ESCOLA POLITÉCNICA DA UNIVERSIDADE DE SÃO PAULO

RENATA BRASIL SILVA

**Análise empírica da influência da experiência do desenvolvedor na
degradação da arquitetura de software em desenvolvimento *open
source***

São Paulo

2020

RENATA BRASIL SILVA

**Análise empírica da influência da experiência do desenvolvedor na
degradação da arquitetura de software em desenvolvimento *open
source***

Versão Corrigida

Dissertação apresentada à Escola Politécnica da Universidade de
São Paulo para obtenção do Título de Mestre em Ciências.

Área de concentração: Engenharia de Computação

Orientador: Prof. Dr. Fábio Levy Siqueira

São Paulo

2020

Autorizo a reprodução e divulgação total ou parcial deste trabalho, por qualquer meio convencional ou eletrônico, para fins de estudo e pesquisa, desde que citada a fonte.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Código de Financiamento 001.

Este exemplar foi revisado e corrigido em relação à versão original, sob responsabilidade única do autor e com a anuência de seu orientador.

São Paulo, 12 de dezembro de 2020

Assinatura do autor:

Renata Brasil Silva

Assinatura do orientador:

Fábio S. S.

Catálogo-na-publicação

Brasil-Silva, Renata

Análise empírica da influência da experiência do desenvolvedor na degradação da arquitetura de software em desenvolvimento open source / R. Brasil-Silva, F. L. Siqueira -- versão corr. -- São Paulo, 2020.
148 p.

Dissertação (Mestrado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais.

1.Arquitetura de Software 2.Experiência de desenvolvedores 3.Mineração de repositórios de software I.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Computação e Sistemas Digitais II.t. III.Siqueira, Fábio Levy

Dedico este trabalho à minha família, em especial aos meus pais Wilson e Maria José e à minha tia Maria Inês (*in memoriam*) cuja presença eu sinto onde quer que eu vá e que, com seu amor, carinho e dedicação genuínos, deve estar muito feliz com esta realização.

AGRADECIMENTOS

Primeiramente, agradeço ao meu orientador Professor Dr. Fábio Levy Siqueira, que me orientou e me ensinou a buscar, com rigor científico, as respostas para concluir minha pesquisa. Gostaria de agradecê-lo por sempre exigir o melhor de mim e por contribuir para o meu crescimento acadêmico e profissional.

Agradeço aos meus pais Wilson e Maria José, que ensinaram a mim e minhas irmãs o valor da união e do companheirismo. Além de sempre apoiarem suas filhas em suas escolhas pessoais e profissionais sem fazer julgamentos.

Agradeço também ao meu padrinho Afonso Maria e minha madrinha Núbia, que acompanham minha trajetória estando presente e torcendo por mim se alegrando a cada conquista.

Agradeço às minhas irmãs, Adriana, Patrícia, Maria e Daniela, que, apesar das brigas, sempre foram o suporte nos momentos difíceis, além de serem companheiras de vários momentos marcantes da minha vida. Em especial à Daniela, que é minha melhor amiga e com quem eu gosto de compartilhar minhas visões mirabolantes sobre a vida. Agradeço também aos meus cunhados André e Thiago, que fazem parte da minha família como se fossem irmãos.

Agradeço também aos meus amigos que marcaram essa jornada. À Alejandra, Douglas e Leiviane, pelas várias conversas, desabafos, risadas e passeios que aliviaram a pressão do dia-a-dia.

Agradeço ao Leonardo e à Bruna que, além de dividirem o lar comigo em 2019, me proporcionaram ótimos momentos de companheirismo, conversas aleatórias e diversão.

Agradeço aos meus gatos, Nikki, Hugo, Pepe, Vivi, Bulbassau, Abgail e Bruna, que iluminam a minha vida e são um desafogo emocional nos tempos de crise.

Por fim, agradeço à Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pelo financiamento deste trabalho.

RESUMO

BRASIL-SILVA, Renata. **Análise empírica da influência da experiência do desenvolvedor na degradação da arquitetura de software em desenvolvimento *open source***. 2020. 148 p. Dissertação (Mestrado em Engenharia de Computação) – Escola Politécnica da Universidade de São Paulo, São Paulo, 2020.

A arquitetura de software tem papel fundamental durante o desenvolvimento e a evolução de um software. Em desenvolvimento *open source*, ela é crucial para possibilitar que os desenvolvedores trabalhem distribuídos geograficamente e ao mesmo tempo; além de ser uma das motivações que atraem novos colaboradores. Quando a arquitetura é afetada por problemas que aumentam o esforço de manutenção e dificultam a acomodação de novas funcionalidades, diz-se que ela está em degradação. Mesmo abordagens de desenvolvimento flexíveis, como as tipicamente usadas no desenvolvimento *open source*, podem ser afetadas por esse processo. São vários os motivos para a degradação da qualidade da arquitetura, dentre os quais, falta de documentação da arquitetura, mudanças de requisitos e alterações feitas por desenvolvedores menos experientes. Uma das medidas para minimizar a degradação que pode ser menos impactante para o processo diz respeito à política de desenvolvimento e a experiência dos desenvolvedores. Sendo assim, neste trabalho, investiga-se a influência da experiência dos desenvolvedores da degradação da arquitetura em desenvolvimento *open source*. Foram analisados quatro projetos grandes e com anos de desenvolvimento por meio da mineração de repositórios de software. Os resultados permitiram concluir que a experiência generalizada não tem relação com a degradação da arquitetura e que, por outro lado, desenvolvedores fazem boas contribuições locais nos componentes em que são especialistas, apesar de não serem conscientes do impacto nos demais componentes.

Palavras-chave: Degradação da arquitetura. Experiência de desenvolvedores. Arquitetura de software. Mineração de repositórios de software.

ABSTRACT

BRASIL-SILVA, Renata. **Análise empírica da influência da experiência do desenvolvedor na degradação da arquitetura de software em desenvolvimento *open source***. 2020. 148 p. Dissertação (Mestrado em Engenharia de Computação) – Escola Politécnica da Universidade de São Paulo, São Paulo, 2020.

Software architecture has a fundamental role in software development and software evolution. In open source software, architecture is essential to support distributed software development and to attract newcomer developers. When an architecture is affected by issues that increase maintenance efforts and hinders accommodation of new requirements, we can say that it degraded. Architectures might degrade even in flexible development processes, as the ones typically used by open source community. There are several factors connected to architecture quality degradation, such as lack of architectural documentation, requirements changes, and modifications made by less experienced developers. One of the less impactful alternatives to mitigate architectural degradation is the adoption of development policies that considers experience of developers. Therefore, in this work, we investigate the influence of developer experience on software architecture degradation in open source projects. We analyzed four large open source projects by mining software repositories. Our results indicate no relation between generalized experience and architecture degradation. On the other hand, our results indicate that specialists in components contribute with better quality solutions for architecture on those components, even though they are not aware of the impacts of their contributions in other components.

Keywords: Architectural degradation. Developer experience. Software architecture. Mining software repositories.

LISTA DE FIGURAS

Figura 2.1 – O papel da arquitetura no contexto do sistema de software.	21
Figura 4.1 - Visão geral da seleção de artigos.	34
Figura 4.2 - Dados usados para quantificar a experiência.	37
Figura 4.3 - Frequência dos termos associados à experiência.	39
Figura 4.4 - Métricas por tipo de experiência.	41
Figura 4.5 - Objetivo dos estudos com métricas de experiência.	42
Figura 6.1 - Interface do módulo de <i>Commits</i>	89
Figura 6.2 - Interface no módulo da Arquitetura.	89
Figura 6.3 - Interface do módulo de Análise Estatística.	90
Figura 7.1 - Degradação da arquitetura nos projetos Ant, Cassandra, Hadoop e Lucene.	94
Figura 7.2 - <i>Box plot</i> da distribuição do delta nos quatro projetos.	95
Figura 7.3 - <i>Box plot</i> da distribuição da experiência nos quatro projetos.	96
Figura 7.4 - Distribuição de <i>commits</i> impactantes por tempo dos autores nos quatro projetos.	97
Figura 7.5 - Gráficos de dispersão da experiência X delta nos quatro projetos.	99

LISTA DE TABELAS

Tabela 4.1 - Dados extraídos dos estudos primários selecionados.	35
Tabela 4.2 - Conceitos nos trabalhos selecionados.	40
Tabela 4.3 - Trabalhos classificados por tipo de experiência.	41
Tabela 5.1 - Causas da degradação da arquitetura.	50
Tabela 6.1 - Perfis de experiência dos desenvolvedores.	66
Tabela 6.2 - Métricas de experiência que utilizam dados disponíveis em repositórios.	68
Tabela 6.3 - Ferramentas que fornecem métricas para a qualidade da arquitetura e as métricas calculadas por elas.	74
Tabela 6.4 - Métricas de qualidade da arquitetura classificadas pelo mapeamento sistemático e que atendem aos critérios desta pesquisa.	75
Tabela 6.5 - Causas tratadas pela Apache.	79
Tabela 6.6 - Características dos sistemas selecionados.	80
Tabela 6.7 – Exemplo de dados utilizados na análise de um <i>commit</i> impactante.	87
Tabela 7.1 - Resumo das estatísticas de <i>commits</i> analisados nos quatro projetos. .	94
Tabela 7.2 - Resumo dos dados sobre o delta nos quatro projetos.	95
Tabela 7.3 - Resumo dos dados sobre a experiência nos quatro projetos.	97
Tabela 7.4 - Resultado do teste estatístico para QP1.	98
Tabela 7.5 - Resultado do teste estatístico para QP2.	100
Tabela 7.6 - Resultado do teste estatístico para QPA.	101

SUMÁRIO

1	INTRODUÇÃO	12
1.1	Objetivo	13
1.2	Justificativa	14
1.3	Contribuições	16
1.4	Escopo	17
1.5	Método	17
1.6	Organização do trabalho	18
2	ARQUITETURA DE SOFTWARE	19
2.1	Definição	19
2.2	O papel da arquitetura	20
2.3	Considerações finais	21
3	DESENVOLVIMENTO <i>OPEN SOURCE</i>	22
3.1	Definição	22
3.2	Características do desenvolvimento <i>open source</i>	23
3.3	Papéis dos desenvolvedores	25
3.4	Arquitetura	27
3.5	Considerações finais	27
4	EXPERIÊNCIA	29
4.1	Definição	29
4.2	Métricas.....	32
4.2.1	Mapeamento sistemático	32
4.3	Considerações finais	43
5	DEGRADAÇÃO DA ARQUITETURA.....	44
5.1	Anomalias arquiteturais	46

5.2	Dívida técnica	47
5.3	Causas	49
5.3.1	Documentação da arquitetura	49
5.3.2	Requisitos	51
5.3.3	Desenvolvedores	52
5.3.4	Processo de desenvolvimento	53
5.4	Métricas	54
5.5	Considerações finais	58
6	ANÁLISE EMPÍRICA	59
6.1	Contexto	59
6.2	Objetivo	61
6.3	Questões de pesquisa	62
6.4	<i>Design</i> da análise	64
6.5	Conceitos e métricas utilizados	65
6.5.1	Métrica para experiência no código	65
6.5.2	Métricas para avaliar a arquitetura	70
6.6	Seleção dos projetos	76
6.6.1	Critérios de seleção	76
6.6.2	Projetos selecionados	78
6.7	Procedimento de coleta dos dados	80
6.7.1	Dados para determinar experiência	81
6.7.2	Dados para a degradação da arquitetura	83
6.8	Procedimento de análise dos dados	84
6.9	Ferramenta	87
6.10	Considerações finais	90
7	RESULTADOS E DISCUSSÕES	92

7.1 Estatística Descritiva	93
7.2 QP1: Relação entre experiência geral e degradação da arquitetura	97
7.3 QP2: Relação entre experiência local e degradação da arquitetura no componente	98
7.4 Análise adicional (QPA).....	99
7.5 Conclusão.....	101
7.6 Ameaças à validade	103
7.6.1 Validade de construção	103
7.6.2 Validade interna.....	105
7.6.3 Validade externa.....	107
7.6.4 Validade de conclusão.....	108
7.7 Considerações finais	108
8 TRABALHOS RELACIONADOS	109
8.1 Experiência de desenvolvedores e qualidade de software	109
8.2 Evolução e degradação da arquitetura do software	114
9 CONCLUSÃO.....	116
9.1 Trabalhos futuros.....	118
REFERÊNCIAS.....	119
APÊNDICE A - INFORMAÇÕES DO MAPEAMENTO SISTEMÁTICO DAS MÉTRICAS PARA A EXPERIÊNCIA.....	131
A.1. Estudos selecionados	131
A.2. Métricas para experiência	133
APÊNDICE B - ANÁLISE DE QP2 E ANÁLISE ADICIONAL	138
B.1. Dados da análise de QP2.....	138
B.2. Dados da análise adicional	146

1 INTRODUÇÃO

A arquitetura de um sistema de software pode ser vista como um conjunto de decisões de projeto que guiam a evolução do software (BUSCHMANN; HENNEY, 2013). Segundo Martin (2017), ela é a forma dada pelos criadores do sistema a partir da definição de como será a divisão de componentes e do modo como se comunicam entre si. Pelo fato de ela servir para compartilhar o entendimento sobre o sistema entre os *stakeholders* (BASS; CLEMENTS; KAZMAN, 2012; NORD; TOMAYKO; WOJCIK, 2004), a arquitetura é fundamental para guiar a evolução de sistemas de software.

Em desenvolvimento *open source*, um dos fatores associados ao sucesso dos projetos é a capacidade do software atrair uma grande comunidade de voluntários (RAYMOND, 1999) que são, ao mesmo tempo, usuários e co-desenvolvedores do sistema. Para Raymond (1999), a arquitetura é uma das motivações que atraem colaboradores. As pessoas que começam a colaborar em um projeto desse tipo, além de serem usuários com necessidades a serem satisfeitas pelo software, geralmente também têm o interesse em aprender sobre bons padrões de código e de arquitetura (YE; KISHIDA, 2003). Uma arquitetura modularizada, que é característica nesse tipo de projeto (MACCORMACK; RUSNAK; BALDWIN, 2006; YE; KISHIDA, 2003), permite que o aprendizado seja mais fácil, porque os módulos são isolados de tal forma que os desenvolvedores iniciantes consigam fazer suas contribuições, sem causar um efeito cascata muito grande. Desse modo, podem se concentrar apenas no local da alteração e aprender aos poucos sobre as relações entre os componentes do software. Além disso, uma arquitetura bem modularizada permite que os desenvolvedores trabalhem em diferentes trechos do sistema de forma simultânea, facilitando a colaboração e o trabalho de equipes distribuídas geograficamente.

Um software evolui e muda à medida que tem suas funcionalidades atualizadas. As atualizações ocorrem por diversas razões, tais como a experiência de uso dos usuários - que traz à tona novas necessidades -, a correção de erros, as mudanças do ambiente e as mudanças no contexto de uso. Essas atualizações ocasionam mudanças que podem aumentar a complexidade do código (CANFORA et al., 2014) ou introduzir violações arquiteturais (BANDI; WILLIAMS; ALLEN, 2013). Desse processo, podem emergir problemas arquiteturais que podem comprometer características de qualidade como a usabilidade, a manutenibilidade e a portabilidade;

o que dificulta as atividades de desenvolvimento do software e tende a diminuir o período em que um sistema ainda se mantém com custos de manutenção viáveis (MARTIN, 2017). Quando isso ocorre, diz-se que a arquitetura está em um processo de degradação.

1.1 Objetivo

As características das pessoas que escrevem o código, em particular, a experiência do desenvolvedor, estão entre as razões que podem levar à degradação da arquitetura. Pretende-se, com esta pesquisa, entender em que medida a inexperiência de desenvolvedores pode ser determinante para a piora da arquitetura, sob a perspectiva da facilidade de manter um sistema de software com eficiência. Diante disso, o objetivo deste trabalho é investigar se há relação entre a experiência de desenvolvedores e a degradação da arquitetura em desenvolvimento *open source*. Isso será feito pela análise de repositórios de projetos *open source*, considerando o histórico de contribuição dos desenvolvedores nesses sistemas e, também, a evolução da arquitetura ao longo de várias versões de um software. Essa análise será realizada mediante a resposta às seguintes questões de pesquisa:

QP1: Existe relação entre a experiência dos desenvolvedores no sistema e a degradação da arquitetura causada por eles?

Esta questão de pesquisa visa investigar se, de modo geral, à medida que os desenvolvedores ficam mais experientes, suas contribuições degradam menos a arquitetura. Se isso for verdade, quanto mais contribuições forem feitas no código por um desenvolvedor, implicará em mais conhecimento adquirido por essa pessoa a respeito da arquitetura e, conseqüentemente, uma arquitetura mais fácil de manter.

QP2: Existe relação entre experiência dos desenvolvedores em um componente específico (experiência local) e a degradação da arquitetura causada por eles no componente?

Essa questão de pesquisa tem como objetivo observar se o acúmulo de experiência no código, em um componente, resulta em contribuições que melhoram a arquitetura do componente. Se essa hipótese for verdadeira, pode-se esperar que os

desenvolvedores mantenham a qualidade arquitetural nos componentes em que são especialistas, mas não garante que eles não degradem a arquitetura de outros componentes.

Em outras palavras, pretende-se investigar o impacto da experiência generalizada no sistema sobre a degradação da arquitetura, bem como o impacto da experiência especializada nos componentes e a degradação da arquitetura neles. As duas questões de pesquisa são importantes para entender como se dá o grau de influência das características dos desenvolvedores na qualidade da arquitetura de um sistema ou em partes dele.

1.2 Justificativa

Como dito anteriormente, a arquitetura tem um papel fundamental no desenvolvimento de software, seja pelo seu papel estrutural, seja pelo seu funcionamento como meio de comunicação entre os *stakeholders* de um sistema. Em projetos *open source*, há ainda uma motivação extra para a preocupação com a qualidade da arquitetura: ela é um dos fatores que estimulam a participação das pessoas e contribui para o sucesso desses softwares (RAYMOND, 1999; YE; KISHIDA, 2003). Quando a arquitetura degrada, vários riscos se apresentam ao projeto, entre os quais, problemas de implementação e dificuldades para corrigir erros e mudar requisitos; além de problemas relacionados ao gerenciamento do processo de desenvolvimento.

Em relação a esse último ponto, se o que está implementado diferir do que foi planejado, essa situação pode dificultar e, até mesmo, impedir o planejamento de novas funcionalidades. Isso ocorre pelo fato da arquitetura – e sua representação – serem o meio pelo qual os *stakeholders* entendem o que está em vigor e discutem como alcançar características de qualidade desejáveis para o sistema.

Problemas que afetam a arquitetura são, portanto, responsáveis por dificuldades de desenvolvimento e manutenção, com possibilidade de representar risco ao projeto. Em contrapartida, os problemas relacionados ao código (p.ex.: anomalia de código) e ao *design* (p.ex.: anomalia de *design*) podem ser tratados mediante a adoção de boas práticas de desenvolvimento e nem sempre eles se refletem em problemas arquiteturais. Por esse motivo, o interesse deste trabalho é o nível arquitetural.

A degradação da arquitetura se dá por diferentes motivos, dentre os quais: a falta de documentação, as mudanças de requisitos, a falta de preocupação com requisitos arquiteturalmente significativos, a arquitetura que não suporta facilmente mudanças e a falta de experiência de desenvolvedores. Alguns desses motivos são inevitáveis, como as mudanças de requisitos, por exemplo, que são inerentes ao processo de evolução de um software. Outros fatores podem ser tratados a partir de mudanças no processo de desenvolvimento; é o caso da política de desenvolvimento que considera a capacidade técnica dos desenvolvedores e o gerenciamento de times de desenvolvedores. Ações e mudanças nesse âmbito podem diminuir o processo de degradação da arquitetura, possibilitando que o software tenha uma vida longa, ao passo que atende as necessidades dos usuários. Além disso, são medidas que, ao serem tomadas, podem ocasionar menos impacto do que refazer uma arquitetura do início, por exemplo.

Apesar de processos mais flexíveis, como os tipicamente executados no desenvolvimento *open source*, minimizarem boa parte desses fatores, problemas arquiteturais ainda surgem e causam transtornos. Nesse contexto, a qualidade e a experiência dos desenvolvedores de um time podem ser fatores mais facilmente controláveis e com menos custo e esforço associado, em comparação com outras medidas para minimizar a degradação. Portanto, a contribuição deste trabalho é estudar se a experiência dos desenvolvedores tem alguma influência na degradação da arquitetura em desenvolvimento *open source*, pois pessoas com menos contato prévio com o código estão mais propensas a introduzir defeitos (BIRD et al., 2011) e dívida técnica (ALFAYEZ et al., 2018). Com isso, pretende-se entender quais as melhores formas para lidar com o problema da degradação arquitetural. Caso essa relação exista, tal conhecimento possibilita a adoção de medidas potencialmente menos custosas para minimizar a degradação; pois, se é verdade que um dos fatores responsáveis pela piora da arquitetura é o nível de experiência dos desenvolvedores, basta que se adotem medidas que reforcem a necessidade de profissionais mais experientes (com base em suas contribuições para o código) como responsáveis por áreas cruciais para integridade da arquitetura. Contudo, isso não exclui a necessidade de se adotar outras medidas baseadas na existência de outras causas de degradação. Outra possível consequência em caso das hipóteses de pesquisa serem verdadeiras, é a possibilidade de realizar o acompanhamento das contribuições e das métricas da

arquitetura de desenvolvedores menos experientes, no decorrer das versões, e evitar o acúmulo de problemas arquiteturais que, se deixados de lado por muito tempo, podem ser mais difíceis de serem resolvidos por várias razões. Um exemplo disso é a falta de tempo e de recursos humanos disponíveis para dedicar à manutenção da estrutura do sistema, enquanto os usuários esperam ter suas necessidades de uso atendidas.

Caso não se confirmem as hipóteses, é preciso continuar investindo no estudo de como as demais causas impactam a qualidade da arquitetura, para que se criem alternativas viáveis para contornar a degradação arquitetural.

1.3 Contribuições

As principais contribuições deste trabalho são:

- Relacionar a experiência e a degradação da arquitetura em desenvolvimento *open source*, considerando *commit-a-commit*;
- Relacionar a experiência local e a degradação da arquitetura em um componente em desenvolvimento *open source*, considerando *commit-a-commit*;
- Analisar a experiência e a qualidade do software do ponto de vista da qualidade da arquitetura, pois é o que permite que se tenha uma evolução sustentável de um software. A maioria dos trabalhos fazem a análise da experiência de desenvolvedores e a qualidade de software pela análise de código;
- Um mapeamento sistemático sobre métricas para medir experiência na Engenharia de Software;
- O desenvolvimento da ferramenta *Mining My Repository*¹, que analisa repositórios e coleta dados para calcular a experiência e degradação da arquitetura *commit-a-commit*.

¹ https://github.com/renatabrasil/mining_my_repository

1.4 Escopo

Este trabalho considera apenas a arquitetura de softwares orientados a objetos - com pacotes representando componentes – devido à métrica adotada para medir a qualidade da arquitetura. Além disso, será analisada a degradação sob a perspectiva da manutenibilidade, que é a característica de qualidade relacionada ao grau de eficiência e efetividade que um software pode ser modificado por seus mantenedores (ISO/IEC, 2011). Isso implica que, quando a arquitetura degrada, tem-se maior grau de esforço e mais custos associados à manutenção de um sistema.

1.5 Método

Pode-se dividir este trabalho em duas fases: (1) a fase de planejamento, que compreende a definição do *design* da pesquisa e das métricas; e (2) a fase operacional, na qual são decididas e realizadas as ações necessárias para a execução da pesquisa, isto é, detalhes sobre o procedimento de coleta de dados e análise de dados, bem como a execução desses dois processos.

Na fase de planejamento, após a definição do objetivo, das questões de pesquisa e do *design* da pesquisa, foi realizada uma revisão bibliográfica sobre a degradação da arquitetura e, principalmente, sobre formas de medir a qualidade a arquitetura implementada. O passo a seguir foi escolher uma forma de medir a experiência, mas, ao contrário do que ocorreu com a degradação, não foram identificados outros trabalhos que faziam um levantamento das métricas para medir a experiência. Por essa razão, foi realizado um mapeamento sistemático sobre métricas para medir experiência na Engenharia de Software. A revisão bibliográfica e o mapeamento sistemático foram úteis à escolha das métricas.

Na fase operacional, foi realizada a análise em repositórios de software (mineração de repositórios), utilizando dados disponíveis por sistemas de controle de versões. Para o procedimento de coleta dos dados, foi realizada a busca por uma ferramenta para fazer a análise de repositórios. Como as opções encontradas não satisfizeram os requisitos correspondentes às métricas e à forma de análise escolhidas, que rastreia *commit-a-commit* de uma versão do software, desenvolveu-se uma ferramenta.

Os processos de coleta de dados e de análise dos dados foram realizados por meio da ferramenta desenvolvida. Foram aplicados métodos estatísticos na análise de dados, devido à natureza quantitativa dos valores. Essas etapas duraram alguns meses e foram executadas de forma iterativa, enquanto a ferramenta era refinada a fim de satisfazer as necessidades da análise.

1.6 Organização do trabalho

Além desta introdução, este documento compreende mais oito capítulos. Os Capítulos 2, 3, 4 e 5 fazem parte da fundamentação teórica do trabalho. O Capítulo 2 discute a arquitetura de software; o Capítulo 3, o desenvolvimento *open source* e suas características; o Capítulo 4 aborda o conceito de experiência em Engenharia de Software e nesta pesquisa; e, por fim, o Capítulo 5 discute a degradação da arquitetura, seus sintomas, suas causas e as métricas que podem avaliar uma arquitetura implementada.

O Capítulo 6 apresenta o protocolo da análise empírica realizada, o que inclui o objetivo, as questões de pesquisa, os conceitos e métricas, o *design* da pesquisa, a ferramenta utilizada e o procedimento de coleta e análise de dados. O Capítulo 7 apresenta e discute os resultados da análise. Além disso, apresenta as ameaças à validade deste trabalho.

O Capítulo 8 traz uma discussão acerca dos trabalhos relacionados, destacando as similaridades e diferenças em relação a este. Por fim, o Capítulo 9 expõe as considerações finais e as perspectivas de trabalhos futuros.

Este documento possui, ainda, dois apêndices. O Apêndice A apresenta informações complementares sobre o mapeamento sistemático a respeito da forma de medir a experiência em Engenharia de Software. O Apêndice B traz detalhes das análises dos resultados do Capítulo 7.

2 ARQUITETURA DE SOFTWARE

A arquitetura de um sistema é a descrição de seus componentes e da forma como eles se comunicam entre si; ou seja, é a representação da estrutura do sistema. Um edifício, uma corporação ou um software são exemplos de sistemas. No caso de um software, os componentes são as entidades de implementação (p.ex.: classes e pacotes) e as relações podem compreender a noção de herança ou de associação entre classes, por exemplo. A arquitetura é fundamental para guiar a evolução de softwares, mesmo em processos de desenvolvimento em que sua importância não seja evidente, ou naqueles em que ela não é tratada com formalidade, tal como nos métodos ágeis e em desenvolvimento *open source*.

2.1 Definição

Há mais de uma definição de arquitetura de software aceita (BASS; CLEMENTS; KAZMAN, 2012; ISO/IEC/IEEE, 2011; MEDVIDOVIC; TAYLOR, 2010; PERRY; WOLF, 1992). A maioria das definições envolve a organização de componentes e relacionamentos entre eles, mas também há definições mais abstratas.

A definição de Perry e Wolf (1992) retrata a arquitetura de software como “um conjunto de elementos arquiteturais que possuem alguma organização” (PERRY; WOLF, 1992, p. 44, tradução nossa). Nessa definição, elementos arquiteturais são componentes que se relacionam uns com os outros satisfazendo certas propriedades e restrições. Para os autores, também fazem parte da arquitetura as decisões que justificam as escolhas feitas tanto para respeitar as restrições quanto para satisfazer os requisitos.

A definição de Bass, Clements e Kazman (2012) é bem similar a essa e conceitua a arquitetura como “o conjunto de estruturas necessárias para raciocinar sobre o sistema, que compreendem elementos de software, relações entre eles e as propriedades de ambos” (BASS; CLEMENTS; KAZMAN, 2012, p. 18, tradução nossa). No entanto, nessa definição, nada é dito quanto às decisões que motivam as relações entre componentes e suas propriedades.

A definição da norma ISO/IEC/IEEE 42010 (2011) inclui, como parte da arquitetura, os princípios de *design* e evolução que regem o sistema: “arquitetura de

um sistema é o conjunto de conceitos fundamentais ou propriedades de um sistema no seu ambiente, que incorpora seus elementos, relacionamentos e seus princípios de *design* e evolução” (ISO/IEC/IEEE, 2011, p. 2, tradução nossa).

Medvidovic e Taylor (2010) têm uma definição mais abstrata que difere das três anteriores, pois não mencionam diretamente elementos, relações e propriedades entre eles. Para eles, a arquitetura de um sistema é “o conjunto das principais decisões de *design* feitas durante seu desenvolvimento e evolução” (MEDVIDOVIC; TAYLOR, 2010, p. 471, tradução nossa).

Essas definições, de alguma forma, falam de arquitetura como uma maneira de organização de componentes que se relacionam entre si, segundo princípios de *design*, satisfazendo requisitos e restrições postas para o sistema. De acordo com essa ideia, a arquitetura de software pode ser vista como um conjunto de decisões de projeto que guiam a evolução do software (BUSCHMANN; HENNEY, 2013).

2.2 O papel da arquitetura

Em geral, a arquitetura segue princípios de *design* para alcançar determinadas características de qualidade desejadas pelos *stakeholders*. Um exemplo disso é a decomposição do sistema em partes, de tal forma que haja independência entre os componentes da arquitetura, possibilitando fácil manutenção; ou seja, permitindo que o software possa ser modificado com um bom grau de eficiência e efetividade por seus mantenedores, que é uma das características de qualidade de software segundo a ISO/IEC 25010 (2011). As características de qualidade, ou atributos de qualidade, são “uma propriedade mensurável ou testável de um sistema que é usada para indicar o quanto o sistema satisfaz as necessidades dos *stakeholders*” (BASS; CLEMENTS; KAZMAN, 2012, p. 63, tradução nossa). A Figura 2.1 mostra o contexto de um sistema, o qual está situado em um ambiente e exibe uma arquitetura. Esse contexto inclui também os *stakeholders*, seus interesses e propósito, e a descrição da arquitetura, que deve expressar a arquitetura do sistema. Em resumo, um sistema situado em um determinado ambiente, com características próprias, exibe uma arquitetura. Essa arquitetura leva em conta os *stakeholders* do sistema; isto é, desenvolvedores, testadores e clientes do projeto, e seus interesses.

Figura 2.1 – O papel da arquitetura no contexto do sistema de software.

Fonte: Traduzido e adaptado de ISO/IEC/IEEE 42010 (2011).

2.3 Considerações finais

A importância da arquitetura se dá, principalmente, pelo seu papel de guiar a evolução do software mesmo em processos de desenvolvimento em que atividades arquiteturais não são conduzidas com formalidade. A arquitetura, particularmente a sua descrição, serve como forma de comunicação entre *stakeholders* e possibilita manter a qualidade da arquitetura, por meio do registro das decisões de *design* e das características de qualidade que se deseja alcançar (BASS; CLEMENTS; KAZMAN, 2012). Além disso, uma boa arquitetura permite, entre outras coisas, que um software evolua com o custo de manutenção viável. O próximo capítulo descreve o desenvolvimento *open source* e o papel e as características da arquitetura nesse contexto.

3 DESENVOLVIMENTO *OPEN SOURCE*

Sistemas de software *open source* vêm se consolidando como alternativa ao software proprietário. Segundo relatório anual publicado pela RedHat², no meio corporativo há uma tendência de queda no uso de softwares proprietários. Em 2019, 55% dos softwares usados pelos entrevistados era software proprietário, enquanto em 2020, esse número é de 42%. Em dois anos, a estimativa é que esse número caia para 32%. Isso pode ser atribuído ao alto custo com licenças caras e inflexíveis, além da dependência de fornecedor. Como alternativa a esses entraves, tem crescido a participação do software *open source* no ambiente empresarial. De acordo o relatório da Red Hat, o uso de softwares *open source* está em trajetória ascendente tanto entre empresas, quanto entre a comunidade de usuários comuns. Estima-se que, nos próximos dois anos, o uso de software *open source* em ambiente corporativo passará de 36% a 44% e o uso entre a comunidade passará de 19% para 21%. Um dos motivos para esse crescimento se deve ao processo de desenvolvimento de sistemas *open source*, que se caracteriza, pelas entregas rápidas e pela proximidade com o usuário final. Além disso, algumas das características destacáveis e que o diferenciam das formas de desenvolvimento de código não aberto são: a forma de coordenação do processo, os meios de comunicação, os papéis dos desenvolvedores e o papel da arquitetura do software.

3.1 Definição

O conceito de software *open source* está intimamente relacionado ao de software livre. Ambos correspondem a softwares nos quais os usuários são livres para usar, copiar e distribuir. O software livre, segundo o Projeto GNU³, “é software que vem com permissão para qualquer um copiar, usar e distribuir, com ou sem modificações, gratuitamente ou por um preço. Em particular, isso significa que o código-fonte deve estar disponível”. A *Open Source Initiative*⁴, por outro lado, define o software *open*

² <https://www.redhat.com/en/enterprise-open-source-report/2020>

³ Categorias de softwares livres e não livres: <http://www.gnu.org/philosophy/categories.html>

⁴ <https://opensource.org/osd>

source não apenas como um software cujo acesso ao código-fonte é aberto, mas cuja distribuição do software deve seguir critérios que, em resumo, garantem que: o software possa ser livremente redistribuído (vendido ou distribuído); que o código-fonte esteja disponível permitindo sua distribuição no formato fonte e no formato compilado; que permita modificações e trabalhos derivados que possam ser distribuídos nos mesmos termos do código original; que a finalidade do software não seja restrita a nenhum grupo de pessoas ou área de atuação (p.ex.: uso por empresas); que os direitos associados ao programa sejam aplicados a todos para os quais o programa é redistribuído; e que a licença não imponha restrições quanto ao uso em conjunto com outros softwares distribuídos. De acordo com Stallman (2014), do Projeto GNU, a diferença está na filosofia por trás de ambos: o software livre fala de liberdade, no sentido social, e de promover solidariedade social por meio de compartilhamento e cooperação; a visão do software *open source* está mais atrelada à ideia dos benefícios práticos de se usar código aberto, e, principalmente, à possibilidade de criar softwares poderosos e confiáveis. Ambos os modelos diferem basicamente quanto às licenças aceitas, mas na prática quase todo software *open source* é também software livre (STALLMAN, 2014).

3.2 Características do desenvolvimento *open source*

A filosofia *open source* se opõe à ideia de que, para um software ter sucesso, é necessária uma política de gerenciamento de desenvolvimento marcada por processos rígidos e entregas espaçadas de versões. Nesse tipo de processo, apelidado de estilo “catedral” (RAYMOND, 1999), por assemelhar-se a um processo de construção de uma catedral - que é considerado sólido e seguro - os desenvolvedores são liderados por alguém que atribui tarefas e prazos e reforça o compromisso com a produtividade da equipe. Nesses processos, é comum que as entregas entre versões levem tempo, pois deseja-se que o produto entregue contenha a menor quantidade de erros quanto possível, para não afetar negativamente o índice de satisfação do usuário final. Além disso, reforça-se a necessidade de documentação abrangente, para assegurar o sucesso do processo e diminuir riscos; a exemplo da fuga de conhecimento, ocasionada pela rotatividade dos envolvidos no projeto. Essa visão de processo também defende a ideia de propriedade de código individual,

segundo a qual setores do sistema pertencem a determinadas pessoas as quais podem ser responsabilizadas por eles. Acredita-se que muitas mãos no código podem levar à perda de qualidade. Outra crença dessa abordagem de desenvolvimento é que mais envolvidos desacelera o processo: “adicionar pessoas a um projeto de software atrasado resulta em um atraso ainda maior” (BROOKS JR, 1995, p.19, tradução nossa).

Em contrapartida, para os defensores do desenvolvimento *open source* ocorre justamente o oposto: ter mais pessoas envolvidas no processo de desenvolvimento implica benefícios. Uma das justificativas para essa ideia é a crença de que, quanto mais pessoas conheçam, estudem e se interessem, mais favorável é a detecção e correção de erros, além de mais provável a expansão das funcionalidades do software (RAYMOND, 1999). O desenvolvimento se caracteriza por entregas rápidas e frequentes de software e pela aproximação com os usuários finais. Nesse ponto, o desenvolvimento *open source* é parecido com os métodos ágeis; com a diferença de que, naquele, há uma comunidade de voluntários que são considerados como co-desenvolvedores do software (RAYMOND, 1999); o que não ocorre necessariamente com métodos ágeis.

Além disso, o sucesso dos softwares *open source* tem relação direta com o tamanho e a atividade da comunidade de voluntários (RAYMOND, 1999; ZHOU; MOCKUS, 2012). Para isso, outros fatores também são importantes (RAYMOND, 1999): a qualidade da arquitetura e do código, que propiciam fácil entendimento para os novatos ou possibilidade de aprendizagem; e o quão bem ele atende aos usuários finais. Esse estilo é apelidado por Raymond (1999) como “estilo bazar”, pelo aparente caos comparado ao estilo tradicional de desenvolvimento (estilo catedral).

