Análise da Influência de Políticas de Proteção de *Branch* na Qualidade de *Software* em *Pull Requests* de Repositórios de Código Aberto

Camila Silva Campos[1]

[1]Bacharelado em Engenharia de *Software*
Instituto de Ciências Exatas e Informática - PUC Minas
Rua Cláudio Manoel, 1.162, Funcionários, Belo Horizonte – MG - Brasil

camila.campos.1216314@sga.pucminas.br

Abstract. *Code review is a process that seeks to improve software quality by evaluating a set of changes. This activity can be based on the Pull Requests (PR) model, allowing discussion and review of proposed changes. For approved PR, it is possible to define branch protection policies to be followed before merge. Therefore, the goal of this work is to analyze the influence of the use of branch protection policies during the merge of code changes on the occurrence of code quality problems related to maintainability, reliability and security. To this end, data from 207 GitHub open source repositories are mined and analyzed. The results show that PRs that follow at least one of the protection policies tend to include less Code Quality Issue (CQI), and it is still possible that they help in the removal of previously added CQI.*

Resumo. *A revisão de código é um processo que busca a melhoria da qualidade de software por meio da avaliação de um conjunto de alterações. Essa atividade pode ser baseada no modelo de Solicitações de Pull (PR), com discussão e revisão das mudanças propostas. Para as PR aprovadas, é possível definir políticas de proteção de branch a serem seguidas antes do merge. Sendo assim, o objetivo deste trabalho é analisar a influência do uso de políticas de proteção de branch durante o merge de alterações de código, na ocorrência de problemas de qualidade de código relacionados à manutenibilidade, confiabilidade e segurança. Para isso, são minerados e analisados dados de 207 repositórios de código aberto do GitHub. Os resultados mostram que PR que seguem pelo menos uma das políticas de proteção tendem a incluir menos Problemas de Qualidade de Código (CQI), e ainda é possível que auxiliem na remoção de CQI adicionados previamente.*

Bacharelado em Engenharia de *Software* - PUC Minas
Trabalho de Conclusão de Curso (TCC)

Orientador de conteúdo (TCC I): Marco Rodrigo Costa - mrcosta@pucminas.br
Orientador acadêmico (TCC I): Laerte Xavier - laertexavier@pucminas.br
Orientador do TCC II: Rodrigo Baroni - baroni@pucminas.br

Belo Horizonte, 06 de dezembro de 2022.

1. Introdução

A revisão de código é um processo de avaliação de um conjunto de alterações por parte de integrantes da equipe de desenvolvimento de *software* diferentes daqueles que submeteram a modificação. A partir da análise da contribuição, o código revisado pode receber comentários e alterações do autor ou outros interessados, até que seja integrado ou não à base de código. O principal objetivo desse processo é a melhoria da qualidade das contribuições e do *software* [Bosu et al. 2017, Thongtanunam and Hassan 2021]. A qualidade de um sistema computacional pode ser definida de maneiras distintas com base no alvo da entrega do produto. Para os usuários, a qualidade é focada na facilidade de uso e conformidade com os requisitos. Já para desenvolvedores, qualidade significa modularização adequada, encapsulamento de funcionalidades, entre outros [Tonella and Abebe 2008]. Essas definições de qualidade são apresentadas respectivamente como qualidade externa e interna do *software* [ISO/IEC 2014].

O processo de revisão é uma prática extensamente recomendada e utilizada no contexto de desenvolvimento de sistemas [McIntosh et al. 2014]. Um mecanismo utilizado para possibilitar a revisão de um novo código são os *branches*, linhas de trabalho independentes que se originam de uma base de código central [Radigan 2022]. A criação de *branches* permite que os desenvolvedores colaborem em projetos de forma paralela sem afetar o desenvolvimento principal. Posteriormente, a nova linha de trabalho passa pelo processo de *merge* para ser integrada ao *branch* de origem [Atlassian 2022, Zou et al. 2019]. Nesse sentido, diferentes políticas de proteção de *branch* podem definir exigências a serem seguidas antes que a alteração possa ser integrada ao restante do sistema. Por exemplo, um projeto pode estabelecer uma quantidade mínima de aprovações em uma alteração ou exigir revisão por parte de proprietários do código [McIntosh et al. 2016]. No entanto, pouco se sabe sobre a relação da qualidade de código com essas diferentes políticas [Kudrjavets et al. 2022]. Sendo assim, o problema que este estudo busca solucionar é **a carência de trabalhos que analisam as diferentes políticas de proteção de *branch*, e o impacto das mesmas na qualidade interna do *software***.

