Abordagem para seleção de exemplos trabalhados para engenharia de software do domínio de sistemas distribuídos

Breno Farias da Silva[1], **Igor Scaliante Wiese**[1],
Rodrigo Campiolo[1], **Marco Aurélio Graciotto Silva**[1]

[1]Universidade Tecnológica Federal do Paraná (UTFPR)
Programa de Pós-Graduação em Ciência da Computação (PPGCC-CM)
Campo Mourão – PR – Brasil

brenofarias@alunos.utfpr.edu.br, {igor, rcampiolo, magsilva}@utfpr.edu.br

Abstract. *This study addresses the scarcity of examples of work in the field of Distributed Systems (SD) relevant to the teaching of Software Engineering (SE). The objective is to identify code examples from open-source projects that demonstrate significant improvements in code design quality and are suitable for teaching SD and ES. The method comprised the development of a heuristic for selecting examples, using code quality metrics (CK), refactoring analysis (RefactoringMiner) and evaluation by experts. The Worked-Example-Miner (WEM) tool automates data collection and analysis. By using heuristics, with the help of the WEM tool, in the Apache Kafka and ZooKeeper projects, it was possible to select candidate codes for worked examples for teaching SD and ES. Analysis of a specific case in Apache ZooKeeper showed significant improvements in quality metrics after refactoring, illustrating the applicability of the approach.*

Resumo. *Este artigo aborda a escassez de exemplos trabalhos do domínio de Sistemas Distribuídos (SD) relevantes para o ensino de Engenharia de Software (ES). O objetivo é identificar exemplos de código de projetos de software livre que demonstrem melhorias significativas na qualidade do design de código e sejam adequados para o ensino de SD e ES. O método compreendeu o desenvolvimento de uma heurística para seleção de exemplos, utilizando métricas de qualidade de código (CK), análise de refatorações (RefactoringMiner) e avaliação por especialistas. A ferramenta Worked-Example-Miner (WEM) automatiza a coleta e análise de dados. Ao usar a heurística, com auxílio da ferramenta WEM, nos projetos Apache Kafka e ZooKeeper (ZK), foi possível selecionar códigos candidatos a exemplos trabalhados para ensino de SD e ES. A análise de um caso específico no ZK mostrou melhorias significativas nas métricas de qualidade após refatoração, ilustrando a aplicabilidade da abordagem.*

1. Introdução

A formação em Engenharia de Software (ES) requer o aprendizado de diversas competências, integrando conhecimentos, habilidades e atitudes. Métodos de aprendizado baseados em problemas e em projetos, além do aprendizado realizado em contextos de trabalho [Barr et al. 2024], colaboram para esse aprendizado. No entanto, a aplicação desses métodos enfrenta barreiras, tais como a dificuldade de criar exemplos representativos do mundo real, que sejam acessíveis em relação à complexidade dos conteúdos, demandando estratégias pedagógicas para sua superação [Tonhão et al. 2023].

Exemplos trabalhados apresentam-se como uma forma de facilitar o escalonamento da dificuldade do aprendizado de técnicas [Atkinson et al. 2000]. Um exemplo trabalhado é definido como um trabalho cognitivo e experimental com o intuito de fornecer uma solução para um problema específico, no qual uma pessoa possa examinar e aprender com a solução proposta [Atkinson et al. 2000]. Assim, exemplos trabalhados facilitam a aprendizagem, promovendo engajamento, retenção de conhecimento e redução da carga cognitiva [Atkinson et al. 2000, Tonhão et al. 2023, Tonhão et al. 2022]. Eles são amplamente aplicados em conceitos de programação e ES [Skudder e Luxton-Reilly 2014, Muldner et al. 2022], aproximando o aprendizado da prática industrial.

Embora o uso de exemplos trabalhados seja comum na ES, exemplos trabalhados são menos comuns em Sistemas Distribuídos (SD). A complexidade de SD, envolvendo coordenação, sincronização e tolerância a falhas, dificulta sua demonstração, além da rápida evolução tecnológica que pode tornar os exemplos obsoletos. Ademais, também é necessário que software do domínio de SD tenha qualidade na perspectiva de ES, facilitando a manutenção corretiva e evolutiva das técnicas complexas implementadas.

Uma barreira quanto ao uso de exemplos trabalhados sobre SD é a seleção de programas adequados para uso como exemplo. De modo geral, a criação de exemplos trabalhados exige a seleção de artefatos que promovam boas práticas e qualidade de software. No entanto, é difícil Métricas que avaliam coesão e acoplamento, como as métricas CK [Chidamber e Kemerer 1994], associadas a limiares [Filó et al. 2024] e princípios de *design* como SOLID [Gamma et al. 1994, Martin 2003, Martin 2008], são úteis e convenientes para avaliar a qualidade de software. Ferramentas como o *RefactoringMiner* [Tsantalis et al. 2018] identificam refatorações alinhadas a esses princípios, evidenciando melhorias no código. Estudos mostram que SOLID impacta positivamente a qualidade de software, com aplicações até em áreas como Aprendizagem de Máquina [Cabral et al. 2024]. No entanto, não se observam estudos similares quanto a SD.

