Universidade Federal de Ouro Preto
Instituto de Ciências Exatas e Aplicadas
Departamento de Computação e Sistemas

Um estudo empírico sobre *core developers* e arquitetura de projetos populares no GitHub

Carla Sanches Nere dos Santos

TRABALHO DE CONCLUSÃO DE CURSO

ORIENTAÇÃO:
Igor Muzetti Pereira

Dezembro, 2018
João Monlevade–MG

Carla Sanches Nere dos Santos

Um estudo empírico sobre *core developers* e arquitetura de projetos populares no GitHub

Orientador: Igor Muzetti Pereira

Monografia apresentada ao curso de Sistemas de Informação do Instituto de Ciências Exatas e Aplicadas, da Universidade Federal de Ouro Preto, como requisito parcial para aprovação na Disciplina "Trabalho de Conclusão de Curso II".

Universidade Federal de Ouro Preto

João Monlevade

Dezembro de 2018

S237e Santos, Carla Sanches Nere dos.
Um estudo empírico sobre core developers e arquitetura de projetos populares no GitHub [manuscrito] / Carla Sanches Nere dos Santos. - 2018.

41f.: il.: color; grafs; tabs.

Orientador: Prof. MSc. Igor Muzetti Pereira.

Monografia (Graduação). Universidade Federal de Ouro Preto. Instituto de Ciências Exatas e Aplicadas. Departamento de Computação e Sistemas de Informação.

1. Engenharia de software. 2. Arquitetura de software. 3. Software - desenvolvimento . 4. Sistemas de Informação. I. Pereira, Igor Muzetti. II. Universidade Federal de Ouro Preto. III. Titulo.

CDU: 004.41

Catalogação: ficha.sisbin@ufop.edu.br

FOLHA DE APROVAÇÃO DA BANCA EXAMINADORA

Um estudo empírico sobre *core developers* e arquitetura de projetos populares no GitHub

Carla Sanches Nere dos Santos

Monografia apresentada ao Instituto de Ciências Exatas e Aplicadas da Universidade Federal de Ouro Preto como requisito parcial da disciplina CSI499 – Trabalho de Conclusão de Curso II do curso de Bacharelado em Sistemas de Informação e aprovada pela Banca Examinadora abaixo assinada:

Igor Muzetti Pereira
MSc.
DECSI - UFOP

Vicente José Peixoto de Amorim
MSc.
Examinador
DECSI - UFOP

Alexandre Magno de Sousa
MSc.
Examinador
DECSI - UFOP

João Monlevade, 18 de dezembro de 2018

Este trabalho é dedicado à minha família, com muito amor.

Agradecimentos

Agradeço primeiramente a Deus, por sempre me proporcionar sabedoria e determinação para lidar com todos os desafios, e por ter colocado pessoas tão importantes em meu caminho.

Aos meus pais e irmãos, que me apoiaram e incentivaram a buscar o melhor em todos os momentos.

Aos amigos e namorado, que me acompanharam neste trajeto até a graduação, e ajudaram a concluir mais uma etapa.

E aos meus professores, em especial ao Igor Muzetti e ao Alexandre Magno, pela orientação e aprendizado que me proporcionaram.

À todos, muito obrigada!

"Se queremos progredir, não devemos repetir a história, mas fazer uma história nova."

— Mahatma Gandhi (1869 – 1948).

Resumo

A prática da propriedade coletiva de código é muito comum no desenvolvimento *Open Source Software* (OSS). Como todos os membros de uma equipe de desenvolvedores podem modificar qualquer parte do código, o senso de propriedade deve ser trabalhado na equipe, a fim de aumentar a contribuição de novos desenvolvedores e daqueles menos ativos. Com uma equipe melhor estruturada e mais colaborativa, a qualidade do software é maior, pois, pode-se dizer que existe uma relação direta entre a estrutura dos módulos de um sistema e a estrutura da equipe. O objetivo principal deste trabalho é analisar projetos OSS com a proposta de avaliar a propriedade coletiva de código, no que diz respeito a modularidade de software do ponto de vista de *core developers* no contexto de desenvolvedores OSS. Para isso, é proposta uma visualização do organograma de sistemas em matrizes que contém, para cada módulo do sistema, a porcentagem de contribuição de seus principais desenvolvedores. Como resultado, em geral, foram encontrados poucos responsáveis por módulos, mas cada módulo possui ao menos um responsável. Também foram detectados desenvolvedores que possuíam grandes quantidades de módulos sob sua responsabilidade, o que geralmente indica uma necessidade de reestruturação da equipe. Utilizando as matrizes, equipes de desenvolvimento podem ter uma visão mais clara sobre a estrutura de seu sistema. A mesma metodologia pode ser aplicada em qualquer repositório Git, e então gerentes de equipes de desenvolvimento podem ter auxílio à tomada de decisão com base nas contribuições dos desenvolvedores da sua equipe.

Palavras-chave: arquitetura de software, propriedade coletiva de código, desenvolvimento OSS.

Abstract

The practice of collective code ownership is very common in *Open Source Software* (OSS) development. Because all members of a team of developers can modify any part of the code, the sense of ownership must be worked on in the team in order to increase the contribution of new developers and those less active. With a better structured team and more collaborative, the quality of the software is higher, because, it can be said that there is a direct relationship between the structure of the modules of a system and the structure of the team. The main objective of this work is to analyze OSS projects with the proposal to evaluate collective code ownership, regarding software modularity from the point of view of core developers in the context of OSS developers. For this, we propose a visualization of the organization of systems in matrices that contain, for each module of the system, the contribution percentage of its main developers. As a result, in general, few developers responsible for modules were found, but each module has at least one responsible. We also detected developers who had large amounts of modules under their responsibility, which generally indicates a need for team restructuring. Using the matrices, development teams can have a clearer view of the structure of their system. The same methodology can be applied to any Git repository, and then development team managers can assist with decision-making based on contributions from their team's developers.

Key-words: software architecture, collective code ownership, OSS development.

Lista de ilustrações

Figura 1 – Número de desenvolvedores de projetos *open source* a partir de 2014 no GitHub. 18
Figura 2 – Diagrama de Classe representando a estrutura de um *commit*. 27
Figura 3 – Árvore de decisão para determinação de principais módulos. 28
Figura 4 – Árvore de decisão para determinação de principais desenvolvedores. . . 30
Figura 5 – Porcentagem de desenvolvedores responsáveis por cada projeto. 32
Figura 6 – Porcentagem de módulos sem responsáveis de cada projeto. 33

Lista de tabelas

Tabela 1 – Resultado final para o projeto Mockito. 24
Tabela 2 – Seleção de repositórios do GitHub para análise. 25
Tabela 3 – Dicionário de dados coletados pelo *framework* RepoDriller. 26
Tabela 4 – Resultados da análise da propriedade de código dos sistemas selecionados. 31

Lista de Algoritmos

1 Gerar *Ownership Matrice* 27

Lista de abreviaturas e siglas

CSV *Comma-separated values*

DOA *Degree-of-Authorship*

FA *First authorship*

DL *Deliveries*

AC *Acceptances*

OSS *Open Source Software*

XP *eXtreme Programming*

TF *Truck Factor*

DOK *Degree-of-Knowledge*

DOI *Degree-of-Interest*

EUA Estados Unidos da América

VCS *Version Control System*

Sumário

1	**INTRODUÇÃO**	**14**
1.1	**O problema de pesquisa**	**14**
1.2	**Objetivos**	**15**
1.3	**Principais resultados e contribuições**	**16**
1.4	**Estrutura do trabalho**	**16**
2	**REVISÃO BIBLIOGRÁFICA**	**17**
2.1	**Conceitos básicos**	**17**
2.1.1	Desenvolvimento *Open Source Software* (OSS)	17
2.1.2	GitHub	18
2.1.3	RepoDriller	19
2.2	**Revisão teórica**	**20**
2.3	**Trabalhos correlatos**	**22**
3	**DESENVOLVIMENTO**	**24**
3.1	**Primeira Etapa: Seleção de softwares**	**24**
3.2	**Segunda Etapa: Coleta de dados de softwares selecionados com o RepoDriller**	**25**
3.3	**Terceira Etapa: Desenvolvimento da ferramenta para *Ownership Matrices***	**26**
3.3.1	Determinação de principais módulos	28
3.3.2	Cálculo da propriedade de código	29
3.3.3	Determinação de principais desenvolvedores	29
4	**RESULTADOS**	**31**
4.1	**Análise e discussão**	**31**
4.2	**Ameaças à validade**	**34**
5	**CONCLUSÃO**	**36**
	REFERÊNCIAS	**37**

1 Introdução

A propriedade coletiva de código é uma prática de desenvolvimento de software comum em sistemas *open source*, em que todos os membros de uma equipe de desenvolvedores podem modificar qualquer parte do código. Ao longo da construção do código, cria-se uma rede de contribuição (PINZGER; NAGAPPAN; MURPHY, 2008), uma associação entre desenvolvedores e os módulos do sistema. De acordo com (CONWAY, 1968), essa associação pode ser de um para um (uma equipe projeta um módulo do sistema) ou de um para vários (uma equipe projeta vários módulos do sistema). Deste modo, por desenvolvedores e módulos estarem diretamente relacionados, é preciso que a propriedade coletiva de código seja aplicada corretamente, trabalhando o senso de propriedade de código da equipe. A propriedade de código pode ser definida como a relação entre o número de responsáveis por um artefato de um sistema, que pode ser um módulo ou arquivo, e a proporção relativa de suas contribuições (GREILER; HERZIG; CZERWONKA, 2015).