O número de repositórios de *Software* de Código Aberto (OSS, do inglês *Open Source Software*) ativos no GitHub teve um aumento significativo entre 2019 e 2020 [GitHub 2020]. Além disso, esses projetos são utilizados em milhões de outros repositórios [GitHub 2019]. Por outro lado, projetos OSS de grandes organizações tiveram uma alta porcentagem de alterações de códigos aprovadas com problemas não identificados pelos seus revisores [Kononenko et al. 2015]. Estudos apontam que, além da cobertura de revisão, outras propriedades importantes e pouco conhecidas possuem impacto na qualidade de código, visto que componentes com alta taxa de cobertura ainda se apresentaram propensos a erros [McIntosh et al. 2014]. Sendo assim, justifica-se esse trabalho pela importância dos projetos e do processo analisado, e pela possibilidade de diminuição de defeitos durante alterações de código por meio da identificação de outros fatores relacionados à qualidade no processo de revisão.

Além da revisão de código ser um elemento essencial em projetos de desenvolvimento de sistemas de *software* maduros, é considerada também uma das práticas mais eficazes para controle de qualidade [Kononenko et al. 2015]. Sendo assim, considerando que o uso das políticas de proteção de *branch* está contido nesse processo, o objetivo geral

deste trabalho é **analisar a influência do uso de políticas de proteção de *branch* durante o *merge* de alterações de código, na ocorrência de problemas de qualidade de código relacionados à manutenibilidade, confiabilidade e segurança**. Têm-se como objetivos específicos: i) caracterizar o uso dessas políticas nos repositórios; ii) investigar o impacto do uso dessas políticas na ocorrência de Problemas de Qualidade de Código (CQI, do inglês *Code Quality Issues*); iii) avaliar a relevância do uso dessas políticas na diminuição de CQI.

O resultado esperado deste estudo é uma análise de como as diferentes políticas de proteção influenciam na qualidade de *software*. Para isso, foram analisados os processos de revisão e *merge* de alterações, assim como a qualidade interna das mudanças, em projetos relevantes na comunidade de código aberto. Observou-se que o uso de pelo menos uma política impacta positivamente na qualidade do código alterado, auxiliando, principalmente, na manutenibilidade do sistema.

O restante deste trabalho está estruturado da seguinte forma: a Seção 2 apresenta detalhes sobre os conceitos pertinentes ao estudo; a Seção 3 descreve os trabalhos relacionados ao tema abordado; a Seção 4 apresenta os materiais e métodos utilizados; a Seção 5 expõe os resultados e análises; a Seção 6 apresenta a discussão e as ameaças à validade; e a Seção 7 apresenta a conclusão e trabalhos futuros.

2. Fundamentação Teórica

Para melhor compreender os conceitos que compõem a solução deste estudo, esta seção apresenta os fundamentos teóricos de Qualidade de *Software* e Revisão de Código.

2.1. Qualidade de *Software*

A ISO/IEC 25000:2014 define qualidade de *software* como a capacidade do programa de computador em satisfazer as necessidades declaradas e implícitas, quando usado sob condições especificadas [ISO/IEC 2014]. A norma ISO/IEC 25010:2011 define dois modelos de qualidade: modelo de qualidade do produto, relacionado com propriedades estáticas e dinâmicas do *software*; e modelo de qualidade em uso, relacionado ao resultado da interação quando um produto é utilizado em determinado contexto de uso.

O modelo de qualidade de produto combina os modelos de qualidade interno e externo definidos na ISO/IEC 9126-1:2001 e se compõe em oito características principais, apresentadas da Figura 1. Os atributos em destaque são os enfatizados neste trabalho. Cada uma das sub-características definidas na figura podem ser medidas por atributos internos, que buscam medir as propriedades estáticas do *software* em relação à sua arquitetura - como o nível de acoplamento; e as externas, que derivam-se da análise do comportamento do sistema - como a quantidade de defeitos encontrados em testes [IEC 2011].