Assim, este trabalho explora a seleção de exemplos trabalhados para uso em atividades de aprendizagem em SD. O objetivo é identificar exemplos de código de projetos de software livre que demonstrem melhorias significativas na qualidade do *design* de código e que possam ser utilizados para ensinar conceitos de SD por meio de exemplos trabalhados. A seleção considerou o alinhamento com currículos sobre SD [NSF/IEEE-TCPP Curriculum Initiative 2023, Raj e Kumar 2022] e a qualidade da implementação de algoritmos relacionados ao tema. Com isso, buscou-se contribuir para a melhoria das práticas de ensino e a preparação de estudantes para enfrentarem os desafios tecnológicos atuais.

As principais contribuições deste trabalho são: (i) uma heurística para selecionar exemplos trabalhados, identificando códigos com melhorias significativas no *design* de software; (ii) a ferramenta *Worked-Example-Miner* (WEM) [da Silva 2023], projetada para automatizar a coleta de dados relevantes à seleção de exemplos com auxílio da heurística; (iii) um conjunto de exemplos de código sobre SD, obtidos pela aplicação da heurística, demonstrando sua eficácia na identificação de casos que potencializam o ensino de práticas de desenvolvimento;

Este artigo é organizado em sete seções. A Seção 2 introduz a base teórica, enquanto a Seção 3 descreve a metodologia empregada, incluindo objetivos, questões de pesquisa, e o processo de seleção e análise de código. A Seção 4 mostra os resultados,

discutidos na Seção 5 em relação às questões de pesquisa. A Seção 6 discute as limitações do estudo, com relação à avaliação de candidatos pelos especialistas. A Seção 7 resume os principais achados e explora limitações e futuras direções de pesquisa.

2. Referencial Teórico

Esta seção apresenta os conceitos fundamentais estudados em SD e discute o uso de exemplos trabalhados, incluindo estudos relevantes sobre sua aplicação na educação e na ES.

2.1. Educação em Sistemas Distribuídos

A educação em SD faz parte do núcleo de cursos de graduação e pós-graduação em muitas universidades no mundo. Estes cursos são projetados para fornecer uma base sólida em teoria e prática, preparando os estudantes para enfrentar os desafios contemporâneos de desenvolvimento e manutenção de software em ambientes distribuídos [Abad et al. 2021].

Geralmente, os cursos de SD abordam uma gama de tópicos que incluem, mas não se limitam a, comunicação entre processos, algoritmos de consenso, escalabilidade, tolerância a falhas e segurança em SD [Abad et al. 2021]. Uma análise de currículos de cursos ofertados nas 100 melhores universidades do ranking *Times Higher Education World University* em Ciência da Computação identificou tópicos-chave frequentemente lecionados em SD, como Processos, Replicação, Chamadas de Sistema, Controle de Concorrência e Sincronização [Abad et al. 2021]. Isso também se observa nos currículos do NSF/IEEE-TCPP e do ACM/IEEE-CS *Computing Curricula* 2023 que, ao abordar arquiteturas, modelos de programação, algoritmos distribuídos e questões de segurança em SD, destacam a importância crescente desses tópicos na formação de profissionais da computação [NSF/IEEE-TCPP 2024, ACM 2023].

A inclusão de SD nos currículos de Computação também apresenta desafios, especialmente no que diz respeito à necessidade de manter o conteúdo atualizado com as rápidas mudanças tecnológicas e garantir que os estudantes adquiram experiências práticas relevantes, dada a complexidade dos SD. Ao dominar os conceitos e as tecnologias associadas a SD, os estudantes estão bem preparados para carreiras em desenvolvimento de software e muitas outras áreas críticas onde SD desempenham um papel central.

No entanto, embora estudos tenham integrado SD em disciplinas introdutórias de computação, como CS1 e CS2 [McQuaigue et al. 2023, Ghafoor et al. 2023], não se observam abordagens para integração de SD com ensino de ES. A complexidade inerente a SD fornece uma rica fonte de casos de estudo para ensino de ES, aprofundando a compreensão dos estudantes e os preparando para desafios reais no desenvolvimento de software, favorecendo a criação de soluções mais robustas [Abad et al. 2021].

2.2. Exemplos Trabalhados

Um exemplo trabalhado é 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 [Atkinson et al. 2000]. Sendo assim, o desenvolvimento e estudo de exemplos trabalhados visa enriquecer a qualidade do que é lecionado em sala de aula, uma vez que a solução ideal pode representar o estado da arte de um tópico, disseminando conceitos e padrões do problema apresentado [Atkinson et al. 2000].

