Aferindo lacunas de aprendizagem utilizando análise estática automatizada em projetos de *software* orientados a objetos

Henrique César R. de Medeiros[1], Sonia Regina Fortes da Silva[1], Jackson Raniel F. da Silva[1]

[1]Campus Caruaru - Universidade de Pernambuco (UPE)
Rod. BR 104, KM 62 - 55.002-971 - Caruaru – PE – Brasil

henrique.medeiros,jackson.florencio,sonia.fortes@upe.br

Abstract. *The quality of software depends on skilled professionals, but educational institutions face challenges in preparing students due to time and resource limitations. This article uses action research as a methodological approach to diagnose gaps in student training related to software quality. The preliminary results indicate the need to equip students with knowledge and tools to assess the criticality of bugs, vulnerabilities, technical debt, test coverage, and code duplication.*

Resumo. *A qualidade do software depende de profissionais capacitados, mas instituições de ensino enfrentam desafios no preparo dos estudantes devido a limitações de tempo e recursos. Neste artigo, fazendo uso da pesquisa-ação como opção metodológica, é apresentado um diagnóstico de lacunas na formação dos estudantes relativas a qualidade do software desenvolvido. Os resultados parciais apresentados apontam para a necessidade de munir os estudantes com conhecimento e ferramentas de avaliação de criticidade de bugs, vulnerabilidades, débito técnico, cobertura de testes e duplicidade de código.*

1. Introdução

O mercado necessita de profissionais capacitados para construir seus sites, aplicações e, de modo geral, precisa que estas soluções de *software* cumpram suas funções seguindo alguns padrões de qualidade, sejam, em segurança, usabilidade sem travamentos ou falhas de funcionamento, cobertura de testes, entre outros. Devido às limitações humanas e de tempo, nem sempre é possível dar a devida atenção a todos os problemas presentes nos projetos produzidos pelos estudantes, o que acarreta em lacunas de aprendizado e de conhecimento.

Em paralelo a isso, existem as ferramentas de análise estática automatizada (FAEA), que, segundo [Sommerville 2011], são ferramentas de *software* que fazem a varredura do texto-fonte de um programa e detectam possíveis defeitos e anomalias, que podem ser úteis. Elas analisam o código fonte do programa procurando por não conformidades ou riscos de falhas e segurança em potencial.

Este artigo relata uma pesquisa-ação em andamento, voltada para analisar, com o uso de uma FAEA, os programas desenvolvidos por estudantes na disciplina de Projeto Orientado a Objetos (POO) de um curso superior de nível de graduação em Sistemas de Informação. Esta opção visa aprimorar o ensino de desenvolvimento de sistemas de *software* nas próximas execuções da mesma disciplina com foco na preparação dos estudantes para critérios de qualidade exigidos pelo mercado.

DOI: 10.5753/sbie.2024.244619

Seguindo orientações de [Thiollent 2022], nesta pesquisa-ação, o objetivo prático é demonstrar que, algumas vezes, ocorre de as instituições de ensino aprovarem estudantes sem que estejam devidamente qualificados para o mercado de trabalho e formarem os mesmos com alguns dos pilares centrais do desenvolvimento de *software* faltando.

Ainda conforme [Thiollent 2022], o objetivo de conhecimento visa obter informações que seriam de difícil acesso por meio de outros procedimentos e aumentar o conhecimento de determinadas situações. O objetivo de conhecimento dessa pesquisa-ação é obter as informações dos projetos para que seja comprovada a presença de imperfeições e apresentar os resultados extraídos pela FAEA como forma de auxílio de ensino para estudantes de ensino superior.

Como [Sommerville 2011] mostra, a análise estática automatizada é mais rápida e mais barata do que revisões de código detalhadas, aplicando-se à proposta deste estudo. Ele também explica a dificuldade e alto custo de montar uma equipe de inspeção, que se agrava em pequenas empresas ou grupos de desenvolvimento pois todos os membros da equipe também podem ser desenvolvedores de *software*, o que enviesaria os testes, reforçando o objetivo de conhecimento. No contexto de ensino, permitindo aos estudantes um suporte para além das possibilidades docentes.