No contexto de projetos OSS, a avaliação da qualidade interna é a mais relevante para o sucesso do projeto, tendo em vista que eles recebem contribuições de diferentes desenvolvedores localizados em diferentes regiões do mundo. Portanto, atributos internos, como compreensão do código e a capacidade de autodocumentação, são fatores-chave para garantia da qualidade adequada nesses sistemas [Tonella and Abebe 2008, Lu et al. 2016]. Entende-se por qualidade interna a definição da ISO/IEC 25000, que a apresenta como os atributos estáticos de um produto de *software* que satisfaz as necessidades declaradas e implícitas quando o produto de *software* é usado em condições

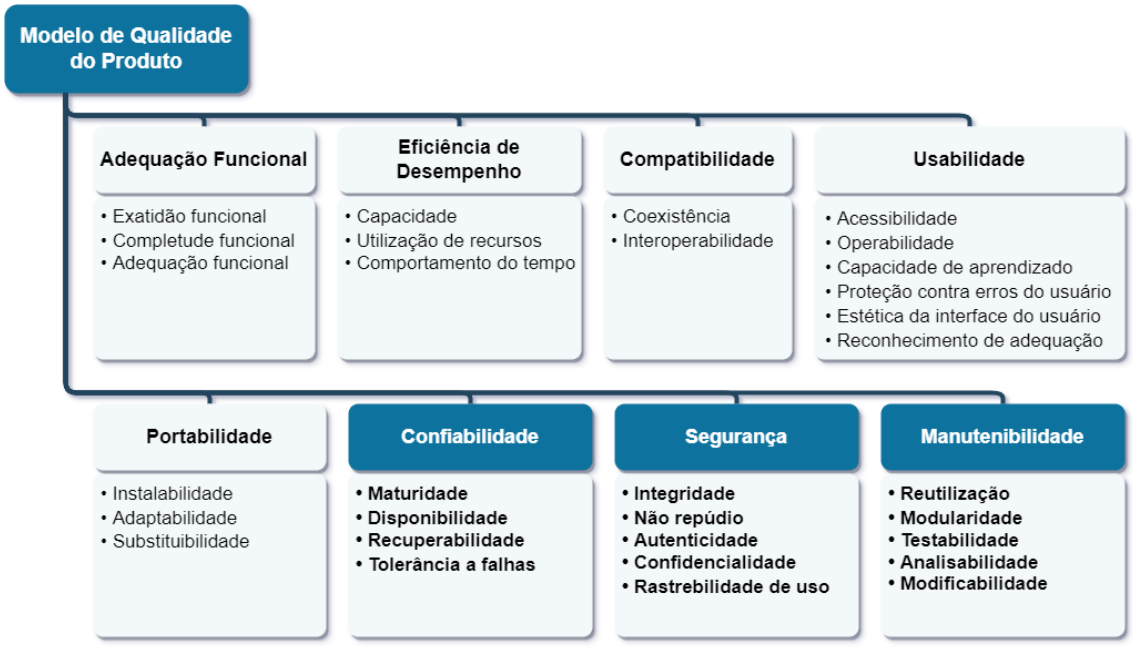

Figura 1. Modelo de Qualidade do Produto Definido na ISO/IEC 25010:2011

especificadas. Por medida interna de qualidade de *software*, define-se o grau em que um conjunto de atributos estáticos de *software* satisfazem essas necessidades. Os atributos estáticos, como a complexidade, podem ser verificados por meio de revisão, inspeção ou ferramentas automatizadas [ISO/IEC 2014].

As medições internas de qualidade podem impactar nas externas. As medidas de código, por exemplo, estão fortemente relacionadas com a manutenibilidade em sistemas de *software* orientados a objetos [Li and Henry 1993]. Já as de complexidade, coesão, acoplamento e tamanho, apresentam maior coerência com a característica de confiabilidade e manutenibilidade [Jabangwe et al. 2015, Al Dallal 2013]. Medir características de qualidade com base em atributos e métricas internas pode tornar as métricas mais precisas, confiáveis e menos dispendiosas, visto que esses atributos e métricas são independentes do ambiente no qual o sistema de *software* opera [Santos et al. 2016].