Dentro do contexto de ES, diversos estudos exploram o uso de exemplos trabalhados para melhorar o ensino e a aprendizagem. Por exemplo, ao investigar a percepção de estudantes sobre a Aprendizagem Baseada em Exemplos (ABE) no ensino de modelagem de software, observou-se que essa abordagem melhora a compreensão, motivação e conexão entre teoria e prática [Bonetti et al. 2023]. Em outra pesquisa, ao analisar os benefícios dos exemplos trabalhados na ES, verificou-se o aumento da motivação e confiança dos estudantes, além da promoção da autonomia [Tonhão et al. 2023].

Em outra pesquisa, ao analisar os benefícios dos exemplos trabalhados no ensino de ES, destacaram-se dificuldades enfrentadas pelos docentes em encontrar exemplos reais, completos e pedagogicamente relevantes [Tonhão et al. 2023]. A pesquisa, que considerou exemplos extraídos de projetos de software livre, mostrou que esses exemplos aumentam a motivação e confiança dos estudantes, promovendo autonomia e preparação para o mercado. Em continuidade, adotou-se a metodologia *Design Science Research* (DSR) para criar e avaliar uma plataforma gamificada que integra exemplos trabalhados em ambientes assíncronos [Tonhão et al. 2022]. Os resultados evidenciaram que tais exemplos enriquecem o aprendizado de conceitos complexos, aumentando engajamento e motivação, especialmente quando associados a casos reais de software livre.

Esses estudos sublinham a eficácia dos exemplos trabalhados no ensino de ES, enfatizando como esses exemplos podem aumentar o engajamento dos estudantes e melhorar a retenção de conhecimento. No entanto, a criação e implementação desses exemplos enfrentam desafios significativos. As dificuldades incluem a complexidade na seleção de exemplos que sejam simultaneamente representativos dos conceitos abordados e compreensíveis para os estudantes. Além disso, garantir a atualidade e relevância dos exemplos frente às rápidas mudanças na tecnologia e nas práticas de desenvolvimento de software é crucial para seu sucesso pedagógico [Tonhão et al. 2023, Tonhão et al. 2022].

3. Método

Objetivou-se nesta pesquisa identificar exemplos de código que mostrem melhorias significativas com relação à qualidade do *design* do código e que possam ser utilizados para ensinar conceitos de SD por meio de exemplos trabalhados. Para atingir este objetivo, o estudo está estruturado em torno de três questões:

- Q1: O exemplo de código-fonte apresenta melhorias na qualidade de *design*?
- Q2: O exemplo é relevante para o ensino de SD?
- Q3: O exemplo é didaticamente adequado para o ensino de SD?

As respostas foram obtidas analisando projetos de software livre, focando em refatorações que melhoraram a qualidade do código sem alterar a funcionalidade. A qualidade foi mensurada por métricas extraídas de cada *commit*, permitindo avaliar a evolução de métricas associadas à qualidade de software.

3.1. Métricas e Critérios Utilizados

Para responder a essas questões, foram selecionadas várias métricas e critérios, aplicados com o intuito de avaliar a qualidade e a relevância dos exemplos de código. A seguir são listadas as métricas e as questões de pesquisa a que elas estão relacionadas:

- **Métrica McCabe** (Q1): Avalia a complexidade ciclomática do código. Este indicador permite verificar melhorias qualitativas que facilitam o ensino de conceitos de *design* eficiente e gestão de complexidade.
- **Métricas CK** (Acoplamento e Coesão) (Q1): Essas métricas buscam entender como componentes do código interagem entre si e como funções e classes são internamente coesas. Melhorias nessas métricas após revisões ou refatorações são evidências de que o código evoluiu para uma estrutura mais robusta e manutenível, ressaltando práticas de *design* sólidas úteis para o ensino de ES e SD.
- **Quantidade de Refatorações** (Q1, Q3): A detecção de refatorações significativas é um indicativo de esforços conscientes para melhorar o *design* do código. Tais códigos podem ser usados didaticamente para mostrar como problemas comuns de *design* podem ser resolvidos de maneira eficaz.
- **Termos e Textos Relacionados a SD** (Q2, Q3): A presença de termos específicos de SD nos comentários do código, na documentação, e nos nomes de métodos e classes destaca a relevância temática do exemplo, facilitando a correlação entre o contexto do código e o currículo de SD. Esta medida é obtida por meio do reaproveitamento da estrutura de visita de nós do código implementados na ferramenta CK, extraindo os termos do código, gerando a métrica *MethodInvocationCounter*. Esta métrica captura a frequência e a ordem de chamadas de método em cada método de uma classe. Para este estudo, essas informações são utilizadas para gerar uma espécie de descrição do método, a qual é utilizada para relacionar com os conceitos previstos nos currículos.
- **Avaliação de Especialistas** (Q2, Q3): A avaliação dos especialistas foi realizada por meio de questionários desenvolvidos pelos autores, com perguntas adaptadas conforme a área de especialização (ES ou SD). A maioria das questões utilizou a escala Likert, com algumas questões de resposta aberta. Para especialistas em ES, as questões abordaram a legibilidade, boas práticas, reutilização, testabilidade, documentação e adequação didática do código. Para especialistas em SD, as questões focaram na relevância do código para o ensino de conceitos de SD e sua adaptação para o ensino prático. Essa avaliação forneceu uma validação externa da relevância e utilidade didática dos exemplos de código.

