UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

BRENO FARIAS DA SILVA

ABORDAGEM PARA SELEÇÃO DE EXEMPLOS DE CÓDIGO DE SISTEMAS DISTRIBUÍDOS PARA A CRIAÇÃO DE EXEMPLOS TRABALHADOS DE ENGENHARIA DE SOFTWARE

CAMPO MOURÃO, PR, BRASIL

2023

BRENO FARIAS DA SILVA

ABORDAGEM PARA SELEÇÃO DE EXEMPLOS DE CÓDIGO DE SISTEMAS DISTRIBUÍDOS PARA A CRIAÇÃO DE EXEMPLOS TRABALHADOS DE ENGENHARIA DE SOFTWARE

Approach for selecting distributed systems code examples for creating software engineering worked examples

Trabalho de Conclusão de Curso de Graduação apresentado como requisito para obtenção do título de Bacharel em Ciência da Computação do Curso de Bacharelado em Ciência da Computação da Universidade Tecnológica Federal do Paraná.

Orientador: Prof. Dr. Marco Aurélio Graciotto Silva

Coorientador: Prof. Dr. Rodrigo Campiolo

CAMPO MOURÃO, PR, BRASIL

2023

AGRADECIMENTOS

Certamente não haveria forma diferente de começar os meus agradecimentos sem expressar minha profunda gratidão ao meu orientador, Prof. Dr. Marco Aurélio Graciotto Silva, e ao co-orientador, Prof. Dr. Rodrigo Campiolo, pelas imensuráveis orientações e feedback constante ao longo deste projeto de pesquisa. A dedicação e apoio de ambos foram fundamentais para o sucesso deste trabalho e por estarem presentes em minha trajetória durante a graduação.

Além disso, quero agradecer ao Prof. Dr. Luiz Arthur, que não esteve diretamente envolvido no projeto, mas que desempenhou um papel crucial como professor, mas também como amigo. Sua amizade e apoio constante tornaram minha jornada de aprendizado uma experiência enriquecedora. Suas conversas e mentorias moldaram minha perspectiva e contribuíram extraordinariamente para o meu crescimento como estudante e como pessoa.

Ao Prof. Dr. Igor Wiese, agradeço seu apoio e sua disposição para compartilhar conhecimento e incentivar minha participação em atividades extracurriculares, abrindo portas que foram fundamentais para o meu desenvolvimento como aluno nessa instituição.

Gostaria também de agradecer a minha namorada, Amanda Carvalho, visto que, todos os dias, ela esteve ao meu lado, oferecendo apoio inabalável, especialmente nos momentos em que duvidava das minhas capacidades, deixando-se sempre disponível para dividir as minhas dores e anseios com ela.

Gostaria de expressar minha profunda gratidão à minha mãe, Márcia Farias, pois, mesmo mantendo pouco contato, sempre esteve preocupada em saber se estou bem e feliz. Agradeço por ser uma fonte inesgotável de amor, por sempre me incentivar a alcançar os meus objetivos que, independente do quão grande possam chegar a ser, sempre me apoiou e nunca duvidou de mim.

Gostaria de enviar um abraço especial aos meus amigos incríveis de Portugal, Bernardo Louro, Diogo Lopes e Simão Farias. Mesmo estando a mais de 7 mil km de distância, vocês fizeram parte de cada passo dessa jornada maluca. Vocês são mais do que amigos; são a prova de que as verdadeiras conexões resistem a qualquer distância. Obrigado por serem parte da minha vida de uma maneira tão única e significativa!

Por último, mas definitivamente não menos importante, dedico um agradecimento especial ao meu pai, Manoel Campos, minha maior inspiração de valor incomensurável. Seu apoio constante e palavras de incentivo não apenas foram um fator motivador crucial, mas também por sempre demonstrar orgulho e apontar que estou no caminho certo. Agradeço sinceramente por ter um pai cuja presença é insubstituível, tornando minha jornada ainda mais significativa.

A todos os que, de alguma forma, contribuíram para a realização desta pesquisa, meu sincero agradecimento.

RESUMO

Esta monografia apresenta uma proposta de pesquisa para o Trabalho de Conclusão de Curso 2 (TCC2), abordando a evolução do código em Sistemas Distribuídos (SDs) por meio da análise de métricas de código. O objetivo principal é desenvolver exemplos trabalhados que sejam aplicáveis ao ensino de Engenharia de Software (ES). A complexidade intrínseca dos SDs, que desempenham papel crucial na infraestrutura computacional contemporânea, ressalta a importância de compreender a evolução do código para formar profissionais qualificados. O referencial teórico deste trabalho fundamenta-se na compreensão da complexidade dos SDs na era digital, justificando a necessidade de estudar a evolução do código para formar profissionais qualificados em ES. Destaca-se a relevância da seleção de métricas de código já selecionadas, as quais fornecem uma visão abrangente da qualidade do código, abordando diferentes aspectos, além de explorar o conceito do que é um exemplo trabalhado e suas aplicações na ES e em SDs. A pesquisa adotou uma abordagem exploratória e qualitativa para investigar a relação entre métricas de código e melhorias em SDs. A heurística a ser desenvolvida no TCC2 para a seleção de códigos representativos é fundamental na criação de exemplos trabalhados que ilustrem, de maneira prática, a dinâmica da evolução do código em SDs. A escolha criteriosa de ferramentas de refatoração e análise de código permitirá uma análise abrangente das métricas de código e das mudanças ao longo do tempo em repositórios de código-fonte de SDs relevantes. A metodologia adotada possibilita uma compreensão holística da relação entre métricas específicas e melhorias no código, incorporando elementos qualitativos na análise. Os resultados desta pesquisa buscam contribuir para a formação de estudantes e profissionais em ES, preenchendo uma lacuna na literatura, pois não existem projetos diretamente comparáveis para servir de referência, sobre a criação de exemplos trabalhados para a ES, utilizando SDs como objeto de estudo. Apesar de resultados prévios não tão animadores, representando desafios esperados, esses desafios oferecem oportunidades para a continuidade da pesquisa. Não suficiente, o cronograma exposto para o TCC2 incorpora elementos já realizados ainda nesta etapa do projeto. Neste estágio inicial da pesquisa, representado apenas por uma proposta, os desafios e limitações intrínsecos ao projeto estão claramente delineados, além de que espera-se impactar positivamente o ensino e a prática profissional na área de ES.

Palavras-chave: métricas ck; evolução de código; exemplo trabalhado; engenharia de software; sistemas distribuídos.

ABSTRACT

This monograph presents a research proposal for TCC2, addressing the evolution of code in SDs through code metrics analysis. The main objective is to develop worked examples applicable to ES education. The intrinsic complexity of SDs, playing a crucial role in contemporary computational infrastructure, emphasizes the importance of understanding code evolution to educate qualified professionals. The theoretical framework is based on understanding the complexity of SDs in the digital era, justifying the need to study code evolution to train qualified professionals in ES. The relevance of the selected code metrics is highlighted, providing a comprehensive view of code quality and addressing different aspects. It also explores the concept of what a worked example is and its applications in ES and SDs. The research adopts an exploratory and qualitative approach to investigate the relationship between code metrics and improvements in SDs. The heuristic to be developed in TCC2 for the selection of representative codes is crucial for creating worked examples that illustrate the dynamics of code evolution in SDs practically. The careful choice of refactoring and code analysis tools will allow a comprehensive analysis of code metrics and changes over time in repositories of relevant SDs source code. The adopted methodology enables a holistic understanding of the relationship between specific metrics and code improvements, incorporating qualitative elements into the analysis. The results of this research aim to contribute to the education of students and professionals in ES, filling a gap in the literature, as there are no directly comparable projects to serve as a reference for creating worked examples for ES using SDs as the object of study. Despite not-so-promising preliminary results, representing expected challenges, these challenges provide opportunities for ongoing research. Moreover, the schedule outlined for TCC2 incorporates elements already completed at this stage of the project. In this early stage of research, represented only by a proposal, the intrinsic challenges and limitations of the project are clearly outlined, with the expectation of positively impacting ES education and professional practice.

Keywords: ck metrics; code evolution; worked examples; software engineering; distributed systems.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

Siglas

ABE	Aprendizagem Baseada em Exemplos
CAP	*Consistency, Availability and Partition Tolerance*
CBO	*Coupling Between Object*
CBOM	*Coupling Between Object Modified*
CCC	*Class Central Coupling*
CCL	*Class Comunication Load*
CK	*Code Metrics*
DIT	*Depth of Inheritance Tree*
ES	Engenharia de Software
IA	Inteligência Artificial
IPC	*Inter-Process Comunication*
IPR	*Interprocess Reuse*
LCOM	*Lack of Cohesion in Methods*
LOC	*Lines of Code*
NOC	*Number of Children*
PLC	*Process-Level Cohesion*
RCC	*Run-time Class Coupling*
RFC	*Response for a Class*
RMC	*Run-time Message Coupling*
SD	Sistema Distribuído
SDs	Sistemas Distribuídos
TCC2	Trabalho de Conclusão de Curso 2

WMC	*Weight Method Class*
ZAB	*Zookeeper Atomic Broadcast Protocol*
ZK	ZooKeeper

SUMÁRIO

1 INTRODUÇÃO

No cenário atual da computação, os Sistemas Distribuídos (SDs) desempenham um papel fundamental, alimentando a infraestrutura de serviços e aplicações que impulsionam nosso mundo digital. De acordo com Steen e Tanenbaum (2016, p. 968), um Sistema Distribuído (SD) é definido como “Um conjunto de computadores independentes que se apresenta a seus usuários como um sistema único e coerente.” Diante disso, emergem desafios inerentes devido à necessidade de apresentar o sistema ao usuário como uma entidade homogênea, embora, intrinsecamente, o sistema seja constituído por diversas partes heterogêneas.

A complexidade inerente aos SDs os torna um campo de pesquisa desafiador, mas, ao mesmo tempo, altamente relevante na computação moderna. Compreender e lidar com essa heterogeneidade e complexidade é essencial para o desenvolvimento, manutenção e escalabilidade de SDs. De igual modo, a crescente dependência de empresas e instituições por esses sistemas torna o estudo de SDs indispensável para a formação de profissionais capacitados e para o avanço da qualidade do ensino em Engenharia de Software (ES).

Nos SDs, as preocupações com a segurança dos dados e da comunicação, o desempenho eficiente e a capacidade de manter a operação contínua, mesmo diante de falhas, são instigações extremamente pertinentes. A necessidade de proteger informações sensíveis, assegurar rápido tempo de resposta, garantir que a troca de mensagens seja eficiente e manter a disponibilidade de serviços torna o estudo da evolução do código em SDs ainda mais vital. Essas complexidades são estudadas em cursos universitários e estão intrinsecamente ligadas ao código que impulsiona esses sistemas, tornando a investigação sobre a evolução do código em SDs uma fonte valiosa sobre como esses sistemas evoluem para atender às demandas constantes.

Os currículos de Ciência da Computação ACM CS2023 Raj e Kumar (2022, p. 3) abordam computação distribuída e paralela como um tópico principal. Um estudo identificou que tópicos como Processos e Threads, Replicação, Chamadas de Sistema, Controle de Concorrência, Tolerância a Falhas, Sincronização, Comunicação, entre outros constituem os elementos-chave em cursos de SDs(ABAD; ORTIZ-HOLGUIN; BOZA, 2021). Contudo, a ausência de pesquisas recentes sobre a elaboração de exemplos trabalhados utilizando SDs como objeto de estudo destaca a importância de investigar a evolução do código nesse contexto. Enfatiza-se, assim, a relevância dessa investigação, especialmente no que se refere ao aprimoramento da educação em ES. Isto é evidenciado pela relevância crítica desses sistemas, explicada pelo avanço de tecnologias de software e hardware, pelo aumento do acesso à internet e número de usuários.

