Comparação de ferramentas de análise estática para detecção de defeitos de software usando mutantes

Yury Alencar Lima[1], **Igor Oliveira Fonseca**[1], **Jonas M. Chagas**[1], **Elder de Macedo Rodrigues**[1], **Maicon Bernardino da Silveira**[1], **João Pablo S. da Silva**[1]

[1]Universidade Federal do Pampa (UNIPAMPA)
Código Postal 97.546-550 – Alegrete – RS – Brasil

{yuryalencar19, igorfonsecaigor, jonaschagas10}@gmail.com

{elderrodrigues, maiconbernardino, joaosilva}@unipampa.edu.br

Abstract. *Static code analysis is a technique applied in the verification step of a software, with the objective of identifying defects without the need to run the application. Currently, there are several static analysis tools available on the market, where each one has its particularities and different scopes of analysis. Therefore, choosing the right tool to use is not a trivial activity. This study aims to compare two static code analysis tools, SonarQube and PMD, analyzing their effectiveness in identifying defects in software code. For this purpose, mutants were created from Open Source projects, which helped to verify the effectiveness of the tools.*

Resumo. *A análise estática de código é uma técnica aplicada na etapa de verificação de um software, com o objetivo de identificar defeitos sem a necessidade da execução da aplicação. Atualmente existem diversas ferramentas de análise estática disponíveis no mercado, onde cada uma possui suas particularidades e diferentes escopos de análise. Portanto, escolher a ferramenta correta a ser usada não é uma atividade trivial. Este estudo tem como objetivo comparar duas ferramentas de análise estática de código sendo elas o SonarQube e o PMD, analisando a sua eficácia na identificação de defeitos em um código de software. Com esta finalidade foram criados mutantes a partir de projetos Open Source, que auxiliaram a verificar a eficácia das ferramentas.*

1. Introdução

O desenvolvimento de *software* com qualidade é um fator essencial para que os produtos atendam às necessidades dos usuários de forma satisfatória [Delamaro et al. 2013]. Na área de qualidade de *software*, são empregadas diversas tecnologias, metodologias e processos com o objetivo de detectar a maior quantidade possível de defeitos, possibilitando o aumentando da confiabilidade e redução na quantidade de erros. A verificação e validação é responsável pela detecção de defeitos dentro de um produto de *software*. Os defeitos, são imperfeições dentro do código de um produto de *software*. A presença de um defeito em uma aplicação pode ocasionar um erro, ou seja, o funcionamento indevido de alguma função ou classe, que por sua vez pode se manifestar como uma falha ou não ao usuário final [Delamaro et al. 2013]. Desse modo, a presença de erros, defeitos e falhas gera impacto na confiabilidade de uma aplicação.

Dentro da verificação existem diversas técnicas, tais como: sala limpa [Mills et al. 1987], inspeção [Ebenau and Strauss 1993] e análise estática de código [Webb 1990]. A análise estática tem como objetivo encontrar defeitos sem a necessidade de executar uma aplicação [Emanuelsson and Nilsson 2008]. Entretanto, essa técnica possui um alto custo, decorrente da necessidade de alocar recursos somente para realizar as análises. A fim de fazer o uso da técnica e reduzir o tempo de execução, aumentar a eficiência e diminuir os custos relacionados, foram desenvolvidas algumas ferramentas para realizar essas análises de forma automatizada. Além dos defeitos de implementação, ferramentas de análise estática podem identificar, entre outros, casos de perda de recursos e vulnerabilidades de segurança. A análise automatizada de código é uma técnica economicamente viável e considerada eficiente em identificar defeitos antes da fase de testes não automatizados [Zheng et al. 2006].

Atualmente, há uma grande variedade de ferramentas – tanto comerciais quanto de *software* livre – destinadas à análise estática de código. A escolha de qual ferramenta de análise utilizar pode implicar na identificação ou não de determinadas categorias de defeitos, impactando na qualidade do *software* desenvolvido e na produtividade do processo de desenvolvimento como um todo. Desse modo, este estudo tem como objetivo comparar duas ferramentas para automação de análise estática de código quanto à sua capacidade de identificação de defeitos. Com esse objetivo, foi utilizada a estratégia de mutantes [Bochmann and Petrenko 1994], ou seja, a inserção de problemas específicos em versões diferentes da aplicação para avaliar o desempenho das ferramentas.