Para avaliar a qualidade e relevância dos exemplos de código de SD na perspectiva da ES, foram adotadas diversas métricas e critérios. A métrica McCabe foi aplicada para avaliar a complexidade ciclomática, destacando exemplos que mostram redução após refatorações, simplificando o entendimento e a manutenção do código. As métricas CK foram utilizadas para entender a interação entre componentes do código e a coesão interna de funções e classes.

As métricas CK fornecem uma abordagem quantitativa para avaliar a qualidade de *design* em sistemas orientados a objetos [Chidamber e Kemerer 1994]. Neste estudo, utilizamos as métricas CK: CBO (acoplamento entre classes), DIT (profundidade da árvore de herança), LCOM (falta de coesão dos métodos), NOC (número de filhos), RFC (respostas de métodos de uma classe) e WMC (complexidade dos métodos) [Chidamber e Kemerer 1994]. Essas métricas permitem analisar diferentes aspectos do código, como coesão, acoplamento e complexidade, sendo que altos valores de CBO e WMC estão associados a maior complexidade e propensão a defeitos [Chidamber e Kemerer 1994, Subramanyam e Krishnan 2003]. Além disso, valores elevados de DIT indicam hierarquias de herança

profundas, o que pode dificultar a manutenção, enquanto altos valores de LCOM sinalizam baixa coesão entre métodos, sugerindo possíveis oportunidades de refatoração. Para a interpretação dessas métricas, adotamos limiares sugeridos por um estudo recente [Filó et al. 2024]. Os valores recomendados e que foram utilizados são: para DIT, até 2 é excelente, entre 2 e 4 regular, e acima de 4 indesejável; para LCOM, até 0,167 é ideal, entre 0,167 e 0,725 regular, e acima de 0,725 indica baixa coesão; para WMC, até 11 é ideal, entre 11 e 34 regular, e acima de 34 indica alta complexidade.

Os padrões de refatoração identificados pela ferramenta *RefactoringMiner* [Tsantalis 2018], como *Extract Method*, *Extract Class*, *Pull Up Method*, *Push Down Method* e *Extract Superclass*, são relevantes por sua contribuição à modularização, reutilização e manutenção do software. Esses padrões impactam nas métricas de *design* do código, como as Métricas CK. Por exemplo, refatorações como *Extract Method* reduzem a complexidade (WMC), enquanto *Extract Class* e *Extract Superclass* diminuem o acoplamento (CBO) ao redistribuir responsabilidades. Já *Pull Up Method* e *Push Down Method* otimizam a hierarquia de classes, melhorando o RFC por meio de uma herança mais eficiente.

No contexto de SD, a identificação de padrões de refatoração que aprimoram componentes e otimizam operações é relevante para aumentar a eficácia e a manutenibilidade do sistema. Além disso, a análise dos termos do código por meio da métrica *MethodInvocationCounter*, quando combinada com currículos de SD no Gemini, facilita a filtragem de candidatos relevantes para o ensino dessa área. Embora o Gemini seja um modelo não determinístico e apresente variações em suas respostas, a avaliação de especialistas pós-execução fortalece a validação da heurística, conferindo maior robustez e a validando.

3.2. Heurística de Seleção de Candidatos para Criação de Exemplos Trabalhados em ES de SD

Os códigos candidatos à utilização como exemplos trabalhados passaram por uma processo de seleção de várias etapas, visando garantir sua relevância educacional e técnica. Para facilitar a aplicação dessa heurística, foi desenvolvida a ferramenta *Worked-Example-Miner* (*WEM*), que automatiza processos como a coleta de dados dos projetos analisados, a geração de gráficos e arquivos CSV, permitindo análises mais precisas e eficientes. Assim, as etapas realizadas com o auxílio da ferramenta foram as seguintes:

- **Análise das métricas** (Q1): Consiste na análise detalhada das métricas de qualidade do código geradas pelo CK desde o início do projeto até o momento em que uma melhoria substancial nas métricas é identificada.
- **Padrão de refatoração** (Q1): Após a identificação de uma melhoria significativa nas métricas, realiza-se a análise do padrão de refatoração detectado pela ferramenta *RefactoringMiner*. Essa análise considera o tipo e a extensão da refatoração ocorrida, avaliando seu impacto no *design* do código e sua relevância para a ES.
- **Análise de código-fonte** (Q2 e Q3): A análise investiga os arquivos *.diff* que documentam a refatoração, avaliando a relevância das mudanças com base nas métricas antes e após a refatoração, o padrão de refatoração adotado, a adequação dos nomes de classes e métodos aos conceitos de SD, e os impactos no sistema.
- **Análise das refatorações entre commits** (Q1 e Q2): Para cada projeto selecionado, as métricas são extraídas e analisadas, aplicando-se um filtro para identificar refatorações que apresentaram redução nas métricas entre os *commits*. Além

disso, são considerados padrões de refatoração que indicam melhorias na qualidade do *design* do código. Também é realizada uma análise dos termos presentes no código para correlacioná-los com conceitos relevantes para o ensino de SD.

- **Avaliação por especialistas** (Q2 e Q3): Um especialista em SD e outro em ES revisam a análise, confirmando a relevância das refatorações tanto do ponto de vista técnico quanto educacional. O especialista em SD avalia se as mudanças são didaticamente apropriadas para serem utilizadas como exemplos no ensino da área, garantindo que os exemplos ilustrem conceitos teóricos e apresentem aplicações práticas para desafios típicos de SD.

Dessa forma, a heurística proposta e a ferramenta desenvolvida auxiliam no processo de seleção, contribuindo para a identificação de códigos candidatos ao uso em exemplos trabalhados, promovendo a compreensão técnica dos estudantes e incentivando a aplicação prática desses conhecimentos em cenários reais de desenvolvimento de SD.

3.3. Processo de Seleção

O processo de seleção dos exemplos de código para uso didático em SD adotou uma abordagem iterativa. Inicialmente, foram coletadas as métricas CK de projetos de software livre amplamente utilizados na indústria para o desenvolvimento de aplicações distribuídas. A partir dessas métricas, foram gerados dados e metadados das classes e métodos de cada repositório, criando um histórico unificado das métricas em cada *commit*, incluindo regressões lineares e a detecção de quedas nas métricas ao longo do tempo. Em seguida, a lista de candidatos com quedas nas métricas foi encaminhada ao Gemini, juntamente com os termos extraídos dos códigos desses *commits*, com o intuito de associar esses termos a tópicos relevantes de SD, determinando sua pertinência para o ensino da área. Por fim, os candidatos considerados relevantes pelo Gemini foram avaliados por especialistas, resultando na lista final de exemplos adequados para a criação de exemplos trabalhados.

Cada exemplo de código identificado foi revisado por um especialista em SD para garantir sua relevância didática. Em tese, o desejável é que a seleção automatizada fosse eficaz na identificação de bons exemplos, sendo que a análise por especialistas também serve para avaliar o desempenho da heurística proposta. Esse processo de revisão não se limita a verificar a adequação técnica, mas também examina a capacidade dos exemplos de ilustrar os conceitos de *design* e manutenção de software em ambientes distribuídos.

Este processo iterativo de seleção e revisão assegura que os exemplos finais escolhidos para ensino sejam não apenas tecnicamente sólidos, mas também relevantes e úteis para estudantes que buscam entender os desafios e soluções na ES e SD.

4. Resultados

A aplicação do método proposto foi realizada utilizando a ferramenta *WEM* [da Silva 2023], que integra as ferramentas mencionadas no método. A ferramenta aplica a heurística descrita anteriormente e foi utilizada em projetos de software livre da área de SD e implementados em Java, restrição essa de linguagem devido às ferramentas utilizadas. A seguir, apresentamos a seleção dos projetos e os resultados conforme as questões de pesquisa.

Para este estudo foram selecionados os projetos *Apache Kafka* [ASF 2011] e *Apache ZooKeeper* [ASF 2008], com base em recomendações de um especialista em SD,

devido à ampla aplicação prática, complexidade técnica e histórico de manutenção ativa. O *Apache Kafka* é um sistema de mensagens distribuído projetado para lidar com grandes volumes de dados em tempo real enquanto o *Apache ZooKeeper* atua como um serviço centralizado de coordenação, garantindo consistência via *Two-Phase Commit Protocol*.

Além dos dois repositórios analisados neste trabalho, a ferramenta WEM [da Silva 2023] foi empregada para processar outros repositórios que atendem aos critérios estabelecidos. Especificamente, foram considerados repositórios Java de código aberto hospedados no GitHub, organizados sob o tópico *distributed-systems*, com no mínimo 50 estrelas e que foram atualizados nos últimos 180 dias. Os candidatos que foram avaliados neste estudo, bem como demais candidatos extraídos dos outros projetos, estão disponíveis no repositório *Worked-Example-Miner Candidates* [da Silva 2024].