1.1 O problema de pesquisa

No cenário ideal da propriedade coletiva de código, projetos *Open Source Software* (OSS) criam equipes "espontâneas" à medida que o código evolui com a colaboração de seus desenvolvedores (PANICHELLA et al., 2014). Porém, novos contribuidores, ou aqueles menos ativos, podem ficar receosos em alterar partes do código que não foram bem entendidas (SEDANO; RALPH; PÉRAIRE, 2016). Com pouca colaboração, a qualidade de software pode ser reduzida e mais suscetível a erros (GREILER; HERZIG; CZERWONKA, 2015). A modularização é outro fator relevante para a qualidade do sistema. É definida por (PARNAS, 1972) como a divisão de responsabilidades de um sistema em módulos, geralmente programados por diferentes equipes. Como há uma associação direta entre a estrutura da equipe e os módulos do sistema, uma equipe mal estruturada torna o sistema mais propenso a falhas. Grandes equipes, por exemplo, podem sofrer uma desintegração de sua estrutura de comunicação (CONWAY, 1968). Então, conforme o estudo de (GREILER; HERZIG; CZERWONKA, 2015), a falta de um proprietário bem definido faz com que se perca a responsabilidade de manter e testar o código, fazendo com que funcionalidades possam ser perdidas. Apesar de suas desvantagens, a modularização também oferece benefícios no gerenciamento e na compreensibilidade do desenvolvimento de um sistema (PARNAS, 1972) e faz parte do projeto de arquitetura de software. A arquitetura de software lida com o *design* e a implementação da estrutura de alto nível do sistema (KRUCHTEN, 1995). No processo de *design* incluem-se a definição dos limites do sistema, a elaboração de seu escopo, a definição de atividades de *design* e delegação de tarefas, a coordenação das

tarefas definidas e a consolidação de *subdesigns* em um único *design* (CONWAY, 1968). O objetivo da arquitetura de software é "satisfazer os principais requisitos de funcionalidade e desempenho do sistema, bem como outros requisitos não funcionais" (KRUCHTEN, 1995). Portanto, o estudo da arquitetura de *open source softwares* (OSS) é importante para a criação de sistemas mais confiáveis e de melhor qualidade.

Como motivação para a pesquisa, a seguinte pergunta pode ser realizada: "Como o organograma de um software *open source* pode ser gerenciado?". Os objetos de pesquisa deste trabalho são repositórios de softwares *open source* populares no GitHub, escolhidos com base no número de estrelas[1], número de desenvolvedores e na linguagem de programação utilizada. Esses critérios possibilitam a obtenção de uma amostra de softwares bem estruturados por desenvolvedores experientes. A maioria deles vêm sendo desenvolvida ao longo dos últimos dez anos, fornecendo assim uma extensa base de dados para análise. A escolha de uma linguagem popular proporciona uma gama maior de sistemas para selecionar. Ao final do processo de desenvolvimento deste trabalho, foi obtida a relação entre os principais desenvolvedores e módulos dos sistemas selecionados, tornando possível avaliar que, no geral, o número de desenvolvedores que possuem conhecimento do código foi grande e suficiente para que os sistemas não passem por muitos problemas com a saída de um *core developer* da equipe.

1.2 Objetivos

O objetivo principal desse trabalho é analisar projetos OSS com a proposta de avaliar a propriedade coletiva de código, no que diz respeito a modularidade de software. Essa análise é realizada do ponto de vista de *core developers* no contexto de desenvolvedores OSS.

Como objetivos específicos deste trabalho, as seguintes perguntas são respondidas:

1. Quantos contribuidores tem cada sistema?
2. Quantos são responsáveis por módulos?
3. Quantos responsáveis tem cada módulo?
4. Quantos módulos possuem cada responsável?
5. Existem módulos com muitos responsáveis? Algum motivo aparente para isso?

[1] No GitHub, o número de estrelas de um repositório indica a sua popularidade.

1.3 Principais resultados e contribuições

A partir do histórico de *commits* de projetos populares no GitHub, foi possível obter matrizes que apresentam, para cada módulo do sistema, a porcentagem de contribuição de seus principais desenvolvedores. Essas matrizes, chamadas de *Ownership Matrices*, foram geradas a partir de uma ferramenta desenvolvida para auxiliar a pesquisa. Um trabalho semelhante para estimar o *Truck Factor* (TF) de 133 repositórios do GitHub foi realizado por (AVELINO et al., 2016). O TF avalia o quanto um programa está preparado para a rotatividade de membros da equipe de desenvolvimento. No contexto desse trabalho, os *core developers*, ou responsáveis pelo projeto, são equivalentes. Além dos valores obtidos, o autor também indica quais são os desenvolvedores responsáveis por cada sistema em geral.

Este trabalho contribui de forma a possibilitar que equipes tenham uma visão mais clara sobre a estrutura de seu sistema. A mesma metodologia pode ser aplicada em qualquer repositório Git e então gerentes de equipes de desenvolvimento podem ter auxílio à tomada de decisão com base nas contribuições dos desenvolvedores da sua equipe. A forma como foi apresentada a visualização do organograma dos sistemas, torna mais fácil focar somente nos resultados de módulos e desenvolvedores mais importantes, sem necessariamente ter grandes quantidades de dados para analisar.

1.4 Estrutura do trabalho

A estrutura deste trabalho é dividida em cinco capítulos: Introdução, Revisão bibliográfica, Desenvolvimento, Resultados e Conclusão. Os próximos capítulos são descritos a seguir:

- Capítulo 2, *Revisão bibliográfica*: apresenta os conceitos fundamentais para o entendimento do trabalho, bem como os trabalhos relacionados à pesquisa.
- Capítulo 3, *Desenvolvimento*: contém a metodologia realizada para obter a *Ownership Matrices*.
- Capítulo 4, *Resultados*: apresenta os resultados obtidos e responde às perguntas propostas a partir da análise dos dados.
- Capítulo 5, *Conclusão*: conclui o trabalho com uma discussão sobre suas limitações, as contribuições realizadas e apresentando sugestões para trabalhos futuros.

2 Revisão bibliográfica

Há vários aspectos no desenvolvimento de um sistema que podem ser analisados a partir do estudo da propriedade coletiva de código. Apesar do enfoque deste trabalho estar na distribuição da propriedade de código entre os desenvolvedores de um sistema, a revisão da literatura nas diversas abordagens auxilia em um melhor estabelecimento do problema. Alguns conceitos básicos são importantes para um melhor entendimento do trabalho.

2.1 Conceitos básicos

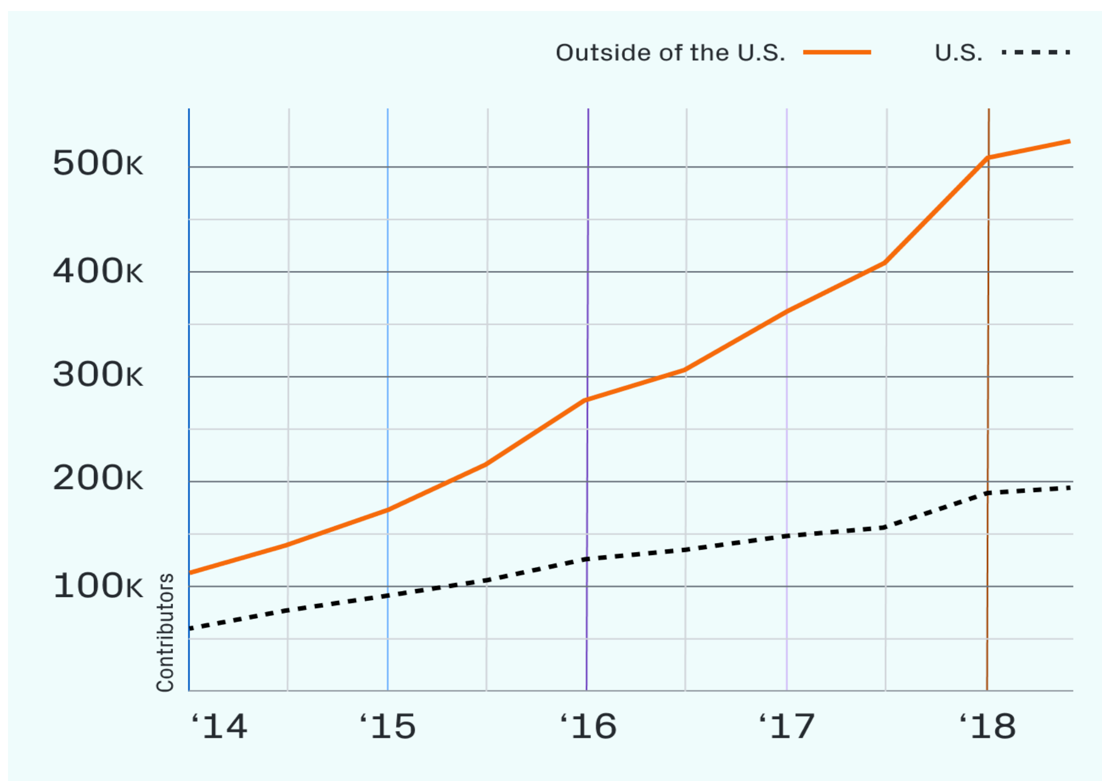
Os conceitos básicos incluem descrições sobre termos e tecnologias utilizadas neste trabalho. Incluem-se termos do desenvolvimento OSS, GitHub, RepoDriller, entre outros.