Este estudo está organizado como descrito a seguir. A Seção 2 apresenta trabalhos relacionados ao tema deste estudo. A Seção 3 apresenta o referencial teórico dos conceitos empregados no desenvolvimento da pesquisa, além de descrever as ferramentas que foram comparadas e em quais sistemas os testes foram executados. A Seção 4 detalha a preparação dos artefatos para os testes. A Seção 5, por sua vez, mostra a análise dos resultados obtidos e o comparativo entre eles. Já a Seção 6 encerra o trabalho e apresenta possibilidades de extensão para a pesquisa apresentada.

2. Trabalhos Relacionados

Com a finalidade de obter uma melhor visão e entendimento sobre o tema deste trabalho, foram selecionados dois estudos por meio de uma busca realizada de forma *ad hoc* na plataforma *Google Scholar* utilizando a *string* de busca "*comparative study* + *analysis static code* + *tools*". A proposta do trabalho de Fatima et al. [Fatima et al. 2018] é realizar a comparação de quatorze ferramentas que dão suporte às linguagens C e C++. O trabalho fornece uma visão geral das ferramentas e realiza a comparação por meio de vinte e oito critérios, como variáveis, ponteiros, injeção, memória, entrada, laços e funções, além de algumas outras verificações, como divisibilidade por zero, onde para cada critério é atribuído um valor sim/não. Ao final, cada ferramenta recebe uma pontuação que sumariza os critérios, podendo assim representar graficamente as comparações.

O trabalho de Li e Cui [Li and Cui 2010] possui como principal objetivo analisar e comparar sete ferramentas de análise estática e ao final propor um método eficiente para detecção de vulnerabilidades. A maioria das ferramentas possuem suporte à linguagem Java, duas suportam C++ e uma suporta a linguagem C. O estudo apontou vantagens e desvantagens de cada ferramenta e verificou a capacidade de detecção de vulnerabil-

idades. Diante da lista de vantagens e desvantagens, o estudo propôs um método de detecção de falha de *software* utilizando uma combinação de análise estática e detecção dinâmica.

3. Referencial Teórico

3.1. Análise Estática

Dentro do desenvolvimento de *software*, é crucial que os artefatos produzidos sejam de qualidade, para auxiliar na garantia da qualidade a atividade de inspeção pode ser uma alternativa [Melo 2009]. A inspeção é um processo sistemático com o intuito de verificar se os artefatos criados durante o ciclo de desenvolvimento, satisfazem a sua respectiva especificação [Wang and Lee 2008]. O processo de inspeção é composto pela revisão de artefatos, que podem ser especificações, modelos, código-fonte ou outro qualquer artefato criado durante o desenvolvimento. Para analisar esses artefatos são utilizados *checklists*, que listam os problemas a serem detectados em cada artefato. Assim, cada participante realizando a inspeção possui um guia de problemas considerados importantes que serão a base para detecção de inconsistências . Como o processo de inspeção não necessita de execução do produto de *software*, ele é classificado como uma análise estática [Melo 2009]. Entretanto, o processo de análise estática tende a ser caro, devido à necessidade de alocação de profissionais destinados somente a essa atividade, que tende a ser longa [Melo 2009]. Desse modo, estratégias automatizadas através de ferramentas de análise estática são capazes de reduzir os custos, aumentando a qualidade do produto de *software*, além de viabilizar um *feedback* mais rápido para os desenvolvedores.

3.2. Ferramentas de Análise Estática de Código

Neste estudo as ferramentas de análise estática foram selecionadas a partir dos seguintes critérios: 1) Ser de código aberto; 2) Possuir uma grande quantidade de usuários ativos (verificado através das estrelas no GitHub); 3) Possuir suporte para as linguagens Javascript e Java, pois são linguagens amplamente utilizadas na criação de aplicações. Uma das ferramentas escolhidas foi o SonarQube por ser uma ferramenta automática de análise de código que visa inspecionar continuamente a qualidade do código, possibilitando encontrar defeitos, *code smells* e vulnerabilidades de segurança [Sonarqube 2021]. A ferramenta presta suporte a uma gama de linguagens de programação, como Java, JavaScript/TypeScript, Python e PHP, que são algumas das mais utilizadas na indústria. O processo dentro da ferramenta ocorre de forma automática, podendo ser configurada no projeto para executar após a finalização do *build*. A documentação é bem detalhada, facilitando realizar a instalação e configuração, contando com vários exemplos e possibilidades de execução, inclusive através de ferramentas de integração contínua.