Nesse contexto, a motivação dos indivíduos, que são usuários do software e, ao mesmo tempo, co-desenvolvedores também pode ser considerado um fator de sucesso, pois eles conhecem as funcionalidades do produto e também suas próprias necessidades como usuários (RAYMOND, 1999); sendo mais fácil encontrar problemas e soluções (RAYMOND, 1999). Dito de outro modo, com tantos indivíduos participando, em algum momento alguém terá a solução para um determinado problema, pois, como usuários do software, só se envolvem na codificação pessoas que realmente querem estar fazendo aquele trabalho e que têm compromisso com a qualidade dele. Desse modo, a motivação pode estar associada a fatores como

reputação na comunidade ou até mesmo a satisfação individual (MOCKUS; FIELDING; HERBSLEB, 2002; RAYMOND, 1999). Ademais, esses fatores são, também, sociais; não apenas técnicos. Elementos sociais, e inerentes à vida em sociedade, ajudam a explicar por que indivíduos se agrupam para produzir e manter um produto de software que é entendido como pertencente à toda essa comunidade, ou seja, como propriedade coletiva da comunidade.

3.3 Papéis dos desenvolvedores

Em projetos desse tipo, apesar de haver um fluxo de pessoas entrando e saindo, em projetos longevos, há um núcleo de desenvolvedores mais ativo que está sempre contribuindo (JOBLIN et al., 2017). Esses desenvolvedores acabam sendo os mais experientes naquele código e, portanto, acabam sendo responsáveis pela evolução do software (MOCKUS; FIELDING; HERBSLEB, 2000). Os desenvolvedores mais ativos têm mais porcentagem de propriedade de código, porque produzem mais e, muitas vezes, acabam fazendo parte do corpo de desenvolvedores principais; se houver um (MOCKUS; FIELDING; HERBSLEB, 2000). Devido à experiência no código, eles têm mais participação na tomada de decisões (JOBLIN et al., 2017; MATRAGKAS et al., 2014). Porém, nem todos os voluntários escrevem código com frequência (MATRAGKAS et al., 2014). Algumas pessoas, simplesmente, contribuem relatando erros ou escrevendo documentação (MOCKUS; FIELDING; HERBSLEB, 2002).

Segundo Nakakoji et al. (2002), os desenvolvedores podem ser classificados nos papéis de:

- Desenvolvedor periférico: contribui, ocasionalmente, com novas funcionalidades ou *features*, ou seja, a contribuição é irregular (não é frequente e contínua);
- Desenvolvedor ativo: aqueles que contribuem regularmente com novas funcionalidades e *features* e corrigem *bugs*;
- Membro principal: responsáveis por guiar e coordenar o desenvolvimento e
- Líder do projeto: geralmente é quem iniciou o projeto.

De acordo com Crowston e Howison (2006) e Joblin et al.(2017), em projetos *open source*, apenas uma pequena fração de desenvolvedores é responsável por desempenhar a maioria do trabalho. Ainda de acordo com eles, a estrutura de um projeto *open source* assemelha-se a uma cebola, pois, no centro, estão os desenvolvedores responsáveis pela maioria do trabalho, em outros termos, os membros principais, que são os envolvidos nas decisões arquiteturais e que fazem parte do projeto durante um longo tempo (FERREIRA; VALENTE; FERREIRA, 2017). Na próxima camada estão os desenvolvedores ativos, que contribuem regularmente. Na camada seguinte, estão os desenvolvedores periféricos, que têm uma rotatividade maior nos projetos e suas contribuições estão mais concentradas em correção de erros e pequenos aprimoramentos (FERREIRA; VALENTE; FERREIRA, 2017). Por fim, na camada mais exterior, estão os contribuidores que não contribuem com o código, mas auxiliam reportando erros e escrevendo documentação, por exemplo.

Quanto à propriedade de código, pode haver uma política de particionamento de código (GUTWIN; PENNER; SCHNEIDER, 2004; MOCKUS; FIELDING; HERBSLEB, 2002), na qual um desenvolvedor pode ser o responsável por um determinado componente. Em muitos casos, alguns desses desenvolvedores mais ativos acabam por dominar a porcentagem de contribuição de alguns componentes, justamente, pela frequência de contribuições e consequentemente pela experiência naquela parte do código, mesmo sem haver uma política de particionamento de código explícita (MOCKUS; FIELDING; HERBSLEB, 2000).

Outro ponto importante, para que um projeto *open source* tenha sucesso, é a forma de coordenação das contribuições e atividades. Segundo Raymond (1999), ela pode não acontecer como no estilo “catedral”, que é centralizada na figura de um gerente, mas sim pela existência de líderes de projeto; eles não designam tarefas, como ocorre no estilo “catedral”, eles coordenam as contribuições. A coordenação ocorre, geralmente, por meio de tecnologias de comunicação, via web (MOCKUS; FIELDING; HERBSLEB, 2000) e realizada pelos próprios voluntários (NAKAKOJI et al., 2002). Os meios de comunicação mais comuns são as listas de e-mail, os fóruns de desenvolvimento e os *issues trackers* (BUTLER et al., 2018; LINÅKER et al., 2016).

3.4 Arquitetura

Outro fator que contribui para o sucesso e o funcionamento do modelo de desenvolvimento dos softwares *open source* é o *design* da arquitetura desses sistemas, que propicia a eficiência do desenvolvimento paralelo e da aquisição de conhecimento (YE; KISHIDA, 2003). O *design* é normalmente pensado para que a arquitetura seja modular, ou seja, que permita a independência e baixo acoplamento entre os componentes do sistema, e a possibilidade de ter várias tarefas sendo desenvolvidas paralelamente, com níveis de dificuldade progressiva, e sem necessidade de entender profundamente as partes que não estão sendo modificadas (MACCORMACK; RUSNAK; BALDWIN, 2006; YE; KISHIDA, 2003). Isso serve para acomodar novos desenvolvedores até que eles comecem a se envolver com tarefas mais difíceis (YE; KISHIDA, 2003). Um exemplo disso é a arquitetura do servidor Web Apache, que, segundo Mockus, Fielding e Herbsleb (2002), foi desenvolvida para separar a funcionalidade principal do servidor das *features*, que estão localizadas em componentes que podem ser compilados e configurados separadamente. Arquiteturas muito complexas são, também, uma dificuldade para adesão de novos desenvolvedores, uma vez que dificultam que novos membros façam contribuições isoladas ou até mesmo entendam o que precisa ser feito (YE; KISHIDA, 2003). Em outros termos, a arquitetura é um dos fatores que estimula a participação das pessoas e contribui para o sucesso dos sistemas de software *open source*, embora haja evidências que desenvolvedores voluntários podem degradar a arquitetura (PAIXAO et al., 2017). A fim de evitar essa situação, as soluções costumam ser discutidas e melhoradas antes de serem incorporadas. Além disso, costuma-se realizar revisão de código, para garantir o seu padrão de qualidade (MOCKUS; FIELDING; HERBSLEB, 2000).

3.5 Considerações finais

Sistemas de software *open source* são marcados por uma estrutura de coordenação hierarquizada, porém descentralizada, na qual o trabalho não é concentrado nas mãos de poucos indivíduos. Além disso, é um tipo de desenvolvimento marcado pelo alto nível de envolvimento da comunidade de usuários, os quais podem se tornar

contribuidores do software, por meio de várias atividades, entre as quais: escrita de documentação, codificação, revisão de código e relato de *bugs*. Outras características destacáveis, do desenvolvimento *open source*, são as entregas frequentes; a aproximação com o usuário final; a arquitetura modularizada, que facilita o entendimento de novatos que desejam contribuir escrevendo código; e o desenvolvimento paralelo, no qual várias pessoas diferentes trabalham em partes distintas do software, ao mesmo tempo, podendo ou não estar geograficamente distribuídas. Os desenvolvedores principais em projetos *open source* são geralmente os desenvolvedores que estão há um longo tempo no projeto e acabam por participar das decisões arquiteturais do software. O grau de experiência de um desenvolvedor será discutido no próximo capítulo, a partir da definição de experiência, e da discussão do modo como ela é quantificada em Engenharia de Software.

4 EXPERIÊNCIA

A noção de experiência está associada à quantidade de exposição a uma determinada circunstância, ou seja, ao acúmulo de conhecimento na execução de uma atividade (OZCAN KINI; TOSUN, 2018). No entanto, em desenvolvimento de software, a experiência de um desenvolvedor em uma determinada atividade, em um determinado projeto, pode ser influenciada por outros aspectos que não, somente, o contato prévio com a atividade em questão (BHATTACHARYA; NEAMTIU; FALOUTSOS, 2014). Medir a experiência pode ser importante para entender se ela é ou não determinante para a qualidade do software. Ela tem sido utilizada, dentre outras coisas, para entender quais políticas devem ser adotadas para melhorar o processo de desenvolvimento de software (BIRD et al., 2011; RAHMAN; DEVANBU, 2011).

4.1 Definição

Uma noção de experiência de um desenvolvedor no sistema como um todo é dada por Rahman e Devanbu, que definem experiência *geral* como: “a contribuição acumulativa do desenvolvedor para o sistema inteiro em um determinado momento” (2011, p. 493, tradução nossa). Uma noção mais específica de experiência é dada por Kini e Tosun, ao afirmarem que “a experiência de um desenvolvedor está relacionada ao seu conhecimento em um contexto particular, isto é, diretório, arquivo ou *commit*” (2018, p. 73, tradução nossa). Rahman e Devanbu (2011) também apresentam a noção de experiência *especializada*, que é associada à experiência específica em artefatos ou partes do código. Todas essas definições podem envolver a contribuição de indivíduos em código e, também, em outras atividades do desenvolvimento, como discussões em fóruns e decisões de projeto.

O conhecimento adquirido por meio da experiência, a partir de contribuições e exposição a atividades em um sistema, é chamado por Robillard (1999) de *conhecimento episódico*. Esse autor também fala em *conhecimento tópico* (ou *semântico*), o qual é adquirido em livros e cursos. Esses dois tipos de conhecimento são diferentes entre si, principalmente, porque o indivíduo que possui apenas conhecimento tópico não é, necessariamente, experiente como o que tem conhecimento episódico. Embora ele tenha acumulado conhecimento a partir da

teoria, ele não obrigatoriamente saberá como agir diante de situações práticas. Autores como Bird et al. (2011), Kini e Tosun (2018) e Rahman e Devanbu (2011) consideram primariamente o conhecimento episódico, uma vez que, de acordo com eles, um indivíduo aumenta seu conhecimento à medida que aumenta suas contribuições, e não por causa de seu conhecimento teórico prévio.

O termo *expertise* também pode ser relacionado à experiência, a exemplo do que foi discutido por Mockus e Herbsleb (2002). Os autores usam o termo afirmando que “*expertise* é definida como a habilidade de um especialista, e, se interpretada quantitativamente, reflete o nível de habilidade de uma pessoa para desempenhar determinadas tarefas” (MOCKUS; HERBSLEB, 2002a, p. 506, tradução nossa). Isso implica que a *expertise* não está relacionada somente à experiência a partir da quantidade da exposição de uma pessoa àquela atividade, mas, também, à qualidade de suas contribuições. No caso da atividade de escrita de código, devido à maior exposição ao código, espera-se que a contribuição dessa pessoa seja de mais qualidade, uma vez que, a cada interação com o código, sua experiência tende a aumentar (RAHMAN; DEVANBU, 2011; TEUSNER; MATTHIES; GIESE, 2017).

Outro conceito relacionado à experiência é a propriedade de código, que pode ser entendida como a informação sobre quem é o responsável por cada parte do código. Autores como Bird et al. (2011) e Rahman e Devanbu (2011) quantificam a propriedade de código por meio da quantidade de atividades que cada desenvolvedor desempenha em um sistema em relação a todas as contribuições dos demais desenvolvedores desse sistema. Por exemplo, Bird et al. (2011) medem a propriedade de código, de um determinado desenvolvedor, pela razão entre a quantidade de *commits* que esse desenvolvedor faz e a quantidade de *commits* realizada por todos os desenvolvedores na parte do sistema considerada. Dessa forma, nesses trabalhos, as pessoas com maior propriedade de código também podem ser entendidas como as mais experientes naquelas porções de código pelas quais ela é responsável.

Em políticas de desenvolvimento que reforçam a propriedade forte de código, existe uma pessoa que possui mais porcentagem de contribuição que as demais. Ela é considerada especialista naquela parte do sistema, podendo ser responsabilizada por aquele componente e, dentre outras coisas, ser consultada em caso de dúvidas (FOUCAULT; FALLERI; BLANC, 2014; RAHMAN; DEVANBU, 2011). Por isso, espera-se que esse indivíduo faça contribuições e fique responsável por tarefas mais

complexas, como novas funcionalidades e mudanças na arquitetura, que são atividades mais propensas a introduzir erros (ZELLER, 2009). A principal desvantagem desse tipo de política de propriedade de código é a restrição do conhecimento a apenas uma pessoa (RAHMAN; DEVANBU, 2011); o que cria uma relação de dependência, pondo em risco o projeto caso essa pessoa venha a se desligar.

Por outro lado, em políticas que reforçam a propriedade coletiva de código, todos são responsáveis por todas as partes do sistema, ou, pelo menos, pela parte em que a propriedade é coletiva (RAHMAN; DEVANBU, 2011). Espera-se que todos os indivíduos acumulem experiência no sistema como um todo, progressivamente, possibilitando que o conhecimento não esteja restrito a uma pessoa, e proporcione soluções de melhor qualidade. Dessa forma, a saída de desenvolvedores não constitui um risco para o projeto, como normalmente seria. Sobre esse assunto, utilizam-se os termos *Truck Factor (TF)*, *Bus Factor* e *Lottery Number* para se referir ao “número mínimo de desenvolvedores que precisam ser atingidos por um caminhão (ou sair do projeto) antes que o projeto seja incapacitado” (FERREIRA; VALENTE; FERREIRA, 2017, p.207, tradução nossa). O *Truck Factor* mede o nível de concentração de conhecimento do projeto nos membros, ou seja, TF baixo implica que o conhecimento está concentrado em poucas pessoas, o que representa mais riscos para o projeto; ao passo que TF alto implica na presença de práticas de propriedade coletiva (FERREIRA; VALENTE; FERREIRA, 2017). É comum encontrar a adoção da propriedade coletiva em métodos ágeis, como na Programação Extrema (BECK, 2000), e em projetos *open source* (RAYMOND, 1999).

Em resumo, a noção de experiência, *expertise*, conhecimento e propriedade de código aparecem frequentemente associadas (BIRD et al., 2011; FERREIRA; VALENTE; FERREIRA, 2017; QIU et al., 2015; RAHMAN; DEVANBU, 2011). Ao longo deste trabalho, a noção de experiência abordada está vinculada à *expertise*; isto é, a experiência de um desenvolvedor está relacionada ao conhecimento acumulado e à habilidade desenvolvida na execução de tarefas de desenvolvimento.

4.2 Métricas

Um mapeamento sistemático da literatura foi conduzido nesta pesquisa com o intuito de identificar quais métricas são usadas na quantificação da experiência por praticantes e em pesquisas em Engenharia de Software. Esta seção apresentará uma visão geral das métricas a partir do resultado do mapeamento sistemático. As métricas são apresentadas na Tabela A-2 no apêndice A. Todo o mapeamento foi realizado seguindo o guia criado por Kitchenham e Charters (2007), que descreve o protocolo para realizar revisões e mapeamento sistemático da literatura. Segundo Kitchenham e Charters (2007), “o objetivo do mapeamento é fornecer uma visão geral e abrangente dos estudos primários em uma área específica que visa identificar quais evidências estão disponíveis sobre o tópico” (p. vii, tradução nossa). O mapeamento foi realizado considerando trabalhos publicados na língua inglesa.

4.2.1 Mapeamento sistemático

O objetivo do mapeamento sistemático foi analisar como a experiência de desenvolvedores é medida na Engenharia de Software. Até a data da publicação desta dissertação, não se tem conhecimento de revisão ou mapeamento sistemático que aborde o levantamento de métricas para essa finalidade. O levantamento com o objetivo mais similar, é a revisão sistemática de Morales et al. (2019), que aborda diferentes elementos que influenciam a experiência dos desenvolvedores. No entanto, a experiência, nesse contexto, está relacionada à vivência dos desenvolvedores; ou seja, envolve ferramentas e ambientes de desenvolvimento que melhoram as condições de trabalho e produtividade dos desenvolvedores. Para alcançar o propósito deste mapeamento, derivou-se do objetivo as seguintes seis questões de pesquisa (Q):

- Q1: Quais métricas são usadas para medir experiência em Engenharia de Software?
- Q2: Quais tipos de dados são usados para medir a experiência?
- Q3: Quais conceitos aparecem associados à experiência?

- Q4: Quais outros conceitos ou assuntos são abordados em conjunto com experiência?
- Q5: Que tipo de experiência é medida?
- Q6: Qual o objetivo dos estudos que medem a experiência?

A partir do objetivo e das questões de pesquisa, foram definidos os seguintes critérios de inclusão (I) e exclusão (E), para a seleção dos estudos:

- I1: Há uma definição de métrica de experiência de desenvolvedor;
- E1: Não há uma definição de métrica de experiência de desenvolvedor;
- E2: Não há uma definição explícita da métrica;
- E3: Não está na língua inglesa;
- E4: Não pertence ao tópico de Engenharia de Software;
- E5: Não é um artigo de conferência (simpósio, workshop) ou periódico;
- E6: Não é um estudo primário e
- E7: Há uma versão mais atualizada do artigo.

Primeiramente, realizou-se uma pesquisa exploratória, com o objetivo de descobrir os termos com os quais a experiência pode ser referenciada em Engenharia de Software. Os termos que apareceram vinculados, além do próprio termo experiência, são: *expertise*, *knowledge* e *code ownership*.

A partir disso, buscou-se em seis das principais bases de pesquisa digitais (ACM Digital Library, IEEE Xplorer, Science Direct, Web Of Science, Engineering Village e Scopus), considerando o período entre 2010 até abril de 2020, e usando a *string* de busca:

((code **AND** ownership) **OR** experience **OR** knowledge **OR** expertise) **AND**
(metric **OR** measurement **OR** heuristic **OR** measure) **AND** (developer **OR**
contributor) **AND** software

A seleção dos trabalhos foi feita em três etapas, utilizando os critérios de inclusão e exclusão. Quando não era possível definir apenas pelos critérios de inclusão e exclusão, o artigo era aceito e conduzido para a etapa seguinte. As três etapas consistiram em:

- Etapa 1: Leitura de título,

- Etapa 2: Leitura de *abstract*, e
- Etapa 3: Leitura do texto completo.

Realizada a busca, foram devolvidos 990 artigos, dos quais, 271 artigos duplicados. Após a leitura dos títulos correspondente à 1ª etapa, restaram 122 artigos para analisar o *abstract*. Após a leitura do *abstract*, dos 122 artigos, 65 artigos foram rejeitados, restando apenas 57 para a leitura do texto completo. Desses 57, 25 artigos foram rejeitados, segundo os critérios de exclusão. Ao final dessas etapas de filtragem, restaram 32 artigos. A extração de dados ocorreu, simultaneamente, com a terceira etapa; isto é, a extração dos dados foi feita durante a leitura do texto completo. Todas essas etapas foram executadas por apenas um pesquisador (a autora desta dissertação), com discussão de dúvidas com um outro pesquisador (o orientador). A Figura 4.1 mostra a visão geral do processo de seleção de artigos.

Figura 4.1 - Visão geral da seleção de artigos.

Fonte: a autora.

Os dados extraídos foram utilizados para dar uma visão geral dos estudos primários encontrados e para responder às 6 questões de pesquisa do mapeamento. A Tabela 4.1 mostra quais foram os dados coletados em cada artigo selecionado, e para quais questões eles têm relevância no mapeamento.

A Tabela A-1 no Apêndice A traz os estudos selecionados pelo mapeamento, e relaciona cada estudo a um identificador que será utilizado, no decorrer deste capítulo,

Tabela 4.1 - Dados extraídos dos estudos primários selecionados.

#	Dado	Descrição	Questão relevante
D1	Índice	ID do artigo	Visão geral
D2	Título	Título do artigo	Visão geral
D3	Lista de autores	Nomes completos de todos os autores do artigo	Visão geral
D4	Ano	Ano de publicação	Visão geral
D5	Local de publicação	Evento ou periódico onde o artigo foi publicado	Visão geral
D6	Tipo de publicação	Periódico, Conferência, Simpósio ou Workshop	Visão geral
D7	Questões de pesquisa	Questões de pesquisa que o artigo visa responder	Visão geral
D8	Resumo	Conteúdo principal do artigo	Visão geral
D9	Métricas de experiência	A fórmula ou descrição da métrica de experiência	Q1
D10	Dados usados na métrica	Quais os dados usados para determinar a experiência: <i>commits</i> , linhas de código, relatório de <i>bugs</i>	Q2
D11	Conceitos associados à experiência	Quais termos estão relacionados conceitualmente à experiência	Q3
D12	Outros conceitos abordados com experiência	Conceitos não relacionados conceitualmente à experiência, mas que aparecem nos estudos selecionados	Q4
D13	Tipo de experiência	Contexto da experiência medida: projeto ou código, método ou linguagem.	Q5
D14	Objetivo do estudo	Qual o propósito do estudo que usa a métrica de experiência	Q6

para referir-se a cada trabalho. O identificador tem o formato Tn , em que n é o número atribuído ao trabalho no mapeamento.

4.2.1.1 Q1: Quais métricas são usadas para medir experiência em Engenharia de Software?

O resultado do mapeamento mostrou que a experiência de uma pessoa tem sido medida de diversas formas em Engenharia de Software, e que as abordagens mais comuns são determinar a experiência pelo cálculo da proporção de *commits* ou o cálculo da proporção de linhas modificadas entre um desenvolvedor e todos os

demaís. Essas informações são obtidas a partir do uso de sistemas de controle de versão (p.ex.: Git, SVN).

Além de os sistemas de controle de versão armazenarem as versões do código do software, eles também armazenam informações a respeito da atividade nele, a partir de *commits*. Os desenvolvedores fazem alterações em suas versões locais e, quando querem confirmar, realizam um *commit*; e dão um *push*, quando querem enviá-las para a base comum de código. Cada *commit* contém, em resumo, informações sobre o autor, as linhas modificadas e os arquivos modificados; servindo como registro de atividade desse desenvolvedor no projeto.

A Tabela A-2 no Apêndice A traz todas as métricas levantadas para medir a experiência em Engenharia de Software, identificando quais os meios usados por elas para chegar até o valor da experiência (p.ex.: fórmulas, métricas de grafos e atribuição da experiência pela própria pessoa). Algumas métricas foram definidas com valores absolutos, ou seja, desconsiderando no cálculo a parcela de contribuição de outras pessoas; outras métricas, entretanto, foram definidas em relação a outros indivíduos.

4.2.1.2 Q2: Quais tipos de dados são usados para medir a experiência?

Os tipos de dados mais frequentemente usados para medir experiência são contagem de *commits* (13; 27%), contagem de linhas de código modificadas (*code churn* – 11; 23%) e senioridade (8; 16%). Outros dados usados nas métricas foram informações de plataformas de codificação social como: Stack Overflow (p.ex.: reputação) e GitHub (p.ex.: quantidade de repositórios); relatório de erros (*bugs*), comentários em revisão de código, contagem de arquivos modificados e outros (p.ex.: quantidade de tarefas concluídas/mês). A Figura 4.2 mostra os dados que são comumente usados nas métricas de experiência, usadas em Engenharia de Software.

No que concerne aos dados mais utilizados, um argumento para o uso do número de *commits* nas métricas é que “um *commit* representa uma única unidade de esforço para fazer um conjunto de alterações logicamente relacionadas” (JOBLIN et al., 2017, p.166, tradução nossa). No entanto, há várias críticas quanto à validade e à capacidade de se obter informações significativas, quando apenas o número de *commits* é considerado individualmente (JOBLIN et al., 2017; ROBBES; ROTH LISBERGER, 2013). Segundo Robbes e Röthlisberger (2013), um *commit* é apenas o resultado final de uma sessão de desenvolvimento, e não a atividade real

Figura 4.2 - Dados usados para quantificar a experiência.

Fonte: a autora.

do desenvolvedor. A quantidade de operações que um desenvolvedor faz no código, em um *commit*, varia de indivíduo para indivíduo. Por exemplo, um desenvolvedor pode ter um estilo de fazer *commits* frequentemente com pequenas mudanças. Nesse caso, ele seria considerado experiente, pois faz *commits* com muita frequência. Robbes e Röthlisberger (2013) comentam que esses diferentes estilos influenciam nas métricas de experiência. Outro problema que pode haver é não considerar a noção de esquecimento com o tempo, ou seja, alguém que fez muitos *commits* no passado não necessariamente mantém conhecimento e experiência atualmente. Essa questão foi abordada por Hattori, Lanza e Robbes (2012) e Robbes e Röthlisberger (2013). De acordo com Robbes e Röthlisberger (2013), métricas adicionais podem evitar esses problemas.

A contagem do número de linhas modificadas segue o mesmo princípio do número de *commits*. Supõe-se que um desenvolvedor que tenha modificado um grande número de linhas possa ter conhecimento no código. A crítica a essa métrica é porque ela não considera a qualidade técnica do desenvolvedor ou seu estilo. Exemplificando, um bom desenvolvedor pode escrever menos linhas de código que

um desenvolvedor menos experiente, numa mesma tarefa, ao escrever um código mais simples e eficiente. Ainda assim, essa métrica tem tido desempenho satisfatório, como se tem mostrado nos resultados e discussões de diversos trabalhos que consideram o número de linhas modificadas como métrica de experiência (BHATTACHARYA; NEAMTIU; FALOUTSOS, 2014; CONSTANTINOU; KAPITSAKI, 2016; JOBLIN et al., 2017; MOCKUS; HERBSLEB, 2002; OZCAN KINI; TOSUN, 2018; QIU et al., 2015).

Para o propósito desta pesquisa, que inclui quantificar a experiência no projeto e o conhecimento da arquitetura do sistema, a quantidade de arquivos distintos modificados por um desenvolvedor em seus *commits* é também uma métrica interessante. Considerar esse número pode ajudar a representar a importância entre *commits*. Se um desenvolvedor costuma fazer *commits* pequenos, por questão de estilo, ele terá muitos *commits*, mas na prática isso não significa grandes contribuições no código. Ao considerar o número de arquivos distintos nos *commits*, é possível representar melhor o que acontece na prática, e se a contribuição está concentrada ou espalhada ao longo do sistema. Segundo observação em Kini e Tosun (2018), se um desenvolvedor faz um *commit* de muitos arquivos, isso pode indicar grandes alterações na estrutura do sistema.

As principais críticas às métricas baseadas em contagem é que elas não consideram as relações inter-desenvolvedores (JOBLIN et al., 2017) e nem os diferentes papéis que eles possuem, além de capturarem uma noção local da experiência; ou seja, quantificam a experiência a partir de uma atividade específica desconsiderando o conhecimento geral desse indivíduo nas demais atividades e suas interações com outros contribuidores (BHATTACHARYA; NEAMTIU; FALOUTSOS, 2014).

As métricas baseadas em contagem também têm sido bastante utilizadas para o propósito de classificar os desenvolvedores de um projeto, nos papéis de principal ou periférico (JOBLIN et al., 2017). Segundo Joblin et al. (2017), quando a abordagem usada é a contagem considerando a atividade do desenvolvedor, a forma mais comum é calcular o limite ou ponto de corte no percentil 80%. Em outras palavras, os desenvolvedores que tiverem número de *commits* acima desse limite são desenvolvedores principais; os que tiverem número abaixo, serão periféricos.

4.2.1.3 Q3: Quais conceitos aparecem associados à experiência?

Os conceitos que aparecem como sinônimo ou como correlacionados à experiência nos trabalhos selecionados foram: propriedade de código, *expertise* e conhecimento. A Figura 4.3 apresenta a frequência de ocorrência desses termos nos trabalhos selecionados. O termo *expertise* é o mais frequente, aparece em 20 dos trabalhos selecionados, seguido pelo termo conhecimento, com 15 ocorrências. O termo propriedade de código aparece 12 vezes. Essa distribuição mostra que o termo mais comumente associado à experiência é *expertise*, embora conhecimento e propriedade de código também apareçam com frequência relevante, o que mostra que esses termos estão semanticamente ligados nas pesquisas em Engenharia de Software.

Figura 4.3 - Frequência dos termos associados à experiência.

Fonte: a autora.

4.2.1.4 Q4: Quais outros conceitos ou assuntos são abordados em conjunto com experiência?

Outros conceitos ou interesses de estudo abordados em trabalhos em que se mediu a experiência de desenvolvedores foram: mineração de repositórios de software, modelos de predição de defeitos, problemas no código/arquitetura, métricas de processo, atividades e características dos desenvolvedores, alocação de tarefas (recomendação de desenvolvedores), algoritmos e abordagens para estimar *bus factor* (*truck factor*, *lottery number*), identificação de especialistas (em componentes, métodos ou correção de erros), triagem de erros (*bugs*), permanência de desenvolvedores no projeto, disseminação de conhecimento e aprendizado em

desenvolvimento de software. A ocorrência de cada um deles nos trabalhos selecionados pode ser vista na Tabela 4.2.

Tabela 4.2 - Conceitos nos trabalhos selecionados.

Termo	Trabalho(s)
Mineração de repositórios de software	[T13], [T21], [T26] e [T28]
Modelos de predição de defeitos	[T1] e [T32]
Problemas no código/arquitetura (<i>code smells</i> , violações arquiteturais)	[T21]
Métricas de processo	[T1], [T7] e [T30]
Atividades e características dos desenvolvedores	[T1], [T3], [T4], [T11], [T13] e [T24]
Alocação de tarefas (recomendação de desenvolvedores)	[T6], [T15] e [T19]
Algoritmos e abordagens para estimar <i>bus factor</i> (<i>truck factor</i> , <i>lottery number</i>)	[T8]
Identificação de especialistas (em componentes, métodos ou correção de erros)	[T8], [T9] e [T28]
Triagem de erros (<i>bugs</i>)	[T9], [T12] e [T19]
Permanência de desenvolvedores no projeto	[T15] e [T16]
Disseminação de conhecimento	[T15]
Aprendizado em desenvolvimento de software	[T14], [T17], [T20], [T26] e [T29]

4.2.1.5 Q5: Que tipo de experiência é medida?

Os tipos de experiência medidas nos trabalhos selecionados foram: em projeto (não somente no código), em Engenharia de Software (e em seus processos), em programação (escrever código em várias linguagens de programação), em correção de erros, em linguagens de programação específicas e em código (de um projeto, componente, arquivo ou método). A maioria das métricas (21; 62%) media a experiência em código, seguida pela experiência em correção de erros (4; 12%) e a experiência em programação (4; 12%), duas (6%) métricas quantificavam a experiência em linguagens de programação específicas e duas (6%) experiência em um projeto de software. Apenas uma (3%) métrica quantificava a experiência em Engenharia de Software (e seus processos). A Tabela 4.3 exhibe os trabalhos associados a cada tipo de experiência, e o gráfico da Figura 4.4 mostra as porcentagens de trabalhos por tipo de experiência.

Tabela 4.3 - Trabalhos classificados por tipo de experiência.

Tipo de experiência	Trabalho(s)
Projeto	[T18] e [T29]
Engenharia de Software	[T14]
Programação	[T11], [T17], [T20] e [T24]
Correção de erros	[T5], [T9], [T12] e [T19]
Linguagem	[T2] e [T6]
Código	[T1], [T2], [T3], [T4], [T5], [T7], [T8], [T10], [T13], [T15], [T16], [T21], [T22], [T23], [T25], [T26], [T27], [T28], [T30], [T31] e [T32]

Figura 4.4 - Métricas por tipo de experiência.

Fonte: a autora.

4.2.1.6 Q6: Qual o objetivo dos estudos que medem a experiência?

Foram identificados, inicialmente, a partir da extração dos dados (D14), cinco tipos de objetivos de estudos: investigação de qualidade de software, nova abordagem de métrica, novos modelos de predição de defeitos, fatores humanos (e qualidade do processo/software) e investigar a degradação da arquitetura. Os objetivos que apareceram apenas uma vez foram agrupados na categoria “Outros”; com exceção do objetivo de investigar a degradação da arquitetura, que foi ressaltado, pois também corresponde ao objetivo deste trabalho. A Figura 4.5 mostra a quantidade de estudos para cada tipo de objetivo identificado no mapeamento.

Figura 4.5 - Objetivo dos estudos com métricas de experiência.

Fonte: a autora.

A maioria dos 32 estudos selecionados (15; 38%) tinha como objetivo apresentar novas abordagens de métrica. Nove (23%) correspondiam a estudos que usaram métricas de experiência em estudos que investigavam a qualidade de software (p.ex.: por meio de defeitos). Quatro estudos (10%) usaram as métricas para estudar modelos de predição de defeitos, que são usados, dentre outras coisas, para estimar custos de versões a serem lançadas. Três estudos (8%) tinham como objetivo investigar fatores humanos (características dos desenvolvedores) em desenvolvimento de software (p.ex.: perfil de quem fica em projetos OSS). Apenas um estudo (3%) usou alguma métrica de experiência (*code churn*), com o objetivo de estudar seu impacto na degradação da arquitetura. Outros sete estudos (18%) correspondem a “outros” tipos de objetivo. Desses outros, por exemplo, há um trabalho que apresenta a forma de resumir a atividade de desenvolvedores, a partir das suas informações de atividades no GitHub (TREUDE; FIGUEIRA FILHO; KULESZA, 2015), ou da investigação empírica do efeito de diferentes padrões de edição em arquivos na qualidade do software (ZHANG et al., 2014).

4.3 Considerações finais

A experiência de um desenvolvedor pode ser associada ao conhecimento acumulado ao longo das atividades que ele desempenha, durante o desenvolvimento de um software. O conhecimento pode ser classificado como episódico, quando pode ser adquirido a partir da experiência e da prática; ou conhecimento tópico, quando pode ser adquirido por meio de livros e cursos. A experiência em um projeto e em seu código está diretamente relacionada ao conhecimento episódico, uma vez que ter conhecimento ou domínio teórico em uma linguagem de programação, ou em Engenharia de Software, não implica necessariamente ser experiente no domínio do sistema.

A forma mais comum de quantificar a experiência no código, nos trabalhos selecionados pelo mapeamento sistemático, é por meio da contagem das atividades de um desenvolvedor. Isso é possível porque se utiliza sistemas de controle de versão (como, por exemplo, o Git) para registrar a atividade no código e para regular as contribuições de cada desenvolvedor. Alguns trabalhos consideram como métricas para propriedade de código a porcentagem de contribuição dos desenvolvedores em um componente do sistema (BIRD et al., 2011; FOUCAULT; FALLERI; BLANC, 2014; GREILER; HERZIG; CZERWONKA, 2015b). Exemplificando, em Bird et al. (2011), a noção de contribuição está relacionada a *commits* no código, pois entende-se que os arquivos envolvidos foram alterados pelo desenvolvedor. Essa mesma forma de quantificar a propriedade de código foi usada por Foucault, Falleri e Blanc (2014) e Greiler, Herzig e Czerwonka (2015b) em seus trabalhos que visavam replicar os resultados de Bird et al. (2011), em outros contextos.

Neste trabalho, a experiência é analisada em relação à degradação da arquitetura. Com isso, no capítulo seguinte, se discute a degradação da arquitetura apresentando sua definição, os sintomas de problemas arquiteturais em um sistema, as prováveis causas da degradação e, por fim, as métricas para avaliar a arquitetura.

5 DEGRADAÇÃO DA ARQUITETURA

Assim como os seres humanos, sistemas de software passam pelo processo de envelhecimento. Isso pode ser causado pela ausência de atualizações das funcionalidades do software, ou pelo aumento da complexidade de suas estruturas (PARNAS, 1994). Quando as estruturas ficam mais complexas, tem-se o comprometimento da qualidade do software, porque fica mais difícil compreender as relações entre componentes e, assim, torna-se mais difícil incorporar novos requisitos e corrigir erros (MARTIN, 2017). Além disso, os custos e esforços associados à manutenção do sistema tornam-se elevados, a ponto do processo de desenvolvimento não ser mais sustentável (VENTERS et al., 2018). A esse processo de aumento de complexidade da arquitetura de um sistema, dá-se o nome de degradação da arquitetura. Trata-se de um processo inevitável, mas entendê-lo pode limitar seus efeitos e reduzir os danos causados, aumentando o tempo de vida do software.

Sistemas de software têm suas funcionalidades atualizadas, dentre outras razões, devido à experiência de uso dos usuários, que traz à tona novas necessidades, correção de erros, mudanças do ambiente e mudanças no contexto de uso. Essas atualizações provocam alterações que podem, entre outras coisas, aumentar a complexidade do código (CANFORA et al., 2014), ou introduzir problemas arquiteturais (BANDI; WILLIAMS; ALLEN, 2013). Tais problemas podem emergir a partir de violações da arquitetura planejada (DE SILVA; BALASUBRAMANIAM, 2012; HOCHSTEIN; LINDVALL, 2005; MACIA et al., 2012; MARTINI; BOSCH; CHAUDRON, 2014; PERRY; WOLF, 1992), mas não somente por conta disso; também podem vir por meio de anomalias arquiteturais (CAI; KAZMAN, 2017; FONTANA et al., 2017a; GARCIA et al., 2009a; LE et al., 2016; VENTERS et al., 2018), que, por sua vez, também podem ser responsáveis por aumentar a complexidade do sistema (LE et al., 2016). Todos esses problemas podem comprometer propriedades de qualidade do software, a exemplo de: eficiência de desempenho, manutenibilidade, portabilidade e confiabilidade (AZADI; FONTANA; TAIBI, 2019); e até contribuir para o fim da vida do software (VENTERS et al., 2018).