2.1.1 Desenvolvimento *Open Source Software* (OSS)

Projetos de desenvolvimento Open Source Software, ou software de código aberto, segundo (HIPPEL; KROGH, 2003), "são comunidades baseadas na Internet de desenvolvedores de software que colaboram voluntariamente para desenvolver software que eles ou suas organizações precisam". Softwares comerciais têm seu código-fonte restrito a funcionários e contratados, geralmente visando a sua venda. A ideia do desenvolvimento OSS é justamente contrária a dos softwares comerciais. O acesso ao código-fonte é gratuito, para que qualquer pessoa com habilidades de programação suficientes possa usar e modificar qualquer parte do código. O conceito inicial de software livre foi dado em 1985 por Richard Stallman, com a criação da Free Software Foundation. Porém, somente se popularizou em 1998, quando foi criado o movimento de software *open source* por Bruce Perens e Eric Raymond. Apesar das diferenças filosóficas entre o software livre e o *open source* software, de acordo com (HIPPEL; KROGH, 2003), o movimento OSS "incorpora essencialmente as mesmas práticas de licenciamento que as pioneiras do movimento de software livre".

Ao longo dos anos, o número de contribuidores em projetos *open source* vem crescendo. Somente o GitHub já conta com a colaboração de cerca de 700 mil desenvolvedores em repositórios *open source* de diversos países. A Figura 1 mostra que desde 2014 este número vem aumentando substancialmente. A linha pontilhada do gráfico refere-se aos desenvolvedores dos Estados Unidos da América (EUA) e a outra linha refere-se aos desenvolvedores do restante dos países.

Figura 1 – Número de desenvolvedores de projetos *open source* a partir de 2014 no GitHub.



Fonte: Blog do GitHub.[1]

2.1.2 GitHub

Segundo a definição do próprio site, o GitHub[2] é uma plataforma de desenvolvimento inspirada na forma como o usuário trabalha. Seja no desenvolvimento OSS ou para negócios, é possível hospedar e revisar código e gerenciar projetos. A empresa foi fundada em 2008 por Chris Wanstrath, PJ Hyett, Tom Preston-Werner e Scott Chacon, sediada em San Francisco, EUA. Hoje o GitHub conta com mais de 800 funcionários, 31 milhões de usuários registrados e 100 milhões de repositórios na plataforma. Este trabalho foi realizado com base em projetos do GitHub, cujos termos relacionados mais importantes para o seu entendimento são detalhados a seguir:

- **Sistema de Controle de Versão:** Do inglês *Version Control System* (VCS), conforme (CHACON; STRAUB, 2014), trata-se de um "sistema que registra alterações em um arquivo ou conjunto de arquivos ao longo do tempo para que se possa recuperar versões específicas posteriormente". O foco deste trabalho está na análise de registro alterações do código-fonte realizado pelo GitHub através dos *commits* do Git.

[1] Disponível em <https://blog.github.com/2018-11-08-100M-repos/> Acesso em dez 2018.
[2] <https://github.com/>

- ***Commit*:** De acordo com a documentação do Git (GIT, 2018), o *commit* consiste de um comando que registra as mudanças realizadas no repositório.

- **Git:** Um eficiente VCS que salva o estado do projeto no momento que o desenvolvedor realiza um *commit*, atualizando a referência somente para os arquivos modificados. O estado do projeto bem como os arquivos que o compõem são salvos no repositório. O Git[3] foi criado em 2005 pela comunidade de desenvolvimento do Linux, com o objetivo de ser um sistema de maior velocidade, possuir *design* simples, suporte robusto a desenvolvimento não linear, ser totalmente distribuído e capaz de lidar eficientemente com grandes projetos como o kernel do Linux (CHACON; STRAUB, 2014).

- ***Pull request*:** No GitHub é possível realizar *pull requests* para informar outros colaboradores de um projeto sobre as alterações realizadas em um repositório. Conforme a documentação (GITHUB, 2018), essa função permite que as alterações sejam discutidas entre desenvolvedores de um projeto antes de serem adicionadas ao código principal.

- ***Branching*:** Neste trabalho, somente o *branch* principal dos repositórios foi analisado. O *branching* é um recurso que permite o desenvolvimento não linear de projetos. Isso quer dizer que um mesmo projeto pode ter diferentes versões ao mesmo tempo, onde cada *branch* (ou ramificação) representa um estado do projeto, construído pelos *commits*. Segundo (CHACON; STRAUB, 2014), o *branch* inicial, denominado pelo Git por padrão como “master”, é como um *branch* qualquer. Porém, é a partir dele que são criadas novas ramificações com novas funcionalidades do projeto, que após serem testadas são mescladas com o *branch* principal para compor uma nova versão do projeto.

2.1.3 RepoDriller

Segundo (ANICHE, 2018), RepoDriller[4] é um *framework* Java *open source* para mineração de repositórios Git. Com ele é possível extrair dados de *commits*, desenvolvedores, modificações, *diffs* e código-fonte. Com a combinação desses dados, é possível obter diversas informações sobre os repositórios. Dentre as suas principais funcionalidades destacam-se:

- Análise de repositórios únicos ou múltiplos;
- Análise somente do *branch* principal;
- Análise de repositórios remotos ou locais;

[3] <https://git-scm.com/>
[4] <https://github.com/mauricioaniche/repodriller>

- Filtragem de *commits* por data, *hash*, *branch* e quantidade de arquivos;
- Utilização de *threads* para acelerar o processo de análise.

O RepoDriller pode ser configurado para extrair dados de *commits* e modificações de arquivos, como nome do autor do *commit*, o *hash* (ou identificador), data, nome do arquivo modificado, mensagem do *commit* e tipo de modificação. Como o código do programa é aberto, outras funcionalidades podem ser implementadas para que o usuário possa adaptar o *framework* à sua necessidade. Mais detalhes de configuração do RepoDriller utilizadas neste trabalho se encontram no Capítulo 3, Seção 3.2.

2.2 Revisão teórica

A propriedade de código compreende estudos em diversas áreas, tais como qualidade de software, gestão do conhecimento e comportamento organizacional. No trabalho de (GREILER; HERZIG; CZERWONKA, 2015), os autores mostram, a partir da investigação de projetos da Microsoft, que a propriedade de código pode se relacionar diretamente com a qualidade do software. A qualidade foi medida pela quantidade de *bugs* corrigidos em um artefato de código. Quanto maior o número de *bugs* encontrados e adicionados para a correção nos softwares analisados, menor a qualidade. Foi identificado que quanto menor a porcentagem de mudanças de colaboradores menores[5] num código, maior a probabilidade de limitação de seu conhecimento. Portanto, maior é a chance de introduzirem erros no sistema. Nesse caso, ocorre o que é chamado pelos autores de falta de propriedade. É uma situação ocasionada pela falta de comunicação entre a equipe, muito comum em softwares OSS e confirma o que diz (SEDANO; RALPH; PÉRAIRE, 2016) sobre "ter o direito de alterar um código, não significa que o desenvolvedor irá se sentir apto para fazer alguma alteração". O ideal é que a equipe esteja engajada para que haja a propriedade colaborativa. Essa propriedade consiste em manter as responsabilidades entre os desenvolvedores bem definidas.

Encontrar a melhor forma de trabalho em equipe é o maior desafio de um gerente de projetos, principalmente no desenvolvimento *open source*, em que há alta rotatividade de desenvolvedores. Segundo (HILTON; BEGEL, 2018), em empresas o custo da má administração da rotatividade pode ser caro. Estima-se o valor de 200% do salário anual de um empregado, somado à perda de conhecimento tácito. Diversos fatores podem levar um desenvolvedor a sair da equipe. Apesar do estudo de (HILTON; BEGEL, 2018) ser voltado para equipes de empresas de desenvolvimento de software, alguns deles também se aplicam ao ambiente cooperativo: não gostar de alguma tecnologia utilizada, não se

[5] Considera-se contribuidores menores aqueles que realizaram menos do que um limite predefinido de mudanças no software.

sentir parte da equipe, almejar novos desafios ou trabalhar com outras pessoas. Para uma boa administração da rotatividade, os autores atentam pela importância de estimular uma boa relação entre os membros da equipe, mentorear novatos para permitir que cresçam rapidamente e compartilhar a propriedade de código, evitando assim, a perda de conhecimento tácito. Uma forma eficiente de compartilhamento da propriedade de código e motivação da equipe observada por (FERREIRA; VALENTE; FERREIRA, 2017) possível de ser utilizada no ambiente OSS, é a utilização de mensagens em *pull requests*[6]. Dessa forma, o histórico de modificação vem acompanhado de sua justificativa, facilitando o entendimento por membros mais novos da equipe.

Ainda no âmbito comportamental, o senso de propriedade é um fator abordado por (SEDANO; RALPH; PÉRAIRE, 2016) que pode influenciar no contexto do sistema, contribuição, qualidade, ajuste do produto em relação às necessidades dos usuários e coesão da equipe. O contexto do sistema é definido pelo conhecimento de todo o código, incluindo o seu propósito e as tecnologias utilizadas. A coesão da equipe é “o grau em que os seus membros se identificam como parte da equipe” (SEDANO; RALPH; PÉRAIRE, 2016). Ameaças à propriedade coletiva podem vir diretamente da cultura de desenvolvimento do sistema. Isto é, quando os membros de uma equipe entendem bem o código e têm uma boa relação, a propriedade coletiva do código é positiva. Mas até mesmo a utilização de métodos ágeis podem se tornar uma ameaça. O estudo utiliza como exemplo o método *eXtreme Programming* (XP). O XP é um método de desenvolvimento ágil de software em que, dentre outras características, “os programadores trabalham em pares e desenvolvem testes para cada tarefa antes de escreverem o código. Quando o novo código é integrado ao sistema, todos os testes devem ser executados com sucesso” (SOMMERVILLE, 2011). Neste método, pode ocorrer de um parceiro somente assistir o outro programar sem efetivamente participar do desenvolvimento. As medidas propostas por (HILTON; BEGEL, 2018) e (FERREIRA; VALENTE; FERREIRA, 2017) são bons exemplos de como evitar essa situação.