2. Trabalhos relacionados

O uso de inspeção de código para extração de métricas de qualidade não é algo novo no contexto de engenharia de *software*. De modo que a temática já foi incluída em currículos de referências para a formação de profissionais da área de tecnologia da informação. [Rocha et al. 2005]. Ainda que essa técnica possa ser ensinada separadamente em um componente curricular, nada impede a sua utilização de forma transversal em outras disciplinas.

Na literatura, é possível encontrar a utilização das técnicas de inspeção de código para o ensino de qualidade de *software*, como é o caso do estudo de [de Andrade Gomes et al. 2017]. Especificamente nesse estudo, os autores avaliam tanto o uso de inspeção de código para o ensino como o uso de uma ferramenta de que sumariza métricas retiradas de um *software* de inspeção para os estudantes. Outro exemplo de uso de inspeção de código para o ensino pode ser encontrado no artigo de [Dietz et al. 2018]. Nesse último caso, os autores listam um conjunto de inspeções a serem feitas por estudantes e aferem a qualidade dos respectivos códigos desenvolvidos antes e depois das inspeções.

O presente artigo distingue-se desses citados por: apresentar a fase exploratória de uma investigação centrada na avaliação da aprendizagem de estudantes já aprovados; discutir as lacunas de formação encontradas; e, propor uma intervenção e averiguação dessa intervenção em turmas futuras, dando início a um estudo longitudinal.

3. Metodologia

Seguindo suas diretrizes [Gil 2002] e [Thiollent 2022], a investigação teve como principais etapas: a) na fase investigativa dos resultados parciais, 1ª fase exploratória da pesquisa-ação, com o *SonarQube*: (i) formulação de problemas encontrados nos *softwares*, (ii) construção de hipóteses, (iii) a realização de um seminário para demonstração

dos resultados parciais e com a seleção da amostra, (iv) coleta de dados resultados, (v) análise e interpretação dos dados; b) na fase final dos resultados da pesquisa-ação: (i) elaboração do plano de ação na disciplina aplicando reformulação nos *software*s e (ii) a divulgação dos resultados finais, após a intervenção no ensino-aprendizagem.

4. Fase exploratória

Seguindo as orientações de [Thiollent 2022] sobre a fase exploratória, a viabilidade de intervenção foi ponderada considerando a possibilidade da adoção das ferramentas necessárias para a pesquisa nos laboratórios do curso superior da disciplina analisada. O professor da disciplina de POO, recebendo a devida autorização, compartilhou os projetos dos estudantes para exposição à análise, essa foi a estratégia metodológica de pesquisa definida. A divisão de tarefas foi voltada a concretizar a estratégia metodológica com a FAEA chamada *SonarQube*. O problema privilegiado nesta pesquisa-ação foi a presença de lacunas de aprendizado no ensino superior quanto ao desenvolvimento dentro do paradigma orientado a objetos, que abre espaço para a hipótese de que as ferramentas de inspeção estática automatizada podem auxiliar na superação das lacunas de aprendizagem.

A apresentação dos resultados parciais da seleção de amostra foi feito por um dos pesquisadores para os demais, juntamente com o docente responsável pelas disciplinas à época do desenvolvimento dos projetos. Na ocasião, foi apresentado o *SonarQube* e seu passo a passo para analisar projetos, e pontuadas as propostas dos participantes e elaboradas as diretrizes da pesquisa-ação. A amostra é constituída de três projetos de *software* orientados a objetos desenvolvidos por grupos compostos por cinco estudantes. Essa amostragem apresentou-se adequada por: serem projetos previamente aprovados na disciplina sob investigação, serem representativos de uma população maior de projetos de estudantes, e permitirem ser tecnicamente aproveitados para a finalidade desse estudo.

