autoria, propriedade de código e a distribuição do conhecimento no processo evolutivo de software *open source*. Com esses resultados foi evidenciado o quão importante é considerar, em estudos que envolvam esses conceitos, o fator evolutivo das contribuições. Caso esse fator não seja observado, um colaborador pode ser considerado *hero* do projeto pelo seu histórico de contribuições, mas possuir baixa quantidade de contribuições nas últimas versões do projeto, o que pode impactar diretamente os resultados das pesquisas. Os resultados também evidenciam aos

gerentes de projetos *open source* a importância que os desenvolvedores *heroes* têm no crescimento do projeto e a necessidade de criar meios de retê-los na equipe.

Os trabalhos futuros são identificados na sequência.

- Replicar o presente trabalho utilizando uma quantidade maior de softwares no *data set*, a fim de identificar padrões na atuação dos *heroes* e na variação da métrica de propriedade de código no decorrer do ciclo evolutivo de projetos *open source*.
- Realizar um estudo calculando a propriedade de código a partir do total de arquivos alterados e efetuar uma análise comparativa com os resultados deste trabalho.
- Investir na pesquisa e no desenvolvimento de métodos mais efetivos para desambiguação de dados de autores de *commits*.
- Criar uma ferramenta que permite a coleta e a análise automatizada da atuação dos desenvolvedores que contribuíram em projetos hospedados no GitHub.

Referências Bibliográficas

Agrawal, A.; Rahman, A.; Krishna, R.; Sobran, A. & Menzies, T. (2018). We don't need another hero. Em *Proceedings of the 40th International Conference on Software Engineering Software Engineering in Practice-ICSE-SEIP*, volume 18.

Avelino, G.; Constantinou, E.; Valente, M. T. & Serebrenik, A. (2019a). On the abandonment and survival of open source projects: An empirical investigation. Em *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 1--12. IEEE.

Avelino, G.; Passos, L.; Hora, A. & Valente, M. T. (2016). A novel approach for estimating truck factors. Em *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pp. 1--10. IEEE.

Avelino, G.; Passos, L.; Hora, A. & Valente, M. T. (2019b). Measuring and analyzing code authorship in 1+ 118 open source projects. *Science of Computer Programming*, 176:14--32.

Bird, C.; Gourley, A.; Devanbu, P.; Gertz, M. & Swaminathan, A. (2006). Mining email social networks. Em *Proceedings of the 2006 international workshop on Mining software repositories*, pp. 137--143.

Bird, C.; Nagappan, N.; Devanbu, P.; Gall, H. & Murphy, B. (2009). Does distributed development affect software quality? an empirical case study of windows vista. Em *Proceedings of the 31st international conference on software engineering*, pp. 518--528. IEEE Computer Society.

Bird, C.; Nagappan, N.; Murphy, B.; Gall, H. & Devanbu, P. (2011). Don't touch my code!: examining the effects of ownership on software quality. Em *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pp. 4--14. ACM.

Bodhuin, T. (1995). An interaction paradigm for impact analysis.

Canfora, G.; Cerulo, L.; Cimitile, M. & Di Penta, M. (2011). Social interactions around cross-system bug fixings: the case of freebsd and openbsd. Em *Proceedings of the 8th working conference on mining software repositories*, pp. 143--152. ACM.

Canfora, G.; Di Penta, M.; Oliveto, R. & Panichella, S. (2012). Who is going to mentor newcomers in open source projects? Em *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pp. 1--11.

Coelho, J. & Valente, M. T. (2017). Why modern open source projects fail. Em *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 186--196.

Coelho, J.; Valente, M. T.; Silva, L. L. & Hora, A. (2018). Why we engage in floss: Answers from core developers. Em *Proceedings of the 11th International Workshop on Cooperative and Human Aspects of Software Engineering*, pp. 114--121.

Crowston, K.; Wei, K.; Li, Q. & Howison, J. (2006). Core and periphery in free/libre and open source software team communications. Em *Proceedings of the 39th Annual Hawaii International Conference on System Sciences (HICSS'06)*, volume 6, pp. 118a--118a. IEEE.

Ferreira, M.; Mombach, T.; Valente, M. T. & Ferreira, K. (2019). Algorithms for estimating truck factors: a comparative study. *Software Quality Journal*, 27(4):1583--1617.