Existem várias medidas que podem ser utilizadas visando a melhoria da qualidade do software, de sua estruturação ou da dinâmica da equipe de desenvolvimento. Porém, conforme (Brooks, F.P., 1987), não há uma bala de prata que resolva os problemas no desenvolvimento de software. Ou seja, uma só metodologia ou tecnologia que irá garantir produtividade, confiabilidade e simplicidade. A propriedade coletiva de código é abordada nesse trabalho de forma a demonstrar a sua distribuição entre os desenvolvedores, indicando quem são os principais e em quais módulos eles atuam. A utilização dessa forma de visualização do sistema por gerentes de projeto pode ter diversas finalidades e ser associada aos métodos de desenvolvimento de software mais adequados para cada sistema.

[6] Recurso de sistemas de controle de versão que usam o Git. Utilizado para enviar um código modificado para revisão antes de ser adicionado à versão principal do projeto.

2.3 Trabalhos correlatos

Um trabalho semelhante foi realizado por (AVELINO et al., 2016) para estimar o *Truck Factor* (TF) de 133 repositórios do GitHub. O TF é uma analogia ao número de pessoas em uma equipe que devem ser atropeladas por um caminhão antes do sistema enfrentar problemas. Ou seja, o quanto um programa está preparado para a rotatividade de membros da equipe de desenvolvimento. Quanto menor o número, mais alta a dependência de contribuidores específicos. O cálculo foi realizado com base no histórico de *commits*, utilizando um modelo denominado *Degree-of-Authorship* (DOA). O DOA foi proposto por (FRITZ et al., 2010) como um método para identificar o conhecimento de desenvolvedores acerca de certas partes do código, e, consequentemente, a propriedade de código.

Também em uma abordagem baseada em *commits*, o estudo de (GREILER; HERZIG; CZERWONKA, 2015) tem o objetivo de confirmar se a propriedade de código influencia diretamente em sua qualidade. Foi realizada uma investigação nos produtos da Microsoft: Office, Office365, Windows e Exchange. A análise de dados foi realizada a nível de diretórios de código (agrupamento de arquivos que contém propriedades em comum). Eles oferecem um nível de granularidade intermediário, proporcionando uma melhor análise de dados em relação a erros. Segundo (GREILER; HERZIG; CZERWONKA, 2015), uma granularidade muito alta traz um número de erros muito grande. Da mesma forma, uma granularidade pequena faria com que o número de erros detectados fosse muito baixo, distorcendo os dados. Os contribuidores foram classificados em contribuidores maiores e menores. Foi considerado um contribuidor maior, aquele que realizou no mínimo 50% das mudanças em um código. Aqueles que realizaram menos de 50% das mudanças foram considerados contribuidores menores.

O modelo *Degree-of-Knowledge* (DOK) é proposto por (FRITZ et al., 2010) para calcular o grau de conhecimento dos desenvolvedores de uma equipe a respeito do código para o qual eles contribuem. O DOK consiste na combinação dos fatores de DOA e *Degree-of-Interest* (DOI). Conforme (KERSTEN; MURPHY, 2006) citado por (FRITZ et al., 2010), "O DOI representa a quantidade de interação – seleções e edições – que um desenvolvedor teve com um elemento de código-fonte". Uma seleção ocorre quando um desenvolvedor abre um arquivo para ser editado. Já a edição ocorre quando há a alteração do arquivo. Um dos estudos de caso do artigo tem o intuito de encontrar especialistas em partes específicas do código. O DOK foi calculado para classes e pacotes de dois projetos, que consistiam de um grupo lógico de pacotes Java. Foram analisados três meses de modificações. Sete desenvolvedores mais experientes participaram da confirmação dos resultados, e, para 55% dos módulos validados, o método foi eficaz. Por conta do histórico de modificações ser de apenas três meses, não foi possível identificar especialistas para alguns módulos.

Em comparação com os trabalhos correlatos, neste trabalho foi utilizado o modelo

DOA para cálculo de propriedade de código, assim como em (AVELINO et al., 2016). O modelo proporciona uma boa estimativa da propriedade de código para diferentes sistemas (FERREIRA; VALENTE; FERREIRA, 2017), além de respeitar o fluxo de experiência do desenvolvedor. Isso acontece porque o grau de conhecimento de um desenvolvedor sobre um determinado arquivo aumenta quando este confirma as alterações no repositório de origem e diminui quando outros desenvolvedores fazem alterações (FRITZ et al., 2010). Diferente do estudo de (AVELINO et al., 2016), este trabalho abrange um número maior de desenvolvedores como parte dos principais contribuidores ao analisar módulos mais específicos dos sistemas. Além disso, também investiga quais são os módulos em que os desenvolvedores contribuem e as suas porcentagens de contribuição. De forma semelhante a (GREILER; HERZIG; CZERWONKA, 2015), este trabalho realiza a análise de diretórios, porém o nível de granularidade é definido pelo número de desenvolvedores que os alteram. A classificação dos contribuidores foi realizada de maneira diferente, utilizando como base o modelo de DOA. Em comparação com a classificação utilizada por (GREILER; HERZIG; CZERWONKA, 2015), o DOA foi elaborado para selecionar os proprietários de módulos de um sistema. Sendo assim, esta se mostrou uma estratégia mais adequada para selecionar os *core developers*. Apesar do DOK ser um método mais completo, diferente do estudo de (FRITZ et al., 2010), no contexto deste trabalho não é possível identificar quando um desenvolvedor acessou um arquivo. Portanto, foi utilizado somente o método DOA. Com o intuito de encontrar desenvolvedores responsáveis para todos os principais módulos, também foi analisado todo o histórico de *commits*.

3 Desenvolvimento

O desenvolvimento deste trabalho pode ser dividido em três etapas principais: (1) seleção de softwares a serem analisados, (2) coleta de dados dos softwares selecionados com o auxílio do *framework* RepoDriller e (3) desenvolvimento de uma ferramenta para auxiliar no cálculo da propriedade de código de desenvolvedores. Ao final do processo, foram obtidas matrizes com as propriedades de código para cada projeto analisado, denominadas *Ownership Matrices*. A Tabela 1 apresenta um exemplo da metodologia aplicada ao projeto Mockito[1]. Os nomes dos principais módulos e desenvolvedores foram ocultados para melhor visualização dos dados. Nesse exemplo, o projeto possui sete *core developers*. A responsabilidade pelo projeto está em sua maior parte no desenvolvedor *Dev 1*. Os módulos de 5 a 8 possuem o maior número de responsáveis e a quantidade de módulos sob responsabilidade de cada desenvolvedor varia de 1 a 10. Mais detalhes sobre os resultados deste trabalho são discutidos no Capítulo 4.

Tabela 1 – Resultado final para o projeto Mockito.

Principais Módulos	Dev 1	Dev 2	Dev 3	Dev 4	Dev 5	Dev 6	Dev 7
	%	%	%	%	%	%	%
Módulo 1	83.68	9.02	0	0	0	0	0
Módulo 2	83.83	8.55	0	0	0	0	0
Módulo 3	84.29	8.25	0	0	0	0	0
Módulo 4	55.17	22.41	0	5.17	0	0	0
Módulo 5	40.08	5.26	0.40	0	0.20	25.30	8.10
Módulo 6	40.01	5.06	0.40	0	0.20	23.62	8.10
Módulo 7	40.01	5.06	0.40	0	0.20	23.62	8.10
Módulo 8	40.01	5.06	0.40	0	0.20	23.62	8.10
Módulo 9	41.67	4.01	0.62	0	0	21.30	10.19
Módulo 10	35.71	14.29	0	0	0	10.71	3.57

3.1 Primeira Etapa: Seleção de softwares

Doze projetos do GitHub foram selecionados de acordo com os seguintes critérios: linguagem de programação utilizada para o desenvolvimento, número de estrelas do repositório e número de desenvolvedores. Foram escolhidos projetos em Java, por se tratar de uma das linguagens mais populares no GitHub. Por convenção, apenas repositórios com no mínimo 1000 estrelas e 50 desenvolvedores entraram na análise. Para facilitar a pesquisa

[1] <https://github.com/mockito/mockito>

pelos repositórios, foi utilizada a ferramenta GitTrends[2]. A ferramenta desenvolvida por (AVELINO et al., 2016), foi criada com o propósito de apresentar o TF de softwares do GitHub. Porém, foi utilizada nesse projeto com o intuito de filtrar repositórios pelos critérios definidos. A Tabela 2 apresenta a seleção de projetos para análise.

Tabela 2 – Seleção de repositórios do GitHub para análise.