A coleta de dados foi realizada utilizando as seguintes ferramentas: o *SonarQube Community*, a linguagem de programação Java e o ambiente de desenvolvimento integrado(IDE, na sigla em inglês) *IntelliJ Idea Community*. O *SonarQube* foi inicializado a partir de uma imagem *Docker* e então adicionados manualmente os projetos para expô-los à análise da ferramenta através do Maven, que é o gerenciador de projetos da *Apache* software *Foundation*, associado ao *IntelliJ Idea Community*.

4.1. Análise e interpretação dos dados

A coleta de dados preliminar deu-se de forma gráfica, como demonstrado na Figura 1. Nessa imagem, é possível observar as diferenças entre os três projetos estruturados no POO. Os projetos desenvolvidos pelas equipes 1 e 2 têm como objetivo ser o sistema de um condomínio e lidar com as informações relacionadas a esse propósito, a terceira aplicação analisada, por outro lado, é voltada para agendamentos para eventos e inscrições.

Como a Figura 1 mostra, problemas presentes nos projetos precisam ser pontuados. Para melhor entendimento do gráfico, o tamanho de cada bolha indica o número de linhas de código para cada arquivo. As cores de bolha, exceto verde, indicam *bugs* ou vulnerabilidades que devem ser examinados. Bolhas no topo ou direita, que é o caso nos três projetos, representam que a saúde do sistema em termos de longo prazo pode estar

correndo risco. Quanto mais à esquerda as bolhas estiverem menor é o débito técnico, e quanto mais próximo do eixo X mais assegurada pelos testes a aplicação está, o que indica que o débito técnico dos dois primeiros projetos demanda atenção.

Figura 1. Comparação das visões gerais dos projetos.

Como existem outros critérios que necessitam de análise para entender com exatidão os problemas dos projetos, os seguintes dados foram compilados na Tabela 1, que divide a análise nas categorias: quantidade de *bugs*, vulnerabilidades, débito técnico, cobertura de testes e duplicidades de código.

Em manutenibilidade a segunda aplicação é a mais problemática em razão do número de *code smells*, que são problemas de legibilidade dos códigos, e também é mais custosa por ter mais pontos de acesso, que são falhas de segurança, possui mais linhas de código repetidas e maior tempo de débito técnico, o que é agravado por ser a segunda aplicação com mais *bugs*. Em comparação com as demais soluções, a segunda aplicação apresenta mais de 9 vezes a quantidade de blocos duplicados das outras duas soluções e mais de 10 vezes a quantidade de linhas duplicadas das demais soluções, além de possuir mais de 13 vezes o número de arquivos duplicados das outras aplicações.

No *SonarQube*, os *bugs* são categorizados da seguinte maneira: *Blockers* são os erros que comprometem completamente a usabilidade do *software*, os *Criticals* são aqueles que comprometem parcialmente a aplicação mas possuem grau de severidade elevado. Além dessas, existem mais duas subdivisões dos *bugs*, são elas, *Major* e *Minor* que, respectivamente, indicam a presença de *bugs* maiores e menores.

Comparando os quatro elementos dos *bugs* (*blocker*, *critical*, *major* e *minor*) observa-se que a solução 3 - S3 é a que oferece menor número de *bugs*. Na avaliação de nível, S3 recebeu E em relação ao *bug*, pois possui um *blocker* que possibilita um dos maiores riscos, travamento total. No entanto, a solução 2 ficou no nível D e a solução

Tabela 1. Problemas no código orientado a objetos em cada solução analisada

Classe	Métrica	Solução 1	Solução 2	Solução 3
Quantidade de *bugs*		4	3	1
	Blockers	4	0	1
	Critical	0	1	0
	Major	2	1	0
	Minor	0	1	0
	Avaliação em relação aos *bugs*	E	D	E
	Esforço para remediação	25 min	15 min	5 min
Vulnerabilidades		1	0	0
	Avaliação de vulnerabilidades	E	A	A
	Pontos de acesso de segurança	20	23	0
	Esforço para remediação de vulnerabilidades	45 min	0	0
Débito técnico		7 dias	12 dias	2h e 30 min
	Proporção do Débito	3,5%	2,6%	0,4%
	Avaliação	A	A	A
	Esforço para alcançar avaliação A	0	0	0
	Code smells	428	714	20
Cobertura de testes		0,00%	0,00%	0,00%
	Linhas a cobrir	1541	3198	426
	Linhas não cobertas	1541	3198	426
	Cobertura de linhas	0,0%	0,0%	0,0%
Duplicidades de Código		2,50%	14,00%	3,60%
	Densidade	2,5%	14%	3,6%
	Linhas duplicadas	105	1323	60
	Blocos duplicados	10	105	4
	Arquivos duplicados	3	40	3