Por fim, o PMD foi escolhido por também ser é uma ferramenta de código-fonte aberto que realiza a análise estática principalmente em códigos nas linguagens Java e JavaScript. Tem como principal funcionalidade analisar o código procurando defeitos, *code smells* e duplicação de código [PMD 2021]. Também se destaca a análise de complexidade ciclomática, ou seja, a quantidade de possíveis caminhos que o código pode executar. A ferramenta executa a análise conforme as regras descritas e configuradas previamente em um arquivo no formato XML. O PMD inclui conjuntos de regras integrados para executar análises rápidas com uma configuração padrão, porém é possível

personalizar as regras de acordo com a necessidade. A documentação do PMD é bastante abrangente, contando com exemplos de uso, lista de ferramentas com as quais possui integração e formatos de relatórios, entre outras informações.

3.3. Sistemas Sob Teste

Com o intuito de avaliar as ferramentas de análise estática é importante testar sua utilização em sistemas com diferentes características. Enquanto para aplicações *web* a maioria dos defeitos se localizam na camada de apresentação, nas aplicações para *desktop* os defeitos são encontrados principalmente na camada lógica [Torchiano et al. 2011]. Assim, foram selecionados dois sistemas para plataformas distintas. Além da diferença na plataforma para a qual foram desenvolvidos, os sistemas têm complexidades distintas, permitindo analisar o desempenho de cada ferramenta levando em consideração o tamanho da aplicação.

A primeira aplicação selecionada é uma API que realiza a integração de dois serviços[1]. Um dos serviços é referente à busca de receitas a partir dos seus respectivos ingredientes, onde o usuário pode apresentar os ingredientes que possui e solicitar que a aplicação procure receitas possíveis com estes itens. Entretanto, o serviço anterior não apresenta uma imagem ou representação visual da receita, sendo esse um fator importante para a adesão dos usuários. Desse modo, foi necessária a integração com um segundo serviço de busca de *gifs*, onde é buscada a representação de cada receita para retornar ao usuário da aplicação. Essa solução foi implementada utilizando a linguagem JavaScript e alguns padrões de projeto, como o *Adapter* e *Strategy*. Desse modo, o projeto completo possui 823 LoC.

A segunda aplicação escolhida foi um simulador *desktop* de cafeteira inteligente denominado SmartCoffee[2]. O simulador realiza todo o controle de pagamento, gerando trocos a partir da quantidade de moedas presentes no seu compartimento. Além disso, o simulador analisa os insumos disponíveis na máquina, enviando e-mails ao responsável técnico para alertar a ocorrência de níveis baixos ou falta de copos, ingredientes e moedas. O envio de alertas também é acionado quando os compartimentos de moedas ficam altos ou cheios. Por fim, o técnico também possui autenticação na aplicação, exigida para realizar as manutenções. Com o objetivo de implementar essa solução foi utilizado o Java como linguagem de programação e para a criação da interface gráfica foi utilizada a plataforma JavaFX. A aplicação, incluindo a interface gráfica, foi implementada utilizando 2.217 LoC.

3.4. Análise de Mutantes

A análise de mutantes tem como principal objetivo detectar a eficácia de casos de testes e inspeções automatizadas [DeMillo et al. 1978]. Essa análise é realizada com base em pequenas modificações sintáticas (mutantes) inseridas de maneira sistemática dentro de um produto de *software*. Desse modo é possível com um único mutante inserido em um *software* analisar e afirmar algumas hipóteses como: os meus testes/inspeções automatizados poderiam detectar essa anomalia? Uma simples anomalia pode apresentar diversos erros acoplados [DeMillo et al. 1978]? O processo para realizar uma análise com

[1]Repositório da API: https://github.com/gitpublications/challenge
[2]Repositório do SmartCoffee: https://github.com/gitpublications/smartcoffee

mutantes geralmente é composto por: 1) execução do programa juntamente com suas inspeções/testes automatizados; 2) geração ou injeção dos mutantes; 3) execução dos mutantes com as inspeções/testes automatizados do programa; 4) análise dos mutantes [Delamaro et al. 2013].