A eficácia do ensino em Ciência da Computação, notadamente no âmbito da ES, seria consideravelmente aprimorada pelo desenvolvimento e aplicação de exemplos trabalhados (do inglês, *worked examples*). Entretanto, é relevante observar que exemplos trabalhados não são tão proeminentemente empregados no contexto da ES. Um exemplo trabalhado é uma ferramenta pedagógica poderosa, definida como um trabalho cognitivo e experimental que oferece

uma solução ideal e praticável para um problema específico, permitindo que aprendizes examinem e aprendam com a solução proposta(ATKINSON *et al.*, 2000). A literatura destaca a importância dos exemplos trabalhados na disseminação de conceitos e padrões, fornecendo uma solução representativa do estado da arte para um tópico específico. A falta de pesquisa dedicada aos exemplos trabalhados nesta área é evidente, e muitos professores enfrentam obstáculos ao incorporar exemplos reais em suas práticas pedagógicas(TONHÃO; COLANZI; STEINMACHER, 2021).

No âmbito da ES, os exemplos trabalhados frequentemente assumem a forma de apresentação passo-a-passo da execução de códigos específicos. No entanto, pesquisas indicam a escassez de estudos específicos sobre exemplos trabalhados na Ciência da Computação, uma vez que tarefas de programação exigem alto teor cognitivo(SKUDDER; LUXTON-REILLY, 2014). Além disso, estratégias como "*Faded Worked Examples*", que envolvem a apresentação gradual de exemplos resolvidos, têm mostrado impacto significativo na aprendizagem, promovendo a abstração do aluno e desenvolvendo a capacidade de recuperar informações(SKUDDER; LUXTON-REILLY, 2014).

1.1 Objetivos

1.1.1 Objetivo geral

Neste contexto, o objetivo geral deste projeto de pesquisa é desenvolver uma heurística para identificar exemplos de código-fonte que representem aprimoramentos no *software* de um SD, por meio da análise de métricas de código utilizadas na ES, com o propósito de criar um exemplo trabalhado para ser empregado em sala de aula. Essa heurística será projetada para auxiliar na seleção de exemplos que ilustrem como o código em SDs se adapta e evolui ao longo do tempo. Em decorrência disso, uma vez conquistada a heurística almejada, objetiva-se criar um exemplo trabalhado, de modo a compreender por que um código específico atende às métricas de código desejadas, focando especialmente na melhoria do desenho e qualidade do código, observadas através das métricas, no âmbito do desenvolvimento de software.

1.1.2 Objetivos específicos

- Desenvolver uma heurística de seleção que possa analisar as métricas de código.
- Identificar exemplos de código-fonte que representem eficazmente a evolução em SDs ao longo do tempo.

- Fornecer um exemplo trabalhado para a ES, identificando em quais aspectos um determinado código evoluiu, reconhecendo padrões, desafios comuns e práticas recomendadas.

1.2 Justificativa

Os avanços da digitalização de serviços e produtos tornou a sociedade dependente de SDs, acessados por usuários a partir de múltiplos dispositivos como computadores pessoais, smartphones, TVs e muito mais. A dependência abrange desde redes sociais até sistemas de transporte, saúde, serviços governamentais e muito mais. Portanto, para conquistar uma boa formação de estudantes e profissionais na área da Computação, faz-se necessário o entendimento de ES e SDs, tornando-se uma necessidade iminente para atender à demanda crescente por profissionais qualificados nesse campo.

Dado o contexto anterior, a investigação sobre a evolução do código em SDs assume destaque. O estudo visa proporcionar uma compreensão mais profunda de como o código-fonte em SDs se adapta ao longo do tempo para atender a demandas como o aumento do número de usuários em escala não esperada, descobertas de vulnerabilidades e novas formas de ataque aos sistemas, além do uso de novos protocolos de comunicação para tentar resolver desafios enfrentados por estes sistemas.

Os objetivos traçados neste trabalho se alinham com a necessidade de abordar essas complexidades. O desenvolvimento de uma heurística para identificar exemplos de código-fonte representativos da evolução em SDs contribuirá para uma melhor formação de estudantes e profissionais tanto na ES quanto em SDs, levando-se em consideração as inúmeras características de SDs já mencionadas anteriormente. Esses exemplos servirão como valiosos recursos de ensino e estudo, facilitando a compreensão das complexidades e desafios dos SDs em um ambiente distribuído e interconectado.

Portanto, a justificativa para este estudo se baseia na aplicação das práticas e técnicas de ES aplicadas a evolução do código em SDs, contribuindo para a formação de profissionais mais capacitados e avançando a qualidade do ensino em ES. Este estudo procura preencher uma lacuna na literatura sobre a criação de exemplos trabalhados em SDs para ser objeto de estudo na ES, fornecendo percepções valiosas para o desenvolvimento e manutenção de SDs e aprimorando a prática profissional neste campo da computação.

1.3 Estrutura do trabalho

A estrutura desta monografia segue uma abordagem organizada, dividida em cinco capítulos para uma compreensão abrangente do trabalho. O Capítulo 2 abrange a fundamentação teórica e revisão de trabalhos relacionados, proporcionando a base conceitual necessária para

a compreensão do trabalho. O Capítulo 3 detalha a abordagem proposta, incluindo questões da pesquisa, ferramentas, métodos e um cronograma para orientar a condução da pesquisa. O Capítulo 4 apresenta os resultados preliminares, a partir da análise das métricas obtidas dos projetos selecionados. Finalmente, o Capítulo 5 oferece uma síntese dos principais resultados, mencionando as limitações do estudo.

2 REFERENCIAL TEÓRICO

Este capítulo oferece uma base conceitual para compreender os principais elementos de SDs e seus desafios. São explorados conceitos acerca de exemplos trabalhados, além de expor alguns estudos sobre sua utilização. Tais estudos englobam não apenas a educação em geral, mas também na área de ES. Dessa forma, este capítulo visa fornecer um contexto abrangente que sustentará a análise e discussão dos resultados obtidos ao longo deste trabalho.

2.1 Sistemas Distribuídos

Um SD é aquele no qual os componentes localizados em computadores interligados em rede se comunicam e coordenam suas ações apenas passando mensagens. A motivação para o desenvolvimento de SDs é impulsionada pela necessidade de melhorar a eficiência na utilização de recursos computacionais, possibilitando que a carga de trabalho seja dividia entre vários servidores, oferecendo suporte a aplicações multitarefa e fornecendo maior confiabilidade por meio da redundância de dados e serviços. A facilidade de acesso remoto a recursos também é uma motivação significativa, permitindo que usuários acessem informações e serviços independentemente da localização física dos recursos(STEEN; TANENBAUM, 2016).

Um estudo abrangente foi conduzido em 2019 e apresentado no artigo "*Have We Reached Consensus? An Analysis of Distributed Systems Syllabi*"(ABAD; ORTIZ-HOLGUIN; BOZA, 2021). Nele, a grade das 51 disciplinas mais famosas sobre SDs de programas de Ciência da Computação foram analisadas, conforme as melhores universidades pelo *Times Higher Education*, visando identificar os principais tópicos abordados. A análise desses dados revelou que conceitos fundamentais, como Processos e Threads, Replicação, Chamadas de Sistema, Controle de Concorrência, Tolerância a Falhas, Sincronização, Comunicação, entre outros, constituem os elementos-chave frequentemente lecionados em disciplinas de SDs. Além disso, a pesquisa abordou a questão das referências utilizadas na elaboração do conteúdo dos cursos, identificando que obras como "*Distributed Systems*" na edição de 2017, de Van Steen e Tanenbaum, e "D*istributed Systems: Concepts and Design*" na edição de 2011, de Coulouris, são fontes comuns amplamente utilizadas para explicar os princípios dos SDs(ABAD; ORTIZ-HOLGUIN; BOZA, 2021).

A implementação de SDs enfrenta diversos desafios, sendo um deles a heterogeneidade, que se manifesta em diferentes níveis, como heterogeneidade de hardware, rede, linguagem de programação e implementações realizadas por diferentes desenvolvedores. Essa diversidade pode exigir o uso de *middlewares* para fornecer uma camada de abstração e facilitar a integração. A ausência de um relógio global em SDs introduz desafios na coordenação de eventos e sincronização entre diferentes partes do sistema. A tolerância a falhas é abordada por meio de técnicas como *checksums* para detecção de erros, mascaramento de falhas

usando retransmissão de dados, além de contar com o uso de operações de *rollback* para manter a consistência, mesmo diante de falhas(COULOURIS *et al.*, 2011).

Para atingir escalabilidade, SDs empregam técnicas como replicação de dados, uso de cache distribuída, distribuição de carga entre múltiplos servidores e utilização de *web proxies*, entre outros. No entanto, a escalabilidade também traz consigo desafios, como a necessidade de manter a consistência entre as réplicas de dados. Adicionalmente, o Teorema *Consistency, Availability and Partition Tolerance* (CAP) estabelece que é impossível garantir fortemente, de forma simultânea, características de Consistência, Disponibilidade e Tolerância a partição em um SD. Assim, durante o desenvolvimento de um SD, é preciso fazer um balanço entre diferentes objetivos(BREWER, 2012).

A troca de mensagens *Inter-Process Comunication* (IPC) é uma questão pertinente em SDs, podendo ocorrer de forma síncrona ou assíncrona, além de ser classificada como bloqueante ou não bloqueante, dependendo do comportamento desejado. Do mesmo modo, a questão da coordenação e do acordo, na eleição de nó líder, por exemplo, também são fundamentais, com técnicas como eleição, protocolos de consenso e algoritmos de ordenação de eventos desempenham papéis cruciais na operação confiável de SDs. Em específico, a ordenação de eventos é tarefa particularmente complexa pela ausência de um relógio global, a qual na maioria das vezes é baseada em relações *happened-before*(COULOURIS *et al.*, 2011).

Em resumo, o desenvolvimento de SDs é desafiador, o que evidencia a importância do estudo desses sistemas. A complexidade desses desafios destaca a importância da ES na busca por soluções melhores. Ao analisar a evolução do código em SDs por meio de estudos de casos, a ES pode extrair *insights* valiosos para a aplicação prática de conceitos teóricos e estratégias de resolução de problemas. A utilização de SDs como cenário de estudo oferece uma base sólida para construir exemplos trabalhados, enriquecendo o ensino em ES. A interseção entre os desafios de SDs e os princípios da ES cria uma oportunidade única para desenvolver materiais de ensino que promovam uma compreensão aprofundada, preparando os estudantes para enfrentar as complexidades do desenvolvimento de software.

2.2 Exemplos trabalhados

Os estudos existentes, como "*Learning from Examples - Instructional Principles from the Worked Examples Research*"(ATKINSON *et al.*, 2000), "*Using Real Worked Examples to Aid Software Engineering Teaching*'(TONHÃO; COLANZI; STEINMACHER, 2021), "Uma Plataforma Gamificada de Desafios Baseados Em *Worked Examples* Extraídos de Projetos de Software Livre Para o Ensino de Engenharia de Software"(TONHÃO *et al.*, 2022) e "*Students' Perception of Example-Based Learning in Software Modeling Education*"(BONETTI *et al.*, 2023) comprovam que o uso de exemplos trabalhados, quando aplicados corretamente, potencializam o engajamento e retenção do conteúdo exposto. Entretanto, a maioria desses trabalhos se con-

centra principalmente na ES, o que pode ser problemático quando consideramos contextos que envolvem sistemas de informação ou SDs mais simples, como cliente-servidor.

Conforme exposto no artigo "*Learning from Examples - Instructional Principles from the Worked Examples Research*"(ATKINSON *et al.*, 2000), um exemplo trabalhado é definido como um trabalho cognitivo e experimental com o intuito de fornecer uma solução ideal, todavia próxima ao praticável, para um problema específico, no qual uma pessoa com escasso ou nenhum conhecimento acerca do tema possa examinar e aprender com a solução proposta. Sendo assim, o desenvolvimento e estudo de exemplos trabalhados enriquecem a qualidade do que é lecionado em sala de aula, uma vez que a solução ideal pode representar muito bem o estado da arte para um determinado tópico, visto que dissemina conceitos e padrões do problema apresentado(ATKINSON *et al.*, 2000).