Ferreira, M. M.; Ferreira, K. A. M. & Valente, M. T. (2016). Distribuição de conhecimento de código em times de desenvolvimento-uma análise arquitetural. *Sociedade Brasileira de Computação–SBC*, p. 89.

Foucault, M.; Falleri, J.-R. & Blanc, X. (2014). Code ownership in open-source software. Em *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, p. 39. ACM.

Fritz, T.; Murphy, G. C.; Murphy-Hill, E.; Ou, J. & Hill, E. (2014). Degree-of-knowledge: Modeling a developer's knowledge of code. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(2):14.

Goeminne, M. & Mens, T. (2011). A comparison of identity merge algorithms for software repositories. *Science of Computer Programming*, 78(8):971--986.

Gousios, G.; Storey, M.-A. & Bacchelli, A. (2016). Work practices and challenges in pull-based development: the contributor's perspective. Em *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 285--296. IEEE.

Greiler, M.; Herzig, K. & Czerwonka, J. (2015). Code ownership and software quality: a replication study. Em *Proceedings of the 12th Working Conference on Mining Software Repositories*, pp. 2--12. IEEE Press.

Joblin, M.; Apel, S.; Hunsen, C. & Mauerer, W. (2017). Classifying developers into core and peripheral: An empirical study on count and network metrics. Em *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 164--174. IEEE.

Kocaguneli, E.; Zimmermann, T.; Bird, C.; Nagappan, N. & Menzies, T. (2013). Distributed development considered harmful? Em *Proceedings of the 2013 International Conference on Software Engineering*, pp. 882--890. IEEE Press.

Kouters, E.; Vasilescu, B.; Serebrenik, A. & van den Brand, M. G. (2012). Who's who in gnome: Using lsa to merge software repository identities. Em *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pp. 592--595. IEEE.

LaToza, T. D.; Venolia, G. & DeLine, R. (2006). Maintaining mental models: a study of developer work habits. Em *Proceedings of the 28th international conference on Software engineering*, pp. 492--501. ACM.

Lee, A.; Carver, J. C. & Bosu, A. (2017). Understanding the impressions, motivations, and barriers of one time code contributors to floss projects: a survey. Em *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pp. 187--197. IEEE.

Lehman, M. M. (1996). Laws of software evolution revisited. Em *European Workshop on Software Process Technology*, pp. 108--124. Springer.

Lehman, M. M. & Belady, L. A. (1985). *Program evolution: processes of software change*. Academic Press Professional, Inc.

Majumder, S.; Chakraborty, J.; Agrawal, A. & Menzies, T. (2019). Why software projects need heroes (lessons learned from 1000+ projects). *arXiv preprint arXiv:1904.09954*.

Martin, R. C. (2013). *Clean Code-Refactoring, Patterns, Testen und Techniken für sauberen Code: Deutsche Ausgabe*. MITP-Verlags GmbH & Co. KG.

McIntosh, S.; Kamei, Y.; Adams, B. & Hassan, A. E. (2016). An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, 21(5):2146--2189.

Meneely, A. & Williams, L. (2009). Secure open source collaboration: an empirical study of linus' law. Em *Proceedings of the 16th ACM conference on Computer and communications security*, pp. 453--462. ACM.

Meyer, B. (1988). *Object-oriented software construction*, volume 2. Prentice hall New York.

Müller, C.; Reina, G. & Ertl, T. (2015). In-situ visualisation of fractional code ownership over time. Em *Proceedings of the 8th International Symposium on Visual Information Communication and Interaction*, pp. 13--20. ACM.

Nagappan, N.; Murphy, B. & Basili, V. (2008). The influence of organizational structure on software quality. Em *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pp. 521--530. IEEE.

Nordberg, M. E. (2003). Managing code ownership. *IEEE software*, 20(2):26--33.

Orfanó, T. S.; Ferreira, K. A. & Bigonha, M. A. (2018). Heurísticas para identificação de ambiguidade de autores em projetos open source.

Palomba, F.; Tamburri, D. A.; Serebrenik, A.; Zaidman, A.; Fontana, F. A. & Oliveto, R. (2018). Poster: How do community smells influence code smells? Em *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pp. 240--241. IEEE.