As violações da arquitetura introduzem inconsistências entre a arquitetura planejada e a arquitetura implementada, a partir do desrespeito às regras arquiteturais estabelecidas (EICK et al., 2001). Elas dificultam as atividades de manutenção do

software, porque, além de nem sempre serem documentadas, frequentemente, correspondem a atalhos de desenvolvimento que diminuem a clareza da arquitetura, tornando mais difícil para os mantenedores entenderem as relações entre componentes (EICK et al., 2001). Anomalias de código também podem estar relacionadas a problemas arquiteturais, mas, às vezes, seus efeitos podem estar restritos ao nível de implementação ou, até mesmo, apenas servir de indício para problemas de *design* (FOWLER, 1999). Algumas pesquisas (EICK et al., 2001; FERREIRA et al., 2014; MACIA et al., 2012) buscaram entender o relacionamento entre problemas de código e problemas arquiteturais, para ajudar na definição de estratégias de manutenção, tal qual a refatoração. O equivalente a anomalias de código, mas no nível da arquitetura, são as anomalias arquiteturais, que “podem derivar de decisões arquiteturais comumente utilizadas, intencionais ou não, e que impactam negativamente a qualidade interna do software” (FONTANA et al., 2017a, p. 433, tradução nossa).

Vários termos têm sido usados para designar problemas arquiteturais. Perry e Wolf (1992) introduziram a ideia de erosão e desvio arquitetural. Segundo eles, erosão ocorre por causa de violações na arquitetura; e desvio, devido à “insensibilidade sobre a arquitetura” (PERRY; WOLF, 1992), descrita por Van Gorp e Bosch (2002) como a existência de regras arquiteturalmente implícitas, que não são claras para as pessoas que vão trabalhar com a arquitetura. Hochstein e Lindvall (2005) falam em degeneração para descrever quando a estrutura de um sistema desvia do que foi planejado, de maneira descontrolada, a tal ponto, que o sistema não é mais sustentável. Macia et al. (2012) chamam de degradação o mesmo fenômeno. Outro termo popular é dívida técnica arquitetural (LI; AVGERIOU; LIANG, 2015), que, no entanto, não se restringe a descrição de desvio da arquitetura. O termo é usado para designar “decisões arquiteturais que comprometem alguns aspectos de qualidade internos, como a capacidade de manutenção” (LI; AVGERIOU; LIANG, 2015, p. 201, tradução nossa).

A definição de *degradação da arquitetura*, a ser considerada ao longo deste trabalho, leva em conta todas as discussões em torno desses termos e considera que a arquitetura está degradando, quando princípios de *design* (p.ex.: abstração, modularidade e encapsulamento) que a caracterizam são violados, a ponto de comprometer as propriedades de qualidade do software; que segundo a norma

ISO/IEC 25010 (2011) são: adequação funcional, confiabilidade, eficiência de desempenho, operabilidade, segurança, compatibilidade, manutenibilidade e portabilidade. Portanto, essa definição de degradação da arquitetura engloba como problemas arquiteturais: anomalias arquiteturais e acúmulo de dívida técnica arquitetural, entre outros, quando eles contribuem para prejudicar o alcance das características de qualidade desejáveis para o software.

Este capítulo apresenta sintomas de problemas arquiteturais: anomalias arquiteturais (Seção 5.1) e dívida técnica (Seção 5.2). Também se discute possíveis causas para o surgimento de problemas arquiteturais (Seção 5.3) e as métricas utilizadas para indicar a qualidade da arquitetura de software.

5.1 Anomalias arquiteturais

A presença de anomalias de código não é suficiente para ser indício de degradação arquitetural, porque muitas anomalias podem não causar problemas estruturais (MACIA et al., 2012). Algumas se limitam a problemas com relação a implementação (métodos, parâmetros e classes) (MACIA et al., 2012), que não precisam do conhecimento das relações arquiteturais entre componentes, para que sejam corrigidas, apenas da aplicação de boas práticas de programação com relação a código (LE et al., 2018). Alguns trabalhos, como o de Ferreira et al. (2014) e de Oizumi et al. (2016), relacionam a ocorrência de algumas anomalias de código ao surgimento de problemas arquiteturais. Porém, Le et al. (2018) e Macia et al. (2012) afirmam que essa relação não é direta e, portanto, a ocorrência de anomalias de código não é o melhor indicativo de degradação da arquitetura.

Por outro lado, as anomalias arquiteturais não podem ser corrigidas simplesmente operando sobre classes e seus métodos. Elas são resultado de más decisões arquiteturais e envolvem o uso equivocado de abstrações no nível arquitetural, ou seja, componentes, conectores, padrões e estilos e, além disso, são independentes dos interesses de processo e organizacionais e abordam concretamente a estrutura interna e o comportamento dos sistemas (GARCIA et al., 2009b). E, assim como as anomalias de código, elas também impactam negativamente propriedades de qualidade do software. Na definição dada por Garcia et al. (2009b), não se considera como anomalia o desvio arquitetural porque uma

arquitetura concreta desviada pode melhorar a manutenibilidade, ainda que viole sua arquitetura pretendida. Isso pode ocorrer no caso de a arquitetura planejada conter elementos de *design* ruins e a arquitetura concreta implementar soluções melhores.

Garcia et al. (2009b) foi um dos primeiros trabalhos a tentar definir e catalogar anomalias arquiteturais. Nele, apresentaram-se quatro tipos de anomalias arquiteturais, sendo uma contribuição importante para que trabalhos seguintes pudessem criar estratégias de detecção automática para essas anomalias. Os autores as descrevem em termos de componentes e conectores, que são elementos arquiteturais definidos em Shaw et al. (1995), porque utilizando unidades de construção da arquitetura, como esses elementos, é possível identificar as anomalias ainda que não se conheça o histórico do sistema. Em seguida, Le et al. (2016) propuseram uma classificação em quatro categorias de anomalias: baseada em interface, baseada em mudança, baseada em dependência e baseada em interesses. Além disso, propuseram a relação de impacto de cada anomalia com algumas das características de qualidade. A partir da evolução do entendimento de anomalias, incluindo a definição e classificação delas, bem como, o tipo de característica de qualidade que elas impactavam, foi possível desenvolver estratégias de detecção que começaram a ser incorporadas em ferramentas de avaliação da qualidade da arquitetura (BEHNAMGHADER et al., 2017a). Em uma contribuição mais recente, (AZADI; FONTANA; TAIBI, 2019) catalogam as anomalias arquiteturais que podem ser detectadas por algumas ferramentas específicas existentes. Eles trazem, também, a análise de um conjunto de ferramentas disponíveis para a detecção de anomalias e descrição da técnica de detecção empregada.

5.2 Dívida técnica

Outra forma de falar sobre degradação da arquitetura é por meio da metáfora da dívida técnica (SEAMAN; GUO, 2011), que utiliza a noção de dívida financeira. O termo foi introduzido na indústria de software por Cunningham em 1992:

A primeira vez que a qualidade de código é comprometida é como entrar em dívida. Uma pequena dívida acelera o desenvolvimento contanto que esse débito seja pago [...]. O risco ocorre quando a dívida não é reembolsada.

Cada minuto gasto com código de baixa qualidade implica em juros sobre essa dívida. (CUNNINGHAM, 1992, p. 30, tradução nossa).

Desde então, conforme observaram Seaman e Guo (2011) e Rubin (2012), o sentido da metáfora vem sendo ampliado e, hoje, corresponde não somente a atalhos com relação à implementação de código como, também, a problemas oriundos de deficiências no processo de desenvolvimento; por exemplo, *design* ruim ou inadequado para as necessidades do produto, cobertura de testes insuficiente e defeitos no software. Uma pequena dívida até pode acelerar o desenvolvimento, mas o acúmulo tende a provocar o efeito inverso (SEAMAN; GUO, 2011).

Há casos específicos em que a dívida técnica não tem necessariamente que ser paga, podendo, inclusive, beneficiar o projeto (GUO; SEAMAN; DA SILVA, 2016). Para exemplificar, considera-se o caso de um módulo ter sido desenvolvido rapidamente e com alguma complexidade, para atender uma demanda específica do usuário, mas raramente haver a necessidade de modifica-lo. Nos demais casos, conforme se acumula, aumentam os problemas com manutenibilidade (LI; AVGERIOU; LIANG, 2015); sendo difícil realizar a manutenção na estrutura existente e acomodar novas funcionalidades. Quando isso ocorre, a tendência é a diminuição da velocidade de desenvolvimento (RUBIN, 2012).

Para lidar com a dívida técnica, é preciso identificá-la e quantificá-la, para que a dívida possa ser paga (SEAMAN; GUO, 2011). Por itens de dívida técnica, entende-se: módulos que precisam ser refatorados, testes que precisam ser escritos, inspeções ou revisões que precisam ser feitas e documentação que precisa ser escrita ou atualizada. No caso de dívida técnica arquitetural, além de questões de conformidade arquitetural, que precisam ser resolvidas, decisões arquiteturais que comprometem características de qualidade também são consideradas itens de dívida técnica (LI; AVGERIOU; LIANG, 2015).

Devido ao sentido amplo que a metáfora adquiriu, com o passar dos anos, identificar dívida técnica em um sistema pode ser um desafio. Tom, Aurum e Vidgen (2013) observaram que a dívida técnica tem sido classificada em dimensões como: dívida de código, dívida de *design*, dívida arquitetural e dívida de teste; para citar alguns. Ela também tem sido classificada em tipologias: dívida estratégica, dívida tática, dívida incremental e dívida não intencional (TOM; AURUM; VIDGEN, 2013). Segundo Ernst et al. (2015), questões arquiteturais estão entre as principais fontes de

dívida técnica, transformando o uso dessa metáfora em um recurso popular, para tratar de tópicos relacionados a problemas arquiteturais.

5.3 Causas

Esta seção discute fatores que podem ser responsáveis pela degradação da arquitetura. Os fatores (ou causas) podem estar associados a diversos elementos do processo de desenvolvimento de software, tais como: documentação da arquitetura, requisitos, desenvolvedores e questões sobre o processo de desenvolvimento em si. Para se chegar às causas, foi realizado um levantamento bibliográfico, consultando os tópicos: degradação da arquitetura, desvio arquitetural, erosão arquitetural e dívida técnica. Além dos trabalhos devolvidos em buscas em bases digitais, como *ACM Digital Library* e *IEEE Explorer*, outros artigos foram consultados a partir da leitura dos primeiros. Um resumo dessas causas é apresentado na Tabela 5.1. Os fatores serão apresentados, mais adiante, em tópicos que representam cada um dos elementos do processo de desenvolvimento ao qual estão relacionados.

5.3.1 Documentação da arquitetura

Por ser importante para servir de comunicação entre os *stakeholders* e como base para análise e construção do sistema (BASS; CLEMENTS; KAZMAN, 2012), a ausência ou falta de atualização da documentação da arquitetura podem trazer problemas ao processo de desenvolvimento.

Os responsáveis pela documentação podem ser os desenvolvedores, ou, quando é um requisito contratual, pessoas que conhecem um pouco do sistema e que foram contratadas especificamente para este fim (PARNAS, 1994). Desenvolvedores conhecem bastante sobre o sistema que estão implantando, mas a qualidade da documentação produzida por eles depende da política adotada pelo gerente e pela organização/processo, e do tipo de notação utilizada. No caso de a documentação ser feita por não-desenvolvedores, a notação utilizada pode não ser compreendida pelos desenvolvedores na fase de manutenção. Como os desenvolvedores que fazem a manutenção do software, muitas vezes, não estiveram envolvidos na concepção do projeto original, o fato de a documentação estar desatualizada, com

Tabela 5.1 - Causas da degradação da arquitetura.

Causas	Descrição
Documentação	Falta de documentação de princípios e decisões de <i>design</i> (PARNAS, 1994). Prejudica principalmente quando mudanças precisam ser feitas no software
	Documentação pobremente escrita ignorada pelos desenvolvedores encarregados pela manutenção do sistema (PARNAS, 1994).
	Falta de documentação da arquitetura pretendida e o fato de muitos dos envolvidos na concepção da arquitetura não permanecerem no projeto (LENHARD et al., 2017; MACIA et al., 2012).
Requisitos	Requisitos imprecisos podem fazer os desenvolvedores escreverem códigos que terão grande chance de serem modificados futuramente (EICK et al., 2001).
	Requisitos mal definidos no início pela falta de conhecimento do negócio (ERNST et al., 2015; MARTINI; BOSCH; CHAUDRON, 2015).
	Requisitos conflitantes que não foram previstos nos estágios iniciais do desenvolvimento (DIMECH; BALASUBRAMANIAM, 2013).
	Mudanças de requisitos (MAIR; HEROLD, 2013) aliados com falta de verificação de conformidade entre a arquitetura pretendida e a arquitetura concreta durante o desenvolvimento podem levar a desvios, que trazem dificuldades ao introduzir modificações posteriormente (ALI et al., 2017; DIMECH; BALASUBRAMANIAM, 2013).
	Falta de especificação ou ênfase em requisitos arquiteturais críticos (MARTINI; BOSCH; CHAUDRON, 2015).
Desenvolvedores	Pessoas que não entendem o <i>design</i> original podem fazer mudanças inconsistentes com o conceito original (PARNAS, 1994).
	Desenvolvedores não conseguem entender ou alterar um código complexo, escrito por seus colegas mais qualificados (EICK et al., 2001).
	Desenvolvedores nem sempre estão cientes dos impactos arquiteturais de suas soluções (PAIXAO et al., 2017).
	Inexperiência: pessoas novas na equipe estão mais sujeitas a acumular dívida técnica arquitetural, isto é, contribuir para a degradação da arquitetura, devido à falta de compreensão sobre a arquitetura e padrões (MARTINI; BOSCH; CHAUDRON, 2015).
	Rotação de pessoas na equipe de desenvolvimento e, principalmente, saída de pessoas do núcleo de desenvolvimento (FERREIRA; VALENTE; FERREIRA, 2017).
Processo de desenvolvimento	Processo de desenvolvimento que não adota a arquitetura modularizada, ou seja, não se tem a preocupação de tentar isolar as partes que têm maior probabilidade de mudanças em pequenos trechos de código. Logo, mudanças vão alterar várias partes do código e isso aumenta a chance de inserir erros (PARNAS, 1994).
	Métodos iterativos. A fase de projeto da arquitetura é geralmente utilizada para fazer o <i>design</i> tentando antecipar mudanças futuras e nos métodos iterativos, isso acontece a cada iteração (VAN GURP; BOSCH, 2002).
	A priorização de características acima da preocupação com a arquitetura do sistema (MARTINI; BOSCH; CHAUDRON, 2015).
	Desenvolvimento paralelo: pessoas diferentes trabalham ao mesmo tempo em partes diferentes do código. Isso é suficiente para o acúmulo de diferenças entre suas decisões de <i>design</i> (MARTINI; BOSCH; CHAUDRON, 2015).

notação inapropriada, ou não haver documentação relatando a arquitetura planejada. faz com que as soluções implementadas possam piorar a qualidade de código e as escolhas arquiteturais (LENHARD et al., 2017; MACIA et al., 2012; PARNAS, 1994).

Em projetos *open source*, a documentação da arquitetura é feita de maneira distinta de projetos comerciais com código fechado (KAZMAN et al., 2016). É mais comum que discussões sobre a arquitetura estejam presentes em meios como e-mails, sistemas de controle de mudança e sistemas de controle de versão. E, raramente, esses projetos se apoiam em descrições detalhadas quanto ao *design* do sistema, ou aos planos e cronogramas do projeto (KAZMAN et al., 2016). As informações arquiteturais em projetos *open source* podem ser obtidas por vias como listas de e-mails, blogs, fóruns, comentários no código e dados de *commit* (KAZMAN et al., 2016; ØSTERLUND; CROWSTON, 2013). Pessoas com papéis e nível de envolvimento distintos devem ter acesso à documentação com nível de detalhamento condizente (ØSTERLUND; CROWSTON, 2013). Desse modo, para pessoas com pouco conhecimento no código, é desejável ter documentação mais explícita, já que eles não têm o *background* para entender apenas informações mais superficiais. De acordo com Ding et al. (2014), apenas 5,4% dos projetos *open source* tem algum tipo de documentação arquitetural.

5.3.2 Requisitos

Nos estágios iniciais de desenvolvimento, há um grau maior de incertezas sobre o domínio e sobre as reais necessidades do usuário final, o que faz com que requisitos imprecisos, conflitantes e mal definidos sejam levantados podendo levar os desenvolvedores a adotarem soluções com grande chance de serem modificadas futuramente (EICK et al., 2001; ERNST et al., 2015). Além disso, no caso de requisitos imprecisos e mal formulados, isso pode resultar em implementações que não foram solicitadas ou que não atendem ao que os usuários desejam. No caso de requisitos conflitantes, eles podem impossibilitar a implementação de funcionalidades e o alcance de características de qualidade. Esses problemas podem emergir por causa de um processo de engenharia de requisitos inadequado (POHL; RUPP, 2012).

No início do desenvolvimento, é difícil ter clareza sobre as regras de negócio e sobre o comportamento desejado pelo usuário final. Certas escolhas, nesse estágio,

podem tornar a arquitetura inapropriada futuramente. A escolha de um padrão arquitetural pode inviabilizar mudanças que serão necessárias em uma etapa de maior maturidade de conhecimento sobre o negócio e sobre o próprio software em construção. Nesse contexto, a arquitetura pode não permitir mudanças ou abstrações exigidas pelo sistema, pois ela não consegue incorporar novos requisitos que não foram pensados nas fases iniciais de levantamento de requisitos (DE SILVA; PERERA, 2015; EICK et al., 2001; MERKLE, 2010). Em geral, requisitos não-funcionais são críticos para a arquitetura (segurança, confiabilidade, portabilidade, manutenibilidade, etc.) e costumam ser negligenciados, com frequência, quando se trata de tomar decisões arquiteturais (CAO; RAMESH, 2008; MARTINI; BOSCH; CHAUDRON, 2015).

Além disso, processos de desenvolvimento iterativos não planejam a arquitetura pensando em antecipar mudanças futuras na mesma intensidade que processos orientados a plano, pois o planejamento ocorre a cada iteração (VAN GURP; BOSCH, 2002). Sendo assim, a probabilidade de uma decisão arquitetural inviabilizar uma outra, feita em uma iteração anterior, é maior.

Em projetos *open source*, a elicitação de requisitos pode não ser conduzida como nos processos clássicos de engenharia de requisitos. De acordo com Sacchi (2002), os requisitos podem aparecer em mensagens de e-mail e em tópicos de discussão em listas de e-mail ou fóruns. Os requisitos, nesse tipo de sistema, emergem da experiência dos participantes da comunidade, a partir de suas experiências pessoais com o software (SCACCHI, 2002).

5.3.3 Desenvolvedores

O nível de envolvimento, autonomia e responsabilidade de um desenvolvedor, em um projeto, depende de alguns fatores; dentre eles, o papel atribuído a ele (p.ex.: líder, quem corrige defeitos, quem faz aprimoramentos) (JOBLIN et al., 2017) e a política de desenvolvimento adotada (p.ex.: propriedade forte de código, propriedade coletiva de código) (BIRD et al., 2011).

No entanto, de uma forma geral, desenvolvedores fazem alterações no código, levando em conta o modelo conceitual da arquitetura que têm em mente, por isso, pessoas que não entendem o *design* podem fazer mudanças inconsistentes com o

conceito original (PARNAS, 1994). E mesmo que entendam o *design*, nem sempre eles estão cientes das mudanças estruturais que a solução adotada por eles pode ocasionar (PAIXAO et al., 2017). Outra causa pode ser a dificuldade do desenvolvedor em entender ou alterar um código delicado e complexo escrito por seus colegas mais qualificados (EICK et al., 2001). Essa dificuldade pode se dar por inexperiência de indivíduos novos no time de desenvolvimento, uma vez que eles estão mais sujeitos a optar por decisões limitadas devido à dificuldade em compreender soluções adotadas na arquitetura (MARTINI; BOSCH; CHAUDRON, 2015). Esse é o caso que é investigado nesta pesquisa.

Outro motivo relacionado aos desenvolvedores, é a saída de pessoas do projeto e, principalmente, a saída de pessoas que fazem parte do núcleo de desenvolvimento (FERREIRA; VALENTE; FERREIRA, 2017; SCACCHI, 2002), uma vez que é mais provável que essas pessoas façam parte das decisões arquiteturais (JOBLIN et al., 2017).

Em projetos *open source*, em especial os bem-sucedidos, é incomum que desenvolvedores com baixa capacidade técnica possam escrever código (JOBLIN et al., 2017). Ainda assim, quando eles são novatos no projeto, o grau de experiência nele pode interferir nas soluções adotadas; embora eles tenham suas soluções avaliadas e revisadas por pessoas mais experientes, e tenham a seu dispor boas práticas de desenvolvimento, como testes e integração contínua, para garantir que suas contribuições não vão quebrar o código.

5.3.4 Processo de desenvolvimento

As abordagens de desenvolvimento de software podem variar quanto à ênfase dada a cada atividade de desenvolvimento, à ordem na qual as atividades devem ser executadas, quanto aos artefatos produzidos em cada atividade, e quanto ao papel das pessoas no processo. Durante o desenvolvimento, os principais problemas que emergem são a dificuldade de cumprir prazos, gerenciar a dívida técnica assumida e a priorização de características, ao invés da construção e manutenção de uma arquitetura de qualidade.

Prazos pressionam os desenvolvedores a assumirem dívida técnica e, portanto, a qualidade das soluções adotadas pode piorar. Além disso, na falta de tempo, o

gerenciamento da dívida técnica arquitetural é deixado de lado (MARTINI; BOSCH; CHAUDRON, 2015) e os desenvolvedores escrevem código de baixa qualidade, sem entender seus efeitos na estrutura existente (EICK et al., 2001). Uma outra causa de degradação é a prioridade dada, por alguns processos, à entrega de funcionalidade, ao invés da preocupação com a qualidade da arquitetura que vem sendo desenvolvida (MARTINI; BOSCH; CHAUDRON, 2015). Alguns processos de desenvolvimento possuem característica de desenvolvimento paralelo, em que os indivíduos trabalham ao mesmo tempo em partes diferentes de código, acumulando diferenças entre os modelos conceituais sobre a arquitetura que eles têm em mente (MARTINI; BOSCH; CHAUDRON, 2015). Essas situações são típicas de métodos ágeis e desenvolvimento *open source*, embora possam estar presentes em outros métodos de desenvolvimento.

Nas etapas posteriores de desenvolvimento, quando a demanda por novas funcionalidades e pela correção de erros tomam boa parte do tempo, as tarefas tornam-se mais consumidoras de esforço. Isso acontece em consequência do aumento da complexidade, devido à evolução natural do sistema, e leva os desenvolvedores a tomar decisões piores. Essas decisões podem ser motivadas pela compreensão dessas pessoas a respeito da arquitetura, ou porque soluções melhores consomem mais tempo do que se dispõe (VAN GURP; BOSCH, 2002).

5.4 Métricas

Existem diferentes formas de avaliar uma arquitetura. Segundo Lindvall, Tvedt e Costa (2003), a avaliação de uma arquitetura pode ser tomada em diferentes pontos do desenvolvimento e com diferentes objetivos. Exemplificando, pode ser feita antes do início do desenvolvimento, para avaliar arquiteturas candidatas para determinados objetivos de qualidade; ou pode ser realizada pós-implementação do sistema. No primeiro caso, utilizam-se dados qualitativos como entrevistas, *workshops* e análise de cenários para determinar se uma arquitetura é adequada para a finalidade pretendida pelo software. Exemplos de métodos para a avaliação de arquitetura nesse estágio, são: ATAM (*Architecture Tradeoff Analysis Method*) (BASS; CLEMENTS; KAZMAN, 2012) e CBAM (*Cost Benefit Analysis Method*) (CLEMENTS; KAZMAN; KLEIN, 2002). No segundo caso, a avaliação pós-implementação, utiliza dados

quantitativos e é usado para comparar versões diferentes da arquitetura, ou para determinar se determinadas características de qualidade ainda estão sendo satisfeitas. São utilizadas métricas, a partir de dados medidos no sistema já implementado, os quais são geralmente coletados automaticamente.

É difícil definir métricas para dizer se uma arquitetura é/está boa o suficiente. Li et al. (2014) afirmam que um dos motivos é que a arquitetura está em um nível de abstração mais alto que o código, talvez, por isso, seja mais comum que a análise de qualidade de software seja feita baseada em métricas de complexidade e qualidade de código, ao invés de métricas que avaliem diretamente a arquitetura. Alguns trabalhos buscam estabelecer alguma relação entre métricas de qualidade de código (p.ex.: tamanho, complexidade, acoplamento e estabilidade) e problemas arquiteturais, como alternativa para avaliar indiretamente a arquitetura (LENHARD; BLOM; HEROLD, 2019; LI et al., 2014; MACIA et al., 2012b); porém, existe uma tendência a sugerir novas métricas e ferramentas para este fim (AZADI; FONTANA; TAIBI, 2019; COULIN et al., 2019; VENTERS et al., 2018). Outra tendência é utilizar métricas de coesão e acoplamento em associação direta à qualidade da arquitetura, porque elas acabam refletindo características estruturais e podem ser extraídas por ferramentas estáticas de código (FREGNAN et al., 2019).

Vários fatores parecem influenciar na ausência de uma métrica ou conjunto de métricas que se sobressaíam como instrumentos de medição da qualidade de uma arquitetura. Além de a arquitetura estar em um nível de abstração alto, o que deve ser medido também vai depender da perspectiva a ser observada; que pode ser qualquer uma das características de qualidade: eficiência de desempenho, confiabilidade, segurança, portabilidade, manutenibilidade etc. (LINDVALL; TVEDT; COSTA, 2003). Dessa forma, se o objetivo for avaliar se a arquitetura degradou sob a perspectiva da eficiência de desempenho, métricas de complexidade devem ser mais relevantes (COULIN et al., 2019). Se for o caso de observar se a arquitetura degradou sob a perspectiva de manutenibilidade, as métricas de coesão e acoplamento podem ser mais adequadas (COULIN et al., 2019; LINDVALL; TVEDT; COSTA, 2003).

Um caminho tem sido correlacionar métricas com relação a código, as quais já são conhecidas e utilizadas, e que são obtidas por meio da relação entre os elementos de implementação (classes, pacotes), como indício de problemas arquiteturais. Li et al. (2014) verificaram que algumas métricas de modularidade, calculadas com base

no código-fonte, tais como Índice de Uso Entre Pacotes (IIPU) e Índice de Extensão Entre Pacotes (IIPE), têm correlação significativa com a qualidade da arquitetura e, mais especificamente, podem estar associadas com a presença de dívida técnica arquitetural. Fontana et al. (2017a), Garcia et al. (2009b) e Le et al. (2016) usaram as métricas de Martin (2017) e métricas com respeito a classe, como CBO e LCOM, para detectar a presença de anomalias arquiteturais. A vantagem dessa abordagem é que as anomalias arquiteturais são padrões problemáticos, que são comumente encontrados em sistemas, e, por isso, são fáceis de serem observadas por desenvolvedores e arquitetos; além disso, podem ser detectadas automaticamente a partir de métricas de código e de relacionamentos entre componentes da arquitetura (GARCIA et al., 2009b). Tem-se tentado, também, alternativas como a criação de índices para quantificar a qualidade da arquitetura, os quais podem ser calculados baseados nos caminhos anteriores, tal como a presença de anomalias arquiteturais (ROVEDA et al., 2018). E há a alternativa de combinar métricas (VENTERS et al., 2018), assim como a Densidade de Anomalias Arquiteturais (ASD, de *Architectural Smells Density*) (LE et al., 2016), que quantifica o quão grande é o número de anomalias detectadas relativo ao número de componentes do sistema (LE et al., 2016).

De acordo com Coulin et al. (2019) e Venters et al. (2018), dentre as métricas popularmente utilizadas para avaliar a qualidade da arquitetura estão as de coesão e acoplamento. Geralmente, elas são utilizadas de forma complementar, utilizando a noção de que um bom *design* arquitetural tem alta coesão e baixo acoplamento (CHIDAMBER; KEMERER, 1994). Na prática, isso indica boa manutenibilidade, porque haverá poucas mudanças entre classes pertencentes a componentes diferentes. A redução do acoplamento diminui o efeito cascata entre componentes e classes; isto é, mudanças secundárias, em decorrência de mudanças primárias, tornando mais fácil a manutenção do sistema (LINDVALL; TVEDT; COSTA, 2003). Em outras palavras, isso permite que desenvolvedores trabalhem paralelamente e que precisem de menos conhecimento sobre o impacto de suas mudanças o que, por sua vez, diminui a chance de introduzir defeitos.

O conjunto de métricas de acoplamento é amplo, bem como, a diversidade de ferramentas desenvolvidas para calculá-las (FREGNAN et al., 2019). Um dos fatores que pode explicar a diversidade de métricas e a falta de uma que se sobressaia na

prática, é o fato do nível de abstração da arquitetura atrapalhar a formalização das métricas, dificultando sua validação por outros trabalhos (COULIN et al., 2019; FREGNAN et al., 2019; STEVANETIC; ZDUN, 2015). Além disso, é comum que as ferramentas disponibilizadas para calcular as métricas propostas sejam softwares proprietários, ou que não tenham o algoritmo para o cálculo disponível publicamente. Sendo assim, outros trabalhos não conseguem avaliar os resultados em um conjunto maior de dados.

A manutenibilidade tem sido uma das perspectivas mais comuns para avaliar a qualidade da arquitetura e, ainda assim, há uma dificuldade em relacionar as métricas arquiteturais com os custos de manutenção (MO et al., 2018; VENTERS et al., 2018). Segundo Lindvall, Tvedt e Costa (2003), a manutenibilidade depende de vários fatores diferentes. Um sistema com uma arquitetura que permite que os desenvolvedores, que fazem a manutenção do código, possam compreender e aprender sobre sua estrutura, além de saber como operá-la para suas necessidades, é mais fácil de entender e manter. Segundo a norma ISO/IEC 25010 (2011), as subcaracterísticas de manutenibilidade são: modularidade, reusabilidade, analisabilidade, modificabilidade e testabilidade. Essas subcaracterísticas estão relacionadas à eficiência com a qual a arquitetura pode ser modificada, de acordo com a necessidade dos mantenedores. Em outras palavras, se a arquitetura é composta por componentes nos quais alterações feitas causam o mínimo impacto em outros componentes, se esses componentes podem ser reusáveis, se é possível avaliar o impacto de mudanças antes de aplicá-las ou conseguir identificar componentes que devem ser alterados por causa de defeitos, se o componente pode ser modificado, sem introduzir defeitos, e se os componentes podem ser facilmente testados.

Boa manutenibilidade pode ser alcançada por meio de regras arquiteturais claras e da adoção de padrões arquiteturais e de código, baixa complexidade, alta coesão e baixo acoplamento. Sistemas pequenos também são mais fáceis de manter. Logo, além de coesão, acoplamento e complexidade, o tamanho também pode ser relacionado a manutenibilidade e, portanto, também é possível avaliar uma arquitetura quanto à perspectiva de manutenibilidade, usando métricas que medem essas propriedades em um sistema. Alguns exemplos dessas métricas, em particular coesão e acoplamento, são a métrica *distância da sequência principal* de Martin (2017) e a métrica de *qualidade de modularidade* (MQ – *Modularization Quality*) de Mitchell e

Mancoridis (2006). A métrica distância da sequência principal, proposta por Martin (2017), mede o quão usável e passível de manutenção é um componente (KOZIOLEK et al., 2013). Outro exemplo, é a métrica MQ, qualidade de modularidade, que foi proposta por Mitchell e Mancoridis (2006) e é dependente da representação do sistema em grafo, para medir o quão bem as partições do grafo satisfazem as relações de alta coesão e baixo acoplamento. Portanto, um valor alto de MQ representa uma arquitetura bem modularizada.

Mais detalhes sobre métricas para medir a qualidade da arquitetura serão discutidos no Capítulo 6, na Seção 6.4.2, que trata de métricas para medir a qualidade da arquitetura.

5.5 Considerações finais

Como discutido ao longo deste capítulo, desvio arquitetural, presença de anomalias de código e anomalias arquiteturais e acúmulo de dívida técnica podem ser indícios de degradação da arquitetura. Isso compromete a manutenibilidade do software, porque arquiteturas são planejadas para serem fáceis de manter e ajudar a garantir qualidade interna do software, durante sua evolução (BRUNET et al., 2012).

Compreender o que leva a arquitetura ao estado de degradação, é útil para estabelecer o relacionamento com práticas de desenvolvimento de software. Esse conhecimento é importante para descobrir quais práticas podem contribuir para a degradação da arquitetura e quais podem minimizar a ocorrência de problemas arquiteturais; o que será tratado no capítulo seguinte, a partir da apresentação da análise empírica a ser realizada nesta pesquisa.

6 ANÁLISE EMPÍRICA

Na Engenharia de Software, pesquisas empíricas podem ser desafiantes porque, em muitos casos, o objetivo do estudo não é apenas verificar se uma teoria é ou não válida. Há ocasiões em que o objetivo da pesquisa pode ser entender um fenômeno sob a perspectivas dos participantes, tal como se faz em outras áreas do conhecimento, por exemplo, nas ciências sociais (WOHLIN; AURUM, 2015). Por isso, muitas vezes, o uso individual de métodos de pesquisa, a exemplo do experimento controlado, *survey* e do estudo de caso, pode não ser a abordagem mais indicada; sendo mais adequado, nesses casos, combinar vários métodos (WOHLIN; AURUM, 2015). O *design* de uma pesquisa, nessa área, pode ser definido baseado em um conjunto de decisões que levem em conta pontos, como: o propósito da pesquisa, a metodologia, o método de coleta de dados e o método de análise de dados (WOHLIN; AURUM, 2015).

Este capítulo descreve o *design* e as etapas do *design* da análise empírica realizada neste trabalho. As seções seguintes detalham: o contexto da pesquisa (6.1), o objetivo de pesquisa (6.2), as questões de pesquisa e as hipóteses de pesquisa (6.3), o *design* da análise (6.4), os conceitos e as métricas utilizadas (6.5), os critérios para a seleção dos projetos e as características dos projetos selecionados (6.6), o procedimento de coleta de dados (6.7), o procedimento de análise de dados (6.8) e a ferramenta desenvolvida e utilizada para coleta e análise de dados (6.9).

6.1 Contexto

Entender como a degradação da arquitetura ocorre ao longo da evolução do software, tem atraído atenção da comunidade científica (AHMED et al., 2015; BEHNAMGHADER et al., 2017; CANFORA et al., 2014; LE et al., 2018; MACIA et al., 2012; MARTINI; BOSCH; CHAUDRON, 2015) e contribuído para o debate a respeito dos sintomas e causas de problemas arquiteturais. Esse conhecimento é útil para prolongar o período em que um sistema se mantém em boa condição de manutenibilidade.

Alguns autores têm se concentrado em investigar os sintomas de degradação da arquitetura, como os problemas no código, associados a problemas arquiteturais,

e as anomalias arquiteturais. Eick et al. (2001), por exemplo, realizaram uma análise quantitativa no histórico de mudanças de um sistema, com milhares de linhas de código, para determinar quando o código entra em decadência. Ferreira et al. (2014), por sua vez, por meio de um estudo de caso exploratório, realizaram uma análise qualitativa, para entender como é possível relacionar problemas no código com anomalias arquiteturais, por meio de métricas. Macia et al. (2012) investigaram a relação entre anomalias de código e seu impacto na degradação da arquitetura. Le et al. (2016) caracterizaram as anomalias arquiteturais e introduziram algumas métricas para medir qualidade arquitetural. Em um trabalho seguinte, os autores investigaram a natureza e o impacto das anomalias arquiteturais na qualidade do software (Le et al., 2018). Fontana et al. (2017) apresentaram uma ferramenta para detecção de anomalias arquiteturais por meio da avaliação de problemas de dependência na arquitetura. Brunet et al. (2012) analisaram como as violações arquiteturais evoluem em um software. Também tem havido interesse em determinar a razão da degradação da arquitetura (DE SILVA; BALASUBRAMANIAM, 2012; HOCHSTEIN; LINDVALL, 2005; MARTINI; BOSCH; CHAUDRON, 2015; VAN GURP; BOSCH, 2002).

Algumas das possíveis causas de degradação da arquitetura, identificadas na Seção 5.3, estão relacionadas aos desenvolvedores e à qualidade do trabalho desses indivíduos. Essas causas vão desde a falta de entendimento do *design* original, e mudanças inconsistentes ocasionadas por isso (PARNAS, 1994), à falta de experiência do desenvolvedor e desconhecimento técnico (EICK et al., 2001; MARTINI; BOSCH; CHAUDRON, 2015). Em outros casos, ainda que não haja diferença técnica notável entre os desenvolvedores, problemas como: não seguir decisões e padrões pensados na concepção da arquitetura, ao longo do sistema, podem ser fonte de problemas arquiteturais, como apontaram (MARTINI; BOSCH; CHAUDRON, 2015). Um exemplo disso, é a presença de diferentes padrões de *design* ou de arquitetura usados para implementar a mesma funcionalidade. Segundo Martini, Bosch e Chaudron (2015), o fato de os desenvolvedores trabalharem, em paralelo, em diferentes partes de código, já é suficiente para o acúmulo de dívida técnica arquitetural. Eles acrescentam, ainda, que em abordagens de desenvolvimento, nas quais as equipes são encorajadas a se auto-organizarem e terem autonomia, amplia-se esse fenômeno, devido à possível divergência na adoção de soluções, como, por exemplo, a política de nomes ou de padrões. Equipes desse tipo podem ser

encontradas em métodos ágeis (BECK et al., 2001) e em projetos *open source* (RAYMOND, 1999).