No mesmo contexto, os principais elementos na elaboração de um exemplo trabalhado de alta qualidade estão intrinsecamente vinculados aos conceitos de *"inter-example feature"* e *"intra-example feature"*. A concepção do termo *"inter-example feature"* destaca a importância de fornecer uma diversidade de exemplos que ilustrem estratégias múltiplas para problemas similares, porém de tipos diferentes, enquanto o termo *"intra-example feature"* trata de características intrínsecas à estrutura interna de cada exemplo(ATKINSON *et al.*, 2000).

O artigo intitulado "*Worked Examples in Computer Science*"(SKUDDER; LUXTON-REILLY, 2014) destaca a escassez de pesquisas sobre exemplos trabalhados na área da Ciência da Computação. Além disso, o trabalho expõe que as tarefas de programação requerem um alto teor cognitivo, logo isso pode ser um indício para haver poucos trabalhos estudando exemplos trabalhados nesta área. Porém, o uso de exemplos trabalhados seria benéfico, visto que diminui a carga cognitiva para compressão do conteúdo(ATKINSON *et al.*, 2000). Adicionalmente, o autor investiga as ramificações de exemplos trabalhados na área da Computação, frequentemente manifestados na apresentação passo-a-passo da execução (do inglês "*code-tracing*"), de um código específico, prática recorrente em disciplinas como análise de algoritmos. Não suficiente, o estudo também explora o emprego do conceito de "*problem solution pair*", no qual um problema é apresentado, estimulando o aluno a tentar resolvê-lo, seguido pela exposição da solução correspondente. Por fim, de maneira menos convencional, são utilizadas técnicas de geração de código enquanto se leciona um conteúdo, o que contribui para esclarecer o raciocínio empregado na elaboração da solução para o problema.

Um estudo recente intitulado "*A Review of Worked Examples in Programming Activities*"(MULDNER; JENNINGS; CHIARELLI, 2022) examina exemplos trabalhados na disciplina de Ciência da Computação, enfocando especialmente em exemplos de rastreamento e geração de código, conforme discutido previamente. O estudo avança na análise de diferentes estratégias de emprego e criação de exemplos trabalhados. Notavelmente, a prática de rastreamento de código, empregando exemplos de modelagem, demonstrou que alunos expostos a tais exemplos obtiveram melhores resultados acadêmicos e exibiram taxas de desistência inferiores em comparação com seus pares que não tiveram acesso a esses recursos.

Em contraste, o estudo anterior expôs que o uso de vídeos instrutivos sugeriu que a presença física do instrutor em sala de aula não influencia de maneira significativa o aprendizado. Ademais, verificou-se que a apresentação estática de exemplos de códigos favorece o processo educativo de estudantes com menor base de conhecimento prévio sobre o assunto. Concluindo, a implementação de ferramentas de visualização, em detrimento das atividades tradicionais de rastreamento de código, mostrou-se capaz de elevar substancialmente tanto o desempenho quanto o aprendizado dos alunos.

No mesmo contexto, no que se refere aos exemplos vinculados à geração de código, o estudo concluiu que não há evidências suficientes para sustentar que o uso de exemplos de modelagem na geração de código resulte em uma melhoria na aprendizagem quando comparado ao uso de exemplos estáticos. Entretanto, os estudantes expressaram considerar os exemplos de geração de código em tempo real como úteis, enquanto os exemplos estáticos possuem a vantagem de permitir que os alunos compreendam os elementos do modelo em seu próprio ritmo de assimilação. Uma abordagem sugerida para aprimorar o uso de exemplos de geração de código é a inclusão de submetas nos exemplos, possibilitando uma melhor fragmentação das etapas necessárias para construir a solução do problema. Desta forma, o emprego de submetas indicou uma melhoria significativa no desempenho dos estudantes universitários.

Não suficiente, o artigo também explorou o uso de exemplos incompletos, que apresentam lacunas na solução para que os estudantes as preencham, é uma abordagem promissora para apoiar alunos em atividades de programação. Exemplos tradicionais com lacunas, onde os alunos completam programas existentes, mostraram-se benéficos. Em comparação com atividades de geração de código, os estudantes que completaram exemplos relataram menor carga cognitiva. Outra modalidade de exemplos incompletos são os quebra-cabeças de *Parsons* (do inglês "*Parsons puzzles*"), nos quais os alunos organizam fragmentos de código desordenados. Embora esses quebra-cabeças tenham reduzido o tempo de conclusão, não há evidências sólidas de que melhorem o aprendizado. O estudo também conclui que a presença de distratores e o tipo de *feedback* afetam os resultados.

A exposição dos alunos a um determinado conteúdo, como já vista, pode ser feita pelo uso de exemplos trabalhados(ATKINSON *et al.*, 2000). Todavia, o uso de *"Faded worked examples"* é uma estratégia de ensino que apresenta exemplos de problemas resolvidos aos alunos de forma gradual. Inicialmente, fornece-se um exemplo completamente resolvido para garantir a compreensão total da solução. Em seguida, ao longo de problemas subsequentes, partes do exemplo resolvido são progressivamente removidas. Os alunos são desafiados a preencher as lacunas, completando as partes ausentes da solução. Isto implica em abster o discente de lidar com situações passivas, tornando-o proficiente em resgatar informações no hipocampo do cérebro(SKUDDER; LUXTON-REILLY, 2014).

O impacto dos exemplos trabalhados na aprendizagem varia conforme o nível de experiência do aprendiz. Para iniciantes, esses exemplos proveem benefícios ao frisar a atenção nos elementos cruciais do problema, facilitando a criação de esquemas de resolução pertinen-

tes e otimizando recursos cognitivos em comparação com a abordagem direta na resolução de problemas. Em contraste, para indivíduos com certa experiência, os exemplos trabalhados podem tornar-se supérfluos, resultando em uma carga cognitiva extrínseca(SKUDDER; LUXTON-REILLY, 2014).

2.3 Exemplos trabalhados na Engenharia de Software

No estado atual da arte, poucos trabalhos são encontrados sobre o uso de exemplos trabalhados em ES, muito menos em SDs. Porém, o artigo intitulado "*Using Real Worked Examples to Aid Software Engineering Teaching*"(TONHÃO; COLANZI; STEINMACHER, 2021) busca compreender o uso de exemplos trabalhados na ES. Para tal, é realizado um *survey* que evidenciou haver pouco uso de exemplos trabalhados reais em sala de aula, devido às dificuldades encontradas na criação dos mesmos. Um questionário também foi feito para os discentes, onde foi apontado que o uso de exemplos trabalhados reais resultou em uma melhoria na percepção e motivação dos alunos.

No mesmo contexto, em um estudo de caso em sala de aula, foram utilizados exemplos trabalhados como material para ensinar a aplicação real de conceitos e para criação de atividades. Os resultados desse estudo de caso mostraram que mais de 80% dos estudantes ficaram mais motivados, sentiram que melhoraram as suas habilidades para o mercado de trabalho e, não suficiente, mais de 90% dos discentes concordaram que os exemplos trabalhados foram relevantes para a qualidade do ensino(TONHÃO; COLANZI; STEINMACHER, 2021).

Um exemplo trabalhado, conforme definido por Tonhão, Colanzi e Steinmacher (2021, p. 1), é um instrumento que compreende a apresentação de um problema, as etapas envolvidas em sua resolução e o desfecho final. Para ilustrar, consideremos um exemplo utilizado no estudo, onde foi feita a explicação do conceito da anomalia *large class*, que se refere a uma classe que assume mais responsabilidades do que seria apropriado. Esse problema pode ser identificado por meio da métrica *Lines of Code* (LOC) ou pela análise de coesão da classe. Em seguida, o contexto do problema em um projeto específico é delineado, no qual uma classe chamada *PackWrite* é examinada. Esta classe, responsável por gerar arquivos de pacote para um conjunto específico de objetos do repositório, incorporava uma classe estática, *ObjectToPack*, encarregada de empacotar um objeto.

Para solucionar essa questão, é necessário extrair a classe *ObjectToPack* para uma nova classe, tornando-a independente da classe *PackWriter* e transferindo todas as suas funcionalidades. Como resultado desse processo, a classe *PackWrite* passou a aderir ao princípio de responsabilidade única, no qual uma classe deve ser dedicada a resolver um problema específico, garantindo, assim, sua coesão integral. Dessa forma, o estudo mostrou, de forma eficiente, o que é um exemplo trabalhado, além do fato deles contribuírem grandiosamente para a qualidade do ensino(TONHÃO; COLANZI; STEINMACHER, 2021). No entanto, a criação de exemplos trabalhados não é trivial e apresenta diversos desafios.

A criação ou busca de exemplos trabalhados pode ser, e geralmente é, bem complexa. Uma das conclusões obtidas pela análise do *survey* realizado pelo estudo anterior aponta que os professores em geral têm dificuldade em encontrar bons exemplos, de acordo com: a complexidade desejada, adequação à disciplina lecionada, tópico procurado, entre outras. As principais dificuldades apontadas são a adequação à disciplina e a complexidade, visto que muitos exemplos são: (i) muito grandes e complexos, ou (ii) muito triviais, sem qualquer conexão com a realidade(TONHÃO; COLANZI; STEINMACHER, 2021). Além disso, outra dificuldade relevante apontada está relacionada ao fato de muitos exemplos não serem representativos do mundo real, por estarem desatualizados. Já é conhecido por outro estudo que um fator relevante para que alunos tenham uma boa ideia do que é exigido no mercado de trabalho é o uso de projetos atualizados(PINTO *et al.*, 2017).

Por meio de uma pesquisa exploratória feita no trabalho "*Students' Perception of Example-Based Learning in Software Modeling Education*"(BONETTI *et al.*, 2023), foi realizado um experimento com professores que nunca haviam utilizado Aprendizagem Baseada em Exemplos (ABE). Professores selecionados implementarem esta abordagem em uma aula sobre modelagem de diagramas de classe. A implementação da ABE foi feita instruindo os professores sobre como a abordagem proposta funciona. Em seguida, os professores deviam construir o plano de aula, utilizando o portal disponibilizado, para então executarem o plano de aula. A referida aula foi sucedida por um questionário que visava avaliar as percepções dos aprendizes em relação ao conteúdo apresentado.

Ainda sobre o mesmo estudo, a análise qualitativa dos resultados teve como objetivo observar os aspectos benéficos e desafiadores desse experimento. Os resultados evidenciaram que a estratégia de ABE não apenas contribuiu significativamente para o grau de compreensão e retenção dos conceitos abordados, mas também facilitou uma maior conexão entre a teoria e a prática. Adicionalmente, o trabalho destacou, como perspectivas futuras, a necessidade de investigar o impacto dessas estratégias em outras áreas da ES.

2.4 Trabalhos relacionados

O trabalho "*DistFax - A Toolkit for Measuring Interprocess Communications and Quality of Distributed Systems*"(FU; LIN; CAI, 2022) tem um cunho de análise de métricas em SDs, porém não com o intuito de criar exemplos trabalhados, mas sim de fornecer uma ferramenta que ajude a analisar determinados aspectos de SDs. Dessa forma, até o momento, não foram encontradas pesquisas sobre a utilização da evolução de código-fonte de SDs por meio de métricas estáticas de código para a criação de exemplos trabalhados na disciplina de ES.

No mesmo contexto, o artigo propõe a ferramenta *DistFax* para a análise de métricas de IPC, que demonstraram correlação com métricas de qualidade de software. As métricas utilizadas na ferramenta são dinâmicas, as quais requerem o uso de relações *happened-before*

que, como já explicado no começo desse capítulo, são um desafio por não haver um relógio global.

As métricas de acoplamento utilizadas no *DistFax* são:

- ***Run-time Message Coupling* (RMC)**: Contagem de mensagens enviadas de um processo para outro.
- ***Run-time Class Coupling* (RCC)**: Razão entre o número de métodos em uma classe dependentes de métodos em outra classe.
- ***Class Central Coupling* (CCC)**: Representa o acoplamento agregado de uma classe em relação a classes em processos remotos.
- ***Interprocess Reuse* (IPR)**: Mede o acoplamento entre processos mediante métodos comuns.
- ***Class Comunication Load* (CCL)**: Avalia a carga de comunicação de uma classe individual com outras classes em todos os processos remotos.