A etapa de geração ou injeção de mutantes dentro de uma aplicação também pode ser diferente de acordo com a tecnologia utilizada. Em cada linguagem de programação é possível utilizar operadores de mutação especializados para a tecnologia [Delamaro et al. 2001]. Neste estudo foram utilizados os seguintes operadores de mutação baseados no estudo [Mirshokraie et al. 2015]: 1) **Variável local/global**: Adição, atualização ou remoção de uma variável; 2) **Parâmetro de função**: Adição, atualização ou remoção de um parâmetro de função; 3) **Condição de laço de repetição**: Atualização de uma condição utilizada em laços de repetição; 4) **Condição em condicional**: Atualização de uma condição utilizada em alguma estrutura condicional; 5) **Retorno**: Atualização do retorno de uma função; 6) **DOM**: Operador específico para aplicações *web*, onde pode ser atualizado algum elemento presente na árvore de elementos presentes no HTML [Raggett et al. 1999].

4. Criação dos Mutantes

Com o objetivo de analisar a qualidade da inspeção realizada por cada ferramenta de análise estática, foram criados dezesseis projetos mutantes, a partir dos dois projetos originais utilizados. Os projetos, que estão disponíveis em um repositório público no GitHub[3], foram divididos por tecnologia e também possuem o projeto original, viabilizando a replicação deste estudo posteriormente.

A criação dos projetos mutantes foi realizada de forma manual e aleatória dentro da aplicação. A partir da versão original das duas aplicações escolhidas, foi gerado um clone do seu repositório para cada defeito a ser inserido. A seguir são apresentados os projetos referentes à tecnologia Java presentes no estudo: 1) **Original**: Projeto original escrito na tecnologia Java e utilizado no estudo; 2) **Mutant 1**: Neste projeto o mutante foi inserido no arquivo *CoinsCompartment*, linha 14, atualizando a inicialização da variável com *"null"*. Classificado como mutante de variável local/global; 3) **Mutant 2**: Mutante inserido no arquivo *IngredientsCompartiment*, onde a linha 22 foi removida. Trata-se de uma remoção de variável, classificando-o como mutante de variável local/global; 4) **Mutant 3**: O mutante inserido localiza-se no arquivo *DrinkMachine*, onde foi adicionado um parâmetro extra na linha 78. Classificado como mutante de parâmetro de função; 5) **Mutant 4**: O mutante foi inserido no arquivo *LoginController*, com a remoção de um dos parâmetros da linha 26. Classificado como mutante de parâmetro de função; 6) **Mutant 5**: Neste projeto o mutante foi inserido no arquivo *DrinkMachine*. Consiste na atualização da condição presente na linha 485. Classificado como mutante de condição de laço de repetição; 7) **Mutant 6**: Localizado no arquivo *DrinkMachine*, local onde foi atualizada a condição presente na linha 33. Classificado como mutante de condição em condicional; 8) **Mutant 7**: Neste projeto o mutante foi inserido no arquivo *DrinkMachine*, atualizando o retorno presente na linha 274. Classificado como mutante de retorno; 9) **Mutant 8**: O mutante foi inserido no arquivo *template.html*, onde foi atualizada a *tag p* para *paragraph* nas linhas 13 e 19. Classificado como mutante de DOM.