4.1. Q1: O exemplo de código-fonte apresenta melhorias na qualidade de *design*?

A lista de candidatos gerados pelo WEM contém vários padrões de refatoração identificados. No entanto, não há informações concretas sobre o percentual de padrões de refatoração desejados em relação ao total de padrões detectados. Ainda assim, nos repositórios analisados mais detalhadamente (*Kafka* e *ZooKeeper*), foram encontrados alguns casos em que padrões de refatoração desejados foram detectados. Dentre esses, destaca-se a classe *org.apache.zookeeper.server.quorum.Follower*, do *ZooKeeper*, na qual foi identificada uma refatoração do tipo *Extract Superclass*.

Essa refatoração resultou na criação da classe *Learner*, que centralizou funcionalidades comuns, reduzindo duplicações no código. Em particular, o commit *ZOOKEEPER-549* reestruturou a hierarquia de classes, tornando-a mais robusta e aderente aos princípios de *design* de software.

4.2. Q2: O exemplo é relevante para o ensino de SD?

A relevância educacional dos exemplos foi verificada por meio da análise dos termos presentes nos códigos e da integração com o modelo de IA generativa do Google Gemini [Google 2023]. No mesmo contexto da classe *org.apache.zookeeper.server.quorum.Follower*, foram identificados termos que refletem conceitos relevantes para o ensino de SD. Termos como *followLeader*, *getZxid*, *readPacket* e *writePacket* destacam a interação do seguidor com o líder, essencial para manter a sincronização entre os nós em um SD. Operações como *truncateLog*, *commit* e *notifyAll* reforçam conceitos de coordenação e consistência de estado, enquanto funções como *validateSession* e *loadData* abordam a integridade das conexões e dados. Esses termos demonstram práticas centrais em SD, tornando a classe um exemplo didático relevante para o ensino de replicação e coordenação em SD.

4.3. Q3: O exemplo é didaticamente adequado para o ensino de SD?

A refatoração identificada na classe *org.apache.zookeeper.server.quorum.Follower*, embora relevante do ponto de vista de ES e SD, apresenta desafios para sua adequação didática em disciplinas introdutórias de SD. A refatoração em questão abrange múltiplos arquivos e inclui alterações em classes complexas, o que pode dificultar a compreensão para estudantes iniciantes. No entanto, os padrões de refatoração detectados, como a reorganização de métodos relacionados à sincronização de nós e à validação de sessões, estão intrinsecamente ligados a problemas centrais de SD, como consistência, replicação,

tolerância a falhas, e coordenação entre componentes distribuídos. Embora a complexidade técnica possa limitar sua aplicabilidade em disciplinas introdutórias, o exemplo oferece um contexto para discussões avançadas sobre coordenação e otimização de operações em SD, embora tenha sido considerado pelo especialista em SD como não plenamente adequado do ponto de vista didático.

5. Discussão dos Resultados

Esta seção apresenta uma análise detalhada dos resultados obtidos a partir da refatoração aplicada à classe *Follower* no contexto do *Apache ZooKeeper*, destacando os impactos positivos nas métricas de qualidade de código, os princípios de *design* aplicados e as implicações didáticas para o ensino de ES e SD.

A análise manual e exploratória dos resultados gerados pela ferramenta proposta visa compreender a evolução da qualidade e identificar candidatos a exemplos trabalhados. No caso do *ZooKeeper*, a análise da classe *org.apache.zookeeper.server.quorum.Follower* revelou tendências decrescentes nas métricas CBO, RFC e WMC, indicando melhorias em modularidade e coesão. Essa classe é essencial para o mecanismo de consenso, em que *followers* participam de votações em um *quorum*.

Refatorações como *Extract Superclass*, *Pull Up Method* e *Pull Up Attribute*, identificadas pelo *RefactoringMiner*, destacam-se por promover modularização, reutilização e manutenção. A refatoração *Extract Superclass* resultou na criação da classe *Learner*, centralizando funcionalidades comuns e reduzindo duplicações. Especificamente, o commit *ZOOKEEPER-549* reestruturou a hierarquia das classes, tornando-as mais robustas e alinhadas a princípios de *design* de software. Métodos como *shutdown()* foram movidos para *Learner* com o *Pull Up Method*, garantindo consistência na gestão de recursos entre os *learners*. O *Pull Up Attribute*, por sua vez, centralizou atributos como *lastQueued*, assegurando um controle de estado unificado e reforçando a integridade do sistema em operações distribuídas. Essas práticas evidenciam como refatorações bem aplicadas aprimoram a arquitetura e a manutenção de SD complexos.