A complexidade ciclomática, que representa a quantidade de caminhos distintos no código, pode ser interpretada como o número mínimo de casos de teste necessário para cobrir todos os caminhos do programa sem repetição, servindo para garantir uma cobertura adequada durante os testes. Portanto, quanto menor a complexidade ciclomática, melhor.

No mesmo contexto, dois modelos de Inteligência Artificial (IA) foram desenvolvidos, para predizer a condição de qualidade de um SDs, dado como entrada as métricas IPC coletadas, onde o modelo supervisionado utilizava a técnica de *Bagging* e o modelo não supervisionado utilizava o algoritmo *K-Means*. No *DistFax*, foram utilizadas seis métricas de IPC (cinco de acoplamento e uma de coesão), correlacionadas com cinco métricas de qualidade de software, sendo elas, tempo de execução, complexidade ciclomática, contagem de caminhos de fluxo de informações, comprimento do caminho do fluxo de informações e superfície de ataque. Observaram-se correlações positivas entre: (i) CCC com o tempo de execução, (ii) RMC e CCC com a complexidade ciclomática, e (iii) CCC com a área de ataque (do inglês, *attack-surface*). Houve também uma correlação negativa entre CCL e *Process-Level Cohesion* (PLC) com a área de ataque(FU; LIN; CAI, 2022).

A única métrica de coesão utilizada no *DistFax* foi PLC, que mede as conexões internas em um processo individual.

A correlação positiva entre CCC e o tempo de execução sugere que um elevado nível de acoplamento está correlacionado a um desempenho inferior. Portanto, nas métricas *Code Metrics* (CK) adotadas em nossa pesquisa, reconhecemos que *Coupling Between Object* (CBO), o qual será explicado na seção 3.2.1, desempenha um papel importante. Além disso, a correlação positiva entre CCC e a área de ataque, em termos de segurança de software, indica que códigos com alto acoplamento são mais vulneráveis a potenciais pontos de ataque. Isso significa

que um invasor pode explorar esses pontos para obter acesso não autorizado ou realizar ações prejudiciais. Esse fato reforça a importância da métrica CBO como um indicador significativo da qualidade do código, especialmente no contexto da segurança do software.

O artigo “RefactorScore: Evaluating Refactor Prone Code”(JESSE; KUHMUENCH; SAWANT, 2023) desenvolve a ferramenta intitulada *RefactorScore*[1], a qual tem o intuito de prover uma métrica de avaliação automática de código que reconhece, de acordo com os *tokens* de um arquivo, áreas propensas a refatoração. Com base nessa identificação, a ferramenta gera uma pontuação correspondente, refletindo o potencial de melhoria nessas áreas específicas. A ferramenta *RefactorScore* está presente no modelo *RefactorBert*, o qual aprendeu com a mudança de códigos entre refatorações, mostrando-se eficiente em detectar erros de qualidade de código. Esse tipo de erro está, por exemplo, associado a código fora do padrão de *design* de código adotado.

O modelo *RefactorBert* foi desenvolvido com base em repositórios de linguagem C/C++(RITCHIE; STROUSTRUP, 1972/1983), contendo mais de um milhão de *commits* ao todo. Contudo, os repositórios examinados nesta monografia não estão presentes no dataset[2] utilizado pela ferramenta *RefactorScore*, o que torna impraticável seu uso para a coleta de dados relacionados à refatoração nos contextos abordados. Adicionalmente, a falta de documentação sobre a utilização da ferramenta *RefactorScore*, juntamente com a limitada presença de apenas três *commits* no repositório da ferramenta, reforça a inviabilidade de seu emprego. Para adaptar a ferramenta ao contexto de repositórios Java(GOSLING, 1996), seria essencial criar um novo *dataset* e treinar o modelo para reconhecer padrões de refatoração em código-fonte do tipo almejado.

2.5 Considerações finais

Este capítulo explorou o referencial teórico necessário para compreender os SDs (SDs) e seus desafios, fornecendo uma base conceitual sólida para a análise e discussão dos resultados desta pesquisa. Os principais Conceitos foram destacados, como a definição de SDs, os desafios inerentes à sua implementação, e a interseção entre esses desafios e os princípios da ES.

Ao analisar os desafios específicos enfrentados pelos SDs, destacamos a importância da heterogeneidade, a ausência de um relógio global, a necessidade de tolerância a falhas e escalabilidade. A discussão sobre o Teorema CAP ressaltou os *trade-offs* cruciais na busca por consistência, disponibilidade e tolerância a partições em um SD.

Explorou-se também o conceito de exemplos trabalhados e sua relevância no contexto da educação em ES. Os exemplos trabalhados oferecem uma abordagem eficaz para ensinar

[1] https://huggingface.co/kevinjesse/RefactorBERT/tree/main

[2] https://huggingface.co/datasets/kevinjesse/ManyRefactors4C

conceitos complexos, proporcionando uma ponte entre a teoria e a prática. No entanto, a criação e seleção de exemplos trabalhados apresentam desafios, especialmente no contexto de SDs.

Além disso, trabalhos relacionados que abordam ferramentas e métricas para análise de qualidade em SDs foram discutidos, destacando o *DistFax* e suas métricas de acoplamento, coesão e correlações significativas com desempenho e segurança do software.

No contexto educacional, evidenciou-se a escassez de pesquisas sobre exemplos trabalhados em SDs, ressaltando a importância de abordagens pedagógicas que integrem de maneira eficaz os desafios específicos desses sistemas.

No próximo capítulo, é apresentada a metodologia adotada para a realização deste estudo, detalhando os procedimentos e técnicas utilizados na coleta e análise dos dados.

3 METODOLOGIA

Este capítulo delineia os objetivos e a abordagem da pesquisa, apresentando uma visão geral das questões fundamentais apresentadas. Além disso, não só explica as métricas, ferramentas, repositórios utilizados, como também o método e resultados esperados, relacionados com o cronograma definido para a conclusão do projeto de pesquisa.

3.1 Questões de pesquisa

As questões de pesquisa foram formuladas com base nos objetivos geral e específicos desta monografia. Elas direcionarão a investigação sobre a evolução de projetos de SDs por meio da análise de métricas de código, de modo a criar um exemplo trabalhado para a ES, como apresentadas abaixo:

1. Como identificar métricas de qualidade de código relevantes para a análise da evolução em SDs?
2. Quais são os padrões e tendências observados que correspondem a uma melhoria clara do código em SDs?
3. Como as melhorias de código se refletem nas métricas selecionadas e qual é a relação delas com características não funcionais como acoplamento, coesão, reusabilidade e manutenibilidade?
4. Quais são as métricas e características mais relevantes para a seleção de exemplos de código apropriados para a criação de exemplos trabalhados na ES?

3.2 Abordagem proposta

O presente trabalho propõe o desenvolvimento de uma heurística para identificar exemplos de código-fonte representativos da melhoria da qualidade de software em SDs, por meio da análise de métricas de código utilizadas na ES. A heurística visa auxiliar na seleção de trechos de código que permitam a criação de exemplos trabalhados que demonstrem como o código em SDs se adapta e evolui ao longo do tempo.

A heurística a ser desenvolvida nesta pesquisa baseia-se nas melhorias que podem ser identificadas através das métricas selecionadas. As métricas devem ser capazes de capturar informações valiosas relacionadas à evolução do código em SDs. Logo, são empregadas ferramentas específicas em repositórios de código aberto cuidadosamente selecionados.

3.2.1 Métricas de software

O artigo "*A Metrics Suite for Object Oriented Design*"(CHIDAMBER; KEMERER, 1994) apresenta um conjunto de métricas para avaliação do design de código orientado a objetos, introduzindo métricas que permitem compreender a complexidade e qualidade do código-fonte. As principais métricas apresentadas no artigo são listadas a seguir.

- **CBO**: Reflete o grau de acoplamento, ou seja, as dependências que uma classe específica possui, indicando a quantidade de classes diretamente associadas a ela, já que utiliza métodos dessas classes. Um valor elevado sugere maior complexidade e menor flexibilidade, pois mudanças na classe podem impactar várias outras. Não suficiente, demasiado acoplamento diminui o grau de modularidade de uma classe. Observar uma redução no CBO ao longo da evolução do código é um indicativo positivo de melhorias na qualidade do código. Um esforço para reduzir o CBO pode levar a um *design* de *software* mais modular, onde as mudanças em uma parte do sistema têm menos probabilidade de exigir mudanças em outra.

- ***Depth of Inheritance Tree* (DIT)**: Reflete a profundidade da árvore de herança à qual uma classe depende, indicando a quantidade de ancestrais que uma classe tem. Um valor elevado de DIT pode indicar uma maior complexidade e potencial para efeitos colaterais ao realizar alterações em alguma das superclasses.

- ***Lack of Cohesion in Methods* (LCOM)**: Representa o coeficiente de falta de coesão em métodos, mensurando a coesão em uma classe. O valor varia de 0 a 1, onde 0 indica alta coesão e 1 indica falta de coesão. Baixos valores são desejados, indicando que os métodos estão fortemente inter-relacionados, contribuindo para uma classe mais coesa.

- ***Number of Children* (NOC)**: Indica o número de filhas diretas de uma classe. Um valor alto pode indicar maior reusabilidade. Isso implica que a referida classe desempenha uma função importante, dada a dependência de outras classes sobre ela.

- ***Response for a Class* (RFC)**: Refere-se ao tamanho do conjunto de respostas de uma classe, ou seja, o número de métodos que podem ser potencialmente executados em resposta a uma mensagem recebida por um objeto dessa classe. Um valor menor de RFC indica que uma classe tem menos comportamentos e, potencialmente, um menor grau de complexidade. Classes com RFC baixo tendem a ser mais coesas e menos acopladas, pois interagem com menos outras classes e componentes, facilitando assim a manutenção e a testabilidade.

- ***Weight Method Class* (WMC)**: Simboliza a soma dos valores de complexidade dos métodos de uma classe específica. Um valor elevado indica que a classe pode ser

complexa, com múltiplos métodos, implicando em um custo significativo para o desenvolvimento e manutenção. Além disso, a existência de vários métodos associados a uma classe sugere que ela pode ser menos genérica, sem mencionar o possível impacto nos filhos dessa classe. Classes com WMC baixo tendem a ser mais coesas, o que significa que elas têm um propósito bem definido e limitado. A redução do WMC pode ser um sinal de que a classe está se tornando mais focada em suas responsabilidades, o que facilita o entendimento, a manutenção e a extensão do código.

Dentre os padrões de refatorações identificados pelo *RefactoringMiner*, tem-se interesse intrínseco pelas refatorações do tipo *Extract Method*, *Extract Class*, *Pull Up Method*, *Push Down Method*, *Extract Superclass*, visto que esses tipos de refatorações são fundamentais para a melhoria da qualidade do código, promovendo a modularização, a reutilização e a manutenção do software. A refatoração *Extract Method* é crucial para reduzir a complexidade de métodos longos, isolando partes do código em novos métodos para melhor compreensão e reuso. *Extract Class* ajuda a combater o antipadrão de classes "*God Class*", distribuindo responsabilidades de uma classe sobrecarregada para novas classes, o que facilita a manutenção e a expansão do sistema. As refatorações *Pull Up Method* e *Push Down Method* são importantes para otimizar a hierarquia de classes, movendo métodos para a classe base ou derivada, respectivamente, promovendo um melhor aproveitamento da herança e da especialização. Por fim, *Extract Superclass* permite a criação de abstrações mais genéricas ao identificar e extrair um conjunto comum de características e comportamentos, facilitando o gerenciamento de dependências e a evolução do software. Tais práticas são essenciais para a engenharia de software, pois contribuem diretamente para a coesão, o acoplamento e a encapsulação do código, pilares para o desenvolvimento de sistemas de software robustos, escaláveis e facilmente mantíveis.

3.2.2 Ferramentas

O CK[1](ANICHE, 2015) é a ferramenta utilizada nesta pesquisa para analisar a qualidade do código-fonte em projetos Java(GOSLING, 1996) por meio de métricas estáticas. Essas métricas fornecem informações valiosas sobre características como complexidade, acoplamento e coesão de classes. A ferramenta em questão apresenta mais de 35 métricas.