2.2. Revisão de Código

A revisão de código é considerada uma estratégia eficaz para encontrar e corrigir defeitos de *software* antes de integrar um conjunto de alterações propostas na base de código principal [Shimagaki et al. 2016]. A revisão pode possuir diferentes processos e contextos. Alguns deles são revisão baseada em ferramentas, como o Gerrit[1]; e o modelo de desenvolvimento baseado em Solicitações de *Pull* (PR, do inglês *Pull Requests*), usado no GitHub [Sadowski et al. 2018]. Ambas ferramentas permitem configurar um conjunto personalizado de regras de proteção de *branch*, como a quantidade e função dos revisores, e a permissão de *merge* automático das mudanças aceitas.

O modelo baseado em PR suportado pelo GitHub incorpora ferramentas leves de revisão de código a cada solicitação de alteração, que permitem a discussão e revisão

[1]Disponível em: `https://www.gerritcodereview.com`. Último acesso: 10/06/2022.

das mudanças propostas. Durante esse processo, outros contribuidores podem adicionar novos códigos, sugerir modificações ou realizar comentários. Por fim, caso as alterações sugeridas sejam aprovadas, elas poderão passar pelo processo de *merge* ao *branch* principal do projeto [GitHub 2022b]. No entanto, ainda é possível que tenham sido definidas regras de proteção para o *branch* de destino, que exigem verificações antes da realização do *merge*. A proteção de um *branch* no GitHub pode ser definida para um ou para todos os *branches* de um repositório. A política de proteção pode ser alterada a qualquer momento, logo, as PR de um mesmo repositório podem ter sido verificadas com base em diferentes regras.

Para este trabalho, foram utilizadas as regras de proteção de *branch* oferecidas pelo GitHub por ser a maior e mais popular plataforma de hospedagem de código para controle e colaboração de versões [Sedhain and Kuttal 2022, Xue et al. 2019]. Acredita-se que, para esta pesquisa, as regras que devem apresentar maior impacto na qualidade do *software* são: 1) exigência de revisão de código, que garante a permissão de *merge* de uma PR somente após a aprovação de algum revisor; 2) quantidade de aprovações necessárias, que pode ser definida entre uma e seis no GitHub; 3) exigência de fechamento de todos os comentários realizados pelos revisores, indicando que as questões levantadas foram resolvidas.

3. Trabalhos Relacionados

Os trabalhos relacionados discutidos nesta seção envolvem a análise de atributos internos de qualidade em projetos da comunidade OSS, relacionados a atributos da dinâmica de revisão de código em PR, tais como: características dos contribuidores, histórico de alterações, *feedback* e avaliação dos revisores. Os trabalhos são apresentados em ordem cronológica.

No estudo de Lu et al. (2016), são analisados aspectos internos de qualidade de código das contribuições de colaboradores casuais em projetos OSS. Foram selecionados 21 projetos do GitHub por meio da Interface de Programação de Aplicativos (API, do inglês *Application Programming Interface*) da plataforma. Os repositórios foram filtrados: 1) pelo uso da linguagem de programação Java, JavaScript e Python; 2) pelo grau de popularidade do projeto dentro do *site*, na sua respectiva linguagem; e 3) por um número mínimo e máximo de *commits*. Os projetos buscados tiveram seus dados relacionados a *commits*, participantes e número de estrelas extraídos. Para análise da qualidade do código, foi utilizada a ferramenta SonarQube[2], integrada com o *plugin* Git[3], que permite a detecção automática de CQI introduzidos pelos *commits*, assim como o armazenamento de informações do autor daquela alteração. Os resultados obtidos mostram que mais problemas, e de maior gravidade, foram introduzidos por colaboradores casuais. Além disso, o artigo apresenta as categorias de CQI mais frequentemente introduzidas nos projetos, pelos colaboradores informais. A metodologia apresentada no estudo é usada como base para este trabalho.

Xue et al. (2019) propuseram uma forma de correção automatizada de CQI em projetos OSS. O trabalho se dedica à análise de atributos internos de qualidade, do ponto de vista dos desenvolvedores. O estudo objetiva a geração de correções de *patches* de

[2]Disponível em: `https://www.sonarqube.org`. Último acesso: 15/05/2022.

[3]Disponível em: `https://git-scm.com/`. Último acesso: 22/05/2022.