[3]Repositório dos mutantes: https://github.com/gitpublications/mutantsjavaandjs

As atualizações dos mutantes derivados do projeto escrito em JavaScript são descritas a seguir: 1) **Original**: Projeto original escrito na tecnologia JavaScript e utilizado no estudo; 2) **Mutant 1**: Neste projeto o mutante foi inserido no arquivo *recipesController*, linha 13. Foi atualizada a inicialização da variável com *"null"*. Classificado como mutante de variável local/global; 3) **Mutant 2**: O mutante localiza-se no arquivo *RecipePuppyService*, linha 5, onde houve uma remoção de variável. Classificado como mutante de variável local/global; 4) **Mutant 3**: Foi inserido no arquivo *giphyService*, adicionando um parâmetro a mais na linha 28. Classificado como mutante de parâmetro de função; 5) **Mutant 4**: Neste projeto o mutante está no arquivo *baseService*, onde foi removido o parâmetro da linha 7. Classificado como mutante de parâmetro de função; 6) **Mutant 5**: O mutante é localizado no arquivo *RecipePuppyService*, local onde foi atualizada a condição presente na linha 20. Classificado como mutante de condição em condicional; 7) **Mutant 6**: Neste projeto o mutante foi inserido no arquivo *giphyService*. Nesse arquivo, foi atualizado o retorno presente na linha 37 para *"null"*. Classificado como mutante de retorno; 8) **Mutant 7**: Localizado no arquivo *giphyService*, foi atualizado o retorno presente na linha 16. Classificado como mutante de retorno; 9) **Mutant 8**: Neste projeto o mutante está localizado no arquivo *giphyService*, onde ocorreu a remoção da linha 5. Classificado como mutante de variável local/global, pois inicializa todas as variáveis da classe "pai" da classe.

5. Análise e comparação

A análise estática dos 18 projetos resultantes foi realizada com as ferramentas SonarQube e PMD, utilizando suas configurações padrão. Nesta seção são apresentados os resultados das ferramentas após a realização das análises, tanto do projeto original como nos mutantes.

5.1. Resultados na ferramenta SonarQube

A análise executada sobre o projeto SmartCoffee em sua versão original (sem a introdução de mutantes). Ela resultou na identificação de três *bugs*, uma vulnerabilidade, um aviso de segurança e 167 *code smells*, além de identificar 9% de código duplicado, distribuído entre 39 blocos de código. No total, o tempo estimado pela ferramenta para a resolução dos problemas indicados é de aproximadamente quatro dias. Os *bugs* identificados no projeto referem-se a um problema de tratamento de exceção (ausência de ação no caso de interrupção de uma *thread*), ausência de um elemento HTML obrigatório em uma das interfaces e ausência de uma fonte utilizada na interface. A vulnerabilidade indicada pela ferramenta diz respeito à funcionalidade de envio de e-mails, onde deve ser habilitada a verificação do certificado de segurança do servidor para evitar possíveis ataques do tipo *man-in-the-middle*. Já o aviso de segurança é relacionado a uma função de *debug* presente no código, o que pode representar um risco, devendo ser revisada pelo desenvolvedor a necessidade de mantê-la e a possibilidade de remoção.

Os *code smells* identificados pela ferramenta são classificados em três graus de importância: 62 são considerados críticos, 60 severos e 45 leves. Os problemas encontrados por sua vez não devem impactar nos resultados após a análise das ferramentas, pois, é executada uma análise estática do código sem a necessidade de sua execução. Desse modo, é analisado somente o código, e através de padrões inferindo o seu comportamento. Os problemas classificados como críticos são principalmente devido à duplicação

de *Strings* literais (45 ocorrências) e instruções *switch* sem um caso padrão especificado (14 ocorrências), sendo os demais casos métodos vazios ou muito complexos. O problema severo mais comum é a utilização de impressões no console, ao invés de utilizar um log (36 ocorrências). Outros problemas severos são expressões booleanas com valor sempre verdadeiro, blocos de código comentados e atribuições não utilizadas, entre outros. Já os problemas considerados leves são relacionados a nomes que não seguem a convenção do Java, importações não utilizadas e problemas de escopo na declaração de métodos.

Quanto ao código duplicado, a maior parte dele se refere ao controlador da interface da ferramenta, que em diversas situações mostra as mesmas mensagens na tela para entradas diferentes. A sequência de instruções para imprimir essas mensagens na tela foi replicada para cada possível situação, o que poderia ter sido evitado. O segundo grupo de código duplicado é referente a duas funções que realizam operações diferentes, porém sobre o mesmo grupo de variáveis onde realizam o mesmo pré-processamento. Esse trecho deveria ser isolados em uma única função auxiliar para eliminar a duplicação de código.