O *RefactoringMiner*[2](TSANTALIS *et al.*, 2018) é uma ferramenta no contexto da ES, especializada em identificar e analisar refatorações de código-fonte. Essa ferramenta fornece uma compreensão aprofundada das mudanças realizadas em um código ao longo do tempo. Com a sua capacidade de reconhecer padrões de refatoração, o *RefactoringMiner* permite que os desenvolvedores analisem como o código foi modificado, de modo a melhorar a qualidade, manutenibilidade e eficiência do software.

[1] https://github.com/mauricioaniche/ck
[2] https://github.com/tsantalis/RefactoringMiner

A biblioteca *PyDriller*[3](SPADINI; ANICHE; BACCHELLI, 2018), disponível para a linguagem de programação Python, é um recurso para análise de repositórios. Sua funcionalidade possibilita aos pesquisadores e desenvolvedores explorar e compreender a evolução do código em projetos Python de uma maneira eficiente. O *PyDriller* oferece recursos para extrair informações, como histórico, autoria e mensagens de *commits*, além de detalhes específicos do estado do código em um determinado momento, contribuindo assim para a pesquisa e prática em ES no contexto de sistemas complexos e interconectados.

3.2.3 Repositórios

A seguir são apresentados os projetos de código aberto disponíveis no https://github.com, que serão analisados neste trabalho de pesquisa.

O *Apache Kafka*(Apache Software Foundation, 2011) é um sistema de mensagens distribuídas, baseado no modelo *publish-subscribe*, amplamente utilizado em infraestruturas de processamento de eventos em tempo real. Conforme o artigo "*Apache Kafka: Next Generation Distributed Messaging System*"(KREPS NEHA NARKHEDE, 2010), o Kafka é projetado para lidar com fluxos de dados em grande escala, permitindo que organizações processem, armazenem e transmitam dados de maneira eficiente, operando em um modelo de log distribuído, no qual as mensagens são registradas em logs (ou tópicos) que podem ser divididos em partições. A estrutura do Apache Kafka é composta por vários componentes, incluindo:

- **Tópicos**: São canais de mensagens nos quais os dados são publicados e armazenados.
- **Produtores**: São responsáveis por publicar mensagens em tópicos do Kafka.
- **Partições**: Os tópicos podem ser divididos em partições para permitir a distribuição e o processamento paralelo de mensagens.
- **Corretores (Brokers)**: São servidores que armazenam e distribuem as mensagens. Eles desempenham um papel central na arquitetura do Kafka.
- **Consumidores**: São aplicativos que se inscrevem nos tópicos para receber mensagens e processá-las.
- **ZooKeeper (ZK) (KRaft em 2023)**: No passado, o Kafka dependia do Apache ZK para gerenciamento de metadados e coordenação. No entanto, em 2023, o Kafka adotou o *KRaft* como substituto, eliminando a dependência do ZK, com o intuito de deixá-lo mais simples e robusto.

[3] https://github.com/ishepard/pydriller

Essa arquitetura permite a escalabilidade, tolerância a falhas e a distribuição das mensagens em várias máquinas e servidores, tornando-o uma escolha popular em empresas como LinkedIn, X (Twitter), Uber, Foursquare, entre outros para a construção de pipelines de dados em tempo real e sistemas de processamento de eventos(KAFKA, 2023). Dessa forma, dada a sua arquitetura, uso no mundo real, além da capacidade do *"Apache Kafka"* lidar com volumes massivos de dados em tempo real, ele torna-se um ótimo candidato a ser estudado.

O ZK(Apache Software Foundation, 2008) é um serviço de coordenação distribuída amplamente utilizado para sistemas de grande escala na internet. Ele fornece um ambiente confiável e altamente disponível para a coordenação de tarefas entre diversos nós em um cluster distribuído. Na ótica do artigo "*ZooKeeper: Wait-free coordination for Internet-scale systems*"(HUNT *et al.*, 2010), o ZK é baseado em um conjunto de servidores distribuídos que formam o chamado "*ensemble*". Cada servidor nesse conjunto mantém uma cópia completa dos dados de configuração e estado do sistema. Os clientes se conectam a qualquer servidor do conjunto e podem ler ou gravar informações. Quando os clientes gravam informações, elas são replicadas para a maioria dos servidores no conjunto antes de serem confirmadas, garantindo assim a consistência dos dados.

O ZK fornece um sistema de coordenação sem espera (do inglês, "*wait-free*"), por meio de um mecanismo de observação para permitir que os clientes armazenem dados em *cache* sem ter que gerenciar o *cache* do cliente diretamente. Além disso, a implementação do algoritmo denominado *Zookeeper Atomic Broadcast Protocol* (ZAB) é responsável por garantir a consistência de dados distribuídos no ZK por meio do uso do *Two-Phase Commit Protocol* para replicar todas as transações para todos os nós do cluster do ZK(VINKA, 2018). O *Two-Phase Commit Protocol* consiste em duas fases. Na primeira fase, um coordenador propõe a execução de uma transação aos participantes, que respondem indicando se estão prontos para realizar a transação. Na segunda fase, se todos concordarem, o coordenador emite o comando de *commit* para que os participantes realizem efetivamente a transação; caso contrário, emite o comando de *rollback*, revertendo as alterações.

O *"ZooKeeper"* é utilizado para coordenação e gerenciamento de serviços distribuídos. Ele oferece um serviço de consenso altamente confiável para SDs, garantindo a consistência e a sincronização entre os nós. Essa funcionalidade permite a implementação de serviços distribuídos confiáveis e escaláveis, tornando o *"ZooKeeper"* uma escolha valiosa para este estudo.

3.3 Métodos

Este trabalho de pesquisa adota uma abordagem exploratória com o propósito de desenvolver uma heurística que possibilite a seleção criteriosa de códigos representativos de melhorias de software, de acordo com métricas específicas de *código* em projetos de SDs. O objetivo principal é aprofundar a compreensão desses casos, proporcionando uma análise detalhada da evolução do código em resposta a mudanças nas métricas escolhidas. Esse estudo visa

contribuir diretamente para a construção de exemplos trabalhados aplicáveis ao contexto da ES.

A natureza exploratória da pesquisa pretende proporcionar uma visão mais ampla sobre como as métricas de código em projetos de SDs refletem melhorias significativas. A heurística em desenvolvimento visa identificar casos exemplares de evolução de código, considerando métricas específicas, selecionadas com base nas melhores práticas e no entendimento profundo das características inerentes aos SDs.

Para alcançar esses objetivos, serão realizadas análises minuciosas de repositórios de código aberto relevantes na área de SDs, considerando não apenas a evolução do código-fonte, mas também as mudanças nas métricas escolhidas ao longo do tempo. Na esfera da segurança, será explorada a possibilidade de correlacionar informações com alterações específicas no código, proporcionando uma análise abrangente e multifacetada. Não suficiente, questões como IPC, disponibilidade, transparência e heterogeneidade também serão estudadas. A metodologia adotada permite, assim, uma compreensão holística da relação entre a qualidade do código e as métricas específicas selecionadas.

Além disso, a pesquisa visa ir além da mera análise quantitativa, incorporando uma abordagem qualitativa ao desenvolver exemplos trabalhados que demonstrem, de forma clara e pedagogicamente eficaz, como a evolução do código em SDs pode ser interpretada e compreendida no âmbito da ES. Essa abordagem qualitativa permite a construção de um conhecimento prático e aplicável que possa enriquecer o ensino e a prática profissional na área.

3.4 Resultados esperados e cronograma

1. Realizar uma revisão da literatura, explorando estudos sobre métricas de código em ES e SDs.
   Resultado esperado: Lista de estudos sobre exemplos trabalhados e as respectivas contribuições.

2. Selecionar um ou mais repositórios *open-source* pertinentes à área de SDs que servirão como fonte de exemplos de código para análise.
   Resultado esperado: Lista de repositórios *open-source* ativamente mantidos e utilizados em aplicações reais selecionados para análise.

3. Selecionar métricas de código específicas e relevantes apropriadas para avaliar a evolução de código em SDs, considerando as melhores práticas da área.
   Resultado esperado: Lista de métricas de código selecionadas, capazes de refletir melhorias, por exemplo, com relações ao acoplamento e coesão de cada classe.

4. Escrita e publicação de um artigo resumido no evento EduComp2024 sobre o projeto e as metodologias utilizadas até o momento.

Resultado esperado: Resumo publicado apresentando o projeto, metodologias e atividades desenvolvidas de fevereiro até abril no EduComp2024.

5. Selecionar exemplos específicos de código trabalhados nos repositórios escolhidos, aplicando a heurística desenvolvida e justificando a seleção com base em métricas de código.
   Resultado esperado: Lista de candidatos potenciais com refatorações significativas, com relação à evolução da qualidade do código.

6. Avaliar a utilidade educacional dos exemplos de código selecionados, usando-os em um contexto de ensino.
   Resultado esperado: Contribuir para o estudo do código de um SD no âmbito da ES.

7. Escrita e publicação de um artigo no *SBES-Education* sobre os resultados obtidos até o momento, destacando descobertas e contribuições.
   Resultado esperado: Artigo publicado apresentando os resultados e contribuições entre abril e junho no *SBES-Education*.

Dessa forma, a Tabela 1 apresenta um cronograma de atividades.

Tabela 1 – Cronograma de atividades

Atividade/Mês	Jan	Fev	Mar	Abr	Mai	Jun
1 - Revisão da literatura	x	x	x	x		
2 - Seleção de repositórios	x	x				
3 - Seleção de métricas	x	x	x			
4 - Submissão de um resumo no EduComp2024		x	x	x		
5 - Seleção de exemplos				x	x	x
6 - Avaliação da utilidade educacional				x	x	x
7 - Submissão de um artigo no *SBES-Education*					x	x

4 RESULTADOS PRÉVIOS

4.1 Seleção de projetos

Os critérios para a seleção de repositórios foram determinados pelos seguintes fatores:

- **Ser um repositório Java:** Essencial para viabilizar a aplicação das métricas CK.
- **Ser de código aberto (*open-source*):** A escolha por repositórios abertos, hospedados em plataformas como o *Github*, tem como propósito assegurar a transparência e acessibilidade do código-fonte. Ao selecionar um repositório disponível no *Github*, é viável, por meio da biblioteca *PyDriller*, percorrer a árvore de *commits*. Desse modo, em cada *commit*, torna-se possível gerar as métricas CK.
- **Ser ativamente mantido e atualizado:** A ativa manutenção é necessária para garantir que os repositórios estejam alinhados com as práticas e avanços mais recentes.
- **Ser utilizado em aplicações do mundo real:** A seleção de repositórios empregados em contextos práticos proporciona uma análise mais relevante e aplicável às situações reais.

Os projetos escolhidos foram o *"Apache Kafka"*(Apache Software Foundation, 2011) e o *"ZooKeeper"*(Apache Software Foundation, 2008). Essa escolha baseou-se no fato de os repositórios serem projetos *open-source* disponíveis na plataforma GitHub, onde ambos são mantidos e atualizados de forma ativa para serem utilizados em ambientes do mundo real. Além disso, esses projetos lidam com questões altamente relevantes em SDs.

Logo, ambos os repositórios têm muito a agregar em termos de implementação e evolução, tanto para a área de SDs quanto para a ES. Assim, a escolha desses repositórios propicia a análise da evolução do código em contextos práticos e desafiadores, contribuindo para uma compreensão mais abrangente das práticas de desenvolvimento em SDs.

4.2 Métricas Preliminarmente Selecionadas

No estágio inicial desta pesquisa, optou-se por selecionar um conjunto específico de métricas, incluindo CBO, *Coupling Between Object Modified* (CBOM), RFC e WMC. Essa escolha foi motivada pela intenção de obter uma combinação abrangente que abordasse diferentes aspectos da qualidade do código. As métricas CBO e CBOM foram escolhidas para fornecer *insights* sobre o nível de acoplamento em classes ou métodos, enquanto RFC foi selecionada para medir o grau de comunicação entre os elementos do código. Por fim, a métrica WMC foi incluída para avaliar a complexidade dos métodos em uma classe.