Existem, também, algumas pesquisas com contribuições no sentido de avaliar como as práticas de desenvolvimento estão associadas com a evolução do software. Costa et al. (2018) estudaram o impacto que os ciclos de desenvolvimento curtos têm no atraso da integração de contribuições de correção de erros. Canfora et al. (2014) investigaram empiricamente a relação entre a complexidade e desorganização do código com quatro fatores, dentre eles, presença de atividades de refatoração e número de desenvolvedores trabalhando no arquivo de código-fonte. Alfayez et al. (2018) realizaram um estudo exploratório, para entender o impacto de vários desenvolvedores trabalhando em paralelo no acúmulo de dívida técnica, em um sistema.

Em um sentido geral, tem sido de interesse de pesquisa empírica em Engenharia de Software, estudar os sintomas da degradação da arquitetura durante a evolução do software; estudar como alguns fatores - por exemplo, as características de desenvolvedores afetam a qualidade da arquitetura, e como as práticas adotadas pelos desenvolvedores, no dia-a-dia, se relacionam com esses sintomas, ou, até mesmo, qual o impacto que essas práticas têm na degradação da arquitetura.

6.2 Objetivo

Os problemas que afetam a arquitetura podem comprometer a manutenção e evolução de um software. Le et al. (2018) mostraram que um dos sintomas de degradação da arquitetura, as anomalias arquiteturais, podem estar relacionadas com aumento de problemas de implementação e que arquivos “infectados” (ou seja, com degradação) são mais propensos a mudanças do que arquivos “limpos”. Problemas com relação a código (como as anomalias de código, do inglês *code smells*) e com relação a *design* (como as anomalias de *design*, do inglês *design smells*) nem sempre se refletem em problemas arquiteturais, como discutido na Seção 5.1; por isso, a arquitetura é o foco deste trabalho.

Para monitorar e manter a qualidade da arquitetura, torna-se necessário identificar os reais fatores que impactam negativamente as características de qualidade do sistema. Alguns desses fatores (causas) podem ser contornáveis por

meio da compreensão de como ocorrem, ou pela adoção de políticas de desenvolvimento mais adequadas, por exemplo. Esse pode ser o caso da degradação provocada por desenvolvedores inexperientes no projeto, ou pelas diferentes características de um desenvolvedor.

O objetivo desta análise empírica é analisar o impacto da experiência dos desenvolvedores na degradação da arquitetura de software em desenvolvimento *open source*. Por meio desse conhecimento, será possível descobrir se uma política mais restrita de autonomia no código deve ser adotada. Caso o nível de experiência dos desenvolvedores influencie pouco na degradação da arquitetura, os esforços para acompanhamento do processo de desenvolvimento poderiam ser concentrados em outros pontos. Além disso, os resultados desta pesquisa contribuem para o corpo de pesquisa que estuda se as características, habilidade e experiência dos desenvolvedores têm impacto na qualidade do software.

A próxima seção apresenta as questões de pesquisa derivadas deste objetivo e as hipóteses nulas e alternativas nas quais se baseia esta análise empírica.

6.3 Questões de pesquisa

Um dos argumentos para a adoção do comportamento de independência e auto organização em equipes de desenvolvimento, como as em projetos *open source*, é a possibilidade de espalhar o conhecimento entre os membros da equipe e diminuir a dependência concentrada em poucas pessoas, o que pode representar um risco à sustentabilidade do projeto (FERREIRA; VALENTE; FERREIRA, 2017). Espera-se que mais indivíduos experientes, envolvidos no desenvolvimento de diferentes partes do sistema, implique maior qualidade de código e, também, da arquitetura (RAYMOND, 1999); apesar de o código e a arquitetura estarem em diferentes níveis de abstração. A hipótese deste trabalho é que, em desenvolvimento *open source*, conforme um desenvolvedor acumula experiência, a qualidade de suas contribuições para a arquitetura tende a crescer. Para verificar se é realmente isso que ocorre, as questões de pesquisa QP1 e QP2 foram formuladas e são apresentadas a seguir, juntamente com as hipóteses derivadas de cada uma:

QP1: Existe relação entre a experiência dos desenvolvedores no sistema e a degradação da arquitetura causada por eles?

Hipótese nula 1 (H₀₁): Não há relação entre a experiência dos desenvolvedores e o impacto de suas contribuições para a qualidade da arquitetura.

Formalmente: $\rho_1 = 0$.

Hipótese alternativa 1 (H_{A1}): Quanto maior é a experiência dos desenvolvedores, melhor é o impacto de suas contribuições para a qualidade da arquitetura.

Formalmente: $\rho_1 < 0$.

Em QP1, avalia-se a correlação entre a experiência de todos os desenvolvedores do sistema e a degradação da arquitetura introduzida nos *commits* (contribuições) em que eles são autores. A variável ρ_1 corresponde ao coeficiente de correlação para esta hipótese. Uma correlação mede a dependência entre duas variáveis e seu coeficiente varia entre -1 e +1 (WOHLIN et al., 2012). Quando o coeficiente é -1, trata-se de uma correlação negativa forte, na qual uma variável tende a aumentar o seu valor quando a outra tende a diminuir. Por outro lado, se o coeficiente é igual a +1, então tem-se uma correlação positiva forte, na qual o valor de uma variável tende a crescer quando o da outra cresce e tende a diminuir quando o valor da outra diminui. Um coeficiente igual a 0 indica que as duas variáveis não estão correlacionadas. Neste trabalho, será adotada a correlação de postos de *Spearman* pelo fato de seu uso não implicar na necessidade de saber a exata distribuição de amostragem e também por ser menos afetada pela presença de *outliers* (D'AMBROS; BACCHELLI; LANZA, 2010; LENHARD; BLOM; HEROLD, 2019), além de ela ser utilizada por estudos similares (ALFAYEZ et al., 2018; SALAMEA; FARRE, 2019).

A segunda questão de pesquisa (QP2) visa responder se a experiência local (ou especializada) – isto é, a experiência derivada das contribuições em um componente específico – está relacionada com a degradação neste componente. Nessa questão de pesquisa, se explora a degradação apenas no componente onde houve a alteração, desprezando-se os demais componentes afetados por ela. Ela é formulada da seguinte forma:

QP2: Existe relação entre experiência dos desenvolvedores em um componente específico (experiência local) e a degradação da arquitetura causada por eles no componente?

Hipótese nula 2 (H₀₂): Não há relação entre a experiência dos desenvolvedores em um componente e o impacto de suas contribuições na qualidade da arquitetura do componente.

Formalmente: $\rho_2 = 0$.

Hipótese alternativa 2 (H_{A2}): Quanto maior é a experiência dos desenvolvedores no componente, melhor é o impacto de suas contribuições para a qualidade do componente.

Formalmente: $\rho_2 < 0$.

Em QP2, avalia-se a correlação entre a experiência local de todos os desenvolvedores do sistema em cada componente e a degradação da arquitetura no componente em *commits* em que eles são autores.

As duas hipóteses nulas (H₀₁ e H₀₂) consideram apenas a condição na qual $\rho = 0$ porque ambas estão na forma de hipótese nula simplificada, que é usada para facilitar a análise dos testes estatísticos (ROSS, 2009). De acordo com Ross (2009), qualquer procedimento que valha para decidir entre a hipótese nula simplificada e a hipótese alternativa também serve para decidir entre a hipótese nula mais realista (que neste caso seria $\rho \geq 0$) e a hipótese alternativa.

O objetivo da análise estatística é rejeitar a hipótese nula com alto nível de significância (WOHLIN et al., 2012). Se isso ocorrer, é possível tirar conclusões para responder as questões de pesquisa. A hipótese alternativa é a hipótese em favor da qual a hipótese nula é rejeitada e a que, afinal, se pretende verificar se é verdadeira. O procedimento de análise e o nível de significância considerado serão apresentados em 6.8.

6.4 Design da análise

O propósito desta pesquisa é testar duas hipóteses sobre a importância da experiência de desenvolvedores na ocorrência de degradação na arquitetura de softwares *open source*. Optou-se por fazer análise em repositórios de softwares (Mineração de Repositórios de Software). Esse tipo de análise permite entender a evolução do software por meio da análise dos dados no repositório e, neste trabalho, verificar se a experiência dos desenvolvedores está correlacionada com a degradação

da arquitetura durante a evolução dos softwares nos repositórios analisados. Os tipos de dados encontrados em repositórios vão desde código-fonte à documentação de projeto e são dados do histórico de desenvolvimento de um sistema. Algumas dessas informações estão armazenadas a partir de um sistema de controle de versões; ou seja, é possível rastrear a evolução desses artefatos.

As métricas de experiência e degradação da arquitetura escolhidas (Seção 6.5) deverão ter a possibilidade de serem extraídas diretamente, e de forma automatizada do repositório, a partir dos *commits*. As próximas seções deste capítulo irão detalhar como a análise em repositórios será executada.

6.5 Conceitos e métricas utilizados

Esta seção apresenta os conceitos e as métricas que serão empregados nesta análise, para medir a experiência e degradação da arquitetura.

6.5.1 Métrica para experiência no código

A experiência no código, neste trabalho, é relacionada exclusivamente à atividade de codificação, desprezando aspectos relacionados à capacidade técnica individual ou outras atividades de colaboração no projeto (p.ex.: revisão de código). Em outras palavras, é a experiência adquirida a partir da exposição do desenvolvedor à escrita de código, isto é, vinculada ao conhecimento episódico (ROBILLARD, 1999). Isso pode se dar em nível geral (global), ou em nível especializado (local), dependendo do quão espalhadas são as contribuições do desenvolvedor no sistema. Desenvolvedores com noção mais abrangente do sistema, teoricamente, conhecem mais a arquitetura do que desenvolvedores com conhecimento mais específico de apenas uma parte do sistema. De qualquer forma, existe mais de uma maneira de quantificar a experiência quanto ao código.

Quando um desenvolvedor faz *commits* em um componente, isto é, quando ele faz *commits* em classes que pertencem ao componente, ele está contribuindo para o componente. Teoricamente, quanto maior é o número de contribuições, maior a experiência do desenvolvedor naquela parte do sistema. A experiência em um componente é o que se pode considerar como experiência especializada (ou local). A

experiência geral (ou global), por outro lado, está relacionada a todas as contribuições de um desenvolvedor nos vários componentes do sistema. Em um projeto, pode-se observar alguns tipos de perfis de experiência baseadas nisso: indivíduos com conhecimento amplo e geral que podem ou não ser especialistas (experientes) em alguns componentes; indivíduos com pouca experiência geral, mas experientes em alguns componentes; ou indivíduos que contribuem esporadicamente e, portanto, possuem pouca experiência geral e pouca experiência em qualquer componente da arquitetura. A Tabela 6.1 mostra um resumo de alguns desses perfis possíveis. O perfil de experiência com mais conhecimento geral, provavelmente, corresponde ao de um indivíduo que está contribuindo há bastante tempo e participando de decisões de projeto e da arquitetura (JOBLIN et al., 2017). Como corresponde a um indivíduo que contribuiu em várias partes do sistema, isso pode implicar maior conhecimento dos componentes e dos relacionamentos entre eles (BASS; CLEMENTS; KAZMAN, 2012). O perfil com mais experiência especializada (sempre com respeito a um componente), e pouca experiência geral, pode estar associado a pessoas que fazem alterações conscientes no componente em que é especialista, mas desconhece os efeitos dessa mudança nos demais componentes e, conseqüentemente, na arquitetura.

Tabela 6.1 - Perfis de experiência dos desenvolvedores.

Experiência geral	Experiência especializada	Característica
+	+	Muita experiência em todo o projeto.
-	-	Pouca experiência em todo o projeto e no componente analisado.
-	+	Pouca experiência no projeto, muita no componente.
+	-	Muita experiência no projeto, pouca no componente.

6.5.1.1 Critérios para a escolha da métrica

Os critérios foram baseados em: (1) formas comuns de quantificar a experiência no código, em pesquisas na área de Engenharia de Software e (2) no tipo de análise a ser realizado, usando técnicas de Mineração de Repositórios de Software.

1. **Formas comuns de quantificar a experiência no código em pesquisas na área de Engenharia de Software:** quais as principais fontes de dados usadas para determinar a experiência de um desenvolvedor no código, em particular, em desenvolvimento *open source*.

2. **Métricas que possam ser calculadas a partir de dados em repositórios de software:** métricas cujas informações utilizadas podem ser obtidas em repositórios de software e que possam ser coletadas de forma automatizada por ferramentas.

6.5.1.2 *Seleção da métrica*

As várias formas de medir a experiência de um desenvolvedor em Engenharia de Software foram apresentadas no Capítulo 4 desta dissertação. As métricas levantadas a partir do mapeamento sistemático, apresentado no Capítulo 4, estão no Apêndice A. O mapeamento identificou que várias métricas utilizam a informação da quantidade de *commits* ou número de linhas modificadas e que isso tem sido suficiente para quantificar a experiência em pesquisas da área (BHATTACHARYA; NEAMTIU; FALOUTSOS, 2014; BIRD et al., 2011; OZCAN KINI; TOSUN, 2018).

Quase todas as métricas utilizam informações disponíveis, a partir de repositórios de código, *issue trackers* ou plataformas de codificação social. Porém, dessas métricas, as que medem, especificamente, experiência no código, só precisam das informações contidas em repositórios, a partir do uso de sistemas de controle de versões. Todos os dados necessários para rastrear as contribuições de uma pessoa no código (arquivos e linhas alteradas, data de alteração e autor da contribuição) estão nos *commits*, que estão disponíveis publicamente em softwares *open source*. A disponibilidade possibilita que a métrica possa ser calculada a partir do uso de ferramentas que automatizam o processo de coleta e processamento dos dados, em repositórios de softwares de código aberto. A Tabela 6.2 mostra as métricas que medem experiência no código e que podem ser obtidas a partir de informações disponíveis em repositórios de software. Isso implica que, na tabela, não estão métricas que precisam de outras fontes de dados, tais como, *issue trackers*, plataformas de codificação social ou atribuição da experiência pela própria pessoa.

6.5.1.3 *Métrica escolhida*

Neste trabalho, a experiência será medida usando três fontes de dados: número de *commits*, número de arquivos distintos modificados e número de linhas de código modificadas. A agregação dessas três fontes de dados foi baseada na abordagem de Constantinou e Kapitsaki (2016), que, em seu trabalho, tinha o objetivo

Tabela 6.2 - Métricas de experiência que utilizam dados disponíveis em repositórios.

Autores	Dados
(OZCAN KINI; TOSUN, 2018)	CLOC e contagem de <i>commits</i>
(CONSTANTINOU; KAPITSAKI, 2016)	CLOC, contagem de arquivos e contagem de <i>commits</i>
(ORRÚ; MARCHESI, 2016)	Contagem de <i>commits</i>
(QIU et al., 2015)	Contagem de <i>commits</i>
(BHATTACHARYA; NEAMTIU; FALOUTSOS, 2014)	CLOC e contagem de arquivos
(WU et al., 2014)	Contagem de <i>commits</i>
(TEUSNER; MATTHIES; GIESE, 2017)	Contagem de <i>commits</i>
(DA SILVA et al., 2015)	Contagem de <i>commits</i>
(SCHILLING; LAUMER; WEITZEL, 2012)	Contagem de <i>commits</i>
(OLSSON; ERICSSON; WINGKVIST, 2017)	CLOC
(THONGTANUNAM et al., 2016)	Contagem de <i>commits</i> e comentários em revisões de código
(FOUCAULT et al., 2015)	CLOC e contagem de arquivos
(ZHANG et al., 2014)	CLOC
(RAHMAN; DEVANBU, 2011)	CLOC
(GREILER; HERZIG; CZERWONKA, 2015)	Contagem de <i>commits</i>
(BIRD et al., 2011)	Contagem de <i>commits</i>

de quantificar a experiência de um desenvolvedor em diferentes linguagens, a partir de suas atividades em plataformas de codificação social (*GitHub*⁵ e *Stack Overflow*⁶).

A abordagem original agregava as informações usando os pesos: 0,4 para o número de *commits*, 0,4 para o número de arquivos distintos modificados, e 0,2 para o número de linhas modificadas em uma fórmula, que também agregava a informação de um fator de impulso (ver a métrica na Tabela A-2 no Apêndice A). Os pesos foram utilizados para representar a importância de cada um dos parâmetros para a quantificação da experiência. Porém, diferentemente da abordagem original, aqui não se utilizou o fator de impulso e pesos diferentes foram aplicados às variáveis. O fator de impulso é uma forma de considerar a frequência de contribuições de um

⁵ <https://github.com/>

⁶ <https://stackoverflow.com/>

desenvolvedor. A intenção de Constantinou e Kapitsaki (2016) foi usar esse fator para impulsionar os valores de experiência, quando o desenvolvedor aumenta sua frequência de contribuição, e penalizar, caso diminua. No entanto, ele não é usado na métrica desta análise, porque a proposta deste trabalho é usar a experiência acumulada. Por isso, também é utilizado o valor acumulado da quantidade de *commits*, dos arquivos distintos modificados e das linhas modificadas, diferentemente do que é feito na métrica original, na qual esses valores são as porcentagens entre os desenvolvedores do sistema.

A *Equação 1* descreve como a experiência será quantificada tanto para QP1 quanto para QP2. Ela é medida a cada *commit* devido à forma de análise dos dados (ver Seção 6.8), e k corresponde ao *commit* atual:

Equação 1 - Métrica de experiência no código.

$$0,2 * (k - 1) * \text{commits} + 0,4 * \sum_{i=1}^{k-1} \text{arquivos distintos modificados} + 0,4 * \sum_{i=1}^{k-1} \text{CLOC}$$

A quantidade de *commits* dados até o *commit* atual tem menor peso do que os demais porque, para fins de experiência no código, ela é a menos relevante dentre as outras informações usadas na métrica. Isso acontece pois ela depende do perfil de trabalho do desenvolvedor no sistema de controle de versões, ou seja, a frequência com a qual ele faz *commits* para salvar o estado atual do trabalho realizado em seu ambiente local⁷. A quantidade de arquivos distintos modificados mostra a extensão da contribuição de um dado desenvolvedor, isto é, mostra se ele contribui em várias partes do sistema, no caso de QP1, ou se ele contribuiu em várias classes do componente, no caso de QP2. O número de linhas modificadas é uma medida direta da exposição do desenvolvedor ao código e corresponde ao indício mais direto de conhecimento e experiência derivados da escrita de código. Linhas em branco são excluídas da contagem para evitar que haja distorção entre a quantidade de linhas que podem alterar o software e linhas que são apenas ignoradas. Quanto a alegação de que a contagem de linhas modificadas está mais suscetível a fatores como ineficiência do desenvolvedor escrevendo código, a agregação com a quantidade de

⁷ Para uma mesma contribuição, um desenvolvedor pode fazer somente 1 *commit* enquanto que um outro desenvolvedor pode fazer diversos *commits*.

arquivos modificados e com o número de *commits* serve para minimizar a possibilidade de distorção.

6.5.2 Métricas para avaliar a arquitetura

Esta seção apresenta a métrica utilizada para medir a degradação da arquitetura. Os critérios adotados para a escolha da métrica foram baseados nos critérios de avaliação dos estudos primários do mapeamento sistemático de Stevanetic e Zdun (2015), que avalia métricas de software para inteligibilidade de estruturas arquiteturais. Além disso, esse mapeamento sistemático foi usado juntamente com a revisão sistemática de Coulin et al. (2019), para uma discussão geral sobre métricas para avaliar arquitetura.

A arquitetura de software trata da estrutura do software, especificamente, quanto aos seus componentes e às relações externas visíveis entre eles (BASS; CLEMENTS; KAZMAN, 2012). Neste trabalho, considera-se a arquitetura de softwares orientados a objetos, com pacotes representando componentes. Portanto, as métricas de interesse são aquelas que medem a arquitetura como um todo, ou métricas que medem itens que podem ser vistos como elementos da arquitetura, assim como um pacote. Um exemplo de itens que não representam uma unidade significativa, para visualizar a estrutura arquitetural do software, são métodos ou atributos. Stevanetic e Zdun (2015) não consideram essas estruturas no nível arquitetural. Segundo esses autores, diferentes trabalhos defendem a importância de métricas que medem componentes individualmente, e de métricas que medem toda a representação arquitetural. As métricas que medem componentes consideram as relações de entrada e saída do componente, e relações entre entidades dentro deles. As métricas que medem toda a representação arquitetural, por sua vez, levam em conta todas as partes que constituem o sistema. A seguir, serão discutidos os critérios de seleção e, em seguida, a apresentação da métrica escolhida.

6.5.2.1 Critérios para a escolha da métrica

Os critérios foram baseados no mapeamento sistemático de Stevanetic e Zdun (2015), porque ele apresenta uma descrição bastante detalhada das métricas, da aplicabilidade e da usabilidade delas, servindo também de base para as demais

métricas consultadas. Esses critérios são: (1) a característica de qualidade afetada pela métrica, pois ter uma característica objetivo para comparação auxilia na interpretação dos dados (LINDVALL; TVEDT; COSTA, 2003); (2) o nível de maturidade da métrica - medido pela existência de mais trabalhos validando para outros conjuntos de dados - e (3) a disponibilidade das ferramentas para o cálculo, uma vez que, dependendo da métrica, nem sempre é trivial obter seu valor.

1. **Característica de qualidade afetada:** segundo a norma ISO/IEC 25010 (2011), as características de qualidade do produto de software são: adequação funcional, eficiência de desempenho, compatibilidade, usabilidade, confiabilidade, segurança, manutenibilidade e portabilidade. Neste trabalho, observa-se a degradação sob a perspectiva da manutenibilidade porque ela está relacionada à capacidade do sistema de atender às mudanças e evoluir. Métricas adequadas para isso podem ser as métricas de coesão e acoplamento, como feito Mitchell e Mancoridis (2006).
2. **Maturidade da métrica**⁸: é considerada do ponto de vista do nível de validação da métrica, seguindo a ideia de Stevanetic e Zdun (2015) que definem quatro níveis de validação. Quanto maior o nível de validação, mais madura é a métrica. Os níveis definidos por Stevanetic e Zdun (2015) são:
 - a. **Nível 1 – Anekdotico:** alguns exemplos são fornecidos pelos autores da métrica, para motivar a utilidade dela.
 - b. **Nível 2 – Pequeno experimento:** a métrica é validada empiricamente, usando um pequeno conjunto de dados; por exemplo, um ou poucos projetos de pequeno ou médio porte.
 - c. **Nível 3 – Grande experimento:** a métrica é validada empiricamente, usando grandes conjuntos de dados; como os conjuntos com muitos sistemas de pequeno/médio porte, ou conjuntos com um ou poucos sistemas de grande porte.
 - d. **Nível 4 – Independentemente validado:** a métrica é validada com sucesso por outras equipes de pesquisa e sua aplicabilidade é confirmada no contexto dado.

⁸ A escala TRL (*Technology Readiness Levels*) (MANKINS, 1995) foi consultada, mas descartada porque a escala define níveis para tecnologias e não métricas.

Neste trabalho, considera-se como métrica madura uma métrica validada empiricamente, usando grande conjunto de dados, ou cuja aplicabilidade é confirmada no contexto dado; ou seja, os níveis 3 e 4 da escala adotada.

3. **Ferramenta:** as métricas fornecidas por ferramentas que estão disponíveis possuem uma maior prioridade, porque facilitam a análise empírica.

6.5.2.2 Seleção da métrica

As métricas que medem a qualidade da arquitetura pela perspectiva da qualidade de manutenibilidade foram levantadas em Stevanetic e Zdun (2015), que classificam métricas de arquitetura, relacionando-as à capacidade de aprendizado sobre a arquitetura e seus componentes. A capacidade de aprendizado também influencia na manutenibilidade do sistema, pois, se as relações entre as estruturas são fáceis de serem compreendidas, é possível entender as implicações das mudanças em outras partes do sistema. Em outras palavras, espera-se que mudanças não tragam um efeito cascata muito grande. De acordo com as revisões sistemáticas avaliadas (COULIN et al., 2019; STEVANETIC; ZDUN, 2015), muitas métricas carecem de maturidade, seja pelo baixo nível de formalismo na definição da métrica, seja na integração entre a métrica e a característica de qualidade que ela tem como objetivo medir; ou, ainda, pelo baixo nível de validação empírica que algumas apresentam.

Por outro lado, a disponibilidade de ferramentas que calculam as métricas levantadas foi obtida a partir do levantamento feito por Stevanetic e Zdun (2015). De acordo com os autores, a maioria das ferramentas que fornecem o valor das métricas, em nível arquitetural, necessitam de informações que são fornecidas por outras ferramentas, que, muitas vezes, são proprietárias ou combinação de proprietária e alguma ferramenta *open source*. Das ferramentas que tiveram continuidade, algumas são ferramentas desenvolvidas em pesquisas e, ou não têm código aberto, ou não disponibilizam os algoritmos para os cálculos das métricas (AZADI; FONTANA; TAIBI, 2019). Alguns trabalhos (AZADI; FONTANA; TAIBI, 2019; FONTANA; ROVEDA; ZANONI, 2016) compararam as ferramentas existentes para avaliar a qualidade da arquitetura. Em geral, esses trabalhos classificam as ferramentas usando o tipo de informações que elas disponibilizam sobre a arquitetura (p.ex.: métricas, capacidade de detectar anomalias arquiteturais ou violações arquiteturais, e sugestão de refatoração) e se são proprietárias ou livres para uso.

A Tabela 6.3 apresenta um resumo de métricas de software, referentes à qualidade da arquitetura, que possuem o apoio de alguma ferramenta para fornecê-las, e mostra que grande parte delas é calculada por ferramentas acadêmicas que não estão disponíveis. Algumas, tal qual a ferramenta Titan (XIAO; CAI; KAZMAN, 2014), são ferramentas acadêmicas que necessitam de informações fornecidas por ferramentas comerciais, corroborando a observação de Stevanetic e Zdun (2015). Dentre as métricas da tabela, algumas estão presentes no mapeamento sistemático de Stevanetic e Zdun (2015), mas há também métricas que foram propostas depois desse mapeamento, tal qual: *Decoupling Level* (MO et al., 2016). A Tabela 6.4 apresenta as métricas desses estudos sistemáticos, que atendem aos três critérios de seleção desta análise empírica. Apenas as métricas acoplamento aferente (Ca)⁹, acoplamento eferente (Ce), instabilidade (I), abstração (A), distância da sequência principal (D), que fazem parte do conjunto de métricas para avaliar *design* de softwares orientados a objetos, em termos de interdependência entre módulos, proposta por Martin (2017), são fornecidas por uma ferramenta disponível. A ferramenta em questão é a Arcan¹⁰ (*ARChitecture ANalyzer*) (FONTANA et al., 2017b). Ela calcula as métricas, a partir do código-fonte, empregando técnicas de recuperação da arquitetura, utilizando engenharia reversa. Ela também pode detectar anomalias arquiteturais. Além disso, é uma ferramenta que tem evoluído continuamente. A subseção seguinte detalhará a métrica.

6.5.2.3 Métrica escolhida : Distância da sequência principal

Das métricas que atendem aos três critérios, a que melhor representa a qualidade da arquitetura, em termos de manutenibilidade, é distância da sequência principal (D). Ela é uma métrica que indica a relação de equilíbrio ideal entre abstração e estabilidade em pacotes; ou seja, os componentes são estáveis, mas têm um grau satisfatório de flexibilidade para evoluir. Quando seu valor for próximo de 0, significa que o componente analisado está próximo do *design* ideal. Por *design* ideal, entende-se boa manutenibilidade; devido à independência entre os componentes e ao baixo

⁹ Acoplamento aferente e Acoplamento eferente foram posteriormente renomeadas pelo autor para *Fan-in* e *Fan-out*, respectivamente em Martin (2017).

¹⁰ Disponível em: <http://essere.disco.unimib.it/wiki/arcan>

Tabela 6.3 - Ferramentas que fornecem métricas para a qualidade da arquitetura e as métricas calculadas por elas.

Ferramenta	Métricas	Entrada	Comercial Disponível	Acadêmica Disponível
(MO et al., 2016)	<i>Propagation Cost (PC)</i> <i>Decoupling Level (DL)</i>	DSM ¹¹ DSM	✓	
(XIAO; CAI; KAZMAN, 2014)	<i>Independence Level (IL)</i> <i>Decoupling Level (DL)</i> <i>Feature Decupling Level (FDL)</i>	DSM DSM FDSM ¹²		X
(LI et al., 2014)	<i>Index of Package Changing Impact (IPCI)</i> <i>Index of Package Goal Focus (IPGF)</i>	Código Código		✓
(BHATTACHARYA et al., 2012)	<i>Average Degree (k)</i> <i>Assortativity (Ass)</i> <i>Modularity Ratio (MR)</i>	Grafo dirigido Grafo Grafo dirigido		X
(BEHNAMGHADER et al., 2017a)	<i>Ratio of Cohesive Interactions (RCI)</i> Instabilidade (I) <i>Modularization Quality (MQ)</i> <i>Bi-directional Component Coupling (BDCC)</i> <i>Architectural Smell Density (ASD)</i> <i>Architectural Smell Coverage (ASC)</i>	Grafo Código Grafo dirigido (MDG) PKG PKG PKG		X
(FONTANA et al., 2017b)	<i>Fan-in</i> (Acoplamento aferente - Ca) <i>Fan-out</i> (Acoplamento eferente - Ce) Instabilidade (I) Abstração (A) Distância da sequência principal (D) <i>Architectural Debt Index (ADI)</i>	Código Código Código Código Código Grafo		✓
(MACCORMACK; RUSNAK; BALDWIN, 2006)	<i>Custered Cost</i>	DSM		X
(ALBATTAH, 2017)	<i>Package Cohesion Metric (CH)</i>	Código		X
(SETHI et al., 2009)	Métricas de Estabilidade e Modularidade de Sethi et al.	UML		X
(SARKAR; KAK; NAGARAJA, 2005)	<i>Module Interaction Index (MII)</i> <i>Non-API Function Closedness Index (NC)</i> <i>API Function Usage Index (APIU)</i> <i>Module Size Uniformity Index (MSIU)</i> <i>Module Size Boundedness Index (MSBI)</i>	DSM	X	X

¹¹ DSM – *Design Structure Matrix*, formato criado por Baldwin e Clark (2000).¹² FDSM – *Feature Design Structure Matrix*, formato criado por MO et al. (2018b).

Tabela 6.4 - Métricas de qualidade da arquitetura classificadas pelo mapeamento sistemático e que atendem aos critérios desta pesquisa.

Autores	Métricas	Nível de maturidade (1-4)	Característica de qualidade	Ferramenta
(SARKAR; KAK; RAMA, 2008)	<i>Association Induced Coupling Metric (AC(S)), Size Uniformity Metric (MU), Common Use of Module Classes (CReuM(S))</i>	3	Manutenibilidade	SIM
(MARTIN, 2017) e (ELISH, 2010) ¹³	<i>Number of Classes (NC), Afferent Coupling (Ca), Efferent Coupling (Ce), Instability (I), Distance (D)</i>	4	Manutenibilidade ¹⁴	SIM
(BHATTACHARYA et al., 2012)	<i>Average Degree (k), Assortativity (Ass) e Modularity Ratio (MR)</i>	3	Manutenibilidade	SIM

Fonte: Stevanetic e Zdun (2015, p. 9)

efeito cascata propagado após alterações (MARTIN, 2017). A seguir, a métrica D será descrita, a partir das demais métricas que a compõem.

- Abstração (A): classes abstratas podem ser estáveis e, ao mesmo tempo, flexíveis, para serem estendidas sem necessidade de modificações (MARTIN, 2017). Isso é desejável, para que um sistema seja flexível e, ao mesmo tempo, estável; sem restringir as opções de *design*. A métrica abstração *A* mede o número total de classes abstratas, em um componente, sobre a quantidade total de classes abstratas no sistema. A métrica varia entre 0 e 1. Quanto mais próximo de 1, maior a proporção de classes abstratas em um componente.
- Instabilidade (I): é uma métrica de dependência, proposta por Martin (2017), e representa uma medida de instabilidade entre componentes de um software. Ela varia entre 0 e 1, e indica o quão instável é um componente. Quanto mais próximo de 1, maior a chance de uma classe em um componente ter que ser modificada, se classes em outros componentes, dos quais ela depende, forem modificadas. Isso é um sinal de baixa manutenibilidade e de que a arquitetura não está bem modularizada. A fórmula para calcular *I* é dada por:
$$\frac{Fan-out}{(Fan-in + Fan-out)}$$
- Fan-out: representa a proporção de classes dentro de um componente, que dependem de classes que pertencem a outro componente.

¹³ Com exceção da métrica NC, as métricas são originalmente de Robert C. Martin. Esse autor apenas fez uma avaliação empírica sobre se as métricas realmente estão correlacionadas com compreensibilidade.

¹⁴ Esse autor considera as métricas com respeito à capacidade de aprendizagem com relação a pacotes, mas o autor original as descreve como métricas importantes para a manutenibilidade.

- Fan-in: representa a proporção de classes fora de um componente, que dependem de classes que pertencem ao componente.
- Distância da sequência principal (D): a sequência principal é uma representação da relação balanceada entre estabilidade (*I*) e abstração (*A*) (MARTIN, 2017). A distância da sequência principal (*D*) mede o quão próximo está um componente da relação ideal. A métrica foi proposta por Martin (2017) e varia entre 0 e ~0,707. Se ela não estiver próxima de 0, a arquitetura não tem boa manutenibilidade. A ferramenta Arcan fornece essa métrica normalizada, variando entre 0 e 1. A fórmula de *D* normalizada é:

$$D = |(A - I - 1)|$$

6.6 Seleção dos projetos

Nesta seção, são apresentados os critérios considerados para a seleção dos projetos que serão objeto da análise. Esses projetos foram escolhidos, visando minimizar os efeitos de fatores que podem afetar as variáveis analisadas (WOHLIN et al., 2012): a experiência do desenvolvedor e a degradação da arquitetura. Isso se faz necessário, porque, na análise de repositórios, não há um alto nível de controle do ambiente e das variáveis independentes, pois as informações são obtidas diretamente dos repositórios de código dos projetos analisados, sem consulta a outras fontes de dados. Os projetos também foram escolhidos, por serem representações de típicos projetos de desenvolvimento *open source* de sucesso, ou seja, projetos com anos de desenvolvimento e uma grande comunidade de contribuidores.

6.6.1 Critérios de seleção

A variável experiência do desenvolvedor tem seu valor determinado usando apenas informações vindas da escrita de código. Porém, outros fatores podem ter efeito sobre o desempenho dos desenvolvedores, entre os quais, a habilidade técnica do indivíduo e seus conhecimentos de Engenharia de Software. Para minimizar a disparidade técnica entre desenvolvedores, os projetos escolhidos devem contar com a adesão

de contribuidores que sigam boas práticas e que tenham mostrado, de alguma forma, experiência em programação e Engenharia de Software, antes de se tornarem contribuidores.

Outro fator que deve ser minimizado é a presença das causas da degradação da arquitetura, levantadas na Seção 5.3, que não têm relação com a ação dos desenvolvedores. São elas: falta de documentação de princípios e decisões de *design*, falta de documentação da arquitetura pretendida, documentação pobremente escrita e ignorada pelos mantenedores, requisitos mal definidos e imprecisos, mudanças de requisitos aliados à falta de verificação de conformidade entre arquitetura implementada e arquitetura pretendida, falta de ênfase em requisitos arquiteturalmente críticos, arquitetura não-modularizada e priorização de características (*features*) acima da preocupação com a arquitetura do sistema.

Um critério, também importante, é o projeto ser um software representativo para o desenvolvimento *open source*. Ele deve apresentar características comumente encontradas em projetos *open source*, como: arquitetura modularizada e possuir comunidade de voluntários engajada. Além disso, é importante que seja um software com várias versões e vários desenvolvedores, com preferência por softwares que possuam alguns anos de desenvolvimento.

Considerou-se, também, apenas sistemas de software orientados a objetos, dada a métrica adotada para medir a qualidade da arquitetura. Além disso, devido à ferramenta que fornece a métrica da arquitetura, restringiu-se as possibilidades a softwares escritos em Java.

Em resumo, foram considerados os seguintes critérios:

- Projetos com desenvolvedores com alto nível técnico;
- Projetos nos quais as outras causas de degradação arquitetural são minimizadas;
- Software com várias versões desenvolvidas e vários desenvolvedores, com todas essas informações disponíveis em um repositório;
- Software orientado a objetos e
- Sistema desenvolvido em linguagem Java.

Levando em consideração esses critérios, optou-se por utilizar projetos da Apache Software Foundation¹⁵. A Apache é uma organização para desenvolvimento de softwares *open source*, sem fins lucrativos, e que possui uma grande comunidade de voluntários e mais de 300 projetos. Os projetos seguem um modelo denominado Modelo de Maturidade do Projeto Apache¹⁶, que serve para guiar como um projeto deve operar. Este modelo contém diretrizes sobre código, licença e *copyright*, versões, qualidade, comunidade, construção de consenso e independência. Isso quer dizer que, por mais que os projetos sejam *open source* e os colaboradores voluntários, todos seguem um rigoroso controle de qualidade que prezam pela sustentabilidade dos sistemas. Nesse contexto, os projetos possuem documentação e estrutura de comunicação (lista de e-mails, *bug tracker* ou canais de IRC), padrões de código, arquitetura modularizada, preocupação constante com a qualidade da arquitetura, hierarquia e descentralização das contribuições de código, adoção de padrões de código e desenvolvedores técnicos. A Tabela 6.5 apresenta um resumo de como os projetos da Apache lidam, de uma forma geral, com as causas da degradação e, na próxima seção, é apresentado um resumo sobre os projetos selecionados.