Entre as oito versões mutantes do projeto SmartCoffee, as únicas que tiveram um resultado diferente da versão original foram a Mutant 3, Mutant 4 e Mutant 7. Em cada uma delas, foram identificados 168 *code smells*, um a mais que na versão original. Isso indica uma eficiência de 37,5% da ferramenta na identificação dos mutantes do projeto em linguagem de programação Java. O problema adicional encontrado na versão Mutant 3 é um parâmetro não utilizado em um método. Na versão Mutant 4, foi identificada uma nova importação não utilizada. A análise da versão Mutant 7 detectou um método com retorno do tipo booleano que sempre retorna o valor verdadeiro, já que este mutante inverteu uma das condições de retorno do método.

No projeto em JavaScript, a análise identificou apenas seis *code smells* na versão original, gerando uma estimativa de 1 hora e 12 minutos de trabalho para a solução dos problemas. Não foram identificados *bugs*, vulnerabilidades, avisos de segurança ou código duplicado. Os *code smells* identificados referem-se ao tratamento inadequado de exceções e problemas na declaração de variáveis. Quanto à análise dos mutantes deste projeto, apenas no Mutant 8 foram identificados 3 *bugs*. Para os outros sete mutantes, nenhum problema adicional foi indicado pela ferramenta. Todos os defeitos detectados no Mutant 8 derivam do problema de que a ferramenta esperava uma chamada ao construtor da superclasse, que não foi encontrada.

5.2. Resultados na ferramenta PMD

Inicialmente o projeto SmartCoffee, codificado em Java, foi analisado na versão original pela ferramenta, que detectou um total de 86 *code smells*. O PMD não apresenta sugestões de correção, somente indica qual linha do arquivo que contém o defeito com uma breve descrição do defeito. A ferramenta reportou 15 classes de *code smells*, descritas a seguir: 1) *UnnecessaryImport* - Importação não utilizada pela classe, com 8 ocorrências e categorizado como “prioridade média-baixa (4)”; 2) *UnusedPrivateField* - Verifica quando um campo é declarado mas não é utilizado, com 2 ocorrências e categorizado como ”prioridade média (3)”; 3) *SingularField* - Verifica variáveis que foram declaradas na classe e são limitadas a somente um método, com 2 ocorrências e categorizado como “prioridade média (3)”; 4) *LooseCoupling* - Verifica a referência utilizada em variáveis e sugere a utilização de interfaces para ter maior flexibilidade quando for

realizada a manutenção do código, com 12 ocorrências e categorizado como “prioridade média (3)”; 5) *UnnecessaryConstructor* - Verifica construtores declarados na classe que são iguais ao *default*, com 1 ocorrência e categorizado como “prioridade média (3)”; 6) *UncommentedEmptyConstructor* - Verifica construtores vazios, com 3 ocorrências e categorizado como “prioridade média (3)”; 7) *UnusedLocalVariable* - Verifica variáveis locais declaradas mas não utilizadas, com 4 ocorrências e categorizado como “prioridade média (3)”; 8) *SimplifyBooleanReturns* - Verifica utilização desnecessária de blocos de condição quando o método retorna um *boolean*, com 2 ocorrências e categorizado como “prioridade média (3)”; 9) *UselessParentheses* - Verifica a utilização de parenteses desnecessários, com 1 ocorrência e categorizado como “prioridade média-baixa (4)”; 10) *LiteralsFirstInComparisons* - Verifica o posicionamento dos argumentos na realização da comparação, com 10 ocorrências e categorizado como “prioridade média (3)”; 11) *ControlStatementBraces* - Verifica a utilização de chaves em blocos condicionais e de repetição, com 25 ocorrências e categorizado como “prioridade média (3)”; 12) *SwitchStmtsShouldHaveDefault* - Verifica a ausência da opção *default* em blocos *switch*, com 14 ocorrências e categorizado como “prioridade média (3)”; 13) *UncommentedEmptyMethodBody* - Verifica a existência de métodos vazios, com 1 ocorrência e categorizado como “prioridade média (3)”; 14) *UnnecessaryLocalBeforeReturn* - Verifica a criação de variáveis locais desnecessárias para retornos de métodos, com 1 ocorrência e categorizado como “prioridade média (3)”; 15) *NonThreadSafeSingleton* - Verifica *singletons* implementados de forma estática, com 1 ocorrência e categorizado como “prioridade média (3)”.