1 com o nível E, semelhante à solução 3. Isso ocorre, pois a solução 2 - S2 possui um *bug* do tipo *critical*, um *major*, e um *minor*, mas não traz nenhum *blocker*. A solução um - S1, no entanto, contém quatro *blockers* e dois *major*, apresentando o maior risco de travamento total entre as três. Assim, tanto a solução 1 como a 2, precisam ser depuradas para posteriormente melhorar a confiabilidade.

Quanto ao esforço para remediação dos *bugs*, S1 exige o maior tempo para a depuração, 25 minutos, enquanto S2 exige 15 minutos e S1 5 minutos. Autores experientes em desenvolvimento de sistemas, como [Spinellis 2016], destacam que existem técnicas e abordagens específicas para depuração efetiva de acordo com o tipo de contexto das soluções, o que se aplica às soluções S1 e S2. Algo que poderia evitar a presença de tantos *bugs* produzindo parte deste débito técnico seria a presença de testes que, conforme [Ramos 2021] esclarece, aumentam a confiança de funcionamento correto do código.

Apesar de repetição de código ser o que a orientação a objetos se propõe a mitigar, o segundo projeto possui 1323 linhas de código repetidas, que representam 14% do projeto, enquanto os demais apresentam uma proporção relativamente menor. Duplicações

de código como estas presentes nas aplicações indicam a possibilidade de aplicação de padrões de projeto, que são soluções padronizadas para problemas corriqueiros no desenvolvimento de sistemas em *software*, catalogados por [Gamma 2009].

Direcionando-se aos problemas de manutenibilidade, o *SonarQube* produz um gráfico associando o mau cheiro de código, por arquivo, à razão do débito técnico pelo nível de proximidade da cor das bolhas ao vermelho. O tamanho da bolha indica o volume de mal cheiro no código, enquanto a posição vertical indica o tempo estimado para corrigir o código. Bolhas verdes e pequenas na borda inferior indicam o cenário ideal. Como é apresentado nos gráficos da Figura 2:

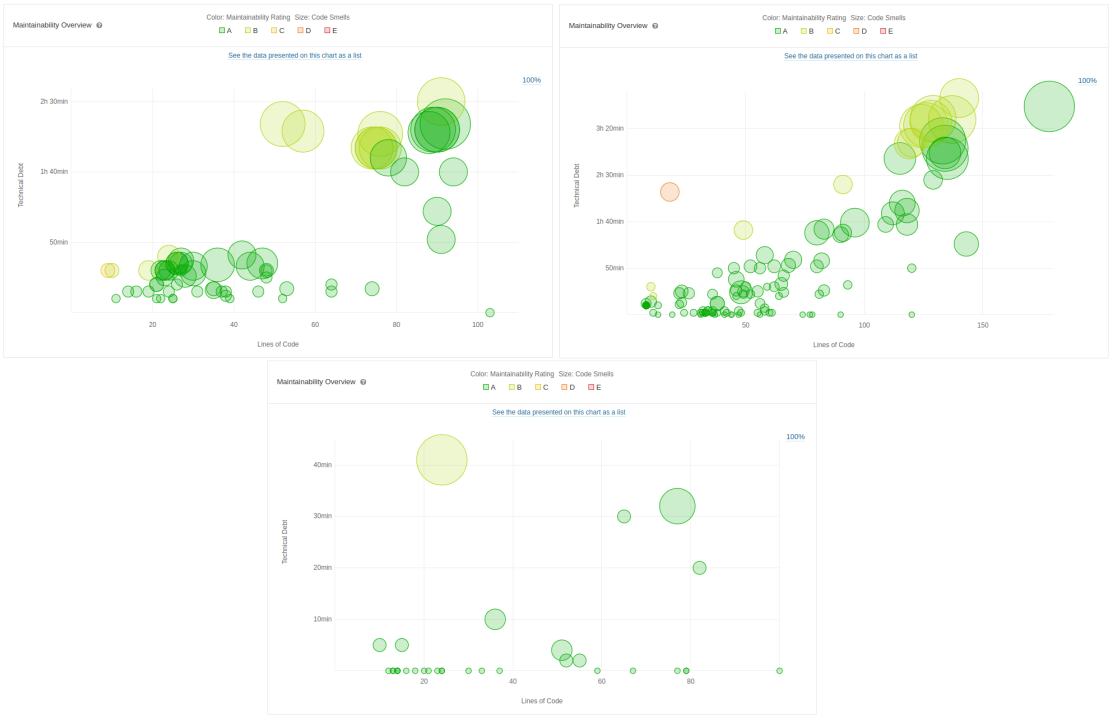

Figura 2. Visões gerais da manutenibilidade dos projetos.

Como apontam os gráficos da fig. 2, os três projetos apresentam bolhas predominantemente verdes, ou seja, por mais que exista demanda de débito técnico, todas as soluções de *software* foram bem escritas em termos de código limpo, conforme alguns dos princípios proposto por [Martin 2009]. Quanto a isso, entre as três aplicações a terceira possui o código mais bem escrito em termos de *code smells*, melhor manutenibilidade, menor proporção de débito e menor débito técnico, enquanto que a segunda aplicação entre as três apresenta menor qualidade.

Partindo para análises relativas à segurança, quanto à gravidade e quantidade das vulnerabilidades que só estão presentes na primeira aplicação, há apenas uma vulnerabilidade, porém, que precisa de estimadamente 45 minutos para ser remediada e que requer prioridade por ser do pior tipo de avaliação possível para uma vulnerabilidade de segurança.

O esforço de retrabalho encontrado para solucionar os problemas constatados pelo *SonarQube* variam de 2 horas e trinta minutos para S3, a 12 dias para S2 e 7 dias para S1,

elevando esse nível de espaço de tempo, demonstrando que a proporção varia conforme o risco de travamento do *software*, constatando que, caso tenha menor tempo de correção, será menor a proporção de débito técnico. Essa questão é discutida por [Torres 2015], que afirma que conforme o débito técnico cresce, mais lento o time de engenharia produto da empresa ficará.

Constatou-se nos escores do débito técnico, proporção do débito técnico e *Code Smells* uma simetria, pois os maiores valores, em ordem decrescente, foram para S2, S1 e depois S3, demonstrando ligação direta entre qualidade do código e o débito técnico. Observa-se também a duplicação de código e a densidade com a proporção de 10 para 1, caracterizando que quanto mais duplicado, mais denso é o código.

5. Discussão

A correção de *bugs* é parte essencial no desenvolvimento de *software* tendo em vista que o funcionamento de um sistema é parte crucial para sua comercialização. Assim, existe a necessidade de que as universidades empoderem os estudantes com conhecimentos, técnicas e ferramentas para auxiliar na capacitação de identificação e correção de *bugs*. As soluções propostas por [Gamma 2009] podem evitar o surgimento desse tipo de problema reduzindo a complexidade do código através da abstração, que é um dos pilares da programação orientada a objetos.

Quanto às falhas de segurança, segundo [Sommerville 2011], existe um conjunto de boas práticas universalmente aceitas, que ajudam a reduzir os defeitos nos sistemas entregues. Como a seção de análise e interpretação dos dados mostrou, existe a necessidade de capacitação em desenvolvimento, identificação de problemas e solução voltados também aos critérios de segurança de *software* no POO.

Sobre a qualidade de *software*, que é intrínseco à manutenibilidade, conforme [Martin 2009] esclareceu, existe a necessidade de capacitação para que seja escrito o código de forma assertiva em termos de clareza de legibilidade do código, o que projetos 1 e 2 reiteram.