Nome do Projeto	Nº de Desenvolvedores	Nº de Estrelas
ElasticSearch	1107	36671
IntelliJ IDEA Community Edition	358	7004
Spring Framework	317	25275
Apache Cassandra	274	4832
Google Guava	167	28581
Pentaho Kettle	162	2204
Apache Hadoop	148	8322
Junit 4	139	7222
Mockito	131	7969
Eclipse Che	106	5182
Eclipse Openj9	103	1548
Apache Maven	68	1409

3.2 Segunda Etapa: Coleta de dados de softwares selecionados com o RepoDriller

O RepoDriller foi utilizado neste trabalho para gerar bases de dados que contém informações necessárias para a criação das tabelas. Essas bases de dados tratam-se de arquivos em *Comma-separated values* (CSV) exportados pelo *framework* ao fim da coleta de dados de cada repositório. A Tabela 3 mostra quais dados foram extraídos pelo RepoDriller. O *hash* do *commit* é um identificador único, utilizado para realizar a busca por modificações. As modificações podem ser dos seguintes tipos:

- ADD: O arquivo foi inserido no projeto.
- RENAME: O arquivo foi renomeado.
- DELETE: O arquivo foi apagado do projeto.
- MODIFY: O arquivo foi modificado.

Para tratar corretamente o histórico de alterações, foi necessário detectar o nome antigo dos arquivos e o nome atual. Estes se diferem caso o arquivo tenha sofrido uma

[2] <http://gittrends.io>

modificação do tipo RENAME. Os nomes e e-mails dos autores dos *commits* foram utilizados para a sua identificação. Um desenvolvedor é considerado autor do *commit* quando este realiza qualquer tipo de modificação em um arquivo do projeto. O termo *committer* é atribuído apenas para desenvolvedores responsáveis pela revisão de código do sistema, que também realizam *commits* para validar a alteração do autor.

Tabela 3 – Dicionário de dados coletados pelo *framework* RepoDriller.

Campo	**Definição**	**Exemplo**	**Notas**
hash	Número de identificação do *commit*	1166901264	Tamanho variável
modificacao	Tipo de modificação realizada	ADD	Valores aceitos: ADD, DELETE, MODIFY ou RENAME
nome_arquivo_atual	Nome do arquivo atualizado no repositório	src/org/junit/Ignore.java	Identificação do arquivo
nome_arquivo_antigo	Nome do arquivo antes de ser renomeado ou inserido no repositório.	org/junit/Ignore.java	
nome_autor	Nome do autor do *commit*.	Kent Beck	Identificação do desenvolvedor
email_autor	E-mail do autor do *commit*.	kent@junit4.com	
nome_committer	Nome do desenvolvedor que realizou *commit* do trabalho do autor.	David Saff	Identificação do desenvolvedor
email_committer	E-mail do desenvolvedor que realizou *commit* do trabalho do autor.	david@junit4.com	

A configuração do RepoDriller foi realizada da seguinte forma: filtros foram adicionados para selecionar apenas o *branch* principal, de forma a analisar *commits* realizados em uma área mais estável do sistema. Apenas arquivos com extensão ".java" passaram pela coleta de dados e diretórios contendo arquivos de teste foram excluídos, a fim de eliminar ao máximo possível arquivos gerados automaticamente. Para ignorar arquivos de teste foi necessário implementar um filtro adicional, devido ao *framework* não suportar essa funcionalidade.

3.3 Terceira Etapa: Desenvolvimento da ferramenta para *Ownership Matrices*

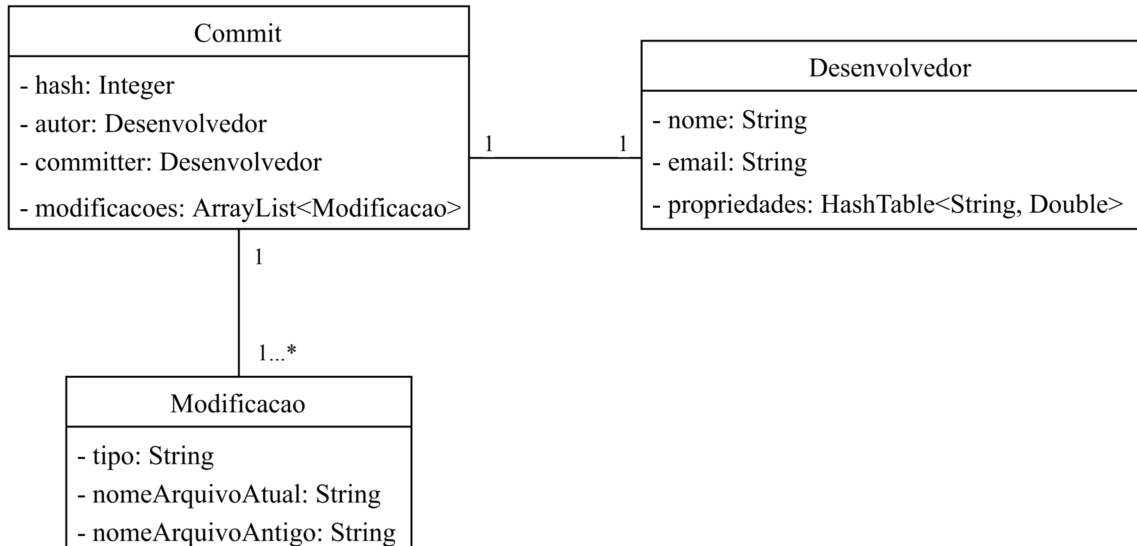
Como os dados obtidos dos *commits* envolveram modificações desde a criação de cada repositório, bases de dados muito extensas foram obtidas. Isso levou à necessidade de implementação de uma ferramenta para gerar as matrizes de propriedade de código, que está disponível no GitHub[3]. Para este trabalho, a ferramenta foi configurada de forma a receber como entrada as bases de dados em CSV geradas pelo RepoDriller. O programa é capaz de automaticamente detectar os principais módulos da base de dados, definir a propriedade de código de cada módulo para cada desenvolvedor e gerar uma matriz contendo a relação desses dados, bastando apenas informar o nome do arquivo de saída e o separador dos valores.

Cada linha de um arquivo da base de dados representa uma modificação. O programa foi implementado de forma a organizar as modificações em *commits*. A Figura 2 representa a estrutura da classe Commit. Um objeto desta classe contém um *hash*, um autor, um

[3] <https://github.com/carlasanches/ownership-matrice>

committer e uma lista de modificações. A estrutura em *commits* foi escolhida para facilitar a busca por modificações, pois, cada *commit* possui um *hash* único e pode ser contado como uma modificação em uma determinada classe de um sistema. O Algoritmo 1 é um resumo em pseudo-código da metodologia para gerar as matrizes. A implementação foi realizada na linguagem Java e a forma como foram desenvolvidas suas principais funcionalidades é descrita a seguir.

Figura 2 – Diagrama de Classe representando a estrutura de um *commit*.



Fonte: Elaborado por autor.

Algoritmo 1: Gerar *Ownership Matrice*

Entrada: Base de dados de *commits* C

Saída: *Ownership Matrice*

1 **início**

2 $renomear(C.arquivosAntigos)$

3 $modulos \leftarrow selecionarModulos(C)$

4 $desenvolvedores \leftarrow selecionarDesenvolvedores(C)$

5 $principaisModulos \leftarrow calcularCentralidade(modulos)$

6 $calcularDOA(desenvolvedores, modulos)$

7 $normalizarDOA()$

8

9 **para cada** $desenvolvedor$ ***em*** $desenvolvedores$ **faça**

10 **para cada** $modulo$ ***em*** $principaisModulos$ **faça**

11 **se** $desenvolvedor.propriedade(modulo) >= 0,05$ **então**

12 **se** $somaPropriedades(modulo) >= 0,5$ **então**

13 $principaisDesenvolvedores.adicionar(desenvolvedor)$

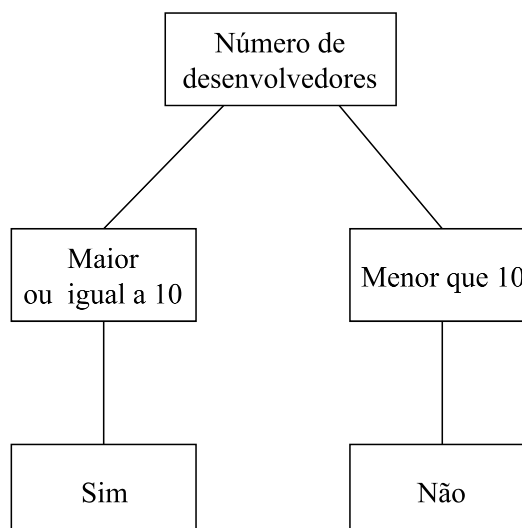
14 **retorna** $matriz(principaisModulos, principaisDesenvolvedores)$

3.3.1 Determinação de principais módulos

Após a estruturação em *commits*, todos os arquivos tiveram seu tipo de modificação avaliado. Caso fosse do tipo “RENAME”, era detectado o nome atual do arquivo e todos os que estiverem com o nome antigo são atualizados. Este passo foi importante para contabilizar corretamente o número de modificações em cada arquivo. A seguir, todos os arquivos existentes foram separados e selecionados para servir de base para a matriz final. Os principais módulos de um sistema foram determinados com base na centralidade de cada módulo, ou seja, o número de desenvolvedores que modificaram um módulo (PINZGER; NAGAPPAN; MURPHY, 2008). A heurística utilizada determina que, quanto mais desenvolvedores modificam um módulo, mais importante ele é considerado.