Contudo, durante o desenvolvimento do estudo, decidiu-se remover a métrica CBOM da análise. Essa decisão foi tomada devido à percepção de que a inclusão de CBOM poderia introduzir poluição nos dados, aumentando a complexidade artificialmente. Isso ocorre, por exemplo, quando a adição de uma chamada a um método da própria classe afeta negativamente a métrica. Dessa forma, a exclusão de CBOM visou manter a integridade e a precisão das métricas escolhidas, proporcionando uma análise mais clara e objetiva da evolução do código.

4.3 Desenvolvimento da ferramenta

As ferramentas desenvolvidas podem ser encontradas no Github [1], estando as três primeiras no diretório "*PyDriller*" e as duas últimas no diretório "*RefactoringMiner*", sendo que todas foram feitas usando a linguagem de programação Python(ROSSUM, 1991).

A primeira ferramenta desenvolvida, denominada *code_metrics.py*(SILVA, 2023a), está integrada ao *PyDriller* e ao CK. Durante esse processo, a ferramenta percorre toda a árvore de *commits* de um repositório específico. Para cada *commit*, o CK é executado, resultando na geração das métricas CK para cada estado do repositório ao final da execução. Adicionalmente, a ferramenta produz um *diff* para cada refatoração realizada no repositório, possibilitando a investigação das refatorações realizadas no código em caso de identificação de uma tendência de melhoria das métricas.

A segunda ferramenta desenvolvida, denominada *metrics_changes.py*(SILVA, 2023b), depende da execução da primeira para operar. Essencialmente, esta ferramenta percorre as métricas de todos os estados do repositório processado, produzindo os seguintes resultados:

- **Evolução das métricas**: Geração de arquivos divididos por classe e métodos, contendo o histórico completo de alterações em cada classe ou método, incluindo suas métricas correspondentes.
- **Predição das métricas**: Criação de arquivos segmentados por classe e métodos, apresentando uma predição baseada na regressão linear aplicada às métricas analisadas, utilizando o histórico gerado anteriormente.
- **Estatísticas das métricas**: Produção de um arquivo para classes e outro para métodos, ordenados pelo número de alterações efetuadas em cada classe ou método.
- **Alterações substanciais nas métricas**: Identificação de padrões de decréscimo do CBO, oferecendo uma análise detalhada das classes ou métodos que apresentam tendências de melhoria ao longo do tempo.

Assim, resumidamente, as ferramentas *code_metrics.py* e *metrics_changes.py* geram metadados que facilitam a análise dos códigos, reduzindo o escopo de busca por refatorações

[1] https://github.com/BrenoFariasdaSilva/Worked-Example-Miner

que, à luz das métricas, são consideradas relevantes para nosso estudo. Vale ressaltar que, enquanto o estudo que desenvolve o *DistFax*(FU; LIN; CAI, 2022) utiliza métricas IPC dinâmicas (geradas em tempo de execução), o presente estudo emprega métricas de código estáticas (geradas apenas pela análise do código-fonte). Isso se deve à inadequação das métricas tradicionais para medir a qualidade das trocas de mensagens em SDs, pois eles não se baseiam em relações explícitas, analisam apenas um computador e não consideram o sistema em sua totalidade. Embora o uso dessas métricas pudesse ser benéfico, o custo computacional pode ser bastante significativo para calcular todas elas.

As Figuras 1, 2 e 3 apresentam exemplos das saídas geradas pelo algoritmo *metrics_changes.py* para a classe *org.apache.zookeeper.server.quorum.Follower*. A seleção dessa classe deve-se ao fato de que, pelo nome, é possível compreender que é uma classe que trata da parte do consenso do protocolo, ou seja, dos participantes (*followers*) entrarem em consenso em uma votação. Dado essa premissa, analisou-se essa classe e percebeu-se uma tendência de queda nas métricas CBO, RFC e WMC.

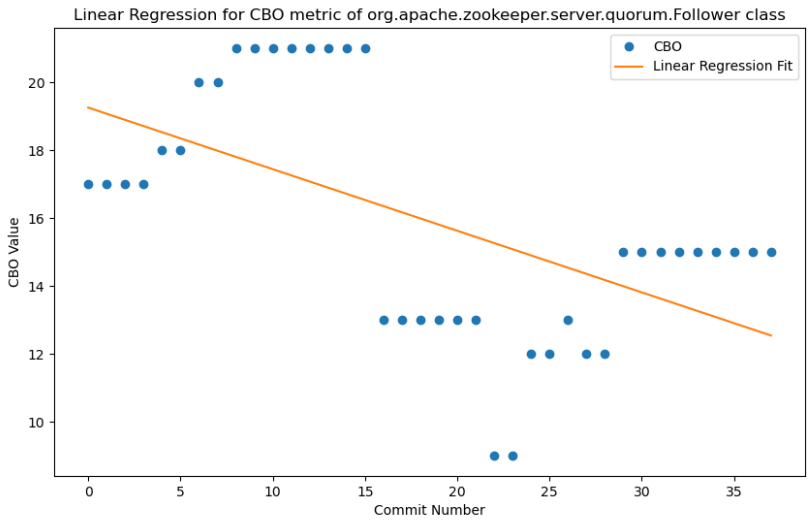

Figura 1 – Regressão linear da métrica CBO da classe org.apache.zookeeper.server.quorum.Follower

No mesmo contexto, as Figuras 1, 2 e 3 também proporcionam uma perspectiva visual das tendências nas métricas CBO, RFC e WMC específicas da classe *org.apache.zookeeper.server.quorum.Follower*. Enquanto isso, as Figuras 4 e 5 apresentam um vislumbre das cinco primeiras linhas dos *CSVs* gerados, delineando a evolução das métricas de tal classe e as estatísticas gerais das classes no contexto do *ZooKeeper*. As primeiras três imagens oferecem uma compreensão aprofundada do comportamento evolutivo dessa classe específica, enquanto a última concentra-se nas classes mais frequentemente modificadas, destacando sua propensão a refatorações significativas ou sua relevância dentro do ecossistema do *ZooKeeper*.

A terceira ferramenta desenvolvida, denominada de *track_files.py*(SILVA, 2023e), mais uma vez, depende da execução da primeira para funcionar. Em suma, esta ferramenta busca

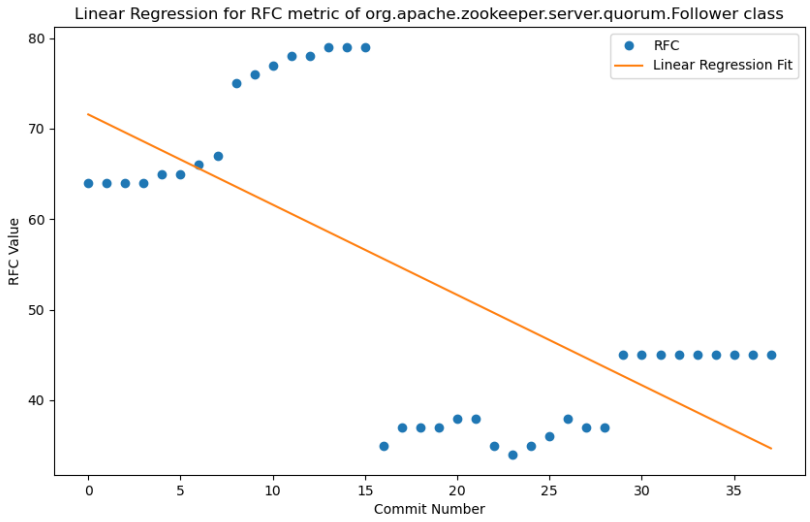

Figura 2 – Regressão linear da métrica RFC da classe org.apache.zookeeper.server.quorum.Follower

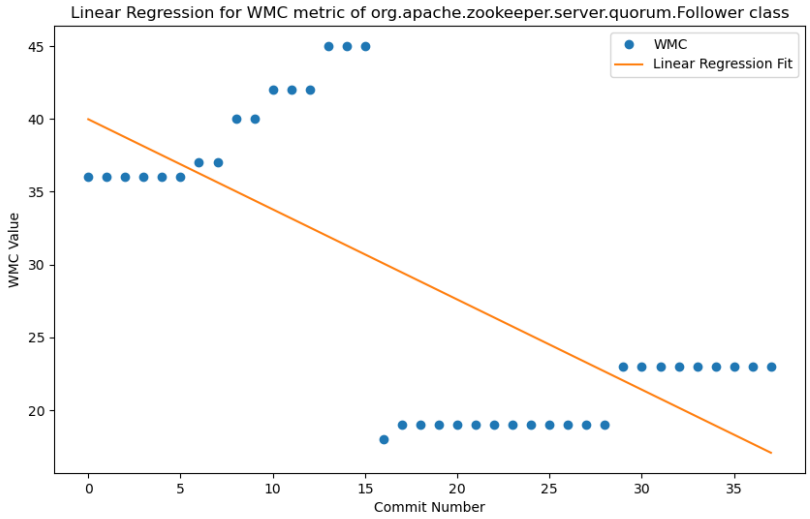

Figura 3 – Regressão linear da métrica WMC da classe org.apache.zookeeper.server.quorum.Follower

por arquivos específicos nos *diffs* dos repositórios. Neste estudo, ela foi utilizada para localizar arquivos de comentários sobre as refatorações realizadas em um *commit*, como o arquivo "CHANGES.txt", uma vez que o espaço da mensagem do *commit* pode oferecer uma visão limitada das refatorações efetuadas.

As ferramentas desenvolvidas, nomeadas *metrics_evolution_refactorings.py*(SILVA, 2023d) (quarta ferramenta) e *repository_refactorings.py*(SILVA, 2023c) (quinta ferramenta), são substancialmente similares, distinguindo-se apenas no escopo de aplicação, onde a quarta incide sobre a totalidade do repositório, enquanto a quinta é específica para classes ou métodos dentro do repositório. Ambas as ferramentas compartilham um funcionamento comum, fazendo uso da ferramenta *RefactoringMiner*(TSANTALIS *et al.*, 2018).

Class	Commit Hash	CBO	WMC	RFC
org.apache.zookeeper.server.quorum.Follower	175-cfd2ecb8d049c1b51b2c1e6d5fd02edd5d3c26e8	17.0	36.0	64.0
org.apache.zookeeper.server.quorum.Follower	176-b4b73a37d43413efdf50427f96aab2720af11baf	17.0	36.0	64.0
org.apache.zookeeper.server.quorum.Follower	177-0a81c16c4c921ef52179b943dd046fc72022ccc6	17.0	36.0	64.0
org.apache.zookeeper.server.quorum.Follower	208-435bc0ff5db851d341c373bc515f6a618691e6c4	17.0	36.0	64.0
org.apache.zookeeper.server.quorum.Follower	211-3a6a4ba7fc538f066d7b691777320fb9b92a5f93	18.0	36.0	65.0

Figura 4 – Evolução das métricas da classe org.apache.zookeeper.server.quorum.Follower

Class	Type	Changed	CBO Min	CBO Max	CBO Avg	CBO Q3	WMC Min	WMC Max	WMC Avg	WMC Q3	RFC Min	RFC Max	RFC Avg
org.apache.zookeeper.server.quorum.QuorumPeer$ServerState	enum	115	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
org.apache.zookeeper.server.quorum.QuorumPeer$QuorumServer	innerclass	115	0.0	4.0	2.139	4.0	1.0	41.0	21.835	41.0	0.0	13.0	6.061
org.apache.zookeeper.server.quorum.QuorumPeer	class	115	21.0	41.0	35.948	41.0	67.0	228.0	161.887	226.0	33.0	137.0	92.53
org.apache.zookeeper.server.quorum.QuorumPeer$ResponderThread	innerclass	115	2.0	5.0	4.304	5.0	10.0	12.0	11.374	12.0	11.0	15.0	14.078
org.apache.zookeeper.ClientCnxn$Packet	innerclass	89	5.0	8.0	6.697	8.0	4.0	8.0	6.135	8.0	11.0	16.0	14.764

Figura 5 – Estatísticas das métricas do ZooKeeper

A utilização das ferramentas associadas ao *RefactoringMiner* é feita por meio da extração do histórico de *commits*, permitindo a análise dos *diffs*, identificando padrões de refatorações. O *RefactoringMiner* possui um banco de dados de refatorações previamente detectadas, permitindo uma análise eficiente das mudanças no código-fonte. A ferramenta gera uma saída estruturada que lista as refatorações identificadas, indicando quais arquivos e linhas de código foram afetados em cada *commit*.