Parikh, G. & Zvegintzov, N. (1983). *Tutorial on SW Maintenance*. IEEE Computer Society Press.

Pfleeger, S. L. (2004). *Engenharia de software: teoria e prática*. Prentice Hall.

Pinto, G.; Steinmacher, I. & Gerosa, M. A. (2016). More common than you think: An in-depth study of casual contributors. Em *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pp. 112--123. IEEE.

Pressman, R. S. (2005). *Software engineering: a practitioner's approach*. Palgrave Macmillan.

Rahman, F. & Devanbu, P. (2011). Ownership, experience and defects: a fine-grained study of authorship. Em *Proceedings of the 33rd International Conference on Software Engineering*, pp. 491--500. ACM.

Ricca, F. & Marchetto, A. (2010). Are heroes common in floss projects? Em *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pp. 1--4.

Robles, G. & Gonzalez-Barahona, J. M. (2005). Developer identification methods for integrated data from various sources. *ACM SIGSOFT Software Engineering Notes*, 30(4):1--5.

Robles, G.; Gonzalez-Barahona, J. M. & Herraiz, I. (2009). Evolution of the core team of developers in libre software projects. Em *2009 6th IEEE international working conference on mining software repositories*, pp. 167--170. IEEE.

Setia, P.; Rajagopalan, B.; Sambamurthy, V. & Calantone, R. (2012). How peripheral developers contribute to open-source software development. *Information Systems Research*, 23(1):144--163.

Sommerville, I. et al. (2007). *Software engineering*. Addison-wesley.

Steinmacher, I.; Conte, T.; Gerosa, M. A. & Redmiles, D. (2015). Social barriers faced by newcomers placing their first contribution in open source software projects. Em *Proceedings of the 18th ACM conference on Computer supported cooperative work & social computing*, pp. 1379--1392.

Thongtanunam, P.; McIntosh, S.; Hassan, A. E. & Iida, H. (2016). Revisiting code ownership and its relationship with software quality in the scope of modern code review. Em *Proceedings of the 38th international conference on software engineering*, pp. 1039--1050.

Thongtanunam, P.; Tantithamthavorn, C.; Kula, R. G.; Yoshida, N.; Iida, H. & Matsumoto, K.-i. (2015). Who should review my code? a file location-based code-reviewer recommendation approach for modern code review. Em *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 141--150. IEEE.

Torchiano, M.; Ricca, F. & Marchetto, A. (2011). Is my project's truck factor low?: theoretical and empirical considerations about the truck factor threshold. Em *Proceedings of the 2nd International Workshop on Emerging Trends in Software Metrics*, pp. 12--18. ACM.

Tufano, M.; Palomba, F.; Bavota, G.; Oliveto, R.; Di Penta, M.; De Lucia, A. & Poshyvanyk, D. (2015). When and why your code starts to smell bad. Em *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pp. 403--414. IEEE.

Ye, Y. & Kishida, K. (2003). Toward an understanding of the motivation of open source software developers. Em *25th International Conference on Software Engineering, 2003. Proceedings.*, pp. 419--429. IEEE.

Zhou, M. & Mockus, A. (2014). Who will stay in the floss community? modeling participant's initial behavior. *IEEE Transactions on Software Engineering*, 41(1):82--99.

Apêndice A

Nesse apêndice são apresentadas as questões que compõem os três *surveys* encaminhados aos autores *heroes*, médios e periféricos. A motivação das perguntas, o público alvo, bem como as respostas da pesquisa estão elucidadas no Capítulo 7.

A.1 Questões Enviadas aos Autores *Heroes*

1. What motivated your contributions to the project?
   a) Studies / Research purpose
   b) Employee in a company responsible for the project
   c) I'm a freelancer
   d) I wanted to increase my expertise
   e) Other: ___

2. What reasons do you identify for making the main contributions to this project?
   a) I contributed to the project for a long time.
   b) I was the only contributor to the project for a given period.
   c) I had great knowledge of the business rules of the project.
   d) I had great knowledge of the programming language and / or technologies used in the project.
   e) I received a high knowledge transfer from the most experienced developers of the project.
   f) Other: ___

3. What were the categories of contributions you made to the project?
   a) New features
   b) Bug fix
   c) Architectural Improvements / Refactor
   d) Changing or updating libraries / plugins

e) Business rules change
f) Other: ___

4. **Option 4.1:** Forwarded to the *heroes* authors who decreased their contributions in the last analyzed periods of time.