Assim, observa-se que a refatoração da classe *Follower* impactou diversas métricas de qualidade, evidenciando melhorias significativas no *design*. A métrica CBO foi reduzida de 21 para 13, indicando um desacoplamento efetivo da classe, facilitado pela introdução da classe base *Learner*, que centralizou a comunicação. A métrica WMC caiu de 45 para 18, demonstrando uma simplificação da complexidade interna da classe *Follower*, graças à realocação de parte da lógica para *Learner*. Já a métrica RFC apresentou uma redução de 79 para 35, sinalizando uma diminuição na quantidade de métodos acessíveis, refletindo uma melhor definição de responsabilidades na hierarquia de classes.

A refatoração da classe *Follower* ilustra a aplicação prática de teoria em SD reais, destacando a importância de um *design* bem planejado na construção de sistemas robustos e escaláveis. Esse caso aborda conceitos-chave como comunicação eficiente entre processos, a fim de impedir falhas como *deadlocks* e *race conditions*; escalabilidade, promovida pela modularidade e baixo acoplamento, que facilita a expansão do sistema sem alterações significativas no código; e tolerância a falhas, onde a modularidade reduz o impacto de falhas individuais, permitindo estratégias eficazes de redundância e recuperação.

Do ponto de vista de ES, a refatoração demonstra como princípios de *design* orientado a objetos são aplicados para melhorar a manutenibilidade, reusabilidade e testabili-

dade de software. Esta refatoração oferece aos estudantes uma visão de como refatorações podem resultar em melhorias tangíveis em projetos de software de grande escala.

6. Ameaças à Validade

As métricas empregadas na Seção 3 são de natureza estática, restringindo-se à análise do código-fonte sem considerar seu comportamento em tempo de execução. A incorporação de métricas dinâmicas poderia mitigar essa limitação, possibilitando uma avaliação mais abrangente e precisa da qualidade do software.

A avaliação dos trechos de código envolveu apenas especialistas locais em ES e SD, o que pode ter introduzido viés. Para mitigar isso, futuras análises poderiam incluir docentes de diferentes instituições que publicam nos principais eventos e revistas da área de Educação em Computação e que atuem em ao menos duas dessas áreas.

Atualmente foram considerados apenas dois projetos de software, o que limita a validade dos resultados. Assim, é necessário considerar um conjunto maior de projetos, permitindo a generalização dos resultados. Quanto a essa limitação, estão sendo analisados mais projetos, viabilizando a continuação deste estudo e análise de mais códigos candidatos para uso em exemplos trabalhados.

7. Conclusão

Neste trabalho desenvolveu-se uma abordagem para identificar exemplos de códigos de software apropriados para o ensino integrado de SD e ES. Para isso, foi definida uma heurística para seleção de código, com base em medidas sobre a qualidade de design e alinhamento com conceitos de SD. A heurística foi aplicada com o auxílio da ferramenta WEM, também desenvolvida neste trabalho, em dois softwares, permitindo a identificação de *commits* com códigos candidatos para utilização em exemplos trabalhados.

De modo geral, esses códigos refletem melhorias na qualidade do *design* e no alinhamento com conceitos de SD. O uso de código refatorado em contextos educacionais permite que os estudantes compreendam melhor os impactos das decisões de *design* em sistemas reais, tornando o ensino mais próximo da prática profissional.

Com base nos resultados, a abordagem proposta parece promissora para a seleção de exemplos trabalhados em SD. No entanto, os dados sugerem que a análise no nível de *commits* pode ser insuficiente. Portanto, recomenda-se examinar as refatorações no nível de *pull requests*, pois isso incluiria o escopo de modificações necessárias para melhorias e correções no código ou implementação de novas funcionalidades.

Como trabalhos futuros, considera-se a integração de métricas considerando aspectos dinâmicos do software e a aplicação da metodologia em uma variedade maior de repositórios de código aberto. Ao combinar métricas estáticas e dinâmicas, será possível obter uma perspectiva mais abrangente e detalhada sobre a qualidade do software, combinando percepções sobre sua estrutura e seu comportamento em condições de uso.

Agradecimentos

O presente trabalho foi realizado com apoio do Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) (Projeto 408812/2021-4), da Fundação Araucária - Governo do Estado do Paraná (Projeto PRD2023361000043) e da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Código de Financiamento 001.

Referências

Abad, C. L., Ortiz-Holguin, E., e Boza, E. F. (2021). Have we reached consensus? an analysis of distributed systems syllabi. In *52nd ACM Technical Symposium on Computer Science Education*, p. 1082–1088, New York, NY, USA. ACM.

ACM (2023). CS2023: ACM/IEEE-CS computing curricula 2023. Disponível em: `https://csed.acm.org/`. Acessado em 21 agosto 2024.

ASF (2008). Zookeeper. Disponível em: `https://zookeeper.apache.org/`. Acessado em 19 setembro 2023.

ASF (2011). Kafka. Disponível em: `https://kafka.apache.org/`. Acessado em 19 setembro 2023.