Os principais módulos são determinados a partir de um número máximo definido para a centralidade. A Figura 3 representa a árvore de decisão da classificação dos principais módulos do sistema. Após a realização de vários testes com as bases de dados, foi definido que se um módulo foi desenvolvido por 10 desenvolvedores ou mais, ele é considerado um dos principais módulos. Caso contrário, ele é desconsiderado. A classificação começa pelos diretórios mais altos da hierarquia do repositório. Definidos os principais diretórios dentre os de nível mais alto, o algoritmo analisa outros níveis abaixo destes, para encontrar outros módulos que atendam ao requisito de centralidade definido. Por exemplo, se existe um diretório de nível mais alto denominado *“src”* e foi detectado que este possui mais de 10 desenvolvedores, o algoritmo, então, calcula a centralidade para cada diretório contido dentro de *“src”*. Esse comportamento se repete recursivamente até encontrar um valor menor do que 10.

Figura 3 – Árvore de decisão para determinação de principais módulos.



Fonte: Elaborado por autor.

3.3.2 Cálculo da propriedade de código

Ainda não há um método definido como o mais correto para o cálculo da propriedade de código. O estudo de (FERREIRA; VALENTE; FERREIRA, 2017) compara três métodos, e o que apresenta os resultados mais confiáveis bem como se aproxima melhor da abordagem deste trabalho é o *Degree-of-Authorship* (DOA). Trata-se de um método criado por (FRITZ et al., 2010), baseado em *commits* e definido pela seguinte fórmula:

$DOA(m_d, f_p) = 3,293 + 1,098 * FA(m_d, f_p) + 0,164 * DL(m_d, f_p) - 0,321 * \ln(1 + AC(m_d, f_p))$.

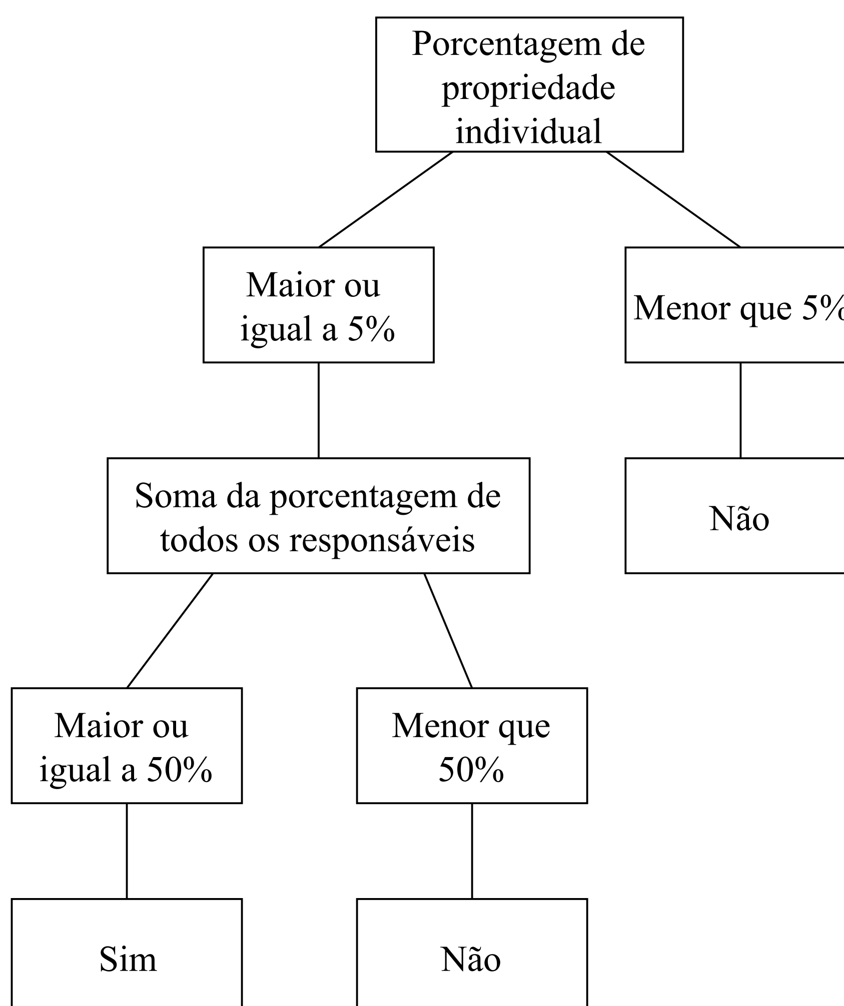
As entradas da função m_d e f_p são respectivamente o desenvolvedor e o caminho do arquivo que está sendo processado. *First authorship* (FA) indica se o desenvolvedor é autor ou não do arquivo, retornando 1 caso verdadeiro e 0 caso falso. *Deliveries* (DL) indica o número de modificações que o desenvolvedor realizou no arquivo e *Acceptances* (AC) é o número de modificações que outros desenvolvedores realizaram no arquivo. Basicamente, neste modelo a autoridade de um arquivo aumenta se um desenvolvedor for autor do arquivo e com a quantidade de modificações que ele realiza e diminui à medida em que outros desenvolvedores modificam o mesmo arquivo.

O cálculo do DOA foi realizado em todos os arquivos para cada desenvolvedor. Os resultados foram normalizados em valores entre 0 e 1 para a definição da propriedade de código para arquivos e módulos. Utilizando o DOA juntamente com os parâmetros definidos por (AVELINO et al., 2016), um desenvolvedor é dito proprietário de um arquivo se o valor do DOA normalizado é maior ou igual a 0,75 e o valor absoluto for maior ou igual a 3,293. O valor final mostrado na matriz é a propriedade dos principais módulos. Este é definido pela porcentagem de arquivos contidos dentro de um módulo os quais um desenvolvedor é proprietário.

3.3.3 Determinação de principais desenvolvedores

A determinação dos principais desenvolvedores foi realizada de forma manual, analisando o resultado do cálculo da propriedade de código. A partir das métricas de (BIRD et al., 2011) citadas em (GREILER; HERZIG; CZERWONKA, 2015) e de (AVELINO et al., 2016), foram definidas as métricas de propriedade individual de módulo e de propriedade de módulo. A propriedade individual é dada pela porcentagem de arquivos em um módulo que são propriedade de um desenvolvedor. Já a propriedade de módulo é aquela que aparece na matriz, dada pela porcentagem de arquivos em um módulo que são propriedade de um desenvolvedor em relação às modificações realizadas por outros desenvolvedores da equipe. A Figura 4 representa a árvore de decisão para classificação dos desenvolvedores. Um desenvolvedor é considerado proprietário de um módulo quando possui propriedade individual de 5% deste, e somados todos os proprietários, a porcentagem é de 50%.

Figura 4 – Árvore de decisão para determinação de principais desenvolvedores.



Fonte: Elaborado por autor.

4 Resultados

Após gerar as matrizes de cada sistema, foi realizada uma análise de seus resultados, visando responder às questões propostas. Todas as *Ownership Matrices* estão disponíveis no GitHub[1]. A Tabela 4 apresenta os resultados para cada sistema analisado. A coluna *Razão* indica a porcentagem de responsáveis de cada projeto, *RespMod (média)* é a média de responsáveis que cada módulo possui, e *ModResp (média)* é, em média, o número de módulos pertencentes a cada responsável.

Tabela 4 – Resultados da análise da propriedade de código dos sistemas selecionados.

Nome do Projeto	Responsáveis	Contribuidores	Razão (%)	Módulos	Módulos sem Responsáveis	RespMod (média)	ModResp (média)
Pentaho Kettle	134	162	83	1032	172	3	23
Apache Hadoop	94	148	64	425	252	3	14
ElasticSearch	83	1107	7	342	23	6	25
IntelliJ IDEA[2]	66	358	18	113	0	7	12
Eclipse Che	49	106	46	393	23	7	57
Apache Cassandra	45	274	16	82	9	5	9
Google Guava	25	167	15	93	10	5	17
Apache Maven	16	68	24	84	0	3	19
Spring Framework	16	317	5	220	0	2	35
Junit 4	13	139	9	19	12	4	6
Eclipse Openj9	11	103	11	26	10	4	8
Mockito	7	131	5	10	0	4	6