Essa saída estruturada proporciona uma base sólida para análises mais aprofundadas, permitindo uma compreensão detalhada das refatorações realizadas ao longo do tempo. A capacidade de distinguir entre o repositório inteiro (quarta ferramenta) e componentes específicos, como classes ou métodos (quinta ferramenta), amplia a flexibilidade dessas ferramentas, adaptando-as às necessidades específicas de investigação e análise de refatorações em projetos de software.

A saída dos algoritmos que utilizam o *RefactoringMiner* consiste em um objeto *JSON*, semelhante à Figura 6, que permite a análise das refatorações por tipo. Destaca-se que, a partir do *JSON* geral, é possível aplicar filtros para armazenar apenas refatorações dos tipos desejados. Essa abordagem proporciona uma visão detalhada e personalizável das transformações efetuadas no código-fonte.

4.4 Análise das métricas dos projetos selecionados

Apesar da intenção inicial de analisar dois repositórios, o *“Apache Kafka”* e o *“ZooKeeper”*, até o momento, concentramos nossos esforços exclusivamente no exame do repositório do *“ZooKeeper”*. Esta escolha se fundamenta no tamanho inferior deste repositório em comparação com o *“Apache Kafka”*, permitindo uma análise mais focalizada e a possibilidade de apresentar resultados preliminares neste trabalho. A decisão foi motivada por considerações de limitação de tempo, pois a análise detalhada de um único repositório já representa um desafio significativo, demandando uma abordagem minuciosa para compreender a evolução do código-fonte e

```
{
 "type": "Pull Up Method",
 "description": "Pull Up Method public getPendingRevalidationsCount() : int from
 "leftSideLocations": [
  {
   "filePath": "src/java/main/org/apache/zookeeper/server/quorum/Follower.java",
   "startLine": 332,
   "endLine": 334,
   "startColumn": 5,
   "endColumn": 6,
   "codeElementType": "METHOD_DECLARATION",
   "description": "original method declaration",
   "codeElement": "public getPendingRevalidationsCount() : int"
  }
 ],
 "rightSideLocations": [
  {
   "filePath": "src/java/main/org/apache/zookeeper/server/quorum/Learner.java",
   "startLine": 78,
   "endLine": 80,
   "startColumn": 5,
   "endColumn": 6,
   "codeElementType": "METHOD_DECLARATION",
   "description": "pulled up method declaration",
   "codeElement": "public getPendingRevalidationsCount() : int"
  }
```


Figura 6 – Refatorações da classe org.apache.zookeeper.server.quorum.Learner

as refatorações realizadas. Neste trabalho, serão mencionados alguns casos analisados que aparentavam ser promissores, no entanto, mostraram algumas dificuldades a mais que iremos lidar nos estágios seguintes deste projeto de pesquisa.

A análise atual revela a presença de vários *commits* no histórico do repositório do *"ZooKeeper"*, e notou-se uma tendência decrescente no valor de métricas, como o CBO. No entanto, observa-se que apenas uma parcela restrita desses *commits* inclui mensagens associadas a melhorias de desempenho, segurança, substituição de algoritmos ou melhorias de modo geral.

Lamentavelmente, até o momento, apenas um *commit* apresentou uma correlação direta entre as vulnerabilidades identificadas no banco de dados do *"CVE-Details"* e os *commits* no repositório do *ZooKeeper*. O único caso encontrado foi para o *commit* 9213f7353b1e6ce4d0fdbc1dca963ace1fd32cec, o qual relaciona-se a vulnerabilidade CVE-2021-21295. Entretando, por mais que o *commit* resolva um problema de vulnerabilidade, constatamos que esta refatoração específica consiste apenas na atualização da versão da biblioteca *Netty* no arquivo pom.xml do Java, o qual provê uma estrutura cliente-servidor de E/S não bloqueante, logo ela não se mostra uma boa candidata para correlacionar métricas de código com a segurança, pois se trata de uma ação direta para resolver uma vulnerabilidade conhecida.

A análise do *commit* 83cf0a93c37759334fab885c2010fa0b7d953f52 foi aprofundada, cuja mensagem é "*ZOOKEEPER-308. Improve the atomic broadcast desempenho in 3x*". Apesar da indicação inicial de um ganho de desempenho em 3 vezes, uma investigação mais detalhada revelou que a alteração se limitava à substituição da classe *FileOutputStream"* por *BufferedOutputStream"*. Embora essa mudança resulte em melhoria de desempenho, não constitui uma refatoração significativa, pois o *commit* altera 19 arquivos, sendo relevante apenas o ar-

quivo “*FileTxnLog.java*”, que consiste na troca das classes mencionadas. Os outros arquivos alterados abordam refatorações relacionadas à validação de objetos nulos antes de operações, mas estão fora do escopo da mensagem do *commit*. Apesar disso, a melhoria de desempenho é reconhecida como válida, embora sua magnitude não possa ser verificada.

Embora a mudança promova efetivamente uma melhoria de desempenho, as métricas tradicionais de código, como CBO, RFC e WMC, podem não refletir diretamente a magnitude do ganho de desempenho obtida. Tais métricas podem não capturar aspectos específicos relacionados à eficiência de operações de entrada e saída. Portanto, apesar da magnitude do aprimoramento não poder ser totalmente verificada por meio das métricas convencionais, é válida a melhoria de desempenho proporcionada pela otimização do uso da classe de E/S, mas não relevante para a criação de um exemplo trabalhado.

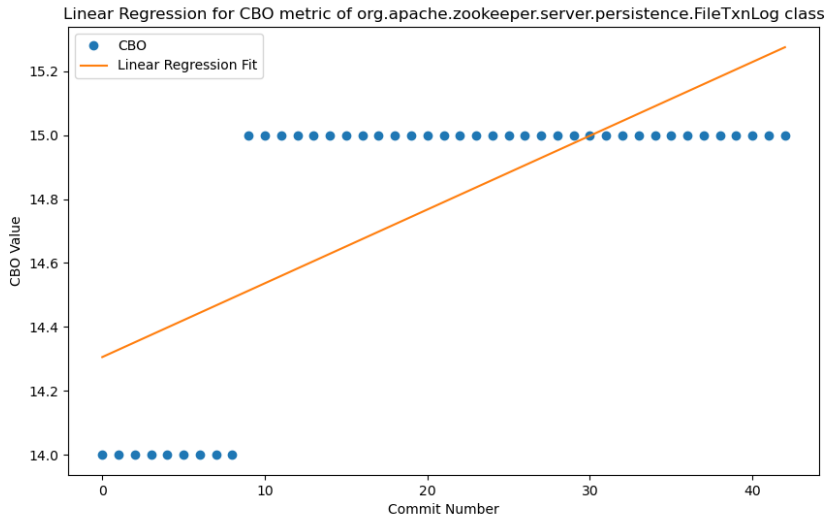

Figura 7 – CBO da classe *org.apache.zookeeper.server.persistence.FileTxnLog*.

A falta de uma refatoração mais substancial e a existência de uma melhoria de desempenho que não pôde ser validada tornam este *commit* inadequado para servir como objeto de exemplo trabalhado. Além disso, uma vez que a classe *“BufferedOutputStream”* está presente desde a versão inicial do Java (lançada em 1996) e o *commit* em questão é de 2009, ou seja, o uso de um *buffered input* devia ter sido feita desde o começo do projeto.

Observou-se que parte da complexidade na análise de um *commit* propriamente dito reside na existência de uma considerável poluição nos dados. Muitos *commits* no repositório do *“ZooKeeper”* abrangem alterações em inúmeros arquivos, frequentemente incorporando refatorações não diretamente correlacionadas às mensagens de *commit*. Essa complexidade aumenta a dificuldade de discernir as motivações subjacentes por trás das mudanças no código-fonte, tornando desafiador o processo de vincular essas alterações a possíveis vulnerabilidades identificadas externamente. Este fenômeno destaca a importância de uma abordagem criteriosa na interpretação do histórico de *commits* para extrair conclusões significativas sobre a evolução do *software* em relação à qualquer aspecto desejado.

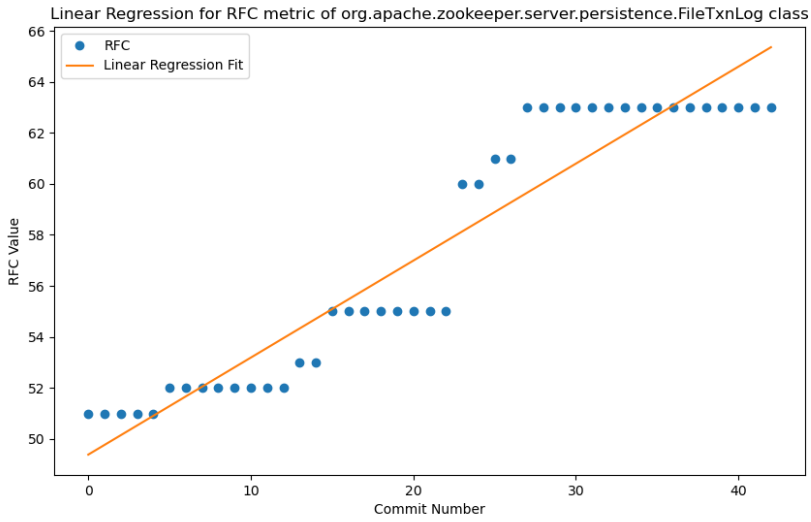

Figura 8 – RFC da classe *org.apache.zookeeper.server.persistence.FileTxnLog*.

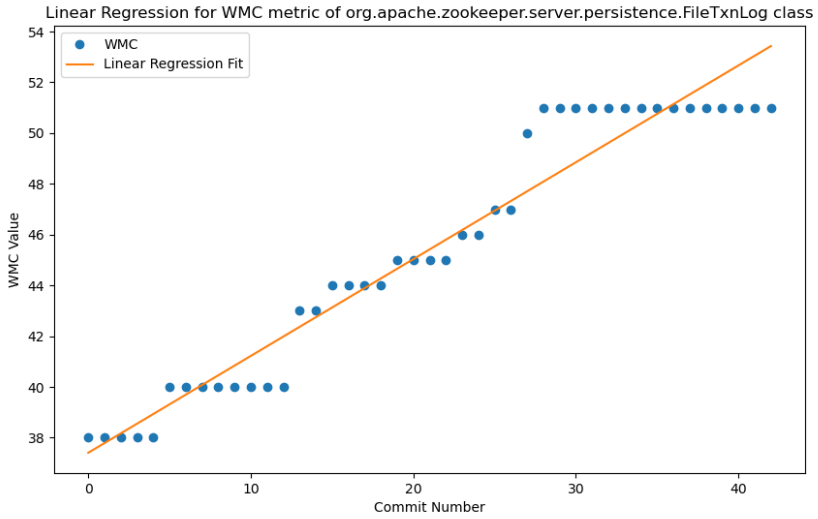

Figura 9 – WMC da classe *org.apache.zookeeper.server.persistence.FileTxnLog*.

De forma resumida, as Figuras 1, 2 e 3 evidenciam uma tendência inicial de aumento nos valores das métricas CBO, RFC e WMC para a classe *org.apache.zookeeper.server.quorum.Follower*. Contudo, na metade da representação gráfica, observa-se uma queda abrupta do valor das métricas. Surpreendentemente, a segunda metade do gráfico revela uma retomada da mesma tendência de crescimento verificada no início. A análise do racional subjacente a essa queda súbita permanece pendente e será abordada de maneira abrangente no âmbito do Trabalho de Conclusão de Curso 2 (TCC2).