6.6.2 Projetos selecionados

Optou-se por analisar mais de um software, porque isso permite testar as hipóteses com times de características diferentes, softwares de tamanhos diferentes e de finalidades diversas, ampliando a capacidade de tirar conclusões sobre as questões de pesquisa. Portanto, foram selecionados 4 projetos da Apache de diferentes domínios escritos em Java e com idades entre 10 e 20 anos. E, além disso, são projetos que já foram selecionados em outras análises empíricas, seja com o propósito de estudar experiência e característica de desenvolvedores, seja para estudar degradação da arquitetura ou qualidade de software. Os 4 projetos são:

- **Ant:** ferramenta para automatização de compilação desenvolvida em Java. É um software baseado em linha de comando, popularmente aceito e usado por

¹⁵ <https://www.apache.org/>

¹⁶ <https://community.apache.org/apache-way/apache-project-maturity-model.html>

Tabela 6.5 - Causas tratadas pela Apache.

#	Causa para degradação	Como é em projetos da Apache
1	Falta de documentação de princípios e decisões de <i>design</i> .	Princípios e decisões de <i>design</i> são documentados.
2	Documentação pobremente escrita ignorada pelos mantenedores.	Além da documentação existente não ser ignorada pelos desenvolvedores, ainda se discute sobre alterações feitas no código.
3	Falta de documentação da arquitetura pretendida e o fato de muitos dos envolvidos na concepção da arquitetura não permanecerem no projeto.	Os líderes do projeto estão nele desde o início e se conhece detalhes sobre a arquitetura pretendida.
4	Requisitos imprecisos que podem fazer os desenvolvedores escreverem códigos que terão grande chance de serem modificados futuramente.	São típicos projetos <i>open source</i> no qual os requisitos são elicitados e especificados com clareza pelo usuário final que muitas vezes é também co-desenvolvedor.
5	Requisitos mal definidos no início pela falta de conhecimento do negócio.	Em geral, os projetos da Apache foram criados por alguém (ou por um grupo) que conhece bem o domínio.
6	Mudanças de requisitos (MAIR; HEROLD, 2013) aliadas a falta de verificação de conformidade entre a arquitetura pretendida e a arquitetura implementada.	Em caso de novos requisitos, sempre se discute os impactos na arquitetura do sistema.
7	Falta de especificação ou ênfase em requisitos arquiteturais críticos.	Há sempre a preocupação com a qualidade e a modularidade da arquitetura.
8	Pessoas que não entendem o <i>design</i> original podem fazer mudanças inconsistentes com o conceito original (PARNAS, 1994).	Qualquer mudança no código passa pela supervisão de outros desenvolvedores com permissão de fazer <i>commit</i> .
9	Desenvolvedores não conseguem entender ou alterar um código complexo, escrito por seus colegas mais qualificados (EICK et al., 2001).	Para se tornar um desenvolvedor com permissão para fazer <i>commits</i> , é preciso se envolver previamente no projeto e se provar bom tecnicamente por meio de sua reputação na comunidade.
10	Arquitetura não modularizada.	A arquitetura é modularizada.
11	Arquitetura inapropriada que não possibilita a incorporação fácil de requisitos que não foram pensados no início do desenvolvimento.	Embora possa ser uma arquitetura complexa, é modularizada e é flexível a novos requisitos.
12	A priorização de características (<i>features</i>) acima da preocupação com a arquitetura do sistema.	Há sempre preocupação com a arquitetura do sistema. As características antes de implementadas são amplamente discutidas, inclusive sob aspecto arquitetural.

muitos desenvolvedores (BARROS; FARZAT; TRAVASSOS, 2015) e por sistemas de software *open source* e industriais. Além disso é um projeto longo: são mais de 20 anos de desenvolvimento.

- **Cassandra:** banco de dados distribuído escalável usado por várias aplicações como Netflix e Uber, por exemplo. São 10 anos de histórico de desenvolvimento.
- **Hadoop:** plataforma para computação distribuída e processamento de grande volume de dados, usada por plataformas como, por exemplo, Amazon e IBM. Sua arquitetura é composta por 5 módulos, dentre eles o *Hadoop Common*; módulo no qual estão as bibliotecas comuns usadas pelos demais módulos.

- **Lucene:** biblioteca que fornece recursos avançados de indexação e pesquisa, com 19 anos de desenvolvimento.

Especificamente, quanto ao Ant, ele possui, atualmente, duas linhas principais de desenvolvimento, que se dividem após a versão 1.9.7: 1.9.x e 1.10.x. A linha 1.9.x trata majoritariamente de correção de erros, enquanto que a linha 1.10.x concentra as melhorias. Neste trabalho, desconsiderou-se as versões entre 1.9.8 e 1.9.14, porque elas pertenciam a ramos paralelos ao ramo 1.10.x de desenvolvimento.

A Tabela 6.6 apresenta as características dos projetos selecionados, o que inclui: o domínio do software, a quantidade de desenvolvedores, o intervalo de tempo acompanhado na análise, o intervalo de versões consideradas e a quantidade de linhas de código de cada um.

Tabela 6.6 - Características dos sistemas selecionados

Sistema	Domínio	Número de desenvolvedores ¹⁷	Intervalo de tempo	Intervalo de versões	LOC
Ant	Automação de compilação	29	07/2000-09/2019	1.1-1.10.7	427k
Cassandra	Banco de dados distribuído	81	04/2010-02/2020	0.6.0-3.11.6	768,2k
Hadoop	Processamento de dados	217*	05/2011-10/2019	0.20.203.0-2.8.5	1,2m
Lucene	Mecanismos de pesquisa	55**	09/2001-03/2020	1.0.1-8.5.0	328,8k

* Todo o projeto Hadoop

** Todo o projeto Lucene

6.7 Procedimento de coleta dos dados

Esta seção descreve o procedimento de coleta de dados, ou seja, quais dados foram coletados e como foi o processo de coleta. Os dados coletados foram utilizados para calcular as métricas de experiência no código e de degradação da arquitetura. Os processos para o cálculo das duas métricas foram detalhados, separadamente, nas Subseções 6.7.1 e 6.7.2, respectivamente.

¹⁷ <https://projects.apache.org/projects.html?number>

6.7.1 Dados para determinar experiência

Retomando a Equação 1 (Seção 6.5.1), para o cálculo da experiência (geral e especializada) de um desenvolvedor em um dado *commit*, na qual k é o *commit* atual, tem-se que:

$$0,2 * (k - 1) * commits + 0,4 * \sum_{i=1}^{k-1} arquivos\ distintos\ modificados + 0,4 * \sum_{i=1}^{k-1} CLOC$$

Para calcular cada um dos itens da Equação 1, é preciso saber a quantidade de *commits*, quantidade de arquivos distintos modificados e a quantidade de linhas modificadas pelo autor do *commit* ao longo dos seus *commits* anteriores. Para QP1, as contribuições em todo o sistema são consideradas. Para calcular a experiência em QP2, serão consideradas apenas contribuições no escopo do componente analisado.

As informações que são recuperadas do sistema de controle de versões, e serão utilizadas para a métrica, são:

- Versão do software,
- Autor do *commit*,
- Data do *commit*,
- Arquivos modificados e diretório onde estão localizados,
- Diff (linhas modificadas no *commit*) e
- Número de linhas modificadas que não são comentários ou linhas em branco.

Considerou-se apenas *commits* que continham arquivos de código-fonte (.java), pois apenas esses arquivos podem modificar a arquitetura do software – e, portanto, impactar as métricas que representam a manutenibilidade do software. Não foram considerados *commits* que pertenciam a versões beta.

Nesta análise, distingue-se o autor e a pessoa que faz o *commit*, tal como o Git faz (CHACON; STRAUB, 2014). O primeiro é quem verdadeiramente escreveu o código, ao passo que o segundo corresponde a quem realizou o *commit*. Em projetos *open source*, é comum ser adotado algum tipo de fluxo de trabalho distribuído e, também, que contribuidores com pouca experiência no projeto não possuam permissão para realizar um *commit* diretamente. Nesses casos, um outro desenvolvedor, que possua permissão, pode fazer o *commit* em seu lugar. Em um

fluxo de trabalho distribuído, os desenvolvedores podem contribuir submetendo *patches* para outro desenvolvedor com papel de membro principal aplicar. Nesse caso, o autor é quem escreveu o código, e o *committer* é quem aplicou o trabalho (CHACON; STRAUB, 2014). Quando o *commit* é feito diretamente no repositório, autor e *committer* são a mesma pessoa.

No Ant, o autor correto de um *commit* foi determinado usando informações do sistema de versionamento, o Git, e, também, por meio da consulta à mensagem do *commit*. Percebeu-se que havia um padrão no qual certos *commits* atribuídos a um autor, na realidade, eram de autoria de outro desenvolvedor (que não tem acesso à escrita direta no repositório). Isso ocorria porque, no início, o Ant usava o SVN como sistema de controle de versões, e o SVN possui apenas o fluxo de trabalho centralizado. Apenas posteriormente, o repositório do software e todo o seu histórico foi migrado para o Git. A informação do real autor vinha nas mensagens de *commits* por meio da frase: “*Submitted by:*” (submetido por), seguido do nome de um desenvolvedor e seu e-mail. Considerou-se, como autor, o desenvolvedor associado à informação de *submetido por*, quando seu nome vinha acompanhado de seu e-mail, pois isso possibilitaria utilizar a estratégia correta de identificação. Em casos em que não há o *submetido por*, acompanhado do nome do autor e de seu e-mail, o autor é o mesmo autor identificado pelo Git. Nos demais projetos, não se observou esse padrão e, portanto, quem fez o *commit* foi considerado o autor.

Em todos os projetos, pode haver o caso de um mesmo desenvolvedor ser autor ou fazer *commits* com nomes e e-mails diferentes, podendo parecer que são desenvolvedores distintos. Para tratar desses casos, seguiu-se a regra de considerar o e-mail como identificador de usuário. Esse é um problema comum, quando se está fazendo mineração em repositórios de software (PONCIN; SEREBRENIK; VAN DEN BRAND, 2011; SANTOS et al., 2018; ZANONI et al., 2014). O procedimento adotado neste trabalho é baseado em estratégias utilizadas por autores que se depararam com o mesmo problema - p.ex.: Businge et al. (2017) a diferença é que esse autor usa primeiro o login e depois o nome. O procedimento seguido é:

- 1) Procurar um usuário com mesmo nome.
- 2) Se não achar, procurar um com o mesmo e-mail.
- 3) Se não encontrar, é um novo desenvolvedor.

Pela Equação 1, a experiência de qualquer desenvolvedor no primeiro *commit* será sempre 0, pois ele não fez qualquer atividade prévia no código. A partir do segundo *commit*, a experiência é determinada considerando todas as contribuições do autor nos *commits* anteriores dele. Isso significa considerar a quantidade de *commits* que ele já deu, e a quantidade de arquivos e linhas que ele alterou até o *commit* atual. Exemplificando, no primeiro *commit* de um autor *A*, ele alterou 3 arquivos e 80 linhas de código no componente θ , e 20 linhas em 2 arquivos do componente λ . No segundo *commit* com autoria de *A*, no qual ele alterou os componentes θ e σ , sua experiência geral atual será, pela Equação 1: $0,2 \times 1 + 0,4 \times (3 + 2) + 0,4 \times (80 + 20) = 42,2$.

A mesma equação é usada para a experiência especializada, porém, só se consideram as contribuições dentro do componente ao qual se refere a experiência que se pretende determinar. No segundo *commit* de *A*, a experiência do autor em θ é: $0,2 \times 1 + 0,4 \times 3 + 0,4 \times 80 = 33,4$, e sua experiência especializada em um outro componente σ é 0, pois esse é o primeiro *commit* de *A* em σ . Assim, a experiência cresce a cada *commit* e não pode ser igual à experiência de um *commit* anterior.

6.7.2 Dados para a degradação da arquitetura

O valor que corresponde à métrica *D* (distância da sequência principal) é extraído dos arquivos compilados referentes aos *commits*. Para obter o valor de *D*, utiliza-se a ferramenta Arcan. Por causa dessa ferramenta, a coleta dos dados da degradação é feita a partir do arquivo jar, resultante da compilação da versão associada a um *commit*. A ferramenta gera um arquivo CSV com as métricas para pacotes de Robert C. Martin (CA, CE, I, A e D). A Arcan recupera a informação arquitetural, a partir do código-fonte, construindo um grafo de dependência com pacotes e classes do sistema mapeados (FONTANA et al., 2017b).

Alguns *commits* não resultaram em versões compiláveis do software por diversos motivos, por exemplo, pelo descuido do desenvolvedor em não incluir uma biblioteca que era necessária para a compilação. Outra situação em que não foi possível compilar versões ocorreu quando as configurações necessárias para fazer a compilação não puderam ser replicadas. Quando não foi possível gerar versões compiláveis e, portanto, não foi possível calcular a métrica de degradação da

arquitetura para o *commit*, ele foi desconsiderado. Além disso, para evitar que dados com origem em erros de medição fossem incluídos, verificou-se, manualmente, situações em que havia arquivos .jar com tamanho incompatível (muito pequeno) com os *commits* anterior e os consecutivos, e cujo valor absoluto da diferença da degradação, entre esses *commits*, era muito distante da média. Caso fosse possível identificar a mudança feita no *commit* e ela fosse compatível com o valor do delta, o *commit* era considerado. Caso contrário, era desconsiderado, por tratar-se provavelmente de um erro de medição¹⁸ (neste caso, compilação).

6.8 Procedimento de análise dos dados

Esta seção detalha como os dados são utilizados para responder à questão de pesquisa desta análise; ou seja, se a experiência de um desenvolvedor tem relação com a qualidade da arquitetura do software. Ao longo desta seção, expõe-se a abordagem utilizada para isso, a análise de impacto de *commits* (BEHNAMGHADER et al., 2017b), que foi utilizada para correlacionar a experiência do autor de um *commit* à qualidade da sua contribuição, no que diz respeito à arquitetura, em cada *commit* realizado entre as versões incluídas na análise, para cada um dos projetos selecionados.

A análise de impacto de *commits* (BEHNAMGHADER et al., 2017b) permite que seja possível rastrear diretamente e individualmente a contribuição de um desenvolvedor, para a qualidade do software. Isso é possível porque se analisa o impacto (com respeito à alguma métrica de software) de cada *commit* ao longo de um período, olhando a diferença entre o impacto do *commit* atual e o impacto do anterior. Para fazer uso dessa abordagem, é necessário o uso de uma ferramenta de análise estática de código, para extrair métricas diretamente do código referente a um dado *commit*, em um repositório gerenciado por um sistema de controle de versões (p.ex.: Git, SVN).

¹⁸ Os *commits* foram compilados de forma automatizada e eles não deveriam gerar arquivos .jar caso houvesse erro de compilação, mas em alguns casos algumas classes foram compiladas apesar do erro de compilação do *commit*. Isso gerou jars muito pequenos. Essa situação ocorreu em todos os projetos, mas no Lucene e no Hadoop a frequência de ocorrência foi desprezível. Porém no Ant e no Cassandra, houveram por volta de 15 a 20 *commits* impactantes nessa situação.

Nesta pesquisa, ao utilizar-se a análise de impacto de *commits*, busca-se associar o resultado de um *commit* impactante à experiência do seu autor. Quanto ao impacto do *commit*, neste trabalho, se refere ao valor da variação na média da métrica D, dos componentes da arquitetura, entre esse *commit* e o *commit* pai. O pai de um *commit* é seu ancestral imediato no histórico de desenvolvimento. A análise de impacto de *commits* prevê a existência de três tipos de *commits* (BEHNAMGHADER et al., 2017b):

- **Normal:** um *commit* com apenas um pai;
- **Merge:** *commit* com dois ou mais pais e
- **Inicial ou órfão:** *commit* sem pai.

Neste trabalho, não se considera *commits* de *merge*, porque, a partir desse tipo de *commit*, não é possível rastrear a contribuição individual de um desenvolvedor, uma vez que um *merge* herda o histórico de *commits* do seu *branch* de origem, sem distinguir os diferentes desenvolvedores envolvidos na evolução desse *branch*. Então, tem-se apenas *commits* normais e órfãos. Além disso, para esta análise considerou-se apenas *commits* no *branch* master.

Além disso, a análise considerará somente o impacto em *commits* no módulo principal, seguindo a sugestão de Behnamghader et al. (2017b). Segundo os autores, o módulo principal corresponde àquele que contém a maioria do código-fonte. Dentre os projetos selecionados, o Ant e o Cassandra possuem apenas um módulo. No Lucene, a maioria do código-fonte está no diretório “core”; por isso ele será considerado o módulo principal. No caso do Hadoop, a partir da versão 2.2.x, a arquitetura do software é composta por vários módulos compilados de forma independente. Um deles, o *Hadoop Common*, concentra grande parte do código-fonte do projeto. Por esse motivo e, também, porque ele continha os arquivos de código-fonte das versões anteriores (antes da divisão em módulos independentes), permitindo, assim, rastrear a evolução da arquitetura nas mesmas estruturas, esse módulo foi considerado o módulo principal. O *Hadoop Common* é, também, o módulo que contém as bibliotecas e arquivos comuns e necessários para os demais módulos do Hadoop.

Na abordagem original, o cálculo de impacto de um *commit* normal c_i para uma determinada característica de qualidade observada é dado por *valor da métrica*(c_i) – *valor da métrica*(c_{i-1}), onde c_{i-1} é o *commit* pai. Neste trabalho, a métrica utilizada

é a métrica D (distância da sequência principal) (MARTIN, 2017), que é uma métrica de pacotes¹⁹. Para medir a qualidade da arquitetura de uma versão, será utilizado a média da métrica D para os componentes da arquitetura. Se essa média tende a 0, indica que a maioria dos componentes têm boa relação entre dependências e capacidade de evolução; isto é, que mudanças nesses componentes implicam em pouco efeito cascata. Por outro lado, se esse valor tende a 1, significa que na média, os componentes desse sistema são instáveis com muitas dependências, tendendo a ter maior efeito cascata. Logo, se a média de D tende a 0, a arquitetura tem boa manutenibilidade.

Portanto, o cálculo de impacto de um *commit* normal c_i quanto à manutenibilidade é dado por: $D(c_i) - D(c_{i-1})$, onde c_{i-1} é o *commit* pai. Caso $D(c_i) \neq D(c_{i-1})$, diz-se que o *commit* é impactante – ou seja, ele impactou a arquitetura. Caso se trate de um *commit* órfão, se ele é um *commit* com código-fonte, logo, ele é um *commit* impactante e considera-se seu impacto sendo apenas igual a $D(c_i)$.

Para maior precisão na medição da contribuição realizada, utilizou-se a mesma estratégia de Amanatidis et al. (2017) e Alfayez et al. (2018), e a informação da quantidade de linhas modificadas no *commit* foi adicionada ao cálculo do impacto. Essa informação dá a noção do quanto um autor contribuiu para a qualidade da arquitetura, por linha de código escrita por ele. Assim, se permite considerar a quantidade de exposição do indivíduo ao código no *commit* em questão. Em teoria, se uma pessoa escreve mais linhas, ela está mais propensa a causar impacto do que uma que faz contribuição de poucas linhas. Com a adição dessa informação, o impacto será calculado agora por: $\Delta = \frac{D(c_i) - D(c_{i-1})}{CLOC(c_i)}$. O delta (Δ) é, portanto, a quantidade de degradação por linha modificada entre dois *commits*. Daqui para frente, o impacto de um *commit* será referenciado apenas como Δ ou delta, para denotar a diferença entre o estado anterior e o atual (do *commit* analisado). Além disso, apenas *commits* impactantes serão analisados, pois busca-se estudar a relação da experiência de quem impacta arquitetura. Assim, *commits* cujo delta é igual a 0 não participam do teste de correlação.

¹⁹ Neste trabalho, baseando-se em Barros, Farzat e Travassos (2015), utiliza-se o mapeamento de componentes da arquitetura para pacotes (diretórios) do software, portanto, deste ponto em diante, quando se fala de componentes, está se falando de pacotes.

Em resumo, a análise é feita considerando para cada *commit* c_i a experiência (geral para QP1, e especializada para QP2) acumulada pelo autor até c_i e o Δ , que representa o impacto arquitetural em c_i . O exemplo da Tabela 6.7 mostra os dados usados para a análise de um *commit* impactante. Os dados usados para responder QP1 estão no par ordenado formado por: (experiência geral, delta). De forma análoga, para QP2 tem-se: (experiência especializada, delta [componente]). No exemplo dado, o par na análise seria: (1013; 0,000048).

Tabela 6.7 – Exemplo de dados utilizados na análise de um *commit* impactante.

Dados	<i>Commit</i> c_{i-1}	<i>Commit</i> c_i
Autor	A	B
Experiência geral	0	1013
CLOC	2	438
D	0,084615	0,105828
D(c) - D(c_{pai})	0	0,021213
Delta	0	0,000048

6.9 Ferramenta

Analisar a grande quantidade de dados, em um repositório, é uma tarefa demorada e repetitiva que pode ser automatizada. Há algumas ferramentas²⁰ disponíveis para fazer mineração em repositórios de software. Elas podem fazer extração de dados a partir de *commits* e fornecer estatísticas sobre a contribuição de desenvolvedores ou, até mesmo, sobre a evolução do código e da arquitetura. Também, existem *frameworks* em várias linguagens, que fornecem funções básicas para extração de dados, e que podem ser usados para construção de ferramentas com funções personalizadas para análises mais específicas.

Devido às características das métricas utilizadas nesta análise, optou-se por desenvolver uma ferramenta nova, uma vez que as ferramentas disponíveis consultadas não disponibilizavam as informações necessárias para o cálculo das

²⁰ <https://github.com/topics/mining-software-repositories>

métricas de experiência e arquitetura deste trabalho. Além disso, a implementação da ferramenta também possibilitou fazer a análise de impacto de *commits*.

Tanto a coleta de dados quanto a análise foram realizadas na ferramenta *Mining my Repository*²¹. Ela foi desenvolvida em Python utilizando o *framework* PyDriller (SPADINI;ANICHE;BACCHELLI,2018), o qual permite analisar repositórios Git. A ferramenta possibilita escolher um projeto por vez para processar as informações, sendo que é preciso carregar, pelo menos uma vez, os *commits* das versões (*releases*) que se deseja analisar, antes de executar os passos para a análise de impacto de *commits* (gerar as versões compiladas de cada *commit*).

A métrica de experiência é calculada no instante em que os *commits* são carregados e armazenados em um banco de dados, no módulo *Commits*. A Figura 6.1 mostra a interface desse módulo na ferramenta. O tempo total gasto nessa etapa depende da quantidade de *commits* a ser analisado. Em média, cada projeto analisado neste trabalho – na versão final da ferramenta - consumiu de 2 a 3 dias para o carregamento de *commits* e o cálculo da experiência.

A compilação de cada *commit* carregado, o cálculo da métrica da arquitetura (D) para cada *commit* e o delta (da análise de impacto de *commits*) são calculados também de forma automatizada, após a etapa de carregamento de *commits*, no módulo *Arquitetura*, cuja interface é apresentada na Figura 6.2. O tempo total desta etapa depende de alguns fatores: a forma de compilação do projeto, a disponibilidade das bibliotecas utilizadas no desenvolvimento e o hardware que está executando os processos da ferramenta. Esta foi a etapa mais trabalhosa do processamento, por ser necessário identificar a forma de compilar os *commits* de cada versão, localizar as bibliotecas e dependências necessárias para compilar os *commits* e configurar o ambiente para isso e, por fim, compilar cada *commit*, o que consumia muitos recursos da máquina em que o software estava rodando. Essa etapa consumiu mais de uma semana por projeto, sendo que o resultado da análise de *commits* levou em média 2 dias por projeto (ela é executada após a compilação das versões). Durante o desenvolvimento da ferramenta, as etapas anteriores (carregamento de *commits*, cálculo da métrica de experiência, compilação de versões e análise de impacto de

²¹ https://github.com/renatabrasil/mining_my_repository

Figura 6.1 - Interface do módulo de *Commits*.

Contribuições - Commits

Informações

Projeto: Apache Hadoop

Versão: release-2.2.0

☒ Carregar commits **Atualizar**

may take several minutes

Aviso

Apenas commits que possuem arquivos .java serão mostrados.

100 commits encontrados.

Page 1 of 2. [next](#) [last](#)

#	hash	Versão	Tem LOC de código:	CLOC (no blank lines)	delta	Committer	Autor	Author Exp
1	bce31805e9b4b1360d50be8e001886d58e087e38	rel/1.1	True	6871	0,00000000	James Duncan Davidson	James Duncan Davidson	0,00
2	282f346ca230a8dec8d1956af05fcc9d511ad672	rel/1.1	True	104	0,00000000	Sam Ruby	Kare Nuorteva	0,00
3	fd883b6209472356b4fd69d79b9b3534863c988	rel/1.1	True	4	0,00000000	Sam Ruby	Glenk Twigg	0,00
4	edb71783adeffc29a56df203f97b5ac0e4566a5d	rel/1.1	True	696	-0,00000504	Stefano Mazzocchi	Stefano Mazzocchi	0,00
5	fd287b3a808daed93021646efdcf52a31957e35a	rel/1.1	True	392	0,00000000	Stefano Mazzocchi	Stefano Mazzocchi	279,80

Fonte: a autora.

Figura 6.2 - Interface no módulo da Arquitetura.

Todas essas ações podem demorar varios minutos.

É importante ter um arquivo de configuração que especifique:

Lembrando que é preciso ter configurado *Apache Ant* e *Apache Maven*, e ter colocado a variavel de ambiente **ANT_HOME**, **M2_HOME** e **JAVA_HOME** nas variáveis de ambiente.

Configurar a variavel de ambiente **JAVA_HOME** e selecionar a versão correta do JDK. É necessário descobrir qual a versão utilizada na versão do software a ser analisado. Esse tipo de informação deve estar em arquivos de configuração.

Diretório para salvar os commits e os compilados:

compiled/apache-hadoop

Local do repositório local: G:/My Drive/Mestrado/USP/programacao/projetos/git/ant

Diretório de arquivos compilados no repositório local: build/classes

Gerar lista de commits

#	Arquivo	Compilados?	Métricas calculadas	Ações
1	compiled/apache-hadoop/commits-rel-release-3.0.0.txt	False		🔍 📄 🗑️
2	compiled/apache-hadoop/commits-release-0.20.203.0.txt	False	11/08/2020 18:05	🔍 📄 🗑️
3	compiled/apache-hadoop/commits-release-1.0.1.txt	False	11/08/2020 18:00	🔍 📄 🗑️
4	compiled/apache-hadoop/commits-release-2.2.0.txt	False	11/08/2020 23:21	🔍 📄 🗑️

Fonte: a autora.

commits) consumiram ainda mais tempo, pois foi preciso enfrentar todos esses obstáculos pela primeira vez. Exemplificando, foi necessário realizar essas etapas, várias vezes, para um mesmo projeto, até que se assegurasse de que a

implementação estava em conformidade com as métricas e passos descritos nesta análise.

No outro módulo da ferramenta, as informações coletadas e processadas nas etapas anteriores são armazenadas no banco de dados para gerar as análises. O módulo de *Análise Estatística* fornece arquivos CSV com informações, tais como: a correlação da experiência com a degradação no sistema, a correlação por desenvolvedor e estatística sobre *commits* impactantes e informações para utilizar na estatística descritiva. A interface desse módulo pode ser vista na Figura 6.3. É a etapa que consome menos tempo, pois, cada CSV com informações consome em média 1 minuto para ser gerado.

Figura 6.3 - Interface do módulo de Análise Estatística.

Fonte: a autora.

6.10 Considerações finais

Este capítulo apresentou o conjunto de decisões para definir o *design* da análise (contexto, o objetivo e as questões de pesquisa e hipóteses) e descreveu as etapas do método de análise, que utiliza técnicas de *Mineração de Repositórios de Software*. etapas do método de análise, que utiliza técnicas de *Mineração de Repositórios de Software*. A apresentação das etapas incluiu a seleção das métricas a serem

utilizadas, a seleção dos projetos analisados, a qual se fez adotando critérios claros e que visavam minimizar algumas ameaças à validade. Uma das principais preocupações foi escolher projetos nos quais não havia a presença de outras causas de degradação. Nas etapas referentes ao procedimento de coleta e à análise de dados, a definição foi feita de acordo com as métricas escolhidas. O procedimento de análise utilizado (BEHNAMGHADER et al., 2017b) já é utilizado em outras pesquisas da área.

O próximo capítulo apresentará os resultados da análise empírica e as ameaças à validade do estudo.

7 RESULTADOS E DISCUSSÕES

Este capítulo apresenta os resultados da análise empírica. A hipótese referente à QP1 considera que, quanto maior é a experiência, menos degradação é introduzida no sistema por um desenvolvedor. Por outro lado, a hipótese relativa à QP2 considera que isso é verdade com relação aos componentes, ou seja, quanto maior é a experiência do desenvolvedor no componente, menor é a degradação que ele introduz nele. Para que essas suposições sejam verdadeiras, a experiência e o delta da degradação, tanto em QP1 quanto em QP2, devem ter correlação negativa. Neste trabalho, a degradação da arquitetura é avaliada sobre o ponto de vista da manutenibilidade.

As subseções deste capítulo estão divididas em: estatística descritiva (7.1), resposta à QP1 (7.2) e resposta à QP2 (7.3). A última seção deste capítulo se refere à discussão sobre os resultados obtidos, incluindo as ameaças à validade. A estatística descritiva é apresentada a partir de *box plots* - que mostram a distribuição de dados e os dados discrepantes (*outliers*) das variáveis, se houverem, - e um resumo dos dados em tabelas. Segundo Wohlin et al. (2012), *box plots* são adequados para representar a dispersão e amplitude das amostras. O resumo dos dados é apresentado por meio de medidas numéricas sumárias (média, desvio padrão, mínimo, máximo e quartis). Quartil é uma medida de localização, que divide a amostra em quatro partes iguais (ROSS, 2009).

O objetivo no teste de hipótese é rejeitar as hipóteses nula formuladas para QP1 e QP2 na Seção 6.3, que dizem que a experiência não tem nenhuma relação com a degradação da arquitetura (denotada pelo coeficiente de correlação $\rho=0$). A hipótese nula é rejeitada em favor da hipótese alternativa. A hipótese alternativa H_{A1} , apresentada na Seção 6.3, enuncia que a experiência do desenvolvedor, baseada em quantidade de linhas desenvolvidas, quantidade de arquivos alterados e quantidade de *commits*, está inversamente relacionada à degradação ($\rho_1 < 0$). Em outras palavras, espera-se que, quanto mais linhas desenvolvidas, arquivos alterados e *commits* realizados, menor será a degradação introduzida por um desenvolvedor em um *commit*. A hipótese alternativa H_{A2} , apresentada na Seção 6.3, afirma que a experiência especializada (experiência apenas no componente) está inversamente relacionada à degradação no componente ($\rho_2 < 0$). Em ambas as hipóteses a

correlação referenciada é a correlação de postos de *Spearman*, que é uma medida não paramétrica e pouco sensível a *outliers* (D'AMBROS; BACCHELLI; LANZA, 2010). Para avaliar a intensidade de uma correlação (fraca, moderada ou forte), pode-se adotar a regra presente em Ross (2009); segundo a qual, uma correlação fraca é aquela cujo ρ tem o valor entre 0 e $|0,5|$, uma correlação moderada tem ρ entre $|0,6|$ e $|0,7|$, e uma correlação forte tem ρ entre $|0,8|$ e $|1|$.

O nível de significância adotado para os testes de hipótese é $\alpha = 0,05$ e a hipótese nula será rejeitada se o valor-p $< \alpha$. O valor-p corresponde à probabilidade de se obter um valor para o teste estatístico, pelo menos, tão contraditório para a hipótese nula (assumindo a hipótese nula sendo verdadeira), quanto o valor que realmente se deu (ROSS, 2009). Portanto, valor-p pequeno indica pouca probabilidade de se obter um valor contraditório observado para a estatística de teste, isto é, pode-se rejeitar a hipótese nula.

7.1 Estatística Descritiva

A Figura 7.1 mostra a degradação da arquitetura nos quatro projetos analisados. O Hadoop é o que apresenta menor variação entre as versões (o eixo x corresponde às versões maiores ou versões estáveis de cada projeto), mas em todos os projetos não há uma tendência óbvia de aumento da degradação. Há momentos alternados entre piora e melhora da métrica D (distância da sequência principal) da arquitetura em todos os projetos; com exceção do Hadoop, que apresenta uma tendência geral de melhora da manutenibilidade ao longo das versões analisadas no gráfico.

A porcentagem de *commits* impactantes nos quatro projetos é em média 21,53%, e a porcentagem de autores que fizeram *commits* impactantes é em média 47,90%. A Tabela 7.1 mostra o resumo desses dados e outras estatísticas gerais nos 4 projetos. A maioria dos *commits* impactantes, nos 4 projetos, teve deltas (que medem a degradação entre *commits* por linhas de código modificadas) que variaram muito próximo de 0, indicando que a quantidade de variação da métrica D, em relação às linhas modificadas, foi muito pequena nesses *commits*. A Figura 7.2 mostra os *box plots* da variável delta, para os 4 projetos analisados. Nesse tipo de gráfico, a barra no centro da caixa é a mediana, a barra inferior da caixa corresponde ao 25º percentil (quartil inferior) e a barra que limita a parte de cima da caixa corresponde ao 75º

Figura 7.1 - Degradação da arquitetura nos projetos Ant, Cassandra, Hadoop e Lucene.

Fonte: a autora.

Tabela 7.1 - Resumo das estatísticas de *commits* analisados nos quatro projetos.

	Estatísticas Gerais			
	Ant	Cassandra	Hadoop	Lucene
Total de <i>commits</i> analisados	6742	7541	3022	9204
<i>Commits</i> compilados²²	6279 (93,13%)	5530 (73,33%)	2478 (81,99 %)	8257 (89,71%)
<i>Commits</i> impactantes	1454 (21,57%)	2654 (35,19%)	266 (8,80%)	1891 (20,55%)
Número de Autores	372	256	142	143
Número de autores que fizeram <i>commits</i> de impacto	158 (43,47%)	135 (52,73%)	61 (42,96%)	75 (52,45%)

percentil (quartil superior). Em todos os projetos, os valores se distribuem se distanciando bastante da média, como é possível notar pelo tamanho da caixa e pelos

²² Os motivos para um *commit* não ter sido compilado são: (1) a compilação não foi possível (por exemplo, por erros de compilação ou ausência de bibliotecas necessárias em versões antigas) e (2) *commits* que não alteram o código.

Figura 7.2 - Box plot da distribuição do delta nos quatro projetos.

Fonte: a autora.

pontos que se espalham ao longo do eixo da caixa, deixando todos os gráficos com formato de cruz. A distribuição do delta é, portanto, heterogênea.

A escala do delta, no gráfico, foi aumentada para prevenir erros de arredondamento, uma vez que os valores para a métrica D já são muito pequenos, pelo fato de D ter valores situados no intervalo $[0,1]$, e o delta ser obtido a partir da divisão pelo número de linhas modificadas. A Tabela 7.2 mostra um resumo dos dados relativos ao delta, por meio de medidas numéricas sumárias (média, desvio padrão, mínimo, máximo e quartis).

Tabela 7.2 - Resumo dos dados sobre o delta nos quatro projetos.

	Delta (10 ⁶)			
	Ant	Cassandra	Hadoop	Lucene
Média	4,94	-4,04	-10,61	-4,32
Desvio padrão	160,01	406,00	130,30	156,85
Mínimo	-1710,59	-1997,14	-1342,71	-1744,13
25%	-1,28	-10,39	-2,60	-2,21
50%	0,22	0,02	-0,03	0,09
75%	3,52	9,62	0,95	2,04
Máximo	1175,26	1976,54	645,08	1552,40

Em relação à variável experiência, a distribuição da experiência em todos os *commits* analisados (impactantes ou não), em todos os projetos, mostra que a maioria dos *commits* pertencem a autores com muito pouca ou nenhuma experiência. A Figura 7.3 mostra os *box plot* da distribuição da experiência nos 4 projetos. Nos gráficos, é possível perceber que a média e a distribuição da experiência nos quartis inferior e superior estão sempre próximas do mínimo. A Tabela 7.3 mostra um resumo dos dados sobre a experiência dos quatro projetos, por meio de medidas numéricas sumárias.

Figura 7.3 - *Box plot* da distribuição da experiência nos quatro projetos.

Fonte: a autora.

A Figura 7.4 mostra a porcentagem de *commits* impactantes por tempo de projeto do desenvolvedor no projeto. Nos 4 projetos, a maioria dos *commits* impactantes é de autoria de desenvolvedores com até 2 anos de projeto. Isso pode indicar que grande parte das contribuições que afetam a estrutura do sistema é de responsabilidade de pessoas com pouco tempo no projeto.

Tabela 7.3 - Resumo dos dados sobre a experiência nos quatro projetos.

	Experiência			
	Ant	Cassandra	Hadoop	Lucene
Média	14782,35	14302,56	4641,59	45465,17
Desvio padrão	16381,38	18492,55	14348,16	50179,22
Mínimo	0,00	0,00	0,00	0,00
25%	1985,25	1564,40	252,65	3454,35
50%	9558,20	5452,00	1034,50	25469,50
75%	21030,90	20491,20	3062,40	73167,55
Máximo	59378,00	79004,80	102939,00	189695,00

Figura 7.4 - Distribuição de *commits* impactantes por tempo dos autores nos quatro projetos.

Fonte: a autora.