Sobre testes de *software*, [Sommerville 2011] afirma que são responsáveis por mostrar que um programa faz o que é proposto a fazer e para descobrir os defeitos do programa antes do uso. As soluções analisadas neste estudo não possuem testes, ou seja, não há garantia de que esses sistemas funcionarão como previsto quando, em produção, for exposta à usuários que não os usarem conforme o planejado.

6. Conclusões

Este artigo apresentou o relato da execução da fase exploratória de uma pesquisa-ação em curso. A análise deu-se com o uso de uma FAEA em projetos de código orientado a objetos desenvolvidos por estudantes da disciplina de POO de uma instituição de ensino superior.

Os objetivos propostos foram atingidos uma vez que, torna-se perceptível que, em razão de limitações humanas e de tempo, não foi possível que o docente responsável pela disciplina percebesse lacunas na formação dos estudantes que foram captadas por essa análise. As lacunas observadas foram categorizadas em: testes de *software*, segurança, qualidade de código e correção de *bugs*.

Diante do exposto, levantam-se as hipóteses de que: i) o ensino de inspeção estática de código pode ajudar os estudantes da disciplina de POO a reduzir as lacunas de aprendizagem evidenciadas nos projetos analisados; e ii) a utilização de FAEAs pode levar os estudantes a encontrar e corrigir mais problemas de código orientado a objeto em comparação com uma análise não automatizada.

Essas hipóteses serão testadas em trabalhos futuros através de um delineamento experimental de quadrados latinos [Aranha and Reis 2020] em que o tratamento seja a adição do *SonarQube* à aula, os materiais experimentais sejam os projetos dos estudantes analisados pela FAEA proposta por esta pesquisa-ação em conjunto com o *Maven* e um ambiente de desenvolvimento integrado compatível e os participantes no experimento sejam os estudantes da disciplina estudada.

Sendo, as variáveis dependentes a quantidade de problemas de código encontrados e a quantidade dos mesmos que foi resolvida, os fatores controlados e não controlados no tratamento fossem, respectivamente a avaliação de aprendizado dos estudantes com e sem o uso do *SonarQube* como ferramenta auxiliar.

Referências

Aranha, E. and Reis, T. (2020). Delineamentos experimentais em informática na educação. *Metodologia de Pesquisa Científica em Informática na Educação: Abordagem Quantitativa, SBC. Disponível em:¡ https://metodologia. ceie-br. org/livro-2*.

de Andrade Gomes, P. H., Garcia, R. E., Spadon, G., Eler, D. M., Olivete, C., and Correia, R. C. M. (2017). Teaching software quality via source code inspection tool. In *2017 IEEE Frontiers in Education Conference (FIE)*, pages 1–8. Ieee.

Dietz, L. W., Manner, J., Harrer, S., and Lenhard, J. (2018). Teaching clean code. In *Proceedings of the 1st Workshop on Innovative Software Engineering Education*.

Gamma, E. (2009). *Padrões de projetos: soluções reutilizáveis*. Bookman editora.

Gil, A. C. (2002). *Como elaborar projetos de pesquisa*. Editora Atlas SA.

Martin, R. C. (2009). *Clean code: a handbook of agile software craftsmanship*. Pearson Education.

Ramos, G. (2021). *JavaScript Assertivo: Testes e qualidade de código em todas as camadas da aplicação*. Casa do Código.

Rocha, M. d. G. B., Nicoletti, M., et al. (2005). Currículo de referência da sbc para cursos de graduação em bacharelado em ciência da computação e engenharia de computação. *SBC, Tech. Rep*.

Sommerville, I. (2011). Engenharia de software, 9a. *São Palo, SP, Brasil*, page 63.

Spinellis, D. (2016). *Effective debugging: 66 specific ways to debug software and systems*. Addison-Wesley Professional.

Thiollent, M. (2022). *Metodologia da pesquisa-ação*. Cortez editora.

Torres, J. (2015). *Gestão de produtos de software: como aumentar as chances de sucesso do seu software*. Editora Casa do Código.