4.1 Análise e discussão

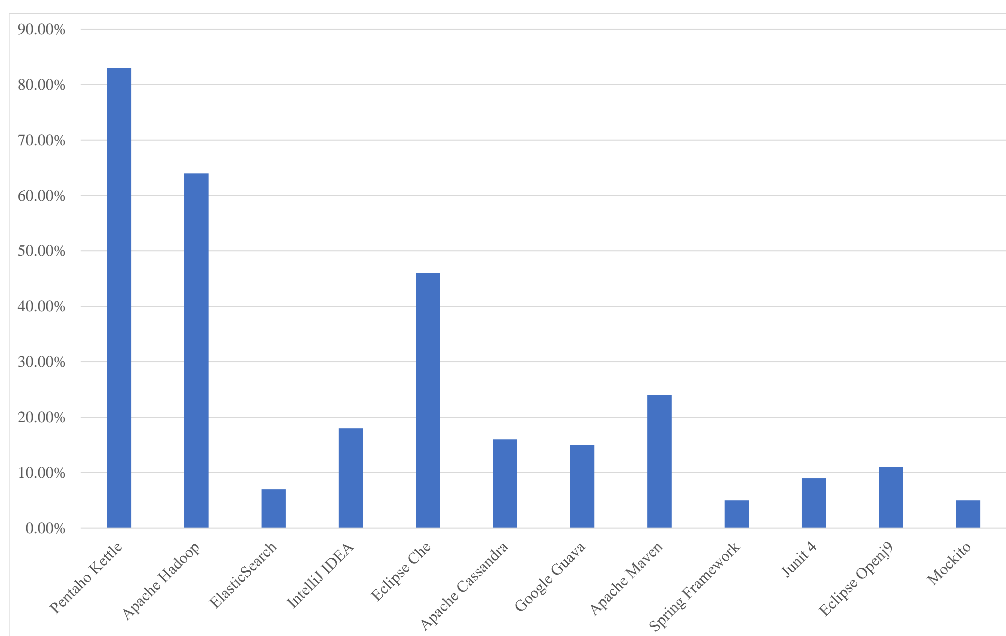
A partir da analise de dados da Tabela 4, é possível responder às perguntas propostas. Respondendo à primeira pergunta: *"Quantos contribuidores tem cada sistema?"*, o número varia entre 68 e 1107 de acordo com a coluna *Contribuidores*. Foram selecionados sistemas com no mínimo 50 contribuidores para que as análises pudessem ser realizadas a partir de um número significativo. Analisar mais contribuidores torna mais fácil encontrar os *core developers*, aqueles que são responsáveis pelo sistema. O número de contribuidores está diretamente ligado ao número de responsáveis. Em um grupo pequeno de desenvolvedores, a chance de todos serem responsáveis pelos módulos do sistema e de obter resultados pouco relevantes é maior. Encontrar uma porcentagem grande de responsáveis também pode indicar que o trabalho da equipe é bem distribuído, ou que há um grande engajamento dos desenvolvedores, como é o caso do Pentaho Kettle.

A última etapa da metodologia (em 3.3.3) apresentou o método utilizado para responder à segunda pergunta: *"Quantos são responsáveis por módulos?"*. A Figura 5 mostra que com o algoritmo utilizado, foi obtido um grande número de responsáveis por

[1] <https://github.com/carlasanches/ownership-matrice/tree/master/matrices>
[2] Community Edition

módulos. O número total de responsáveis encontra-se na coluna 2 da Tabela 4. Dois projetos contém o número de responsáveis acima de 50%. São eles o Pentaho Kettle e Apache Hadoop, com 83% e 64% dos desenvolvedores considerados responsáveis, respectivamente. O Eclipse Che fica próximo, com 46% dos desenvolvedores responsáveis. Isso indica que geralmente há um número menor de chances dos projetos passarem por desafios em sua reestruturação tanto da equipe, quanto do sistema, caso alguns de seus responsáveis deixem o desenvolvimento do programa. Uma provável justificativa dos resultados obtidos encontrarem-se mais altos do que em outros estudos, é a exploração de módulos realizada pelo algoritmo. No trabalho de (AVELINO et al., 2016), foram analisados somente os dois primeiros módulos do organograma de um projeto, enquanto neste trabalho não há um limite de profundidade estabelecido.

Figura 5 – Porcentagem de desenvolvedores responsáveis por cada projeto.

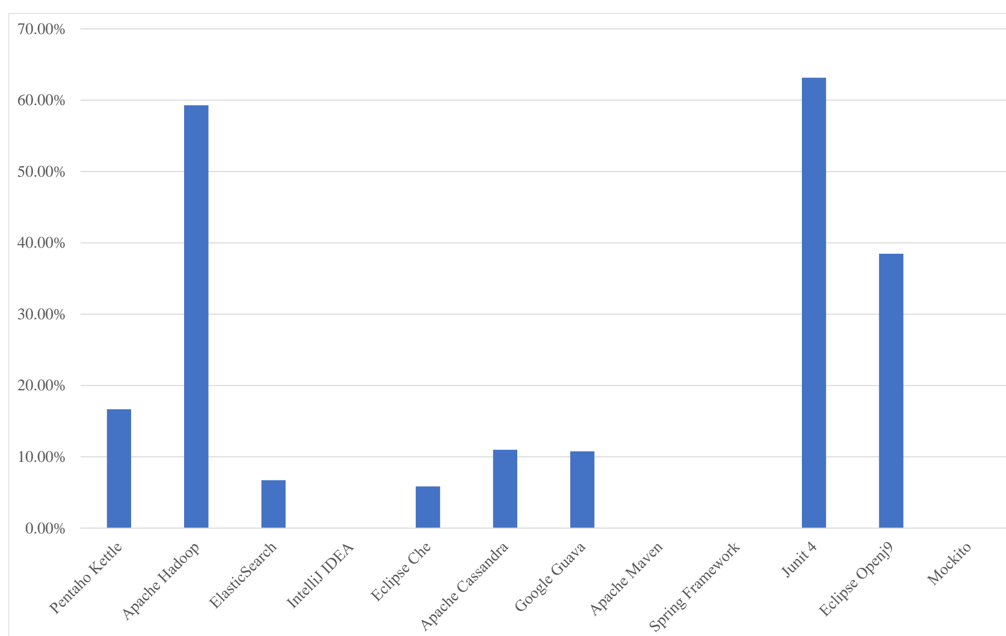


Fonte: Elaborado por autor.

A terceira pergunta proposta é *"Quantos responsáveis tem cada módulo?"*. De acordo com a coluna *RespMod (média)* da Tabela 4, em média, a maioria dos projetos possui 4 responsáveis por módulo. Possuir mais desenvolvedores responsáveis por módulos pode ser útil no caso de algum deles desistir do projeto. Porém, conforme (GREILER; HERZIG; CZERWONKA, 2015), a falta de um proprietário bem definido pode fazer com que as responsabilidades de manutenção do código sejam deixadas de lado, tornando-o mais sujeito a falhas. Os projetos que estariam mais propensos a esta situação são o Junit

4, com 12 dos 19 módulos (63%) sem responsáveis, o Apache Hadoop com 252 de 425 (59%) e o Eclipse Openj9, com 10 de 26 (38%) dos módulos sem responsabilidade definida. Poucos projetos possuem responsáveis para todos os módulos. São eles o IntelliJ IDEA, Apache Maven, Spring Framework e Mockito. A Figura 6 apresenta em porcentagem a razão de módulos que não possuem responsáveis pela quantidade de módulos principais de cada projeto.

Figura 6 – Porcentagem de módulos sem responsáveis de cada projeto.



Fonte: Elaborado por autor.

Respondendo a *"Quantos módulos possuem cada responsável?"*, a coluna *ModResp (média)* indica que as médias variaram entre 6 e 57 módulos por responsável. Como foi especificado na primeira etapa da metodologia de desenvolvimento da ferramenta (em 3.3.1), todos os módulos foram modificados por 10 ou mais desenvolvedores. Médias altas indicam desenvolvedores mais sobrecarregados. Este é o caso da maioria dos projetos analisados, destacando-se o Eclipse Che, com uma média de 57 módulos sob responsabilidade de um desenvolvedor. Os valores mais razoáveis, abaixo de dez módulos, referem-se aos projetos Apache Cassandra, Junit 4, Eclipse Openj9 e Mockito.

Por fim, a última questão diz: *"Existem módulos com muitos responsáveis? Algum motivo aparente para isso?"*. Considerando que os resultados foram obtidos a partir de dados desde o início de cada projeto, os métodos utilizados nesse trabalho não detectaram um

grande número de responsáveis por módulos. Projetos *open source* sofrem constantemente uma alta rotatividade de membros da equipe, e desenvolvedores que foram considerados responsáveis estão distribuídos ao longo de cerca de 10 anos na maioria dos projetos. No decorrer de seu desenvolvimento, equipes vão surgindo e se desfazendo, e então os contribuidores que continuam no projeto devem se reorganizar (PANICHELLA et al., 2014). Como contribuidores mais antigos possuem mais conhecimento sobre o código, estes desenvolvem mais e podem ter mais atribuições de responsabilidades. A quantidade de responsáveis pelos módulos também varia de acordo com a cultura de desenvolvimento da equipe e com a sua estruturação. Em equipes organizadas, a designação de atividades para os subgrupos da equipe fazem parte do projeto do sistema (CONWAY, 1968).

4.2 Ameaças à validade

Considerando que todas as informações obtidas foram através da coleta de dados e análise do histórico de *commits*, algumas limitações podem ser apontadas. A maioria dos softwares sofreu muitas mudanças em sua estrutura ao longo dos anos de desenvolvimento Uma quantidade elevada de *commits* antigos pode ocultar o trabalho atual do resultado final. Não existe uma data correta para iniciar a análise, mas o ideal seria que o algoritmo considerasse o trabalho de contribuidores mais antigos sem diminuir a propriedade de contribuidores mais recentes. Também devido aos *commits* mais antigos, desenvolvedores que já deixaram a equipe podem ser classificados pelo algoritmo como parte dos desenvolvedores principais. Outro ponto relevante sobre a contribuição é a atribuição de autoria a *committers*. Muitas vezes contribuidores realizam *commit* do trabalho de outros desenvolvedores. No contexto desse trabalho isso influencia fortemente a propriedade de código para certos desenvolvedores, sem que haja realmente a necessidade destes alterarem o código. Em um trabalho futuro, pode ser considerado retirar o peso do atributo "autoria" dos *committers*.