7.2 QP1: Relação entre experiência geral e degradação da arquitetura

Esta análise tem como objetivo avaliar se H_{A1} é verdadeira, ou seja, descobrir se há correlação entre a experiência e a degradação da arquitetura (a partir do delta), ao longo das versões analisadas em todos os projetos. A Tabela 7.4 apresenta o coeficiente de *Spearman* ρ_1 e o valor-p para a análise de cada um dos projetos. Em

todos os projetos o resultado não é estatisticamente significativo (valor- $p > 0,05$). Além disso, o ρ_1 é muito próximo de 0 indicando que se houver correlação, ela é muito fraca. A análise gráfica dos diagramas de dispersão da experiência pelo delta, na Figura 7.5, já sugere essa conclusão, uma vez que há uma grande dispersão dos valores dessas variáveis nos quatro projetos.

Tabela 7.4 - Resultado do teste estatístico para QP1.

Projeto	ρ_1	Valor-p
Ant	0,0208	0,4275
Cassandra	-0,0092	0,6364
Hadoop	0,0500	0,4167
Lucene	-0,0192	0,4048

7.3 QP2: Relação entre experiência local e degradação da arquitetura no componente

Esta análise visa verificar se H_{A2} é verdadeira, ou seja, se o acúmulo da experiência em um componente pode significar menor degradação neste componente. A experiência especializada corresponde à experiência acumulada em codificação em um determinado componente. Para analisar apenas esse tipo de experiência, considera-se somente a degradação no componente alterado e despreza-se a degradação que ocorre em outros componentes. Se houver correlação, entende-se a importância e a contribuição da experiência local para a qualidade da arquitetura no componente, pois quanto mais experiência o desenvolvedor adquire naquela parte do sistema, menos degradação ele induz naquele componente. Em outras palavras, o desenvolvedor pode manter a qualidade em um nível local, ainda que ele possa provocar degradação nos demais componentes do software; isto é, ele estaria degradando a arquitetura do sistema de forma involuntária.

Diferentemente da análise de H_{A1} , a análise de H_{A2} é feita considerando isoladamente cada componente em que houve impacto arquitetural (delta diferente de 0). As informações completas sobre cada projeto podem ser encontradas no Apêndice B. A Tabela 7.5 apresenta um resumo sobre QP2 nos componentes de cada projeto, ou seja, mostra a quantidade de componentes nos quais houve *commits* de impacto, a quantidade e porcentagem de componentes nos quais H_{A2} é rejeitada, e a quantidade

Figura 7.5 - Gráficos de dispersão da experiência X delta nos quatro projetos.

Fonte:

Fonte: a autora.

de componentes nos quais houve correlação fraca, moderada e forte (apenas os estatisticamente significantes). Todas as correlações estatisticamente significativas encontradas foram negativas; desse modo, o acúmulo de experiência local de um autor implica menos degradação no componente. Houve apenas dois componentes (1 no Cassandra e 1 no Hadoop) onde a correlação encontrada foi forte.

De uma maneira geral, a maioria das correlações foi fraca, com exceção do Hadoop, onde a maior parte das correlações estatisticamente significantes é moderada.

A quantidade pequena de componentes do Hadoop e do Lucene é devido ao fato de, nesses dois projetos, terem sido considerados apenas um módulo do software (diferentemente do Ant e do Cassandra).

7.4 Análise adicional (QPA)

As hipóteses relativas à QP1 e QP2 analisam a relação entre experiência e a

Tabela 7.5 - Resultado do teste estatístico para QP2.

Projeto	Componentes com <i>commits</i> de impacto	Componentes com H ₀ 2 rejeitada	% H ₀ 2 rejeitada			
				ρ fraco	ρ moderado	ρ forte
Ant	71	12	16,90%	10	2	0
Cassandra	97	40	41,24%	33	6	1
Hadoop	29	7	24,14%	2	4	1
Lucene	15	4	26,67%	3	1	0

degradação, considerando todas as contribuições impactantes de todos os desenvolvedores do sistema; isto é, o teste de hipótese considera todas as experiências de todos os autores de *commits*. Nesta seção, avalia-se a relação entre a experiência geral e a degradação da arquitetura no sistema (similar à QP1) para um único desenvolvedor; ou seja, se existe relação entre a evolução da experiência do desenvolvedor no sistema e a degradação da arquitetura causada por ele. Sendo assim, tem-se a questão de pesquisa adicional:

QPA: Existe relação entre a experiência de um desenvolvedor no sistema e a degradação da arquitetura causada por ele?

Hipótese nula adicional (H_{0A}): Não há relação entre a experiência do desenvolvedor e o impacto de suas contribuições para a qualidade da arquitetura.

Formalmente: $\rho_A = 0$.

Hipótese alternativa adicional (H_AA): Quanto maior é a experiência do desenvolvedor, melhor é o impacto de suas contribuições para a qualidade da arquitetura.

Formalmente: $\rho_A < 0$.

A Tabela 7.6 mostra um resumo da análise de QPA, na qual se evidencia que não há correlação entre a experiência e a degradação para a maioria dos desenvolvedores em cada projeto. A porcentagem de desenvolvedores em que há relação é muito baixa. Na tabela, são apresentados a quantidade de autores (desenvolvedores) que fizeram *commits* de impacto, o número de autores para os quais a hipótese nula foi rejeitada, e a quantidade de correlações fracas, moderadas

e fortes, além da quantidade de correlações que foram positivas e negativas. As informações relativas às correlações na tabela se referem somente a correlações estatisticamente significantes. Algumas correlações são positivas, isto é, contrariam a hipótese H_{A1} . Por este fato, fica claro que não há relação entre o acúmulo de experiência geral e menor degradação da arquitetura introduzida por um autor, nos projetos analisados. Os dados completos dessa análise podem ser encontrados na Tabela B-5 do Apêndice B.

Tabela 7.6 - Resultado do teste estatístico para QPA.

Projeto	Autores com commits de impacto	Autores com H_0A rejeitada	% H_0A rejeitada	ρ			$\rho > 0$	$\rho < 0$
				ρ fraco	ρ moderado	forte		
Ant	33	3	9,09%	0	0	3	2	1
Cassandra	51	9	17,65%	4	1	4	5	4
Hadoop	27	2	7,41%	0	0	2	1	1
Lucene	47	8	17,02%	2	0	6	3	5

7.5 Conclusão

Em todos os projetos analisados, embora não seja possível rejeitar H_01 , todas as correlações encontradas foram muito próximas de 0, indicando que, se houver correlação, ela é muito fraca. Pela análise visual nos gráficos de dispersão, foi possível perceber nos projetos que não há qualquer relação entre a experiência geral e a degradação em QP1. Com relação a QP2, nos componentes nos quais H_02 foi rejeitada, todas as correlações foram negativas, indicando que, neles, houve relação entre o acúmulo de experiência local e menos degradação induzida por esses autores em tais componentes. O resultado dos testes de QP1 sugere que um desenvolvedor experiente não degrada menos a arquitetura que um inexperiente. Em contrapartida, o resultado dos testes de QP2 sugere que desenvolvedores com experiência em um componente podem contribuir com mais qualidade nesse componente. Uma possível conclusão desses resultados é que desenvolvedores, mesmo os experientes, podem não estar cientes do impacto de suas mudanças nos demais componentes, tendo apenas uma noção local; ou que, nos projetos analisados, os desenvolvedores inexperientes no projeto são suficientemente bons tecnicamente e fazem boas contribuições arquiteturais.

Outra observação, a partir dos dados dos projetos, foi que a maioria dos *commits* impactantes em todos os projetos foi de autoria de desenvolvedores com até 2 anos de projeto. Em todos os projetos, também foi possível perceber que, conforme a experiência dos desenvolvedores aumentava, esses desenvolvedores eram autores de menos *commits*. Além disso, as faixas de experiência nas quais os autores mais impactaram a arquitetura – de acordo com o delta – contêm contribuições que melhoraram e pioraram a arquitetura. Ou seja, não parece haver um padrão.

Uma análise adicional foi realizada considerando a degradação da arquitetura por desenvolvedor, a questão de pesquisa adicional QPA, e não foi possível concluir se há relação entre experiência geral e degradação. Houve poucas correlações estatisticamente significantes, nos quatro projetos, e algumas delas foram positivas, contrariando a hipótese de pesquisa. Esses resultados indicam que pessoas inexperientes não obrigatoriamente degradam a arquitetura, e que desenvolvedores com muita experiência também podem degradar. Os resultados também sugerem que pessoas com pouca experiência também podem fazer contribuições que melhoram a arquitetura.

A ausência de conclusões estatisticamente significantes para QP1 pode ser devido a alguns fatores. Um deles pode ser o fato de os quatro projetos não degradarem com o tempo, o que, talvez, seja um indício de que o processo da Apache lide bem com a ocorrência de degradação arquitetural. O fato de os projetos não degradarem, contraria o entendimento de que a arquitetura tende a degradar com o tempo (LE et al., 2018). Outro fator pode ser devido à forma de medir a experiência, cuja outra opção a se considerar, seja o uso de métricas de experiência que considerem mais que a atividade de codificação, uma vez que desenvolvedores experientes podem adquirir *expertise* por meio de outros tipos de atividade. Além disso, outras variáveis podem influenciar na degradação da arquitetura (como foi visto na Seção 5.3). Apesar de várias causas terem sido minimizadas pela escolha do projeto, mais fatores que não foram considerados até então podem ter tido influência nesses resultados.

Os demais fatores que podem ter influenciado o resultado obtido são discutidos na seção a seguir.

7.6 Ameaças à validade

As ameaças à validade deste trabalho foram divididas segundo a classificação apresentado por Wohlin et al. (2012) em: validade de construção (7.7.1), validade interna (7.7.2), validade externa (7.7.3) e validade de conclusão (7.6.4).

7.6.1 Validade de construção

A validade de construção se refere à forma com a qual as métricas utilizadas foram operacionalizadas e se elas são, realmente, a representação dos conceitos sobre os quais a questão de pesquisa foi construída (RUNESON et al., 2012).

A experiência medida neste trabalho se baseia, exclusivamente, no histórico da atividade de codificação de um desenvolvedor via *commits*. Isso se deve ao fato de que a análise feita considera apenas a experiência de um desenvolvedor no código, e porque ela é realizada a partir de dados de arquivamento. Porém, além de não se levar em conta a existência de aspectos de habilidade técnica e conhecimento de Engenharia de Software que podem influenciar a medida de experiência, os *commits* não são o único meio de se rastrear esse tipo de experiência. Dados de repositórios, como o GitHub, também podem ser usados para rastrear a atividade de desenvolvedores via *pull requests* e contribuições em vários repositórios, por exemplo. Para minimizar essa ameaça à validade, foi conduzido um mapeamento sistemático sobre métricas de experiência, no qual concluiu-se que contagem de *commits* e de linhas modificadas estão entre as fontes de informação mais comumente usadas para quantificar a experiência de um desenvolvedor em Engenharia de Software.

A partir do mapeamento, também ficou claro que havia críticas quanto ao uso de apenas uma dessas informações na métrica (ROBBES; ROTH LISBERGER, 2013): elas poderiam não ser capazes de capturar com acurácia a noção de experiência, quando utilizadas sozinhas. Por isso, outra solução adotada foi usar uma métrica composta de mais de uma informação. No fim, a métrica utilizada neste trabalho levou em conta as duas informações mais utilizadas para medir experiência em Engenharia de Software, e a essas duas informações foram acrescentadas, ainda, a quantidade de arquivos distintos modificados, que foi incluída para representar a extensão da contribuição de um dado desenvolvedor; isto é, mostrar se ele contribui em várias

partes do sistema, o que pode representar maior conhecimento dos componentes e do relacionamento entre eles. Um detalhe interessante é que a quantidade de linhas modificadas exclui as linhas em branco; ação que não foi identificada em outros trabalhos que usam essa medida. As linhas em branco foram excluídas por não trazerem informação relevante sobre o acúmulo de experiência. A composição da métrica foi feita usando pesos em cada uma das três informações utilizadas, de acordo com a influência de cada uma para a experiência, sob o ponto de vista arquitetural. O lado negativo dessa abordagem é que a métrica agrega muitas informações ao mesmo tempo, o que pode torná-la sensível a ajustes.

Além dos pontos acima, outra ameaça é que a métrica não leva em conta fatores como o decaimento da experiência, que ocorre devido ao efeito de perda da memória ao longo do tempo (HATTORI; LANZA; ROBBES, 2012). Essa situação pode acontecer quando alguém passa muito tempo sem contribuir ou não contribui com frequência no projeto.

Em trabalhos futuros, considera-se acrescentar um fator de decaimento da memória na métrica e, também, o uso de outros tipos de informação de atividades dos desenvolvedores, durante o processo de desenvolvimento, como a revisão de código, por exemplo. Um exemplo, é a métrica de Thongtanunam et al. (2016), que acrescenta informações sobre a atividade de revisão de código à contagem de *commits* para calcular a experiência de um desenvolvedor.

Outra ameaça encontrada foi a possibilidade associar um autor errado em um *commit*. Esse é um problema que pode ser recorrente em mineração de repositórios de software (PONCIN; SEREBRENK; VAN DEN BRAND, 2011; SANTOS et al., 2018; ZANONI et al., 2014). Para contorná-lo, adotou-se uma estratégia (que foi descrita na Seção 6.6.1) para identificar o autor correto baseado no seu nome, login e e-mail.

A métrica utilizada para medir a qualidade da arquitetura, distância da sequência principal (D), embora seja uma métrica conhecida, faz parte de um conjunto com várias opções possíveis de métricas e ao qual não há nenhuma métrica consolidada que se sobressaia. A escolha de D foi feita por meio de uma análise que levou em conta o nível de maturidade, mais especificamente o nível de validação da métrica. Ela foi calculada com o auxílio da ferramenta Arcan (FONTANA et al., 2017b), que, além de recuperar a arquitetura implementada do software, calcula métricas de classes, pacotes e detecta anomalias arquiteturais. O uso dessas outras métricas é

uma alternativa de estudar a degradação da arquitetura e vem sendo discutida em trabalhos como o realizado por Le et al. (2018)

A métrica D é originalmente usada para avaliar a qualidade de pacotes (componentes) da arquitetura, mas, nesta pesquisa, ela foi empregada para avaliar a arquitetura do ponto de vista da manutenibilidade. Para isso utilizou-se a média de D em todos os componentes que integram a arquitetura do software, seguindo a sugestão do próprio autor da métrica (MARTIN, 2017). Além disso, como a métrica original tem valor no intervalo de $[0,1]$, os valores são sempre menores que 1, com delta como resultado da divisão desse número pelo número de linhas modificadas (inspirado em Alfayez et al. (2018)). Isso sempre resulta em valores muito pequenos, que, para evitar erros de arredondamento, foram multiplicados pelo fator 10^6 .

7.6.2 Validade interna

A validade interna é o aspecto de validação que analisa se há fatores além dos fatores investigados influenciando os resultados obtidos (RUNESON et al., 2012). As principais ameaças de validade interna, desta pesquisa, estão atreladas a capacidade técnica dos desenvolvedores; outras possíveis causas de degradação arquitetural não isoladas e características do projeto analisado. Outra ameaça, diz respeito a *commits* que não puderam ser compilados.

Os desenvolvedores que podem fazer *commits* nos projetos selecionados só têm essa permissão após algum tempo no projeto; seja escrevendo documentação, seja reportando erros. Por isso, além de eles terem um conhecimento mínimo de Engenharia de Software, seu grau de experiência não é de uma pessoa realmente novata no projeto. De fato, o desenvolvedor não tem experiência de alterar o código, mas conhece o projeto e a documentação do sistema. Portanto, eles podem possuir uma experiência suficientemente adequada da arquitetura. Os projetos foram escolhidos de forma a minimizar a diferença técnica (não necessariamente ligada ao código) entre os desenvolvedores, mas trouxe à tona essa ameaça. Em trabalhos futuros, deve ser considerado também o uso de projetos com a característica de ter desenvolvedores sem experiência prévia no sistema e no código.

A presença de outras causas de degradação poderia interferir no resultado da análise. Na escolha dos projetos procurou-se minimizar a maioria delas: a falta de

documentação ou documentação pobremente escrita; requisitos imprecisos, mal escritos ou conflitantes; falta de ênfase em requisitos arquiteturalmente críticos; desenvolvedores que não entendem o *design* original; arquitetura não modularizada ou inapropriada para suportar mudanças facilmente e priorização de características em prejuízo da preocupação com a arquitetura. Por outro lado, algumas causas de degradação não puderam ser minimizadas ou controladas, uma vez que são projetos *open source* reais. Essas causas são: mudanças de requisitos, desenvolvedores nem sempre cientes do impacto arquitetural de suas contribuições, rotação de desenvolvedores, ausência de uma fase de planejamento para a arquitetura e desenvolvimento paralelo.

Outra ameaça é que não foi considerado os diferentes níveis de dificuldade das tarefas aos quais os *commits* estão associados. Tarefas mais complexas, tais como, uma nova funcionalidade, terão maior probabilidade de impactar a arquitetura do que uma de correção de erros, pois podem afetar várias áreas do sistema. Devido ao maior risco de impacto ao sistema, essas tarefas podem ter maior chance de serem executadas por desenvolvedores mais experientes. Em trabalhos futuros, é interessante fazer análise das questões de pesquisa deste trabalho distinguindo os *commits* quanto à categoria da sua tarefa de origem (p.ex.: aprimoramento, correção de erro). A escolha do projeto deve levar em conta a possibilidade de se fazer isso automaticamente. Uma das dificuldades a serem enfrentadas para isso, é conseguir rastrear a relação entre uma tarefa e os *commits* associados a ela (ANTONIOL et al., 2008). O problema é que alguns softwares não seguem uma convenção que possibilite fazer isso automaticamente. Esse é um obstáculo comumente encontrado na área de mineração de repositórios de software (KALLIAMVAKOU et al., 2014).

Na obtenção dos dados referentes à arquitetura, houve alguns problemas na geração de algumas versões (*commits*). Isso se deu, em grande parte, por problemas de configuração no ambiente de desenvolvimento (p.ex.: ausência de bibliotecas em versões muito antigas). Esse problema pode ter levado a valores discrepantes na amostra. Para minimizar essa ameaça, foi realizada uma etapa de inspeção e verificação manual nas versões compiladas. Aquelas, cujo tamanho do arquivo da versão compilada gerada era muito pequeno (em relação ao da versão compilada do *commit* anterior e do próximo) e cuja mensagem do *commit* não justificasse a

ocorrência de tal divergência, foram desconsideradas porque, nesses casos, há a probabilidade de se tratar de um erro de medição.

Por fim, nos projetos Hadoop e Lucene, há mais de um módulo e eles são compilados separadamente. Nesses dois projetos, apenas um desses módulos foi selecionado para análise. Na abordagem de análise de impacto de *commits*, apenas o módulo principal é considerado; no Lucene o módulo “Core” foi selecionado e no Hadoop, o Common. Em ambos, a experiência geral foi calculada considerando *commits* em todos os módulos, mas a análise de *commits* impactantes somente nos módulos selecionados, conforme foi discutido na Seção 6.8. Por isso, houve uma quantidade menor de componentes analisados quando se compara ao Ant e ao Cassandra, cujo projeto possuía apenas um módulo principal.

7.6.3 Validade externa

A validade externa refere-se ao aspecto de validade relacionado à capacidade de generalização dos resultados encontrados (RUNESON et al., 2012). As principais ameaças à validade externa deste trabalho são o número de projetos analisados e o uso de apenas projetos da Apache Foundation para a análise. O número de projetos não é representativo, considerando toda a população de projetos *open source* existente, porém, neste trabalho, não foi possível realizar uma análise em um conjunto maior de projetos, porque a análise de impacto de *commits* demanda um grande custo computacional e operacional. É necessário carregar e gerar o executável para um enorme número de *commits* por projeto. Para que fosse possível ampliar a quantidade de projetos a serem analisados, seria necessária uma infraestrutura melhor que permitisse escalabilidade no processamento.

Outra ameaça é que a análise está sujeita aos modelos de processo seguidos pela Apache (p.ex.: modelo de maturidade de processo, modelo de contribuição). Contudo, a escolha desses projetos garantiu que grande parte das ameaças à interpretação dos resultados, vindas, principalmente, da presença de outras causas da degradação, fossem minimizadas. Além disso, todos os projetos escolhidos já foram usados em outros trabalhos, em análises, seja de degradação da arquitetura, seja do impacto das atividades dos desenvolvedores. Acredita-se que resultados semelhantes podem ser obtidos analisando projetos que possuam essas

características. Porém, a fim de ampliar o alcance dos resultados, projetos com diferentes origens e processos de desenvolvimentos poderiam ser usados.

7.6.4 Validade de conclusão

A validade de conclusão refere-se a questões que afetam a capacidade de tirar a conclusão correta a partir do resultado de uma análise (WOHLIN et al., 2012).

Como os dados coletados foram muito heterogêneos, utilizou-se o teste de correlação de Spearman, para analisar a relação entre a experiência (geral e local) e a degradação, pois esse teste é não paramétrico e é menos sensível a *outliers*. Outros trabalhos similares (ALFAYEZ et al., 2018; SALAMEA; FARRÉ, 2019) também utilizaram esse teste devido à heterogeneidade dos dados, que têm origem em diferentes projetos *open source*.

O delta, variável que representa a degradação, é obtido por meio da diferença de degradação entre um *commit* e o *commit* ancestral imediato, dividida pelo número de linhas modificadas. Neste trabalho, acredita-se que essa seja uma representação correta da contribuição de um desenvolvedor, para a degradação da arquitetura. Essa mesma noção é baseada em Alfayez et al. (2018), que leva em conta a divisão por número de linhas modificadas, como uma forma de considerar o peso das contribuições de cada autor.

7.7 Considerações finais

Este capítulo apresentou o resultado da análise empírica para a relação entre a experiência geral e a degradação da arquitetura (questão de pesquisa 1). Não se encontrou correlação estatisticamente significativa. Por outro lado, os resultados para a experiência local (questão de pesquisa 2) mostraram que há alguma relação entre a experiência local e os impactos das contribuições nos componentes em nível arquitetural. O capítulo seguinte apresenta os trabalhos relacionados e a contribuição desta pesquisa nessas áreas.

8 TRABALHOS RELACIONADOS

Este trabalho está relacionado a pesquisas que investigam a relação entre a experiência de desenvolvedores e a qualidade de software e, também, a trabalhos que investigam a degradação da arquitetura de software, durante a evolução de um sistema. Tais pesquisas serão discutidas nas Seções 8.1 e 8.2, respectivamente.

8.1 Experiência de desenvolvedores e qualidade de software

Um dos possíveis fatores responsáveis pela piora da qualidade de um software, é a pouca ou a falta de experiência dos desenvolvedores, pois pessoas com menos contato prévio com o código estão mais propensas a introduzir defeitos (BIRD et al., 2011) e dívida técnica (ALFAYEZ et al., 2018). A presença de muitas pessoas trabalhando nos mesmos arquivos também faz com que os desenvolvedores acumulem pouca experiência, devido à baixa propriedade de código (BIRD et al., 2011). A propósito, a propriedade de código é uma das formas de medir a experiência e a distribuição de responsabilidades, em determinados trechos do sistema (componentes), ou no sistema inteiro.

Nesse contexto, Rahman e Devanbu (2011) e Bird et al. (2011) são trabalhos que estudam a experiência de desenvolvedores via propriedade de código e utilizam a taxa de defeitos como meio para medir a qualidade de um software. Uma das premissas consideradas por esses trabalhos é a de que o aumento da quantidade de defeitos está associado ao aumento dos custos de manutenção, dificultando a manutenção e incorporação de novos requisitos. Em Rahman e Devanbu (2011), os autores estudaram propriedade de código em quatro projetos *open source*, mas considerando a propriedade em linhas de código em arquivos. A noção de contribuição leva em conta linhas de código modificadas em *commits* associados à correção de erros; e a relação entre essas contribuições e a experiência do desenvolvedor. A investigação consistiu em verificar se fragmentos de código que estiveram envolvidos em correção de erros são resultado da contribuição de muitos desenvolvedores. Os resultados desse trabalho indicam que a experiência de uma pessoa em arquivos específicos importa mais do que a experiência no sistema inteiro, pois, apenas a falta de experiência local esteve de forma consistente associada à

fragmentos de código envolvidos em *commits* que introduziram defeitos nos sistemas analisados. Recomenda-se que, para corrigir erros, é melhor que se escolha alguém com experiência naquela parte do sistema, ao invés de atribuir a alguém com experiência mais geral. Isso sugere, tal como a análise de QP2 nesta pesquisa, que a experiência local tem relação com a qualidade de software, naquele trecho do sistema. Porém, nesse estudo, se mede a qualidade do sistema apenas em relação a defeitos de software; enquanto, neste trabalho, a qualidade do software é medida via degradação da arquitetura.

Do ponto de vista de componentes, que podem ou não incluir vários arquivos de código, Bird et al. (2011) realizaram uma pesquisa que associou a propriedade de código à experiência de desenvolvedores e investigou se ela tem influência na quantidade de defeitos do software. Para isso, eles analisaram dois grandes projetos de código fechado: Windows Vista e Windows 7. Esses projetos utilizaram processos e políticas que reforçam a propriedade de código “forte”, isto é, a existência de partes do sistema que possuem um “dono” (um desenvolvedor com a maior parte das contribuições). Os resultados obtidos mostraram que a taxa de defeitos tem: (1) uma correlação positiva significativa com a quantidade de desenvolvedores secundários em um componente, (2) uma correlação positiva fraca com o número total de desenvolvedores em um componente e (3) uma correlação negativa com grandes níveis de propriedade de um desenvolvedor em um componente. Em outras palavras, há evidências de que a quantidade de desenvolvedores com pouca propriedade de código em componentes, isto é, desenvolvedores com pouca experiência local, está inversamente relacionada à quantidade de defeitos nesse componente em uma versão do software. Essa conclusão é semelhante à conclusão da análise da experiência local neste trabalho (QP2), pois há indícios de que, com relação a componentes, a experiência local acumulada está relacionada a contribuições que degradam menos a arquitetura nesses componentes. A diferença entre as duas conclusões, conforme Bird et al. (2011), é que o indicativo de qualidade para os softwares analisados é a taxa de defeitos, ao passo que, nesta pesquisa, se adota a degradação da arquitetura. Além disso, a relação entre a propriedade de código e a contribuição para os defeitos não é individualizada por desenvolvedor, tornando impossível rastrear a relação entre quem possui mais ou menos experiência (propriedade de código), e a quantidade de defeitos que essa pessoa introduziu. Em

contrapartida, nesta pesquisa, considerou-se a relação entre a experiência do autor do *commit* e o impacto arquitetural, ao empregar a análise de impacto de *commits*. Outra diferença, é que, em Bird et al. (2011), foram analisados projetos de código fechado, com políticas de desenvolvimento específicas da organização à qual os desenvolvedores pertenciam, a Microsoft. Nesta pesquisa, os projetos analisados eram *open source*. Há replicações de Bird et al. (2011); uma, utilizando projetos de código fechado (GREILER; HERZIG; CZERWONKA, 2015), na qual se confirmaram os resultados do estudo original, e outra, utilizando softwares *open source* (FOUCAULT et al., 2015), em que não foi possível confirmar as conclusões do estudo original.

Uma outra forma utilizada para determinar a experiência de desenvolvedores em um projeto – e que está mais próximo do que foi proposto por este trabalho - é usar as informações sobre as atividades desempenhadas no desenvolvimento. Alguns trabalhos (ALFAYEZ et al., 2018; AMANATIDIS et al., 2017; SALAMEA; FARRE, 2019) estudaram a relação entre a experiência e a qualidade de software, a partir da atividade de desenvolvedores no código (experiência), e da dívida técnica (qualidade). Nesses trabalhos, a dívida técnica é medida usando o SonarQube²³, que é uma ferramenta de análise estática de código que se utiliza de um conjunto de regras de qualidade, para detectar problemas no código. O SonarQube diagnostica a dívida técnica, com base em estimativas de esforço necessárias para corrigir problemas de manutenção detectados; ou seja, nesses trabalhos, a dívida técnica está relacionada apenas a código.

Amanatidis et al. (2017) realizaram um estudo de caso em quatro projetos *open source* em PHP, no qual investigaram se a dívida técnica era adicionada uniformemente pelos desenvolvedores do sistema, quais os tipos mais comuns de violação introduzidas no código e se a maturidade de um desenvolvedor no projeto (em dias) estava relacionada às contribuições que acumulavam dívida técnica. As conclusões desse trabalho mostraram que, nos projetos analisados, a dívida técnica introduzida estava concentrada nas contribuições de um grupo pequeno de desenvolvedores, e que a maioria dos desenvolvedores introduziam uma pequena quantidade de dívida técnica. Contudo, não foi possível dizer se há correlação entre a

²³ <https://www.sonarqube.org/>

maturidade no projeto (experiência) e a dívida técnica introduzida. Tanto desenvolvedores com bastante tempo de projeto, quanto os mais inexperientes contribuíram acumulando bastante dívida técnica, que é um resultado consistente com a conclusão da análise de QP1 deste trabalho. A análise da experiência por quantidade de dívida técnica acumulada foi individualizada e feita para cada desenvolvedor que realizou uma quantidade mínima de 10 *commits* no código, ou seja, cada desenvolvedor possui uma experiência e a quantidade de dívida técnica que acrescentou nos *commits* analisados. Sendo assim, a unidade de análise são os desenvolvedores dos projetos analisados, diferentemente deste trabalho, no qual a unidade de análise são os *commits*. Por fim, foram considerados apenas os *commits* do ano mais recente.

Alfayez et al. (2018) fez um trabalho similar que explorou como as características de contribuição de diferentes desenvolvedores influenciam no acúmulo de dívida técnica no código. As características de contribuição referenciadas no trabalho são: senioridade do desenvolvedor (tempo contribuindo no projeto em meses), frequência de *commits* e intervalo de tempo entre *commits*. A qualidade foi avaliada por meio do acúmulo de dívida técnica em código ao longo de *commits*. Os dados foram extraídos de *commits* de 38 projetos Java e a dívida técnica foi obtida pelo uso de ferramentas de análise estática de código. Esse trabalho utilizou a abordagem de análise de impacto de *commits* (BEHNAMGHADER et al., 2017b); a qual possibilitou rastrear as contribuições individuais de cada desenvolvedor durante a evolução do sistema (acompanhada a partir dos *commits*), para a qualidade do código via dívida técnica. O procedimento de análise foi o mesmo utilizado neste trabalho, com a diferença de que, ao invés da dívida técnica, este utilizou a medição da degradação da arquitetura. Cada *commit* tem a experiência de seu autor e um delta, que contabiliza a quantidade de dívida técnica acrescentada; isto é, a unidade de análise são os *commits*, assim como nesta pesquisa. As conclusões desse trabalho indicam que: (1) quem faz mais *commits*, introduz menos dívida técnica; (2) quanto maior o tempo do desenvolvedor no sistema, mesmo fazendo poucos *commits*, menos dívida técnica ele adiciona, e (3) quanto maior o intervalo entre *commits*, mais dívida técnica é adicionada. Se comparada com a análise de QP1 desta pesquisa, a conclusão (1) contraria as conclusões deste trabalho, pois não houve correlação entre experiência e degradação da arquitetura, embora haja diferenças na métrica de

experiência (que considera informações além da quantidade de *commits*) e na qualidade da arquitetura. Por outro lado, a conclusão (2) é difícil de ser comparada às desta pesquisa, porque não há uma relação direta entre senioridade em um projeto e experiência alta nesta pesquisa. Pode haver desenvolvedores que fazem poucos *commits* e estão no projeto há muito tempo (pouca experiência acumulada), ou pessoas com pouco tempo de projeto, mas que são muito ativos (muita experiência acumulada). Por fim, a conclusão (3) indica que se alguém passa muito tempo sem contribuir, a chance de contribuir negativamente para a qualidade do software é maior, mas esse item não foi abordado nesta pesquisa.

Outra contribuição interessante de Alfayez et al. (2018) foi verificar que, de modo geral, nos projetos analisados, a introdução ou diminuição da dívida técnica não está uniformemente distribuída entre os desenvolvedores. Uma conclusão semelhante à de Amanatidis et al. (2017), mas com projetos em linguagens diferentes (PHP e Java). As principais diferenças para esta pesquisa é que, em Alfayez et al. (2018), a experiência é medida em termos de tempo de projeto, a partir da variável senioridade; ao passo que, nesta pesquisa, a experiência medida leva em conta a quantidade de *commits*, quantidade de arquivos distintos modificados e quantidade de linhas de código modificadas. Outra diferença é que, em Alfayez et al. (2018), se usa como indicativo de qualidade a dívida técnica; enquanto, neste trabalho, se usa a degradação da arquitetura.

Salamea e Farre (2019) também investigaram se fatores relacionados aos desenvolvedores têm alguma relação com o acúmulo da dívida técnica em um sistema. Para isso, foram investigados se o nível de participação (a partir da quantidade de linhas de código modificadas), a experiência (tomada em quantidade de meses no projeto) e o nível de comunicação em *commits* (tamanho dos comentários em *commits*) estavam correlacionadas com a quantidade de dívida técnica associada a um dado desenvolvedor ao longo dos *commits* de dois projetos *open source*. Os resultados concluíram que o nível de participação e experiência dos desenvolvedores têm correlação moderada positiva com a dívida técnica; e que a forma de comunicação de um desenvolvedor, em seus *commits*, tem uma correlação fraca com o acúmulo de dívida técnica de responsabilidade desse desenvolvedor. Nesse trabalho, a experiência considerada também se refere ao tempo no projeto, porém a quantidade de linhas modificadas foi utilizada como indicativo de participação

de um desenvolvedor no código; ou seja, que esse parâmetro está mais relacionado às características de contribuição do desenvolvedor do que à experiência. Além disso, diferentemente do realizado nesta pesquisa, a qualidade de software é medida via acúmulo de dívida técnica.

8.2 Evolução e degradação da arquitetura do software

A outra área relacionada ao objetivo de pesquisa deste trabalho é a evolução e a degradação da arquitetura. Para estudar os efeitos da degradação da arquitetura, durante a evolução de um software, é necessário ser possível medir a qualidade de uma arquitetura. Essa tarefa é complexa porque, no mínimo, é preciso possível recuperar a arquitetura implementada, o que não é trivial, e detectar problemas arquiteturais. Nesse sentido, há diversos trabalhos que avançaram nesses fundamentos (LE et al., 2018; PAIXAO et al., 2017). Esses trabalhos estão interessados em estudar os impactos da degradação da arquitetura, sobretudo, na manutenibilidade de sistemas e, também, em determinar o que pode estar relacionado à degradação.

Le et al. (2018) estudaram a natureza e o impacto das anomalias arquiteturais, pois muitos efeitos negativos de problemas arquiteturais são conhecidos, apenas, a partir do relato de experiências dos praticantes; mas não se sabe ao certo os reais impactos em um sistema. Para isso, a abordagem utilizada foi tentar relacionar os arquivos que continham a presença de anomalias a problemas de implementação; isto é, erros reportados nos *issue trackers* à quantidade de *commits* que envolviam esses arquivos. Essas relações poderiam evidenciar que, na presença de anomalias arquiteturais, haveria mais erros e, portanto, problemas de implementação e, também, que esses arquivos estão mais suscetíveis a sofrerem modificação. Ambas as situações afetam a manutenibilidade do sistema. Os autores consideraram apenas alguns tipos de anomalias. A abordagem foi aplicada em projetos *open source*, que confirmaram que os arquivos que continham anomalias estavam mais propensos a apresentar problemas de implementação do que arquivos “limpos”. Isso implica maior dificuldade de manutenção e evidencia uma relação entre anomalias arquiteturais e degradação da arquitetura, porém nada é discutido sobre o que pode causar esses problemas arquiteturais e nem sobre a qualidade dos desenvolvedores do sistema.

Paixao et al. (2017) estudaram se os desenvolvedores estão cientes do impacto arquitetural de suas mudanças e se isso pode levar a melhores decisões a fim de evitar a degradação da arquitetura. Para isso, os autores analisaram o histórico das mudanças realizadas a partir dos *commits* e das discussões ocorridas no processo de revisão de código. Nessa análise, foi considerado o motivo das mudanças (p.ex.: correção de erros, aprimoramento e nova funcionalidade) e se o que foi discutido sobre as soluções propostas continha preocupação ou conhecimento sobre os impactos arquiteturais que elas iriam implicar. A análise das discussões sobre os *commits* foi manual, aplicando técnicas de análise qualitativa nos comentários realizados durante a revisão de código. Porém, a etapa de avaliação do código nos *commits* foi automatizada com o auxílio de uma ferramenta para avaliação da arquitetura, a partir do código-fonte, que fornecia métricas de coesão (LCOF – *Lack of Cohesion of Files*) e acoplamento (*FanOut*). Tal como nesta pesquisa, apenas *commits* com mudanças que causaram impacto arquitetural (variação nas métricas LCOF e *FanOut*) são considerados. Os resultados apresentados mostraram que, nos projetos analisados, não é frequente que entre desenvolvedores se discuta os impactos arquiteturais de suas contribuições; mas o mesmo não ocorre quando são contribuições que explicitamente visam melhorar a arquitetura, como no caso de refatorações. Nesses casos, os desenvolvedores se mostraram conscientes das consequências de suas soluções em nível arquitetural. Isso evidenciou que, mesmo em projetos com boa estrutura de comunicação e com atividades como revisão de código e preocupação com a arquitetura, muitas vezes as pessoas podem não estar cientes do impacto de suas contribuições na arquitetura do software; o pode contribuir para a degradação da arquitetura ao longo do tempo. Essa conclusão é interessante, pois sugere que uma boa comunicação e atividades de revisão de código, por si só, não são suficientes para evitar degradação da arquitetura. Isso pode explicar o porquê de, na análise dos projetos realizada neste trabalho, desenvolvedores experientes também degradarem, mesmo compreendendo o *design* da arquitetura. Apesar da conclusão sobre desenvolvedores e degradação da arquitetura, o nível de experiência dos desenvolvedores não é tratado nesse trabalho, o que é feito aqui.