Atkinson, R. K., Derry, S. J., Renkl, A., e Wortham, D. (2000). Learning from examples: Instructional principles from the worked examples research. *Review of Educational Research*, 70(2):181–214.

Barr, M., Andrei, O., Morrison, A., e Nabi, S. W. (2024). The development of students' professional competencies on a work-based software engineering program. In *55th ACM Technical Symposium on Computer Science Education*, p. 81–87, New York, NY, EUA. ACM.

Bonetti, T. P., Dias, M. M., Silva, W., e Colanzi, T. E. (2023). Students' perception of example-based learning in software modeling education. In *XXXVII Brazilian Symposium on Software Engineering (SBES 2023)*, p. 67–76, New York, NY, EUA. ACM.

Cabral, R., Kalinowski, M., Baldassarre, M. T., Villamizar, H., Escovedo, T., e Lopes, H. (2024). Investigating the impact of SOLID design principles on machine learning code understanding. In *3rd International Conference on AI Engineering – Software Engineering for AI*, p. 1–11. ACM.

Chidamber, S. e Kemerer, C. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493.

da Silva, B. F. (2023). Worked-example-miner (wem): A comprehensive tool for analyzing java repositories. Software. Acessado em 4 de dezembro de 2023.

da Silva, B. F. (2024). Worked-example-miner-candidates: Repositório de candidatos para análise e construção de exemplos trabalhados no domínio de sistemas distribuídos. Conjunto de dados. Acessado em 19 de fevereiro de 2025.

Filó, T. G. S., Bigonha, M. A. S., e Ferreira, K. A. M. (2024). Evaluating thresholds for object-oriented software metrics. *Journal of the Brazilian Computer Society*, 30(1):315–346.

Gamma, E., Helm, R., Johnson, R., e Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Pearson Education.

Ghafoor, S., Brown, D. W., Rogers, M., e Haynes, A. (2023). Faculty development workshops for integrating PDC in early undergraduate curricula: An experience report. In Qasem, A., Thiruvathukal, G., e Bunde, D., editors, *11th Workshop on Education for High-Performance Computing (EduHPC-23)*, p. 1–8.

Google (2023). Google AI research. Disponível em: `https://ai.google.dev`. Acessado em 9 de setembro de 2024.

Martin, R. C. (2003). *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, USA.

Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, USA, 1 edition.

McQuaigue, M., Saule, E., Subramanian, K., e Payton, J. (2023). Data-driven discovery of anchor points for PDC content. In Qasem, A., Thiruvathukal, G., e Bunde, D., editors, *11th Workshop on Education for High-Performance Computing (EduHPC-23)*, p. 1–8.

Muldner, K., Jennings, J., e Chiarelli, V. (2022). A review of worked examples in programming activities. *ACM Trans. Comput. Educ.*, 23(1).

NSF/IEEE-TCPP (2024). Nsf/ieee-tcpp curriculum initiative on parallel and distributed computing. Online. Disponível em: `https://tcpp.cs.gsu.edu/curriculum/?q=home`.

NSF/IEEE-TCPP Curriculum Initiative (2023). EduPar-23: 13th NSF/TCPP Workshop on Parallel and Distributed Computing Education. In *In conjunction with 37th IEEE International Parallel and Distributed Processing Symposium*, St. Petersburg, Florida, USA. National Science Foundation (NSF) and the TCPP Curriculum Initiative on Parallel and Distributed Computing. EduPar-23 Workshop.

Raj, R. K. e Kumar, A. N. (2022). Toward computer science curricular guidelines 2023 (cs2023). *ACM Inroads*, 13(4):22–25.

Skudder, B. e Luxton-Reilly, A. (2014). Worked examples in computer science. In *6th Australasian Computing Education Conference*, volume 148, p. 59–64, Darlinghurst, Austrália. Australian Computer Society.

Subramanyam, R. e Krishnan, M. (2003). Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects. *IEEE Transactions on Software Engineering*, 29(4):297–310.

Tonhão, S., Colanzi, T., e Steinmacher, I. (2023). Using real worked examples to aid software engineering teaching. In *35th Brazilian Symposium onSoftware Engineering (SBES 2021)*, p. 133–142, New York, NY, EUA. ACM.

Tonhão, S., Silva, W., Colanzi, T., e Steinmacher, I. (2022). 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*, p. 33–38, Porto Alegre, RS, Brasil. SBC.

Tsantalis, N. (2018). *RefactoringMiner: A tool for detecting refactoring patterns*. Available in `https://github.com/tsantalis/RefactoringMiner`.

Tsantalis, N., Mansouri, M., Eshkevari, L. M., Mazinanian, D., e Dig, D. (2018). Accurate and efficient refactoring detection in commit history. In *40th International Conference on Software Engineering*, p. 483–494, New York, NY, USA. ACM.