Introduzindo os códigos mutantes, o analisador reportou resultado diferente do projeto original apenas na análise do Mutant 4, que identificou 87 *code smells*. O *code smell* adicional na análise do Mutant 4 é referente à classe *UnnecessaryImport*. Assim como o projeto anterior, o projeto em JavaScript foi analisado primeiramente em sua versão original (sem mutantes), o qual reportou trinta (30) *code smells*. Os três (3) tipos de *code smells* identificados e descritos nessa análise são: 1) *AvoidTrailingComma* - Esta regra verifica a portabilidade do código relacionado as diferenças no tratamento dos navegadores, com 1 ocorrência e categorizado como “prioridade alta (1)”; 2) *GlobalVariable* - Esta regra verifica variáveis declaradas acidentalmente com escopo global, com 24 ocorrências e categorizado como “prioridade alta (1)”; 3) *UnnecessaryBlock* - Verifica blocos de código desnecessários, como blocos condicionais, com 5 ocorrências e categorizado como “prioridade média (3)”.

Neste projeto, após serem introduzidos os códigos mutantes, o analisador reportou diferença em relação o projeto original em dois mutantes: Mutant 2 e Mutant 8. Ambos apresentaram 29 *code smells*, um a menos que o projeto original. O *code smell* que não apareceu nos Mutants 2 e 8 é do tipo *GlobalVariable*. Isso ocorreu porque os dois mutantes inseridos tratam-se de remoção de variáveis e, no projeto original, essas variáveis haviam sido identificadas pela ferramenta como acidentalmente declaradas em escopo global. Assim, considera-se que estes mutantes não foram identificados pelo PMD.

5.3. Comparativo das Ferramentas

Usando a linguagem Java, a ferramenta SonarQube detectou somente três dos oito mutantes injetados durante a etapa de mutação das aplicações. A ferramenta PMD, por sua vez, detectou apenas um dos oito mutantes inseridos no projeto em Java. Desse modo, utilizando as configurações padrão das ferramentas, de acordo com o contexto deste es-

tudo a eficácia verificada do SonarQube é maior do que a do PMD para a linguagem de programação Java para defeitos que utilizam os padrões de mutantes utilizados neste estudo. Usando a linguagem JavaScript, a ferramenta SonarQube detectou apenas um dos oito mutantes inseridos. Enquanto o PMD, em dois casos, apresentou um problema a menos, ou seja, não foi capaz de identificar o problema na mutação inserida e mascarando problemas preexistentes. Desse modo, a eficácia do SonarQube também é maior do que a do PMD em relação à linguagem de programação JavaScript, levando em consideração as configuração padrão das ferramentas.

Analisando somente os resultados das análises estáticas, pode-se afirmar que a ferramenta SonarQube é superior ao PMD. Entretanto, ambas ferramentas possuem a possibilidade do seu utilizador implementar novas regras e adicioná-las ao conjunto padrão, ou até mesmo remover alguma validação habilitada inicialmente. Assim, somente essa análise não seria completa o suficiente para escolher entre as duas ferramentas em qualquer caso. Desse modo, foram analisados outros aspectos capazes de influenciar na escolha de qual ferramenta de análise utilizar. Em relação ao relatório gerado o SonarQube, apresentou uma solução mais intuitiva e completa, sem necessidade de redirecionamentos e identificando problemas no projeto como um todo. Um exemplo dessa análise de projeto é o percentual de código duplicado. Além disso, o SonarQube se mostrou superior ao aceitar também outros códigos JavaScript, como por exemplo utilizando o *framework* ReactJS, que não é compatível com a análise do PMD.