   4.1 We have seen a decrease in your contributions to the project over the last few years. What is the main reason for this?
   a) Role change in the team, but I have remained in the same project.
   b) I have been a developer, but I contribute to other projects as well.
   c) I no longer contribute to Open Source projects.
   d) Other: ___

   Option 4.2: Forwarded to the *heroes* authors who increased their contributions in the last analyzed periods of time .

   4.2 We have seen an increase in your contributions to the project over the last few years. What is the main reason for this?
   a) I became the main developer of the project after other developers left it.
   b) I became the only developer of the project.
   c) Due to the increase of my knowledge in the project and its architecture.
   d) I am dedicating more time to the project.
   e) Other: ___

5. How was the transfer of software knowledge made for you? (For example: information about existing business rules, architectural decisions made in the project, etc.)
   a) In person
   b) Internal Chat
   c) Whatsapp / Messenger
   d) Skype / Hangout
   e) PDF / DOC Documentation
   f) There was no transfer of knowledge
   g) Other: ___

6. How did you transfer software knowledge to the other project members?
   a) In person
   b) Internal Chat
   c) Whatsapp / Messenger

d) Skype / Hangout
e) PDF / DOC Documentation
f) There was no transfer
g) Other: ___

7. How do you evaluate your performance in this project? And the evolution of your knowledge of the system over the years?

8. Besides you, what other developers do you consider as fundamental contributors to the project's success? Why?

A.2 Questões Enviadas aos Autores Médios

1. What motivated your contributions to the project?
   a) Studies / Research purpose
   b) Employee in a company responsible for the project
   c) I'm a freelancer
   d) I wanted to increase my expertise
   e) Other: ___

2. Did you intend to become one of the main developers of the project?
   a) Yes, but I had no time availability
   b) Yes, but I didn't get help from the other developers
   c) Yes, but I didn't know the required language / technology knowledge
   d) Not
   e) I don't know
   f) Other: ___

3. What were the categories of contributions you made to the project?
   a) New features
   b) Bug fix
   c) Architectural Improvements / Refactor
   d) Changing or updating libraries / plugins
   e) Business rules change
   f) Other: ___

4. How was the transfer of software knowledge made for you? (For example: information about existing business rules, architectural decisions made in the project, etc.)

a) In person
b) Internal Chat
c) Whatsapp / Messenger
d) Skype / Hangout
e) PDF / DOC Documentation
f) There was no transfer of knowledge
g) Other: ___

5. How did you transfer software knowledge to the other project members?
   a) In person
   b) Internal Chat
   c) Whatsapp / Messenger
   d) Skype / Hangout
   e) PDF / DOC Documentation
   f) There was no transfer
   g) Other: ___

6. How do you evaluate your performance in this project? And the evolution of your knowledge of the system over the years?

7. Which developers do you think have mostly contributed to the success of the project? Why?

A.3 Questões Enviadas aos Autores Periféricos

1. What motivated your contributions to the project?
   a) Studies / Research purpose
   b) Employee in a company responsible for the project
   c) I'm a freelancer
   d) I wanted to increase my expertise
   e) Other: ___

2. What reason do you identify for not becoming a frequent developer on this project?
   a) I didn't know the required language / technology knowledge
   b) I had no time availability
   c) I had no interest
   d) I was interested, but I didn't get help from the other developers

e) I'm a frequent developer in ElasticSearch repository
f) Other: ___

3. What were the categories of contributions you made to the project?
   a) New features
   b) Bug fix
   c) Architectural Improvements / Refactor
   d) Changing or updating libraries / plugins
   e) Business rules change
   f) Other: ___

4. About the contributions made in this project, did you receive help from other contributors to carry out your own contribution?
   a) I received no help and there was no need.
   b) I received no help, but I needed it.
   c) Yes (Describe how you received help in the 'other' option)
   d) Other: ___

5. How do you evaluate your performance in this project? And the evolution of your knowledge in this project?