4.5 Heurística

A heurística, ainda a ser concluída no TCC2, tem como finalidade auxiliar na criação de exemplos trabalhados para a ES por meio da análise da evolução do código nos repositórios selecionados, a qual baseou-se em uma abordagem sistemática que envolve a escolha cuidadosa dos projetos, a seleção de métricas apropriadas e o desenvolvimento de ferramentas específicas para a análise. Primeiramente, foram estabelecidos critérios para a seleção de repositórios por meio de fatores como, ser um SDs, desenvolvido em Java, *open-source*, ativamente mantido e utilizado em aplicações do mundo real. Os projetos escolhidos, *Apache Kafka* e *ZooKeeper*, destacaram-se por sua relevância no mercado de trabalho. Assim sendo, embora o cronograma delineado na Seção 3.3 revele um avanço em relação a determinadas tarefas, ressalta-se que a análise de códigos se destaca como a atividade mais exigente e onerosa em termos de complexidade e tempo.

4.6 Limitações

Este trabalho apresenta uma limitação, a qual reside na impossibilidade de realizar análises empíricas de certas melhorias de desempenho, como a análise das trocas de mensagens em tempo real, devido à restrição ao uso de métricas estáticas de código. Isso impede a avaliação prática por meio da execução de SDs e simulações em grande escala, representando uma restrição na obtenção de percepções detalhadas sobre o desempenho em contextos de uma quantidade massiva de requisições sendo enviadas ao SD. Além disso, poucos repositórios usam a convenção de *commits*, o que facilitaria obter refatorações significativas, não poluídas por refatorações irrelevantes(Conventional Commits, 2023).

5 CONCLUSÕES

Neste trabalho, o qual representa uma proposta de trabalho para o TCC2, foram desenvolvidas ferramentas e uma heurística em andamento para analisar a evolução de projetos de SDs por meio da avaliação de métricas de código. As métricas selecionadas inicialmente, como CBO, RFC, e WMC, foram escolhidas para proporcionar uma visão abrangente da qualidade do código, cobrindo diferentes aspectos. A métrica CBOM foi removida durante o desenvolvimento do estudo devido à percepção de que poderia introduzir poluição nos dados.

Para facilitar a análise da evolução do código, cinco ferramentas específicas foram desenvolvidas: *code_metrics.py*, *metrics_changes.py*, *track_files.py*, *metrics_evolution_refactorings.py* e *repository_refactorings.py*. Todas essas ferramentas foram implementadas em Python, utilizando o *PyDriller* e o *RefactoringMiner*, e estão disponíveis no Github[1].

A heurística em desenvolvimento visa oferecer uma abordagem sistemática e contextualizada para a análise da evolução do código em SDs, reconhecendo as complexidades e desafios inerentes a essa tarefa. No entanto, a conclusão definitiva dessa heurística está prevista para o TCC2.

Embora os resultados prévios não sejam tão animadores quanto o esperado, eles fornecem *insights* valiosos para a continuidade da pesquisa. A falta de padrão nos *commits* e o elevado número de refatorações por *commit*, muitas vezes não diretamente alinhadas à mensagem do *commit*, representam desafios significativos a serem abordados.

Este trabalho estabeleceu um referencial teórico abrangente, um cronograma bem definido e uma metodologia detalhada. As fases da pesquisa, desde a revisão da literatura até a avaliação da utilidade educacional dos exemplos escolhidos, foram claramente delineadas no cronograma, proporcionando uma abordagem sistemática e eficiente na realização dos objetivos propostos, além de expor completo conhecimento das limitações e desafios que este trabalho está imerso.

Em resumo, a proposta deste estudo busca preencher uma lacuna na literatura sobre a criação de exemplos trabalhados em ES, utilizando SDs como objeto de estudo. Ao contribuir para a formação de estudantes e profissionais em ES, espera-se que a compreensão mais profunda da evolução do código em SDs e os exemplos trabalhados pedagogicamente valiosos impactem positivamente o ensino e a prática profissional nesta área.

[1] https://github.com/BrenoFariasdaSilva/Worked-Example-Miner

REFERÊNCIAS

ABAD, C. L.; ORTIZ-HOLGUIN, E.; BOZA, E. F. Have we reached consensus? an analysis of distributed systems syllabi. *In*: **Proceedings of the 52nd ACM Technical Symposium on Computer Science Education**. New York, NY, USA: Association for Computing Machinery, 2021. (SIGCSE '21), p. 1082–1088. ISBN 9781450380621. Disponível em: https://doi.org/10.1145/3408877.3432409.

ANICHE, M. **Java code metrics calculator (CK)**. [*S.l.*], 2015. Available in https://github.com/mauricioaniche/ck/.

Apache Software Foundation. **Apache ZooKeeper GitHub Repository**. 2008. https://github.com/apache/zookeeper. Acessado em 09 de novembro de 2023.

Apache Software Foundation. **Apache Kafka GitHub Repository**. 2011. https://github.com/apache/kafka. Acessado em 09 de novembro de 2023.

ATKINSON, R. K. *et al.* Learning from examples: Instructional principles from the worked examples research. **Review of Educational Research**, SAGE, EUA, v. 70, n. 2, p. 181–214, jun. 2000. ISSN 0034-6543.

BONETTI, T. P. *et al.* Students' perception of example-based learning in software modeling education. *In*: **Proceedings of the XXXVII Brazilian Symposium on Software Engineering**. New York, NY, EUA: ACM, 2023. p. 67–76. ISBN 9798400707872.

BREWER, E. Cap twelve years later: How the "rules"have changed. **Computer**, v. 45, n. 2, p. 23–29, Feb 2012. ISSN 1558-0814.

CHIDAMBER, S.; KEMERER, C. A metrics suite for object oriented design. **IEEE Transactions on Software Engineering**, v. 20, n. 6, p. 476–493, 1994.

Conventional Commits. **Conventional Commits**. 2023. Acessado em 31 de outubro de 2023. Disponível em: https://www.conventionalcommits.org/en/v1.0.0/.

COULOURIS, G. *et al.* **Distributed Systems: Concepts and Design**. 5th. ed. USA: Addison-Wesley Publishing Company, 2011. ISBN 0132143011.

FU, X.; LIN, B.; CAI, H. Distfax: A toolkit for measuring interprocess communications and quality of distributed systems. *In*: **2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)**. [*S.l.*: *s.n.*], 2022. p. 51–55.

GOSLING, J. **Java Programming Language**. 1996. Acessado em 17 de novembro de 2023. Disponível em: https://www.java.com/en/.

HUNT, P. *et al.* Zookeeper: Wait-free coordination for internet-scale systems. *In*: **Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference**. USA: USENIX Association, 2010. (USENIXATC'10), p. 11.

JESSE, K.; KUHMUENCH, C.; SAWANT, A. Refactorscore: Evaluating refactor prone code. **IEEE Transactions on Software Engineering**, v. 49, n. 11, p. 5008–5026, 2023.

KAFKA, A. **Powered by**. 2023. Acessado em 3 de dezembro de 2023. Disponível em: https://kafka.apache.org/powered-by.

KREPS NEHA NARKHEDE, J. R. J. Apache kafka: Next generation distributed messaging system. **International Journal of Scientific and Engineering Research**, v. 3, n. 8, p. 1–4, 2010. Disponível em: https://ijsetr.com/uploads/436215IJSETR3636-621.pdf.

MULDNER, K.; JENNINGS, J.; CHIARELLI, V. A review of worked examples in programming activities. **ACM Trans. Comput. Educ.**, Association for Computing Machinery, New York, NY, USA, v. 23, n. 1, dec 2022. Disponível em: https://doi.org/10.1145/3560266.

PINTO, G. H. L. *et al.* Training software engineers using open-source software: The professors' perspective. *In*: **2017 IEEE 30th Conference on Software Engineering Education and Training**. [*S.l.*: *s.n.*], 2017. p. 117–121.

RAJ, R. K.; KUMAR, A. N. Toward computer science curricular guidelines 2023 (cs2023). **ACM Inroads**, Association for Computing Machinery, New York, NY, USA, v. 13, n. 4, p. 22–25, nov 2022. ISSN 2153-2184. Disponível em: https://doi.org/10.1145/3571092.

RITCHIE, D.; STROUSTRUP, B. **C/C++ Programming Languages**. 1972/1983. Acessado em 19 de janeiro de 2024. Disponível em: https://en.cppreference.com/.

ROSSUM, G. van. **Python Programming Language**. 1991. Acessado em 17 de novembro de 2023. Disponível em: https://www.python.org/.

SILVA, B. F. da. **Code Metrics Generator: Python Script for generating the code metrics using CK metrics and PyDriller**. 2023. https://github.com/BrenoFariasdaSilva/Worked-Example-Miner/blob/main/PyDriller/code_metrics.py. Acessado em 4 de dezembro de 2023.

SILVA, B. F. da. **Metrics Changes: Python Script for generating the metrics changes using PyDriller**. 2023. https://github.com/BrenoFariasdaSilva/Worked-Example-Miner/blob/main/PyDriller/metrics_changes.py. Acessado em 4 de dezembro de 2023.

SILVA, B. F. da. **Metrics Evolution Refactorings: Python script for tracking the metrics evolution refactorings using Refactoring Miner**. 2023. https://github.com/BrenoFariasdaSilva/Worked-Example-Miner/blob/main/RefactoringMiner/metrics_evolution_refactorings.py. Acessado em 4 de dezembro de 2023.

SILVA, B. F. da. **Repository Refactorings Tracker: Python script for tracking the metrics evolution of refactorings using Refactoring Miner**. 2023. https://github.com/BrenoFariasdaSilva/Worked-Example-Miner/blob/main/RefactoringMiner/repository_refactorings.py. Acessado em 4 de dezembro de 2023.

SILVA, B. F. da. **Track Files: Python script for tracking files changes**. 2023. https://github.com/BrenoFariasdaSilva/Worked-Example-Miner/blob/main/PyDriller/Scripts/track_files.py. Acessado em 4 de dezembro de 2023.

SKUDDER, B.; LUXTON-REILLY, A. Worked examples in computer science. *In*: **Proceedings of the Sixteenth Australasian Computing Education Conference**. Darlinghurst, Austrália: Australian Computer Society, 2014. v. 148, p. 59–64. ISBN 978-1-921770-31-9.

SPADINI, D.; ANICHE, M.; BACCHELLI, A. PyDriller: Python framework for mining software repositories. *In*: **Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018**. New York, New York, USA: ACM Press, 2018. p. 908–911. ISBN 9781450355735. Disponível em: http://dl.acm.org/citation.cfm?doid=3236024.3264598.

STEEN, M. van; TANENBAUM, A. S. A brief introduction to distributed systems. **Computing**, v. 98, n. 10, p. 967–1009, Oct 2016. ISSN 1436-5057. Disponível em: https://doi.org/10.1007/s00607-016-0508-7.

TONHÃO, S.; COLANZI, T.; STEINMACHER, I. Using real worked examples to aid software engineering teaching. *In*: RIBEIRO, M. *et al.* (Ed.). **35th Brazilian Symposium on Software Engineering (SBES 2021)**. New York, NY, EUA: ACM, 2021. p. 133–142. ISBN 978-1-4503-9061-3.

TONHÃO, S. *et al.* Uma plataforma gamificada de desafios baseados em worked examples extraídos de projetos de software livre para o ensino de engenharia de software. *In*: **XVII Simpósio Brasileiro de Sistemas Colaborativos**. Porto Alegre, RS, Brasil: SBC, 2022. p. 33–38.

TSANTALIS, N. *et al.* Accurate and efficient refactoring detection in commit history. *In*: **Proceedings of the 40th International Conference on Software Engineering**. New York, NY, USA: ACM, 2018. (ICSE '18), p. 483–494. ISBN 978-1-4503-5638-1. Disponível em: http://doi.acm.org/10.1145/3180155.3180206.

VINKA, E. **Zookeeper Atomic Broadcast Protocol (ZAB) and implementation of Zookeeper**. 2018. Acessado em 10 de novembro de 2023. Disponível em: https://www.cloudkarafka.com/blog/cloudkarafka-zab.html.