Assim, caso o desenvolvedor necessite realizar a análise estática de um código utilizando algum *framework frontend*, é importante considerar uma ferramenta com suporte a tal tecnologia, como é o caso do SonarQube para o ReactJS. Apesar dos pontos citados, o PMD também possui a possibilidade de criação de regras personalizadas por projeto, além de apresentar uma solução mais leve ao usuário. Caso o desenvolvedor possua alguma limitação de *hardware*, por exemplo, o PMD com regras personalizadas pode mitigar o problema.

6. Considerações Finais

Partindo das análises realizadas é possível verificar que a ferramenta SonarQube teve uma eficácia melhor na detecção de problemas. Tendo em vista que no projeto Java detectou mais problemas que a ferramenta PMD e após a inserção dos mutantes o SonarQube foi capaz de detectar ainda mais defeitos. Mesmo que no projeto em Javascript a ferramenta PMD tenha detectado mais problemas, após a inserção de mutantes a ferramenta dectou menos problemas do que na versão original. Desso modo, o PMD apresentou um comportamento inesperado, provendo menos segurança na análise de projetos Javascript que o SonarQube. Entretanto, existem diversas ferramentas de análise estática para esta finalidade, desse modo, como trabalho futuro adicionar mais ferramentas e avaliar com mais linguagens de programação é essencial para verificar a evolução das ferramentas de análise estática de código.

References

Bochmann, G. V. and Petrenko, A. (1994). Protocol testing: review of methods and relevance for software testing. In *Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, pages 109–124.

Delamaro, M., Jino, M., and Maldonado, J. (2013). *Introdução ao teste de software*. Elsevier Brasil.

Delamaro, M., Pezze, M., Vincenzi, A. M. R., and Maldonado, J. C. (2001). Mutant operators for testing concurrent java programs. In *Proceedings of the Brazilian Symposium on Software Engineering*, pages 272–285. Citeseer.

DeMillo, R. A., Lipton, R. J., and Sayward, F. G. (1978). Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41.

Ebenau, R. G. and Strauss, S. H. (1993). *Software inspection process*. McGraw-Hill, Inc.

Emanuelsson, P. and Nilsson, U. (2008). A comparative study of industrial static analysis tools. *Electronic notes in theoretical computer science*, 217:5–21.

Fatima, A., Bibi, S., and Hanif, R. (2018). Comparative study on static code analysis tools for c/c++. In *2018 15th International Bhurban Conference on Applied Sciences and Technology (IBCAST)*, pages 465–469. IEEE.

Li, P. and Cui, B. (2010). A comparative study on software vulnerability static analysis techniques and tools. In *2010 IEEE international conference on information theory and information security*, pages 521–524. IEEE.

Melo, S. M. (2009). Inspeção de software. *University of São Paulo: São Carlos, SP*.

Mills, H. D., Dyer, M., and Linger, R. C. (1987). Cleanroom software engineering.

Mirshokraie, S., Mesbah, A., and Pattabiraman, K. (2015). Guided mutation testing for javascript web applications. *IEEE Transactions on Software Engineering*, 41(05):429–444.

PMD (2021). Documentação pmd - versão 6.34. `https://pmd.github.io/latest/index.html`. Accessed: 2021-05-16.

Raggett, D., Le Hors, A., Jacobs, I., et al. (1999). Html 4.01 specification. *W3C recommendation*, 24.

Sonarqube (2021). Documentação sonarqube - versão 8.9. `https://docs.sonarqube.org/latest/`. Accessed: 2021-05-16.

Torchiano, M., Ricca, F., and Marchetto, A. (2011). Are web applications more defect-prone than desktop applications? *International journal on software tools for technology transfer*, 13(2):151–166.

Wang, M.-H. and Lee, C.-S. (2008). An intelligent ppqa web services for cmmi assessment. In *2008 Eighth International Conference on Intelligent Systems Design and Applications*, volume 1, pages 229–234. IEEE.

Webb, J. (1990). Static analysis (software testing). In *IEE Colloquium on Software Testing for Critical Systems*, pages 4–1. IET.

Zheng, J., Williams, L., Nagappan, N., Snipes, W., Hudepohl, J. P., and Vouk, M. A. (2006). On the value of static analysis for fault detection in software. *IEEE transactions on software engineering*, 32(4):240–253.