Em relação à escolha de principais módulos, é um desafio estimar, sem nenhum conhecimento sobre o código, classes e diretórios mais importantes. Nesse trabalho foi assumido que aqueles que possuem mais de 10 desenvolvedores são considerados importantes. Porém, para obter resultados mais precisos é interessante adaptar a centralidade dos módulos para cada projeto. Conhecendo a estrutura de desenvolvimento pode ser levado em consideração o número de desenvolvedores em cada equipe, por exemplo. Essa adaptação também é possível de ser realizada no cálculo da propriedade, definindo diferentes valores para os limites de contribuição, de acordo com o que já se sabe sobre a dedicação dos desenvolvedores da equipe.

Outra limitação ocorre devido às dificuldades em obter o conteúdo de todas as modificações realizadas nos arquivos de um projeto. Não há como saber detalhes como

por exemplo o número de linhas modificadas. A contribuição de cada desenvolvedor foi contabilizada de forma igual, de acordo com o número de *commits*. Logo, quanto mais *commits*, maior a porcentagem de contribuição da matriz, independente do impacto da modificação para o desenvolvimento do software.

5 Conclusão

Existem detalhes que dificultam a obtenção de resultados precisos para a pesquisa, como a falta de conhecimento detalhado sobre o desenvolvimento do projeto e da importância de cada modificação realizada por um desenvolvedor. Apesar dessas limitações, o objetivo foi atingido ao responder à pergunta "Como o organograma de um software *open source* pode ser gerenciado?". Gerentes de equipes de desenvolvimento podem utilizar as *Ownership Matrices* para auxílio à tomada de decisão, tanto da estrutura do sistema quanto da estrutura da equipe. A forma como as matrizes foram construídas faz com que o foco seja somente nos dados mais importantes, simplificando a sua visualização. Melhores resultados são obtidos quando um projeto possui um grande número de desenvolvedores, mas a pesquisa também pode ser aplicada em pequenas equipes, com o intuito de medir o nível de dedicação de cada desenvolvedor.

Em comparação com outros métodos de detecção de *core developers*, como o de (AVELINO et al., 2016) e (FRITZ et al., 2010), foram encontrados grandes números de responsáveis pelos projetos, que, em teoria, teriam maiores chances de ter continuidade, mesmo com a perda de desenvolvedores importantes. Diferente dos estudos de (FRITZ et al., 2010) e (GREILER; HERZIG; CZERWONKA, 2015), que analisam módulos e pacotes específicos dos sistemas, este trabalho se destaca por analisar todo o código-fonte desenvolvido pelos colaboradores, e por apontar não só desenvolvedores, mas os módulos mais críticos do sistema, que precisam de maior atenção por parte da equipe.

A análise geral do histórico de *commits* proporcionou encontrar muitas características dos projetos selecionados. Porém, explorando mais recursos da mineração de dados, é possível ampliar ainda mais o estudo em trabalhos futuros. Outros dados podem ser coletados dos repositórios e ter seu uso potencial para análises estatísticas de *Ownership Matrices*. Exemplos desses dados são a data e hora e as mensagens de *commits*. As mensagens geralmente são deixadas por contribuidores para explicar quais foram as mudanças realizadas no *commit* e podem ser utilizadas para descobrir mais detalhes sobre o impacto dessas mudanças no código. As datas podem mostrar estados do projeto em um determinado intervalo de tempo. A partir delas, os *commits* podem ser separados por meses, por exemplo, mostrando uma matriz para cada mês. Assim, é possível ampliar as análises do trabalho e discutir mais questões relacionadas à mudança de responsáveis pelo projeto e do estado dos módulos a cada mês. Uma das questões que podem ser levantadas diz respeito à possibilidade de um aumento no número de desenvolvedores em um mês estar relacionado a um mês de *release*. Também pode-se medir a frequência de modificação de um desenvolvedor em cada módulo, a fim de descobrir onde se concentra sua dedicação utilizando os dados de hora do *commit*.

Referências

ANICHE, M. *RepoDriller*. [S.l.]: GitHub, 2018. <https://github.com/mauricioaniche/repodriller>. Citado na página 19.

AVELINO, G. et al. A novel approach for estimating truck factors. In: IEEE. *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*. [S.l.], 2016. p. 1–10. Citado 7 vezes nas páginas 16, 22, 23, 25, 29, 32 e 36.

BIRD, C. et al. Don't touch my code!: examining the effects of ownership on software quality. In: ACM. *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. [S.l.], 2011. p. 4–14. Citado na página 29.

Brooks, F.P., J. No silver bullet essence and accidents of software engineering. *Computer*, v. 20, n. 4, p. 10 –19, april 1987. ISSN 0018-9162. Disponível em: <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1663532>. Citado na página 21.

CHACON, S.; STRAUB, B. *Pro Git*. [S.l.]: Apress, 2014. Citado 2 vezes nas páginas 18 e 19.

CONWAY, M. E. How do committees invent. *Datamation*, v. 14, n. 4, p. 28–31, 1968. Citado 3 vezes nas páginas 14, 15 e 34.

FERREIRA, M.; VALENTE, M. T.; FERREIRA, K. A comparison of three algorithms for computing truck factors. In: IEEE PRESS. *Proceedings of the 25th International Conference on Program Comprehension*. [S.l.], 2017. p. 207–217. Citado 3 vezes nas páginas 21, 23 e 29.

FRITZ, T. et al. A degree-of-knowledge model to capture source code familiarity. In: ACM. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. [S.l.], 2010. p. 385–394. Citado 4 vezes nas páginas 22, 23, 29 e 36.

GIT. *Git Commit*. [S.l.]: Git, 2018. <https://git-scm.com/docs/git-commit>. Citado na página 19.

GITHUB. *About pull requests*. [S.l.]: GitHub, 2018. <https://help.github.com/articles/about-pull-requests/>. Citado na página 19.

GREILER, M.; HERZIG, K.; CZERWONKA, J. Code ownership and software quality: a replication study. In: IEEE PRESS. *Proceedings of the 12th Working Conference on Mining Software Repositories*. [S.l.], 2015. p. 2–12. Citado 7 vezes nas páginas 14, 20, 22, 23, 29, 32 e 36.

HILTON, M.; BEGEL, A. A study of the organizational dynamics of software teams. In: ACM. *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. [S.l.], 2018. p. 191–200. Citado 2 vezes nas páginas 20 e 21.

HIPPEL, E. v.; KROGH, G. v. Open source software and the “private-collective” innovation model: Issues for organization science. *Organization science*, INFORMS, v. 14, n. 2, p. 209–223, 2003. Citado na página 17.

KERSTEN, M.; MURPHY, G. C. Using task context to improve programmer productivity. In: ACM. *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*. [S.l.], 2006. p. 1–11. Citado na página 22.

KRUCHTEN, P. Architectural blueprints—the “4+ 1” view model of software architecture. *Tutorial Proceedings of Tri-Ada*, v. 95, p. 540–555, 1995. Citado 2 vezes nas páginas 14 e 15.

PANICHELLA, S. et al. How the evolution of emerging collaborations relates to code changes: an empirical study. In: ACM. *Proceedings of the 22nd International Conference on Program Comprehension*. [S.l.], 2014. p. 177–188. Citado 2 vezes nas páginas 14 e 34.

PARNAS, D. L. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, ACm, v. 15, n. 12, p. 1053–1058, 1972. Citado na página 14.

PINZGER, M.; NAGAPPAN, N.; MURPHY, B. Can developer-module networks predict failures? In: ACM. *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. [S.l.], 2008. p. 2–12. Citado 2 vezes nas páginas 14 e 28.

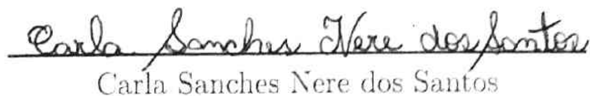
SEDANO, T.; RALPH, P.; PÉRAIRE, C. Practice and perception of team code ownership. In: ACM. *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*. [S.l.], 2016. p. 36. Citado 3 vezes nas páginas 14, 20 e 21.

SOMMERVILLE, I. *Engenharia de software. Tradução Ivan Bosnic e Kalinka G. de O. Gonçalves; revisão técnica Kechi Hirama–*. [S.l.]: São Paulo: Pearson Prentice Hall, 2011. Citado na página 21.

TERMO DE RESPONSABILIDADE

Eu, **Carla Sanches Nere dos Santos** declaro que o texto do trabalho de conclusão de curso intitulado "***Um estudo empírico sobre core developers e arquitetura de projetos populares no GitHub***" é de minha inteira responsabilidade e que não há utilização de texto, material fotográfico, código fonte de programa ou qualquer outro material pertencente a terceiros sem as devidas referências ou consentimento dos respectivos autores.

João Monlevade, 18 de dezembro de 2018



Carla Sanches Nere dos Santos

UNIVERSIDADE FEDERAL DE OURO PRETO
INSTITUTO DE CIÊNCIAS EXATAS E APLICADAS
COLEGIADO DO CURSO DE SISTEMAS DE INFORMAÇÃO

UFOP
Universidade Federal de Ouro Preto

DECLARAÇÃO DE CONFORMIDADE

Certifico que o(a) aluno(a) Carla Sanches Nere dos Santos, autor do trabalho de conclusão de curso intitulado "Um estudo empírico sobre *core developers* e arquitetura de projetos populares no GitHub" efetuou as correções sugeridas pela banca examinadora e que estou de acordo com a versão final do trabalho.

João Monlevade, 07 de janeiro de 2019.

Igor Muzetti Pereira
Professor (a) Orientador (a)

Rua Trinta e seis, 115 – Bairro Loanda – CEP 35931-008 – João Monlevade – MG – Brasil
http://www.icea.ufop.br – cosi@decea.ufop.br – **(31) 3852-8709**