9 CONCLUSÃO

Este trabalho apresentou uma análise sobre a hipótese de que a experiência de desenvolvedores no código estaria inversamente relacionada à degradação da arquitetura em desenvolvimento *open source*. Essa hipótese foi levantada porque, segundo alguns autores, pessoas com menos contato prévio com o código estão mais propensas a introduzir defeitos (BIRD et al., 2011) e dívida técnica (ALFAYEZ et al., 2018). Pretendia-se, com isso, investigar se a experiência também se relacionava diretamente com a qualidade da arquitetura, já que a arquitetura é fundamental para a evolução do software com custo viável; além de uma boa arquitetura ser um fator para atrair mais voluntários para a comunidade de desenvolvedores, que é um dos motivos para o sucesso de projetos *open source*. Concluiu-se que, nos projetos analisados, a experiência não tem relação com a degradação da arquitetura em no que concerne ao sistema, mas, por outro lado, pessoas especialistas em componentes contribuem para a manutenção da qualidade da arquitetura nesses componentes.

Se as hipóteses fossem verdadeiras, a principal contribuição deste trabalho teria sido mostrar que, por meio de mudanças em políticas de desenvolvimento, seria possível minimizar a degradação da arquitetura, pois bastaria que se ajustassem as políticas de desenvolvimento para evitar que desenvolvedores inexperientes pudessem modificar regiões críticas para a integridade da arquitetura. Esta seria uma alternativa a medidas mais impactantes, tal como refazer a arquitetura do zero. Assim, os projetos *open source* poderiam manter arquiteturas de qualidade e atrair mais usuários para contribuir.

Não obstante, a principal conclusão deste trabalho é que a experiência geral dos desenvolvedores não é um motivo direto para a degradação da arquitetura, pelo menos, em projetos *open source* com as características dos projetos analisados. Contudo, desenvolvedores especialistas em componentes fazem contribuições melhores para a arquitetura nesses componentes. Isso quer dizer que todos degradam a arquitetura, independente da experiência que acumularam no sistema inteiro (experiência geral), mas que, em nível local, um especialista em um determinado trecho desenvolve soluções melhores, embora isso possa significar

impacto em outros componentes do sistema sem que ele tenha necessariamente consciência disso.

Essa conclusão sugere que todos, independente da experiência acumulada, devem ter atenção quanto ao impacto arquitetural de suas contribuições, pois todos estão suscetíveis a piorá-la. Esse resultado é consistente com Paixao et al. (2017), que concluiu que os desenvolvedores nem sempre estão cientes dos impactos arquiteturais de suas contribuições, ou seja, uma pessoa experiente também pode degradar a arquitetura porque pode não ser capaz de perceber e antecipar as consequências de suas alterações na estrutura do software.

Em resumo, concluiu-se que não se deve combater a degradação da arquitetura apenas atuando sobre as consequências da experiência de desenvolvedores no código, adquirida por meio de suas contribuições ao longo do tempo. As outras causas não podem ser desconsideradas. Deve-se investir em medidas para minimizar a existência delas nos projetos, a fim de preservar a qualidade da arquitetura. Apesar disso, é preciso considerar que este trabalho foi conduzido analisando apenas projetos da Apache Foundation. Isso significa que todos os projetos analisados possuíam basicamente as mesmas características de processo de desenvolvimento. Além disso, em todos eles, os desenvolvedores só começavam a contribuir escrevendo código depois de algum tempo contribuindo de outras formas no projeto²⁴ (p.ex.: reportando erros), ou seja, eles tinham alguma noção e familiaridade com o sistema. Com isso, pode ser que os desenvolvedores inexperientes (com base na métrica de contagem de contribuição no código), na realidade, não teriam pouco conhecimento, e que essa familiaridade com o sistema seja a razão pela qual desenvolvedores novatos também fizeram contribuições que melhoraram a arquitetura. Se isso for verdade, o processo de gerenciamento seguido pela Apache minimiza a ameaça à arquitetura relacionada à experiência dos desenvolvedores.

²⁴ Em Pugh (2020), o autor fala de sua experiência de ter se tornado um *committer* de um projeto da Apache *apenas* (de acordo com ele) após 4.662 dias depois do ingresso como colaborador; e compartilha suas dicas para acelerar esse processo.

9.1 Trabalhos futuros

As seguintes questões podem ser consideradas para a continuidade deste trabalho, algumas referentes às ameaças a validade:

- Verificar projetos de diferentes origens, que não somente a Apache Foundation, e com diferentes processos de desenvolvimento;
- Aumentar o conjunto de dados a ser analisado; o que implica analisar uma quantidade maior de projetos, para que seja possível ampliar a capacidade das conclusões. Para isso, é necessário desenvolver uma solução para a análise que seja escalável, tendo em vista o alto custo computacional e o tempo consumido pelo o processo para cada projeto;
- Considerar o grau de dificuldade das tarefas associadas aos *commits*; afinal, é possível que desenvolvedores mais experientes se envolvam em tarefas mais complexas, com maior chance de impacto arquitetural; e
- Melhorar da ferramenta *Mining My Repository*:
 - Deixar a ferramenta mais configurável, para que seja possível mudar as métricas mais facilmente e para que seja possível rastrear *commits* para tarefas.
 - Permitir que se possa adicionar mais projetos para a análise, e possibilitar que todas as configurações, para a execução deles, seja disponibilizada para alteração na própria interface da ferramenta. Hoje é preciso modificar manualmente no código.

REFERÊNCIAS

ALBATTAH, W. An Empirical Investigation of the Correlation between Package-Level Cohesion and Maintenance Effort. **International Journal of Advanced Computer Science and Applications**, v. 8, n. 3, p. 186–191, 2017.

ALFAYEZ, R. et al. An Exploratory Study on the Influence of Developers in Technical Debt. **2018 ACM/IEEE International Conference on Technical Debt**, p. 1–10, 2018.

AMANATIDIS, T. et al. **Who is producing more technical debt?** Proceedings of the XP2017 Scientific Workshops on - XP '17. **Anais...**New York, New York, USA: ACM Press, 2017

ANTONIOL, G. et al. Is it a bug or an enhancement? A text-based approach to classify change requests. **Proceedings of the 2008 Conference of the Center for Advanced Studies, CASCON'08**, 2008.

AZADI, U.; FONTANA, F. A.; TAIBI, D. Architectural Smells Detected by Tools: a Catalogue Proposal. **International Conference on Technical Debt (TechDebt 2019)**, n. March, 2019.

BALDWIN, C. Y.; CLARK, K. B. **Design rules: The power of modularity**. [s.l.] MIT press, 2000.

BANDI, A.; WILLIAMS, B. J.; ALLEN, E. B. **Empirical evidence of code decay: A systematic mapping study**. 2013 20th Working Conference on Reverse Engineering (WCRE). **Anais...**IEEE, out. 2013

BANITAAN, S.; ALENEZI, M. **DECOBA: Utilizing Developers Communities in Bug Assignment**. 2013 12th International Conference on Machine Learning and Applications. **Anais...**IEEE, dez. 2013

BARROS, M. DE O.; FARZAT, F. D. A.; TRAVASSOS, G. H. Learning from optimization: A case study with Apache Ant. **Information and Software Technology**, v. 57, n. 1, p. 684–704, jan. 2015.

BASS, L.; CLEMENTS, P.; KAZMAN, R. **Software Architecture in Practice**. 3rd. ed. New Jersey, United States: Addison-Wesley Professional, 2012.

BECK, K. **Extreme Programming Explained: Embrace Change**. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

BECK, K. et al. **Manifesto for agile software development**. Disponível em: <<http://agilemanifesto.org/>>. Acesso em: 29 abr. 2017.

BEHNAMGHADER, P. et al. A large-scale study of architectural evolution in open-source software systems. **Empirical Software Engineering**, v. 22, n. 3, p. 1146–1193, 2017a.

BEHNAMGHADER, P. et al. **Towards Better Understanding of Software Quality Evolution through Commit-Impact Analysis**. 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS). **Anais...IEEE**, jul. 2017b

BERGERSEN, G. R. et al. **Inferring Skill from Tests of Programming Performance: Combining Time and Quality**. 2011 International Symposium on Empirical Software Engineering and Measurement. **Anais...IEEE**, set. 2011

BHATTACHARYA, P. et al. Graph-based analysis and prediction for software evolution. **Proceedings - International Conference on Software Engineering**, p. 419–429, 2012.

BHATTACHARYA, P.; NEAMTIU, I.; FALOUTSOS, M. **Determining Developers' Expertise and Role: A Graph Hierarchy-Based Approach**. 2014 IEEE International Conference on Software Maintenance and Evolution. **Anais...IEEE**, set. 2014

BIRD, C. et al. **Don't touch my code!** Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE '11. **Anais...New York, New York, USA: ACM Press**, 2011

BROOKS JR, F. P. **The Mythical Man-Month**. [s.l.] Addison-Wesley Longman Publishing Co., Inc., 1995.

BRUNET, J. et al. On the evolutionary nature of architectural violations. **Proceedings - Working Conference on Reverse Engineering, WCRE**, p. 257–266, 2012.

BUSCHMANN, F.; HENNEY, K. Architecture and agility: Married, divorced, or just good friends? **IEEE Software**, v. 30, n. 2, p. 80–82, 2013.

BUSINGE, J. et al. **Code Authorship and Fault-proneness of Open-Source Android Applications**. Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering - PROMISE. **Anais...New York, New York, USA: ACM Press**, 2017

BUTLER, S. et al. **An investigation of work practices used by companies making contributions to established OSS projects**. Proceedings of the 40th International Conference on Software Engineering Software Engineering in Practice - ICSE-SEIP '18. **Anais...New York, New York, USA: ACM Press**, 2018

CALIKLI, G.; BENER, A.; ARSLAN, B. **An analysis of the effects of company culture, education and experience on confirmation bias levels of software**

developers and testers. Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10. **Anais...**New York, New York, USA: ACM Press, 2010

CANFORA, G. et al. How changes affect software entropy: an empirical study. **Empirical Software Engineering**, v. 19, n. 1, p. 1–38, 14 fev. 2014.

CHACON, S.; STRAUB, B. **Pro Git book.** Disponível em: <<https://git-scm.com/book/en/v2/Git-Basics-Viewing-the-Commit-History>>. Acesso em: 10 mar. 2020.

CLEMENTS, P.; KAZMAN, R.; KLEIN, M. **Evaluating software architectures: methods and case studies.** [s.l.] Addison-Wesley, 2002.

CONSTANTINOU, E.; KAPITSAKI, G. M. **Identifying Developers' Expertise in Social Coding Platforms.** 2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). **Anais...**IEEE, ago. 2016

COSTA, D. A. DA et al. **The impact of rapid release cycles on the integration delay of fixed issues.** [s.l: s.n.]. v. 23

COULIN, T. et al. Software Architecture Metrics: a literature review. **CoRR**, v. abs/1901.0, p. 1–15, 25 jan. 2019.

CROWSTON, K.; HOWISON, J. Hierarchy and centralization in free and open source software team communications. **Knowledge, Technology & Policy**, v. 18, n. 4, p. 65–85, dez. 2006.

CUNNINGHAM, W. **The WyCash portfolio management system.** Addendum to the proceedings on Object-oriented programming systems, languages, and applications (Addendum) - OOPSLA '92. **Anais...**New York, New York, USA: ACM Press, 1 abr. 1992

D'AMBROS, M.; BACCHELLI, A.; LANZA, M. **On the Impact of Design Flaws on Software Defects.** 2010 10th International Conference on Quality Software. **Anais...**IEEE, jul. 2010

DA SILVA, J. R. et al. **Niche vs. breadth: Calculating expertise over time through a fine-grained analysis.** 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER). **Anais...**IEEE, mar. 2015

DE SILVA, L.; BALASUBRAMANIAM, D. Controlling software architecture erosion: A survey. **Journal of Systems and Software**, v. 85, n. 1, p. 132–151, jan. 2012.

DING, W. et al. How do open source communities document software architecture: An exploratory survey. **Proceedings of the IEEE International Conference on**

Engineering of Complex Computer Systems, ICECCS, p. 136–145, 2014.

EICK, S. et al. Does code decay? Assessing the evidence from change management data. **IEEE Transactions on Software Engineering**, v. 27, n. 1, p. 1–12, 2001.

ELISH, M. O. Exploring the Relationships between Design Metrics and Package Understandability: A Case Study. 2010.

ERNST, N. A. et al. **Measure it? Manage it? Ignore it? software practitioners and technical debt**. Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015. **Anais...**New York, New York, USA: ACM Press, 2015

FERREIRA, M. et al. **Detecting architecturally-relevant code anomalies**. Proceedings of the 29th Annual ACM Symposium on Applied Computing - SAC '14. **Anais...**New York, New York, USA: ACM Press, 2014

FERREIRA, M.; VALENTE, M. T.; FERREIRA, K. A Comparison of Three Algorithms for Computing Truck Factors. **IEEE International Conference on Program Comprehension**, p. 207–217, 2017.

FONTANA, F. A. et al. Automatic detection of instability architectural smells. **Proceedings - 2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016**, p. 433–437, 2017a.

FONTANA, F. A. et al. **Arcan: A Tool for Architectural Smells Detection**. 2017 IEEE International Conference on Software Architecture Workshops (ICSAW). **Anais...**IEEE, abr. 2017b

FONTANA, F. A.; ROVEDA, R.; ZANONI, M. **Tool support for evaluating architectural debt of an existing system**. Proceedings of the 31st Annual ACM Symposium on Applied Computing - SAC '16. **Anais...**New York, New York, USA: ACM Press, 2016

FOUCAULT, M. et al. On the usefulness of ownership metrics in open-source software projects. **Information and Software Technology**, v. 64, p. 102–112, ago. 2015.

FOWLER, M. **Refactoring: improving the design of existing code**. [s.l.] Addison-Wesley Professional, 1999.

FREGNAN, E. et al. A survey on software coupling relations and tools. **Information and Software Technology**, v. 107, n. November 2018, p. 159–178, 2019.

GARCIA, J. et al. **Identifying Architectural Bad Smells**. 2009 13th European Conference on Software Maintenance and Reengineering. **Anais...**IEEE, 2009a

GARCIA, J. et al. Toward a Catalogue of Architectural Bad Smells. In: **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)**. [s.l: s.n.]. v. 5581 LNCSp. 146–162.

GREILER, M.; HERZIG, K.; CZERWONKA, J. **Code Ownership and Software Quality: A Replication Study**. 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories. **Anais...IEEE**, maio 2015

GUO, Y.; SEAMAN, C.; DA SILVA, F. Q. B. Costs and obstacles encountered in technical debt management – A case study. **Journal of Systems and Software**, v. 120, p. 156–169, 2016.

GUTWIN, C.; PENNER, R.; SCHNEIDER, K. **Group awareness in distributed software development**. Proceedings of the 2004 ACM conference on Computer supported cooperative work - CSCW '04. **Anais...New York, New York, USA: ACM Press**, 2004

HATTORI, L. P.; LANZA, M.; ROBBES, R. Refining code ownership with synchronous changes. **Empirical Software Engineering**, v. 17, n. 4–5, p. 467–499, 16 ago. 2012.

HOCHSTEIN, L.; LINDVALL, M. Combating architectural degeneration: a survey. **Information and Software Technology**, v. 47, n. 10, p. 643–656, jul. 2005.

ISO/IEC/IEEE. **42010: Systems and software engineering: Architecture description**, 2011.

ISO/IEC. **25010: Systems And Software Engineering: Systems And Software Quality Requirements And Evaluation (Square) – System And Software Quality Models**, 2011.

JOBLIN, M. et al. Classifying Developers into Core and Peripheral: An Empirical Study on Count and Network Metrics. **Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering, ICSE 2017**, p. 164–174, 2017.

KALLIAMVAKOU, E. et al. The promises and perils of mining GitHub. **Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014**, p. 92–101, 2014.

KAPITSAKI, G. M.; FOUTROS, P. **Dear Developers, your Expertise in One Place**. 2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA). **Anais...IEEE**, ago. 2017

KAZMAN, R. et al. Evaluating the Effects of Architectural Documentation: A Case Study of a Large Scale Open Source Project. **IEEE Transactions on Software Engineering**, v. 42, n. 3, p. 222–247, 2016.

KITCHENHAM, B.; CHARTERS, S. Guidelines for performing Systematic Literature reviews in Software Engineering Version 2.3. **Engineering**, v. 45, n. 4ve, p. 1051, 2007.

KOZIOLEK, H. et al. Measuring Architecture Sustainability. **IEEE Software**, v. 30, n. 6, p. 54–62, nov. 2013.

KURIC, E.; BIELIKOVÁ, M. **Estimation of student's programming expertise**. Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM '14. **Anais...**New York, New York, USA: ACM Press, 2014

LATORRE, R. Effects of Developer Experience on Learning and Applying Unit Test-Driven Development. **IEEE Transactions on Software Engineering**, v. 40, n. 4, p. 381–395, abr. 2014.

LE, D. M. et al. Relating architectural decay and sustainability of software systems. **Proceedings - 2016 13th Working IEEE/IFIP Conference on Software Architecture, WICSA 2016**, p. 178–181, 2016.

LE, D. M. et al. **An Empirical Study of Architectural Decay in Open-Source Software**. 2018 IEEE International Conference on Software Architecture (ICSA). **Anais...**IEEE, abr. 2018

LENHARD, J. et al. **Are code smell detection tools suitable for detecting architecture degradation?** Proceedings of the 11th European Conference on Software Architecture Companion Proceedings - ECSA '17. **Anais...**New York, New York, USA: ACM Press, 2017

LENHARD, J.; BLOM, M.; HEROLD, S. Exploring the suitability of source code metrics for indicating architectural inconsistencies. **Software Quality Journal**, v. 27, n. 1, p. 241–274, 8 mar. 2019.

LI, Z. et al. **An empirical investigation of modularity metrics for indicating architectural technical debt**. Proceedings of the 10th international ACM Sigsoft conference on Quality of software architectures - QoSA '14. **Anais...**New York, New York, USA: ACM Press, 2014

LI, Z.; AVGERIOU, P.; LIANG, P. A systematic mapping study on technical debt and its management. **Journal of Systems and Software**, v. 101, p. 193–220, 2015.

LINÅKER, J. et al. How Firms Adapt and Interact in Open Source Ecosystems: Analyzing Stakeholder Influence and Collaboration Patterns. In: DANEVA, M.; PASTOR, O. (Eds.). **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)**. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016. v. 9619p. 63–81.

LINDVALL, M.; TVEDT, R. T.; COSTA, P. An empirically-based process for software architecture evaluation. **Empirical Software Engineering**, v. 8, n. 1, p. 83–108, 2003.

MACCORMACK, A.; RUSNAK, J.; BALDWIN, C. Y. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. **Management Science**, v. 52, n. 7, p. 1015–1030, 2006.

MACIA, I. et al. **On the Relevance of Code Anomalies for Identifying Architecture Degradation Symptoms**. 2012 16th European Conference on Software Maintenance and Reengineering. **Anais...IEEE**, mar. 2012

MANKINS, J. C. Technology readiness levels: A white paper. **Office of Space Access and Technology, NASA**, n. April, p. 1–5, 1995.

MARTIN, R. C. **Clean Architecture: A Craftsman's Guide to Software Structure and Design**. 1st. ed. [s.l.] Prentice Hall Press, 2017.

MARTINI, A.; BOSCH, J.; CHAUDRON, M. Investigating Architectural Technical Debt accumulation and refactoring over time: A multiple-case study. **Information and Software Technology**, v. 67, p. 237–253, nov. 2015.

MATRAGKAS, N. et al. **Analysing the “biodiversity” of open source ecosystems: the GitHub case**. Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014. **Anais...New York, New York, USA: ACM Press**, 2014

MEDVIDOVIC, N.; TAYLOR, R. N. **Software architecture: Foundations, Theory, and Practice**. Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10. **Anais...New York, New York, USA: ACM Press**, 2010

MITCHELL, B. S.; MANCORIDIS, S. On the automatic modularization of software systems using the Bunch tool. **IEEE Transactions on Software Engineering**, v. 32, n. 3, p. 193–208, mar. 2006.

MO, R. et al. Decoupling Level: A New Metric for Architectural Maintenance Complexity. **Proceedings of the 38th International Conference on Software Engineering - ICSE '16**, p. 499–510, 2016.

MOCKUS, A.; FIELDING, R. T.; HERBSLEB, J. **A case study of open source software development: the Apache server**. Proceedings of the 2000 International Conference on Software Engineering. ICSE 2000 the New Millennium. **Anais...ACM**, 2000

MOCKUS, A.; FIELDING, R. T.; HERBSLEB, J. D. Two case studies of open source software development: Apache and Mozilla. **ACM Transactions on Software Engineering and Methodology**, v. 11, n. 3, p. 309–346, 1 jul. 2002.

MOCKUS, A.; HERBSLEB, J. D. Expertise Browser: A Quantitative Approach to Identify Expertise. **Proceedings of the 24th International Conference on Software Engineering. ICSE 2002**, v. 93, p. 379–380, 2002.

MORALES, J. et al. Programmer eXperience: A Systematic Literature Review. **IEEE Access**, v. 7, p. 71079–71094, 2019.

NAKAKOJI, K. et al. **Evolution patterns of open-source software systems and communities**. Proceedings of the international workshop on Principles of software evolution - IWPSE '02. **Anais...**New York, New York, USA: ACM Press, 2002

NORD, R.; TOMAYKO, J. E.; WOJCIK, R. **Integrating Software-Architecture-Centric Methods into Extreme Programming (XP)**. Pittsburgh, PA: [s.n.].

OIZUMI, W. et al. Code Anomalies Flock Together Exploring Code Anomaly Agglomerations for Locating Design Problems. 2016.

OLSSON, T.; ERICSSON, M.; WINGKVIST, A. **The relationship of code churn and architectural violations in the open source software JabRef**. Proceedings of the 11th European Conference on Software Architecture Companion Proceedings - ECSA '17. **Anais...**New York, New York, USA: ACM Press, 2017

ORRÚ, M.; MARCHESI, M. **A case study on the relationship between code ownership and refactoring activities in a Java software system**. Proceedings of the 7th International Workshop on Emerging Trends in Software Metrics - WETSoM '16. **Anais...**New York, New York, USA: ACM Press, 2016

ØSTERLUND, C.; CROWSTON, K. Boundary-spanning documents in online FLOSS communities: Does one size fit all? **Proceedings of the Annual Hawaii International Conference on System Sciences**, p. 1600–1609, 2013.

OZCAN KINI, S.; TOSUN, A. **[Research Paper] Periodic Developer Metrics in Software Defect Prediction**. 2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM). **Anais...**IEEE, set. 2018

PAIXAO, M. et al. **Are developers aware of the architectural impact of their changes?** ASE 2017 - Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering. **Anais...**2017

PARNAS, D. L. **Software aging**. Proceedings of 16th International Conference on Software Engineering. **Anais...**IEEE Comput. Soc. Press, 1994

PERRY, D. E.; WOLF, A. L. Foundations for the study of software architecture. **ACM SIGSOFT Software Engineering Notes**, v. 17, n. 4, p. 40–52, 1 out. 1992.

POHL, K.; RUPP, C. **Fundamentos da Engenharia de Requisitos**. São Paulo: T&M

Testes, 2012.

PONCIN, W.; SEREBRENIK, A.; VAN DEN BRAND, M. Process mining software repositories. **Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR**, p. 5–13, 2011.

PUGH, E. **Success at Apache: I Became an Apache Solr Committer in 4,662 Days. Here's how you can do it faster!** Disponível em: <<https://blogs.apache.org/foundation/entry/success-at-apache-i-became>>. Acesso em: 1 out. 2020.

QIU, Y. et al. **An Empirical Study of Developer Quality**. 2015 IEEE International Conference on Software Quality, Reliability and Security - Companion. **Anais...IEEE**, ago. 2015

RAHMAN, F.; DEVANBU, P. **Ownership, experience and defects**. Proceeding of the 33rd international conference on Software engineering - ICSE '11. **Anais...New York**, New York, USA: ACM Press, 2011

RAYMOND, E. The cathedral and the bazaar. **Knowledge, Technology & Policy**, v. 12, p. 23–49, 1999.

ROBBES, R.; ROTHLIBERGER, D. **Using developer interaction data to compare expertise metrics**. 2013 10th Working Conference on Mining Software Repositories (MSR). **Anais...IEEE**, maio 2013

ROBILLARD, P. N. The role of knowledge in software development. **Communications of the ACM**, v. 42, n. 1, p. 87–92, 1 jan. 1999.

ROSS, S. M. **Introduction to Probability and Statistics for Engineers and Scientists**. 4 edition ed. [s.l.] Academic Press, 2009.

ROVEDA, R. et al. Towards an Architectural Debt Index. **2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)**, p. 408–416, 2018.

RUBIN, K. S. **Essential Scrum: A Practical Guide to the Most Popular Agile Process**. 1st. ed. [s.l.] Addison-Wesley Professional, 2012.

RUNESON, P. et al. **Case Study Research in Software Engineering**. Hoboken, NJ, USA: John Wiley & Sons, Inc., 2012.

SALAMEA, M. J.; FARRE, C. **Influence of Developer Factors on Code Quality: A Data Study**. 2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C). **Anais...IEEE**, jul. 2019

SANTOS, A. et al. Mining Software Repositories to Identify Library Experts. **ACM International Conference Proceeding Series**, n. i, p. 83–91, 2018.

SARKAR, S.; KAK, A. C.; NAGARAJA, N. S. **Metrics for analyzing module interactions in large software systems**. 12th Asia-Pacific Software Engineering Conference (APSEC'05). **Anais...IEEE**, 2005

SARKAR, S.; KAK, A. C.; RAMA, G. M. Metrics for Measuring the Quality of Modularization of Large-Scale Object-Oriented Software. **IEEE Transactions on Software Engineering**, v. 34, n. 5, p. 700–720, set. 2008.

SCACCHI, W. Understanding the requirements for developing open source software systems. **IEE Proceedings - Software**, v. 149, n. 1, p. 24, 2002.

SCHETTINO, V. et al. **Towards Community and Expert Detection in Open Source Global Development**. 2019 IEEE 23rd International Conference on Computer Supported Cooperative Work in Design (CSCWD). **Anais...IEEE**, maio 2019

SCHILLING, A.; LAUMER, S.; WEITZEL, T. **Who Will Remain? An Evaluation of Actual Person-Job and Person-Team Fit to Predict Developer Retention in FLOSS Projects**. 2012 45th Hawaii International Conference on System Sciences. **Anais...IEEE**, jan. 2012

SEAMAN, C.; GUO, Y. Measuring and Monitoring Technical Debt. **Advances in Computers**, v. 82, p. 25–46, 2011.

SETHI, K. et al. **From retrospect to prospect: Assessing modularity and stability from software architecture**. 2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture. **Anais...IEEE**, set. 2009

SHAW, M. et al. Abstraction for software architecture and tools to support them. **IEEE Trans. Software Eng**, v. 21, n. 4, p. 314–335, 1995.

STALLMAN, R. **Why Open Source misses the point of Free Software**.

STEVANETIC, S.; ZDUN, U. **Software metrics for measuring the understandability of architectural structures**. Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering - EASE '15. **Anais...New York, New York, USA: ACM Press**, 2015

TEUSNER, R.; MATTHIES, C.; GIESE, P. **Should I Bug You? Identifying Domain Experts in Software Projects Using Code Complexity Metrics**. 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS). **Anais...IEEE**, jul. 2017

TEYTON, C. et al. **Automatic extraction of developer expertise**. Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering - EASE '14. **Anais...**New York, New York, USA: ACM Press, 2014

THONGTANUNAM, P. et al. **Revisiting code ownership and its relationship with software quality in the scope of modern code review**. Proceedings of the 38th International Conference on Software Engineering - ICSE '16. **Anais...**New York, New York, USA: ACM Press, 2016

TOM, E.; AURUM, A.; VIDGEN, R. An exploration of technical debt. **Journal of Systems and Software**, v. 86, n. 6, p. 1498–1516, 2013.

TREUDE, C.; FIGUEIRA FILHO, F.; KULESZA, U. **Summarizing and measuring development activity**. Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015. **Anais...**New York, New York, USA: ACM Press, 2015

TUFANO, M. et al. When and Why Your Code Starts to Smell Bad (and Whether the Smells Go Away). **IEEE Transactions on Software Engineering**, v. 43, n. 11, p. 1063–1088, 1 nov. 2017.

VAN GURP, J.; BOSCH, J. Design erosion: problems and causes. **Journal of Systems and Software**, v. 61, n. 2, p. 105–119, mar. 2002.

VENTERS, C. C. et al. Software sustainability: Research and practice from a software architecture viewpoint. **Journal of Systems and Software**, v. 138, p. 174–188, 2018.

WOHLIN, C. et al. **Experimentation in Software Engineering**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012. v. 1

WOHLIN, C.; AURUM, A. Towards a decision-making structure for selecting a research design in empirical software engineering. **Empirical Software Engineering**, v. 20, n. 6, p. 1427–1455, 2015.

WU, W. et al. **DREX: Developer Recommendation with K-Nearest-Neighbor Search and Expertise Ranking**. 2011 18th Asia-Pacific Software Engineering Conference. **Anais...**IEEE, dez. 2011

WU, Y. et al. **The Influence of Developer Quality on Software Fault-Proneness Prediction**. 2014 Eighth International Conference on Software Security and Reliability. **Anais...**IEEE, jun. 2014

WYRICH, M.; GRAZIOTIN, D.; WAGNER, S. A theory on individual characteristics of successful coding challenge solvers. **PeerJ Computer Science**, v. 5, n. 2, p. e173, 4 fev. 2019.

XIAO, L.; CAI, Y.; KAZMAN, R. Titan: a toolset that connects software architecture with quality analysis. p. 763–766, 2014.

YADAV, A.; SINGH, S. K.; SURJ, J. S. Ranking of software developers based on expertise score for bug triaging. **Information and Software Technology**, v. 112, n. March, p. 1–17, ago. 2019.

YE, Y.; KISHIDA, K. **Toward an understanding of the motivation of open source software developers**. 25th International Conference on Software Engineering, 2003. Proceedings. **Anais...IEEE**, 2003

ZANONI, M. et al. Pattern detection for conceptual schema recovery in data-intensive systems. **Journal of Software: Evolution and Process**, v. 26, n. 12, p. 1172–1192, 2014.

ZELLER, A. **Why Programs Fail, Second Edition: A Guide to Systematic Debugging**. [s.l.] Elsevier, 2009.

ZHANG, F. et al. An empirical study of the effect of file editing patterns on software quality. **Journal of Software: Evolution and Process**, v. 26, n. 11, p. 996–1029, nov. 2014.

ZHOU, M.; MOCKUS, A. **Developer fluency**. Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering - FSE '10. **Anais...New York, New York, USA: ACM Press**, 2010

ZHOU, M.; MOCKUS, A. **What make long term contributors: Willingness and opportunity in OSS community**. 2012 34th International Conference on Software Engineering (ICSE). **Anais...IEEE**, 4 jun. 2012

APÊNDICE A - INFORMAÇÕES DO MAPEAMENTO SISTEMÁTICO DAS MÉTRICAS PARA A EXPERIÊNCIA

Este Apêndice traz informações relativas aos estudos selecionados e às métricas do mapeamento sistemático das métricas de experiência, descrito no Capítulo 4.

A.1. Estudos selecionados

A Tabela A-1 traz os trabalhos selecionados no mapeamento sistemático. A coluna ID é o identificador com o qual cada estudo é referenciado ao longo do Capítulo 4.

Tabela A-1 - Artigos selecionados no mapeamento sistemático.

ID	Título	Artigo
T1	[Research Paper] Periodic Developer Metrics in Software Defect Prediction	(OZCAN KINI; TOSUN, 2018)
T2	Identifying Developers' Expertise in Social Coding Platforms	(CONSTANTINOU; KAPITSAKI, 2016)
T3	A Case Study on the Relationship between Code Ownership and Refactoring Activities in a Java Software System	(ORRÚ; MARCHESI, 2016)
T4	An Empirical Study of Developer Quality	(QIU et al., 2015)
T5	Determining Developers' Expertise and Role: A Graph Hierarchy-Based Approach	(BHATTACHARYA; NEAMTIU; FALOUTSOS, 2014)
T6	Dear Developers, your Expertise in One Place	(KAPITSAKI; FOUTROS, 2017)
T7	The Influence of Developer Quality on Software Fault-Proneness Prediction	(WU et al., 2014)
T8	Should I Bug You? Identifying Domain Experts in Software Projects Using Code Complexity Metrics	(TEUSNER; MATTHIES; GIESE, 2017)
T9	DREX: Developer Recommendation with K-Nearest-Neighbor Search and Expertise Ranking	(WU et al., 2011)
T10	Using developer interaction data to compare expertise metrics	(ROBBES; ROTH LISBERGER, 2013)

ID	Título	Artigo
T11	An analysis of the effects of company culture, education and experience on confirmation bias levels of software developers and testers	(CALIKLI; BENER; ARSLAN, 2010)
T12	DECOBA: Utilizing developers communities in bug assignment	(BANITAAN; ALENEZI, 2013)
T13	Influence of Developer Factors on Code Quality: A Data Study	(SALAMEA; FARRE, 2019)
T14	Effects of Developer Experience on Learning and Applying Unit Test-Driven Development	(LATORRE, 2014)
T15	Niche vs. breadth: Calculating expertise over time through a fine-grained analysis	(DA SILVA et al., 2015)
T16	Who Will Remain? An Evaluation of Actual Person-Job and Person-Team Fit to Predict Developer Retention in FLOSS Projects	(SCHILLING; LAUMER; WEITZEL, 2012)
T17	Inferring Skill from Tests of Programming Performance: Combining Time and Quality	(BERGERSEN et al., 2011)
T18	Towards Community and Expert Detection in Open Source Global Development	(SCHETTINO et al., 2019)
T19	Ranking of software developers based on expertise score for bug triaging	(YADAV; SINGH; SURI, 2019)
T20	A theory on individual characteristics of successful coding challenge solvers	(WYRICH; GRAZIOTIN; WAGNER, 2019)
T21	The Relationship of Code Churn and Architectural Violations in the <i>Open Source</i> Software JabRef	(OLSSON; ERICSSON; WINGKVIST, 2017)
T22	Revisiting Code Ownership and its Relationship with Software Quality in the Scope of Modern Code Review	(THONGTANUNAM et al., 2016)
T23	On the usefulness of ownership metrics in open-source software projects	(FOUCAULT et al., 2015)
T24	Summarizing and measuring development activity	(TREUDE; FIGUEIRA FILHO; KULESZA, 2015)
T25	An empirical study of the effect of file editing patterns on software quality	(ZHANG et al., 2014)
T26	Refining code ownership with synchronous changes	(HATTORI; LANZA; ROBBES, 2012)

ID	Título	Artigo
T27	Ownership, Experience and Defects: A Fine-Grained Study of Authorship	(RAHMAN; DEVANBU, 2011)
T28	Automatic extraction of developer expertise	(TEYTON et al., 2014)
T29	Developer fluency: Achieving true mastery in software projects	(ZHOU; MOCKUS, 2010)
T30	Estimation of student's programming expertise	(KURIC; BIELIKOVÁ, 2014)
T31	Code Ownership and Software Quality: A Replication Study	(GREILER; HERZIG; CZERWONKA, 2015)
T32	Don't Touch My Code! Examining the Effects of Ownership on Software Quality	(BIRD et al., 2011)

A.2. Métricas para experiência

Esta Seção traz as métricas do mapeamento sistemático, apresentadas na Tabela A-2. A coluna *Dados* traz as informações necessárias para calcular as métricas, a coluna *Relativa/Absoluta* informa se no cálculo da métrica consideram-se os dados de contribuição dos demais desenvolvedores (relativa) ou não (absoluta); e a coluna *Origem da métrica* traz a informação sobre meio pelo qual a métrica é calculada. Nesta tabela, d corresponde a um desenvolvedor e D o conjunto de desenvolvedores de um sistema.

Tabela A-2 - Métricas do mapeamento sistemático da experiência.

Artigo	Dados	Relativa/Absoluta	Origem da métrica	Tipo de experiência	Métrica
T1	CLOC e Contagem de <i>commits</i>	Relativa	Fórmula	Código (componente e arquivo)	<p>ExpLoc(d)/ExpDirectoryLoc(d): $\frac{ CLOC }{\sum_D CLOC(d) }$, para arquivo e para diretório.</p> <p>ExpCommit(d)/ExpDirectoryCommit(d): $\frac{ commits }{\sum_D commits(d) }$, para arquivo e para diretório.</p>
T2	CLOC, Contagem de arquivos e Contagem de <i>commits</i>	Relativa	Fórmula	Linguagem e Código (projeto)	<p>Exp(d, linguagem): $média(ProjExp(d, k)_{ling}) * \frac{\max(PArquivos_{ling})}{\max(PArquivos)}$</p> <p>ProjExp(d, k)_{ling}: $cw * CommitExp(d, k) + fw * CommitFileExp(d, k) + lw * ChangedLOCExp(d, k)$</p> <p><i>CommitExp</i>, <i>CommitFileExp</i> e <i>ChangedLOCExp</i> são calculados usando a proporção de <i>commits</i>, arquivos distintos e linhas modificadas por <i>d</i>. K é o ano.</p>
T3	Contagem de <i>commits</i>	Relativa	Fórmula	Código (arquivo)	<p>Subjective Ownership[SO](d, f): $\frac{ commits(d, f) }{\sum_f^F commits(d, f) }$, $f \in F$, sendo F arquivos que fazem parte de um <i>commit</i>.</p> <p>Relational Ownership[RO](d, f): $\frac{ commits(D, f) }{\sum_d^D commits(d, f) }$</p>
T4	Contagem de <i>commits</i>	Relativa	Fórmula	Código (componente e projeto)	<p>Ownership(d) para um componente: $\frac{ commits }{\sum_D commits(d) }$</p> <p>Ownership(d) para o projeto com M componentes: $\frac{\sum_M commits }{\sum_M \sum_D commits(d) }$</p>
T5	CLOC, Contagem de arquivos, Tempo (codificando e corrigindo erros), Severidade do erro (<i>issue tracker</i>) e Quantidade de erros corrigidos	Relativa	Fórmula	Código (projeto) Correção de erros	<p>source-code expertise profile(d): $(CLOC_d, arquivos_d , senioridade_código_d, ownership_d)$</p> <p>bug-fixing expertise profile(d): $(contagem_erros_d, errosev_d, senioridade_erro_d, erros_corrigidos_d)$</p> <p>ownership(d): $\frac{CLOC}{\sum_D CLOC(d)}$, erros_corrigidos(d): $\frac{ erros_corrigidos(d) }{\sum_D erros_corrigidos(d) }$</p>
T6	Informações de Plataforma de Codificação Social: GitHub, Bitbucket, Stack Overflow, etc.	Relativa	Fórmula	Linguagem	<p>Expertise(EXP): $w_1 * (w_{1a} * \frac{followers_{GH}}{maxEAfollowers_{GH}} + w_{1b} * \frac{accountDuration_{GH}}{maxEAaccountDuration_{GH}}) + w_2 * \frac{followers_{BB}}{maxEAfollowers_{GH}} + w_3 * (w_{3a} * \frac{badges_{gold}}{maxEAbadges_{gold}} + w_{3b} * \frac{badges_{silver}}{maxEAbadges_{silver}} + w_{3c} * \frac{badges_{bronze}}{maxEAbadges_{bronze}})$</p> <p>Knowledge/MostExperienced: $w_1 * repositories_{GH} + w_2 * repositories_{BB} + w_3 * (w_{3a} * answers_{SO} + w_{3b} * upvotes_{SO}) + w_4$</p>

Artigo	Dados	Relativa/Absoluta	Origem da métrica	Tipo de experiência	Métrica
					$\ast (w4a \ast tweets + w4b \ast retweets + w4c \ast likes)$
T7	Contagem de <i>commits</i>	Relativa	Fórmula	Código (projeto, componente e arquivo)	Ownership(d): $\frac{ commits }{\sum_D commits(d) }$
T8	Contagem de <i>commits</i>	Absoluta	Fórmula	Código (Projeto e componente)	scoreT(d): $q_i \ast \ln(1 + commits(d, T))$, T é um ponto no tempo qi: $\min\left(\frac{increasesT(a)}{decreasesT(a)}, 1\right)$, considera quando há alteração nas métricas de complexidade para calcular a qualidade das contribuições.
T9	Comentários em relatórios de erros (<i>issue tracker</i>)	Absoluta	Grafo (Rede Social)	Correção de erros	Nós: desenvolvedores, Arestas: comentários nas <i>issues</i> . Especialista e experiência determinados por um ranking construído usando medidas em grafos.
T10	Contagem de <i>commits</i>	Relativa	Fórmula	Código (arquivo)	Exp(d): $\frac{ commits }{\sum_D commits(d) }$ ExpDecay(d,t): $\frac{\sum_{c=1}^C \frac{1}{t - Date(c)}}{\sum_D commits(d) }$ C: todos os <i>commits</i> que <i>d</i> deu no arquivo e t: instante que a experiência é medida
T11	Tempo (em anos)	Absoluta	Atribuído/informado pela pessoa	Programação	Experiência: tempo (em anos)
T12	Comentários em relatórios de erros (<i>issue tracker</i>)	Absoluta	Grafo (Rede Social)	Correção de erros	Nós: desenvolvedores, Arestas: comentários nas <i>issues</i> . Especialista e experiência determinados por um ranking construído usando medidas em grafos.
T13	Tempo (em meses)	Absoluta	Fórmula	Código (projeto)	Timepart(d): $ultimo_commit_d - primeiro_commit_d$
T14	Tempo (em anos)	Absoluta	Atribuído/informado pela pessoa	Engenharia de Software	Experiência: tempo (em anos)
T15	Contagem de <i>commits</i>	Relativa	Fórmula	Código (projeto, componente, arquivo e método)	Expertise de um desenvolvedor (ED): $ commits(d) $, $z - score = \frac{(x - \mu)}{\sigma}$, x: $ commits(d) $, μ: média de x de todos os desenvolvedores no artefato e σ: desvio padrão de todos os x no artefato
T16	Contagem de <i>commits</i>	Absoluta	Fórmula	Código (projeto)	Níveis de experiência: baixa: em média 0-3 <i>commits</i> , intermediária: em média 4-94 <i>commits</i> , alta: em média > 94 <i>commits</i> .
T17	Tempo (em anos), Nível de experiência e CLOC	Absoluta	Atribuído/informado pela pessoa	Programação	InProfExp: log(tempo (em anos)) SEJavaExp: escala Likert de 1 a 5, 1: novato, 5: experiente InLOCJava: log(linhas escritas em Java(estimativa))
T18	Informações de Plataforma de Codificação Social –	Absoluta	Grafo (Rede Social)	Projeto	Nós: desenvolvedores, Arestas: comentários de revisão de código em <i>pull requests</i> .

Artigo	Dados	Relativa/Absoluta	Origem da métrica	Tipo de experiência	Métrica
	GitHub: <i>pull requests</i>				<ul style="list-style-type: none"> - Contribuições mais antigas têm peso menor (arestas com peso menor). - O peso é exponencialmente menor considerando os dias entre a data considerada e o dia que interação ocorreu.
T19	Metadados de erros em relatórios de erros (<i>issue tracker</i>)	Absoluta	Fórmula	Correção de erros	<p>Developer Expertise Score (DES): $(\alpha_1 * \mu_d + \alpha_2 * v_d + \alpha_3 * t_d)$</p> <p>$\mu_d: \sum_{i=1}^5 \omega_i * \left(\frac{\beta_i^d}{\beta_i}\right)$, prioridade dos erros corrigidos por d.</p> <p>$v_d: -\sum_{j=1}^K p_j^d * \log(p_j^d)$, $p_j^d: \left(\frac{\sigma_j^d}{\xi_j}\right)$, versatilidade e índice de profundidade em relação a <i>expertise</i> nos produtos que d já fez correção de erros.</p> <p>$t_d: \left(\frac{\lambda_r^d}{\lambda_b^d}\right)$, tempo médio de correção de erros</p>
T20	Tempo (em anos), Pontuação em desafios de código, Reputação no Stack Overflow e Número de <i>pull requests</i> no GitHub	Absoluta	Atribuído/informado pela pessoa	Programação	<p>Experiência (anos): tempo (em anos)</p> <p>Experiência (pontuação em desafios de código): experiência com desafios no último ano (numa escala de frequência de 6 pontos)</p> <p>Experiência (baseada em Stack Overflow): reputação no Stack Overflow</p> <p>Experiência (baseada em GitHub): quantidade de <i>pull requests</i></p>
T21	CLOC	Relativa	Fórmula	Código (arquivo)	Coefficiente de Gini usando CLOC do desenvolvedor no arquivo.
T22	Contagem de <i>commits</i> e Comentários em revisões de código	Relativa	Fórmula	Código (projeto, componente e arquivo)	<p>Traditional code ownership metric(d,M): $\frac{ commits }{\sum_D commits(d) }$, M: módulo ou componente</p> <p>Review-Specific Ownership Heuristics(d,M): $\frac{\sum_{k=1}^{r(D,M)} p(D,k)}{ commits }$, $r(d,M)$: número de revisões de código que d participou em M e $p(d,k)$: normalização das contribuições de d no <i>commit</i> k.</p>
T23	CLOC e Contagem de arquivos	Relativa	Fórmula	Código (projeto, componente e arquivo)	<p>Ownership(peso:arquivos): $\frac{ arquivos }{\sum_D arquivos(d) }$</p> <p>Ownership(peso:LOC): $\frac{ CLOC }{\sum_D CLOC(d) }$</p>
T24	Tempo (em anos)	Absoluta	Atribuído/informado pela pessoa	Programação	Experiência: tempo (em anos)
T25	CLOC	Relativa	Fórmula	Código (projeto, componente e arquivo)	Ownership(d): $\frac{ CLOC }{\sum_D CLOC(d) }$
T26	Linhas adicionadas	Relativa	Fórmula	Código (arquivo)	own(d,f): $own(d,fn-1) \frac{LOC-linhas_add}{LOC} +$

Artigo	Dados	Relativa/Absoluta	Origem da métrica	Tipo de experiência	Métrica
					$\begin{cases} \frac{linhas_add}{LOC}, & \text{se o autor do commit é } d \\ 0, & \text{se o autor do commit não é } d \end{cases}$ <p>Métrica considerando a curva de esquecimento:</p> $own(d,t): own(d,fn-1) + \begin{cases} e^{\frac{-\Delta t}{s}}, & \text{se o autor do commit é } d \\ 0, & \text{se o autor do commit não é } d \end{cases}$ <p>s: tamanho da memória. Quanto tempo a pessoa esquece o conhecimento. É escolhido de acordo com fatores como número de projetos que a pessoa contribui, número de desenvolvedores no projeto atual. T: é dado por número de dias entre a primeira mudança no arquivo desde o último <i>commit</i> até o momento que a propriedade de código é medido.</p>
T27	CLOC	Relativa e absoluta	Fórmula	Código (projeto e linhas de código em arquivos)	<p>Experiência geral(d): $\sum CLOC$, no sistema todo.</p> <p>Experiência especializada(d): $\frac{ CLOC }{\sum_D CLOC }$, linhas de código modificadas em linhas escritas para corrigir erros (<i>implied code</i>).</p>
T28	CLOC	Relativa	DSL/Fórmula	Código (projeto, componente, arquivo e método)	Contagem de todas as linhas modificadas em todos os <i>commits</i> de um d que representem habilidades definidas na DSL.
T29	Tempo (em anos)	Absoluta	Atribuído/informado pela pessoa	Projeto	Experiência: tempo (em anos)
T30	Tempo de execução de tarefa e Número de atividades necessárias para terminar uma tarefa	Relativa	Fórmula	Código (projeto)	<p>Exp(T C A): $\sum EXP(T C A)(d, tc)$</p> <p>ExpertiseTime(d,T): $\sum_{tc \in T} \frac{1}{\log_2(1+TD(d,t))}$, tc: tarefa, T: tempo para terminar tc</p> <p>ExpertiseActivity(A): $\sum_{tc \in T} \frac{MED_{EA}(tc)}{EA(d,tc)}$, EA: número de eventos relacionados a atividades para entregar tc.</p> <p>ExpertiseComplexity(C): $\sum_{tc \in T} \frac{MED_{PM}(tc)}{PM(d,tc)}$,</p>
T31	Contagem de <i>commits</i>	Relativa	Fórmula	Código (arquivo)	Ownership(d): $\frac{ commits }{\sum_D commits(d) }$
T32	Contagem de <i>commits</i>	Relativa	Fórmula	Código (componente)	Ownership(d): $\frac{ commits }{\sum_D commits(d) }$

APÊNDICE B - ANÁLISE DE QP2 E ANÁLISE ADICIONAL

Este Apêndice traz as tabelas com os dados da análise de QP2 e os dados da análise adicional, o que foi discutido no Capítulo 7. O item B1 refere-se aos dados da análise de correlação de QP2 para cada componente em que houve *commit* de impacto. O item B2, por sua vez, refere-se aos dados da análise adicional.

B.1. Dados da análise de QP2

Os dados da análise de QP2 para todos os componentes para os quais houve *commits* de impacto serão apresentados nas tabelas a seguir. Cada tabela é de um projeto e contém o nome do componente, o valor do coeficiente de *Spearman* para QP2, ρ_2 , o valor-p obtido pelo teste de correlação e o número de desenvolvedores diferentes que já fizeram *commits* no componente. As colunas em cinza e os valores em negrito com asterisco referem-se aos resultados estatisticamente significantes.

B.1.1. Ant

No Ant poucos componentes (16,90%) apresentam correlação entre experiência local e degradação do componente. Em dois dos principais componentes da arquitetura do Ant, o `src/main/org/apache/tools/ant/taskdefs`, e o `src/main/org/apache/tools/ant` houve correlação fraca e negativa. Nos demais componentes nos quais também se encontra a correlação, há uma considerável quantidade de pessoas que fazem contribuições e há também correlações negativas fracas (10) e moderadas (2).

Tabela B-1 - Dados relativos ao teste de correlação em QP2 para o Ant.

Componente	ρ_2	Valor-p	Número de desenvolvedores
<code>src/main/org/apache/tools/ant/taskdefs</code>	-0,1647	0,0000	119
<code>src/main/org/apache/tools/tar</code>	-0,0107	0,9417	13
<code>src/main/org/apache/tools/ant/taskdefs/optional</code>	-0,0865	0,2431	47

Componente	ρ_2	Valor-p	Número de desenvolvedores
src/main/org/apache/tools/ant	-0,2135	0,0000	68
src/main/org/apache/tools/ant/taskdefs/optional/javacc	-0,2736	0,0796	19
src/main/org/apache/tools/ant/taskdefs/optional/ejb	-0,2270	0,0166	39
src/main/org/apache/tools/ant/taskdefs/optional/vss	0,0170	0,9110	23
src/main/org/apache/tools/mail	0,0514	0,7913	13
src/main/org/apache/tools/ant/types	-0,0661	0,4152	30
src/main/org/apache/tools/ant/taskdefs/optional/jlink	-0,1390	0,4982	12
src/main/org/apache/tools/ant/taskdefs/optional/depend	-0,1201	0,4266	14
src/main/org/apache/tools/ant/taskdefs/optional/depend/constant pool	-0,3558	0,0685	13
src/main/org/apache/tools/ant/taskdefs/optional/dotnet	-0,4201	0,0070	13
src/main/org/apache/tools/ant/util	-0,0976	0,1254	34
src/main/org/apache/tools/ant/taskdefs/optional/clearcase	-0,0484	0,8349	14
src/main/org/apache/tools/ant/taskdefs/optional/sound	-0,1447	0,4900	13
src/main/org/apache/tools/ant/taskdefs/optional/metamata	-0,3577	0,1327	6
src/main/org/apache/tools/ant/taskdefs/compiler	-0,2440	0,0227	26
src/main/org/apache/tools/ant/taskdefs/optional/junit	-0,1930	0,0061	34
src/main/org/apache/tools/zip	-0,1308	0,2991	13
src/main/org/apache/tools/ant/taskdefs/optional/pvcs	0,0231	0,8984	16
src/main/org/apache/tools/ant/taskdefs/rmic	-0,1527	0,2849	13
src/main/org/apache/tools/ant/taskdefs/optional/ccm	-0,0386	0,8648	14
src/main/org/apache/tools/ant/taskdefs/condition	-0,1751	0,0896	18
src/main/org/apache/tools/ant/taskdefs/optional/jsp/compile rs	-0,3864	0,0138	14
src/main/org/apache/tools/ant/taskdefs/optional/jsp	-0,1369	0,3934	13
src/main/org/apache/tools/ant/util/regexp	-0,2343	0,1894	13
src/main/org/apache/tools/ant/taskdefs/optional/i18n	-0,4613	0,0053	14
src/main/org/apache/tools/bzip2	-0,4342	0,0340	9
src/main/org/apache/tools/ant/listener	-0,0353	0,8035	17
src/main/org/apache/tools/ant/taskdefs/optional/sos	0,2041	0,4019	11
src/main/org/apache/tools/ant/taskdefs/email	-0,1078	0,4379	19
src/main/org/apache/tools/ant/taskdefs/optional/net	-0,0220	0,8415	20
src/main/org/apache/tools/ant/taskdefs/optional/sitraka/bytecode	-0,1503	0,6590	5

Componente	ρ_2	Valor-p	Número de desenvolvedores
src/main/org/apache/tools/ant/filters	-0,1810	0,1397	16
src/main/org/apache/tools/ant/filters/util	0,0149	0,9474	8
src/main/org/apache/tools/ant/helper	-0,0993	0,4206	16
src/main/org/apache/tools/ant/taskdefs/optional/splash	-0,3296	0,1445	11
src/main/org/apache/tools/ant/util/depend	-0,8000	0,2000	3
src/main/org/apache/tools/ant/taskdefs/optional/j2ee	-0,2301	0,2908	11
src/main/org/apache/tools/ant/taskdefs/cvslib	-0,0972	0,4931	17
src/main/org/apache/tools/ant/types/optional/depend	-0,0652	0,7973	7
src/main/org/apache/tools/ant/taskdefs/optional/extension	-0,3421	0,0596	10
src/main/org/apache/tools/ant/util/facade	-0,3834	0,3084	6
src/main/org/apache/tools/ant/taskdefs/optional/extension/resolvers	-0,0855	0,7813	11
src/main/org/apache/tools/ant/input	-0,1932	0,3344	11
src/main/org/apache/tools/ant/util/optional	-0,2545	0,3600	8
src/main/org/apache/tools/ant/taskdefs/optional/unix	-0,1588	0,3479	12
src/main/org/apache/tools/ant/taskdefs/optional/windows	-0,4554	0,3641	4
src/main/org/apache/tools/ant/loader	0,0251	0,9352	6
src/main/org/apache/tools/ant/types/selectors	-0,2234	0,0737	17
src/main/org/apache/tools/ant/launch	-0,3006	0,0358	16
src/main/org/apache/tools/ant/types/selectors/modifiedselector	-0,1547	0,3750	9
src/main/org/apache/tools/ant/taskdefs/optional/sitraka/bytecode/attributes	-0,5000	0,6667	2
src/main/org/apache/tools/ant/dispatch	-0,6935	0,0262	7
src/main/org/apache/tools/ant/taskdefs/optional/condition	-0,8660	0,3333	2
src/main/org/apache/tools/ant/taskdefs/repository	-0,6236	0,0303	5
src/main/org/apache/tools/ant/types/mappers	0,0000	1,0000	5
src/main/org/apache/tools/ant/taskdefs/optional/native2ascii	-0,3768	0,2533	5
src/main/org/apache/tools/ant/taskdefs/optional/javah	-0,3246	0,2378	6
src/main/org/apache/tools/ant/types/resources	-0,0024	0,9860	8
src/main/org/apache/tools/ant/types/resources/comparators	-0,1245	0,5907	9
src/main/org/apache/tools/ant/types/resources/selectors	-0,3069	0,2013	8
src/main/org/apache/tools/ant/util/java15	0,7746	0,2254	4
src/main/org/apache/tools/ant/types/spi	0,2052	0,7406	4

Componente	ρ_2	Valor-p	Número de desenvolvedores
src/main/org/apache/tools/ant/taskdefs/optional/script	-0,1741	0,6542	6
src/main/org/apache/tools/ant/types/optional	0,4238	0,2557	5
src/main/org/apache/tools/ant/property	-0,0556	0,8502	6
src/main/org/apache/tools/ant/attribute	-0,1118	0,8579	3
src/main/org/apache/tools/ant/taskdefs/optional/image	-0,5000	0,6667	2
src/main/org/apache/tools/ant/taskdefs/optional/junitlauncher/co nfined	0,8660	0,3333	3

B.1.2. Cassandra

O resultado no Cassandra foi o mais expressivo dos quatro projetos analisados, pois em uma parcela considerável de componentes (41,24%) houve correlação negativa entre experiência e degradação da arquitetura. São 33 correlações fracas, 6 moderadas e apenas 1 forte.

Tabela B-2 - Dados relativos ao teste de correlação em QP2 para o Cassandra.

Componente	ρ_2	Valor-p	Número de desenvolvedores
src/java/org/apache/cassandra/db/filter	-0,3489	0,0002	26
src/java/org/apache/cassandra/tools	-0,1960	0,0001	71
src/java/org/apache/cassandra/service	-0,2016	0,0000	83
src/java/org/apache/cassandra/concurrent	-0,2280	0,0911	19
src/java/org/apache/cassandra/auth	-0,3614	0,0012	21
src/java/org/apache/cassandra/io/util	-0,2442	0,0011	33
src/java/org/apache/cassandra/db/commitlog	-0,3179	0,0000	31
src/java/org/apache/cassandra/gms	-0,1732	0,0261	25
src/java/org/apache/cassandra/config	-0,2727	0,0000	55
src/java/org/apache/cassandra/db	-0,1888	0,0000	74
src/java/org/apache/cassandra/hadoop	-0,3039	0,0005	24
src/java/org/apache/cassandra/cli	-0,1725	0,0677	16
src/java/org/apache/cassandra/client	0,1771	0,4964	10
src/java/org/apache/cassandra/io/sstable	-0,2085	0,0002	38
src/java/org/apache/cassandra/net	-0,3159	0,0000	48
src/java/org/apache/cassandra/utils	-0,1813	0,0029	42
src/java/org/apache/cassandra/avro	-0,1893	0,3649	4
src/java/org/apache/cassandra/db/migration	0,0829	0,6258	4

Componente	ρ_2	Valor-p	Número de desenvolvedores
src/java/org/apache/cassandra/locator	-0,2062	0,0091	33
src/java/org/apache/cassandra/dht	-0,3189	0,0002	33
src/java/org/apache/cassandra/streaming	-0,0810	0,3072	30
src/java/org/apache/cassandra/db/marshal	-0,2325	0,0061	24
src/java/org/apache/cassandra/thrift	-0,1531	0,0273	35
src/java/org/apache/cassandra/cache	-0,3219	0,0021	22
src/java/org/apache/cassandra/scheduler	0,2110	0,5334	5
src/java/org/apache/cassandra/db/columniterator	-0,4252	0,0004	17
src/java/org/apache/cassandra/cql	-0,1851	0,1046	12
src/java/org/apache/cassandra/security	0,0494	0,8507	13
src/java/org/apache/cassandra/net/sink	0,5000	0,6667	1
src/java/org/apache/cassandra/db/compaction	-0,1872	0,0002	57
src/java/org/apache/cassandra/db/context	-0,1758	0,3805	7
src/java/org/apache/cassandra/io/compress	-0,2889	0,0077	23
src/java/org/apache/cassandra/db/index/keys	-0,1978	0,2622	11
src/java/org/apache/cassandra/db/index	-0,5127	0,0000	27
src/java/org/apache/cassandra/utils/obs	-0,7483	0,0005	9
src/java/org/apache/cassandra/cql3/statements	-0,2084	0,0002	41
src/java/org/apache/cassandra/cql3	-0,3577	0,0000	35
src/java/org/apache/cassandra/utils/vint	-0,2041	0,6606	7
src/java/org/apache/cassandra/transport	-0,4457	0,0000	35
src/java/org/apache/cassandra/transport/messages	-0,3451	0,0056	21
src/java/org/apache/cassandra/hadoop/pig	-0,4220	0,0008	15
src/java/org/apache/cassandra/cql3/operations	-0,7500	0,0522	1
src/java/org/apache/cassandra/tracing	-0,2834	0,0563	16
src/java/org/apache/cassandra/io	-0,5822	0,0289	7
src/java/org/apache/cassandra/streaming/compress	-0,0524	0,7418	20
src/java/org/apache/cassandra/db/index/composites	-0,1358	0,3974	12
src/java/org/apache/cassandra/metrics	-0,0881	0,3983	41
src/java/org/apache/cassandra/exceptions	0,1008	0,6812	15
src/java/org/apache/cassandra/cli/transport	0,0000	1,0000	3
src/java/org/apache/cassandra/cql3/functions	-0,1868	0,1332	15
src/java/org/apache/cassandra/service/paxos	-0,4479	0,0131	13
src/java/org/apache/cassandra/triggers	0,0101	0,9663	8
src/java/org/apache/cassandra/hadoop/cql3	-0,2155	0,0872	17
src/java/org/apache/cassandra/streaming/messages	-0,1486	0,4091	17
src/java/org/apache/cassandra/notifications	-0,1260	0,7663	6
src/java/org/apache/cassandra/service/pager	-0,4419	0,0049	15
src/java/org/apache/cassandra/repair	-0,3362	0,0024	29
src/java/org/apache/cassandra/serializers	-0,3243	0,0574	13
src/java/org/apache/cassandra/db/composites	-0,1120	0,6288	12

Componente	ρ_2	Valor-p	Número de desenvolvedores
src/java/org/apache/cassandra/io/sstable/metadata	-0,0666	0,7629	16
src/java/org/apache/cassandra/utils/btree	0,1746	0,5338	9
src/java/org/apache/cassandra/utils/concurrent	-0,4786	0,0382	9
src/java/org/apache/cassandra/utils/memory	-0,1049	0,6599	11
src/java/org/apache/cassandra/repair/messages	-0,0725	0,7364	13
src/java/org/apache/cassandra/cql3/selection	-0,1152	0,5753	13
src/java/org/apache/cassandra/io/sstable/format	-0,0151	0,9172	26
src/java/org/apache/cassandra/io/sstable/format/big	-0,3193	0,0325	24
src/java/org/apache/cassandra/schema	-0,4333	0,0017	20
src/java/org/apache/cassandra/cql3/restrictions	-0,3338	0,0500	15
src/java/org/apache/cassandra/db/compaction/writers	-0,5551	0,0168	11
src/java/org/apache/cassandra/tools/nodetool	-0,2021	0,1353	37
src/java/org/apache/cassandra/db/lifecycle	-0,4995	0,0249	11
src/java/org/apache/cassandra/utils/progress/jmx	-0,2582	0,7418	4
src/java/org/apache/cassandra/index/sasi/conf/view	-0,8804	0,0206	4
src/java/org/apache/cassandra/index/sasi/analyzer	-0,6669	0,2189	4
src/java/org/apache/cassandra/db/partitions	-0,3942	0,0512	14
src/java/org/apache/cassandra/index	-0,1135	0,6871	9
src/java/org/apache/cassandra/index/internal	0,0090	0,9768	10
src/java/org/apache/cassandra/db/rows	-0,2326	0,1434	16
src/java/org/apache/cassandra/db/transform	-0,5270	0,1796	5
src/java/org/apache/cassandra/batchlog	-0,2739	0,4758	8
src/java/org/apache/cassandra/hints	-0,1313	0,5055	18
src/java/org/apache/cassandra/dht/tokenallocator	-0,3214	0,4821	4
src/java/org/apache/cassandra/repair/consistent	-0,6325	0,3675	4
src/java/org/apache/cassandra/index/sasi/plan	-0,5404	0,0307	10
src/java/org/apache/cassandra/index/internal/composites	0,0730	0,8519	7
src/java/org/apache/cassandra/db/view	-0,3485	0,1564	12
src/java/org/apache/cassandra/index/internal/keys	-0,1118	0,8579	4
src/java/org/apache/cassandra/index/sasi	-0,5037	0,0793	8
src/java/org/apache/cassandra/index/sasi/conf	-0,5192	0,0571	6
src/java/org/apache/cassandra/index/sasi/disk	-0,6152	0,0252	9
src/java/org/apache/cassandra/index/sasi/memory	-0,6547	0,0781	6
src/java/org/apache/cassandra/index/sasi/utils	0,1118	0,8579	4
src/java/org/apache/cassandra/index/sasi/sa	-0,7746	0,2254	3
src/java/org/apache/cassandra/auth/jmx	0,8660	0,3333	3
src/java/org/apache/cassandra/index/sasi/analyzer/filter	-0,8660	0,3333	2
src/java/org/apache/cassandra/tools/nodetool/stats	0,2750	0,4739	7

B.1.3. Hadoop

No Hadoop também houve poucos componentes (24,14%) com correlação entre experiência e degradação da arquitetura. Todas as correlações foram negativas, sendo 2 fracas, 4 moderadas e 1 forte.

Tabela B-3 - Dados relativos ao teste de correlação em QP2 para o Hadoop.

Componente	ρ_2	Valor-p	Número de desenvolvedores
src/java/org/apache/hadoop/fs	-0,7071	0,1817	4
hadoop-common-project/hadoop-common/src/main/java/org/apache/hadoop/io/retry	0,7071	0,1817	5
hadoop-common-project/hadoop-common/src/main/java/org/apache/hadoop/ipc	-0,4126	0,0211	22
hadoop-common-project/hadoop-common/src/main/java/org/apache/hadoop/fs	-0,4098	0,0008	34
hadoop-common-project/hadoop-common/src/main/java/org/apache/hadoop/ha	-0,5186	0,0396	9
hadoop-common-project/hadoop-common/src/main/java/org/apache/hadoop/net	0,3947	0,4387	5
hadoop-common-project/hadoop-common/src/main/java/org/apache/hadoop/security/token/delegation	-0,6547	0,1583	5
hadoop-common-project/hadoop-common/src/main/java/org/apache/hadoop/fs/shell	-0,0230	0,9211	14
hadoop-common-project/hadoop-common/src/main/java/org/apache/hadoop/util	-0,4636	0,0527	10
hadoop-common-project/hadoop-common/src/main/java/org/apache/hadoop/security	-0,2601	0,0921	27
hadoop-common-project/hadoop-common/src/main/java/org/apache/hadoop/security/token	0,3536	0,5594	5
hadoop-common-project/hadoop-common/src/main/java/org/apache/hadoop/http	-0,7326	0,0387	6
hadoop-common-project/hadoop-common/src/main/java/org/apache/hadoop/fs/permission	-0,7826	0,1176	3
hadoop-common-project/hadoop-common/src/main/java/org/apache/hadoop/net/unix	0,3354	0,5811	4
hadoop-common-project/hadoop-common/src/main/java/org/apache/hadoop/security/authorize	0,3536	0,5594	5
hadoop-common-project/hadoop-common/src/main/java/org/apache/hadoop/crypto/key	-0,6860	0,0198	7
hadoop-common-project/hadoop-common/src/main/java/org/apache/hadoop/crypto	-0,3834	0,3084	6
hadoop-common-project/hadoop-common/src/main/java/org/apache/hadoop/security/token/delegati on/web	-0,7120	0,0314	6

Componente	ρ_2	Valor-p	Número de desenvolvedores
hadoop-common-project/hadoop-common/src/main/java/org/apache/hadoop/crypto/key/kms	-0,8300	0,0002	9
hadoop-common-project/hadoop-common/src/main/java/org/apache/hadoop/conf	-0,2907	0,2748	10
hadoop-common-project/hadoop-common/src/main/java/org/apache/hadoop/io	-0,2312	0,4939	8
hadoop-common-project/hadoop-common/src/main/java/org/apache/hadoop/io/nativeio	-0,2582	0,7418	4
hadoop-common-project/hadoop-common/src/main/java/org/apache/hadoop/tracing	0,0000	1,0000	2
hadoop-common-project/hadoop-common/src/main/java/org/apache/hadoop/metrics/ganglia	-0,8660	0,3333	2
hadoop-common-project/hadoop-common/src/main/java/org/apache/hadoop/metrics2/lib	0,8660	0,3333	3
hadoop-common-project/hadoop-common/src/main/java/org/apache/hadoop/metrics2/sink	0,0000	1,0000	3
hadoop-common-project/hadoop-common/src/main/java/org/apache/hadoop/security/alias	0,3536	0,5594	4
hadoop-common-project/hadoop-common/src/main/java/org/apache/hadoop/security/http	-0,8660	0,3333	3
hadoop-common-project/hadoop-common/src/main/java/org/apache/hadoop/io/erasurecode	0,0000	1,0000	3

B.1.4. Lucene

No Lucene também houve poucos componentes (26,67%) em que se encontrou correlação entre experiência e degradação da arquitetura, sendo 3 fracas e 1 moderada.

Tabela B-4 - Dados relativos ao teste de correlação em QP2 para o Lucene.

Componente	ρ_2	Valor-p	Número de desenvolvedores
src/java/org/apache/lucene/analysis/de	-0,2556	0,3221	5
src/java/org/apache/lucene/analysis	-0,0867	0,4474	14
src/java/org/apache/lucene/index	-0,2352	0,0755	12
src/java/org/apache/lucene/util	-0,1708	0,1738	15
src/java/org/apache/lucene/document	-0,1790	0,1075	15
src/java/org/apache/lucene/search	-0,1943	0,0056	20
src/java/org/apache/lucene/analysis/standard	-0,4754	0,0013	13
src/java/org/apache/lucene/queryParser	0,0514	0,6913	14
src/java/org/apache/lucene/store	-0,2871	0,0103	16

src/java/org/apache/lucene/analysis/ru	-0,0732	0,8633	4
src/java/org/apache/lucene/search/spans	-0,1559	0,3010	12
src/java/org/apache/lucene/search/payloads	-0,6147	0,0442	4
src/java/org/apache/lucene/search/function	-0,3426	0,1392	7
src/java/org/apache/lucene/analysis/tokenattributes	-0,3135	0,3210	6
src/java/org/apache/lucene/messages	-0,3162	0,6838	2

B.2. Dados da análise adicional

A Tabela B-5 apresenta os dados referentes à hipótese adicional para os quatro projetos, ou seja, o coeficiente da correlação de *Spearman* ρ e o número de *commits* impactantes para cada autor que tenha dado pelo menos um *commit* impactante no projeto. Por esses dados, é possível notar muitas correlações positivas, o que vai contra a hipótese de QPA. Além disso e do baixo número de correlações estatisticamente significante em todos os projetos, algumas delas ocorreram para desenvolvedores que fizeram poucos *commits* impactantes. Esses dados evidenciam a falta de uma relação entre experiência geral e degradação da arquitetura.

Tabela B-5 - Coeficiente ρ_A por desenvolvedor nos quatro projetos.

#	Ant		Cassandra		Hadoop		Lucene	
	ρ_A	Commits impactantes	ρ_A	Commits impactantes	ρ_A	Commits impactantes	ρ_A	Commits impactantes
1	0,1786	7	-0,0765*	690	-0,0569	28	-0,0501	33
2	-0,0167	9	-0,0503	37	0,2000	4	-1,0000*	3
3	-0,4551	18	0,0443	62	0,2523	7	0,5000	3
4	-0,8000	4	0,1553*	189	0,3857	15	0,1166	18
5	-0,0609	489	0,2182	29	0,1429	7	0,4765	16
6	0,5000	3	-0,1021	351	0,0912	16	0,3714	6
7	-0,1165	76	-0,2000	4	-0,3000	5	0,1727	11
8	-0,0802	113	-0,0293	56	0,5000	5	0,3179	22
9	-0,4000	4	-0,1579	39	-0,2696	17	0,2000	4
10	1,0000*	5	0,1312	84	0,3738	16	-0,1444	36
11	-0,0456	38	-0,5000	3	-0,3000	5	0,2652	18
12	-1,0000*	3	0,1687*	141	-0,5000	3	0,6571	6
13	0,2471	16	-0,0020	129	0,3143	6	-0,0224	456

#	Ant		Cassandra		Hadoop		Lucene	
	ρ_A	Commits impactantes	ρ_A	Commits impactantes	ρ_A	Commits impactantes	ρ_A	Commits impactantes
14	0,1000	5	0,1748	90	0,0357	7	0,0794	31
15	-0,5000	3	0,0476	8	0,2485	10	-0,4667	10
16	1,0000*	3	0,0661	30	-0,0607	15	0,0364	11
17	-0,2000	4	-0,5000	3	0,1786	7	-0,0165	174
18	-0,0109	23	0,4000	4	0,2000	5	-0,0285	341
19	0,0725	14	-0,0539	38	-1,0000*	3	0,1434	48
20	0,4000	4	-0,0392	59	0,5000	3	-0,1726*	134
21	0,0999	30	-0,5000	3	1,0000*	3	-0,5000	3
22	0,5000	3	0,1635	24	0,5476	8	-1,0000*	3
23	0,0000	4	0,1537	51	-0,5714	7	0,1189	12
24	-0,0582	133	0,0766	21	0,1640	11	0,1667	9
25	-0,1785	25	-0,2974	23	0,4000	4	-0,6429	8
26	-0,1674	48	-0,6182*	29	-0,4000	4	1,0000*	3
27	-0,1483	132	-0,4333	9	0,5000	3	-0,3143	6
28	-0,3439	19	-0,3217	12	-		-0,1045	22
29	0,1765	16	-0,2339	64	-		0,1404*	231
30	-0,8660	3	-0,3846	13	-		0,6000	6
31	0,2464	15	-0,3731*	30	-		-0,0114	61
32	0,4000	4	-0,0787	50	-		0,2848	10
33	-0,1601	36	-0,2000	10	-		-0,3000	5
34	-	-	0,7000	5	-		0,0549	14
35	-	-	-0,1126	35	-		0,4945	14
36	-	-	-0,8000	5	-		-0,5714	8
37	-	-	0,1801	27	-		0,2733	11
38	-	-	0,0091	11	-		-0,5000	3
39	-	-	-0,5000	3	-		-0,1786	7
40	-	-	0,5000	3	-		-0,3714	6
41	-	-	0,0714	7	-		-1,0000*	3
42	-	-	0,0000	5	-		-0,7143	6
43	-	-	0,2059	35	-		0,5000	3
44	-	-	-0,2619	8	-		-1,0000*	3
45	-	-	-0,2000	5	-		1,0000*	3
46	-	-	1,0000*	3	-		-0,4424	10

#	Ant		Cassandra		Hadoop		Lucene	
	ρ_A	<i>Commits</i> impactantes	ρ_A	<i>Commits</i> impactantes	ρ_A	<i>Commits</i> impactantes	ρ_A	<i>Commits</i> impactantes
47	-	-	0,9000*	5	-		-0,5000	3
48	-	-	-0,5000	3	-		-	-
49	-	-	0,5000	3	-		-	-
50	-	-	1,0000*	3	-		-	-
51	-	-	-1,0000*	3	-		-	-