CQI, por meio da mineração de padrões de correção. Esses padrões foram extraídos usando a ferramenta GumTree. O estudo utiliza a API do GitHub para coleta de projetos escritos em Java; e a ferramenta SonarQube, para detecção e coleta de correções de CQI no histórico de alterações dos repositórios. Foram extraídas cerca de 30 mil instâncias de correção de CQI e coletados 68 padrões de correções comuns para 56 tipos de problemas. Utilizando a métrica Densidade de CQI para medir qualidade de código, foi descoberto que projetos de grande escala possuem menos problemas. A metodologia utilizada do estudo se relaciona com este trabalho quanto à análise de qualidade, e ao processo e características dos projetos utilizados.

Han et al. (2020) analisaram a violação de convenções de codificação ao longo do processo de revisão de código. Essa pesquisa contribuiu com uma investigação de quais e quantas violações são introduzidas e removidas durante as revisões de código. Para análise das correções e seus respectivos comentários, foram buscados do conjunto de dados denominado CROP, metadados de projetos escritos em Java. Para identificação das violações de convenção, foi utilizada a ferramenta de análise estática de código Checkstyle. Os resultados obtidos mostram que o número de violações aumenta proporcionalmente ao crescimento da base de código. Além disso, foi possível notar que os desenvolvedores ainda verificam manualmente a existência de violações de convenções durante a revisão de código, e que essa prática pode causar atraso na inspeção [Han et al. 2020]. O artigo se relaciona com este trabalho por estudar a qualidade de código com base em processos e projetos similares. No entanto, esse estudo e o presente trabalho se diferem na ferramenta de coleta dos repositórios e nos atributos de qualidade analisados.

Thongtanunam e Hassan (2021) examinaram e caracterizaram a relação da dinâmica de revisão de código com a propensão a defeitos de um *patch*. Para isso, foi realizado um estudo empírico sobre os projetos de código aberto OpenStack e Qt. Para caracterização da dinâmica, foram analisadas métricas de dados históricos dos *patches*. Após a identificação das dinâmicas relevantes, foi examinada a associação entre as métricas de dinâmica definidas e probabilidade de defeito em um *patch*. Os resultados indicam que a decisão de avaliação de um revisor é impactada pelas informações visíveis sobre uma alteração em revisão. Por exemplo, a probabilidade de um voto positivo aumenta à medida que crescem: 1) a quantidade de revisões da mesma composição de autor e revisor; e 2) a proporção de votos anteriores favoráveis. No entanto, essa influência na decisão de avaliação de um *patch* tem pouco impacto na qualidade do *software*. O artigo se compara com este por relacionar a qualidade de código com a dinâmica utilizada no processo de revisão, mas se diferem na escolha das dinâmicas e atributos de qualidade analisados.

Recentemente, Khoshnoud et al. (2022) identificaram e caracterizaram erros não encontrados durante a revisão de código de PR. Para isso, foi utilizado o conjunto de dados SmartShark para coletar dados de PR após o *merge*. A base de dados utilizada é composta por metadados de 77 projetos do GitHub. Três analistas realizaram uma investigação manual nas PR, rotulando os que apresentavam problemas. Ao fim da coleta, os *bugs* foram categorizados e tiveram sua distribuição calculada. Foram encontradas 224 PR problemáticas de um total de 3.261. As categorias de erros mais frequentemente perdidos em revisões de código são: semântica, construção e verificação de análise, com distribuições de 51,34%, 15,5% e 9,08%, respectivamente. A categoria de memória foi a

menos encontrada, com apenas 1,6% de distribuição. Este trabalho busca complementar os resultados obtidos no artigo, analisando se as políticas de proteção de *branch* diminuem a ocorrência de *bugs* não percebidos durante o processo de revisão.

4. Materiais e Métodos

A pesquisa proposta é do tipo quantitativa, tendo em vista que realiza a mineração e coleta de dados para análise do impacto das regras de proteção de *branch* na qualidade de código [Wohlin et al. 2012]. São obtidos dados de repositórios de código aberto relevantes e algumas de suas respectivas PR para realização desta investigação. Nesta seção, são apresentadas as etapas para alcance dos objetivos propostos, sendo descritos os materiais e métodos para coleta, filtro e análise dos dados.

4.1. Conjunto de Dados

Para coleta de dados, é desenvolvido um *script* que faz uso da API GraphQL do GitHub. Inicialmente, são buscados 50 projetos de código aberto com alta popularidade, e no máximo 20 PR para cada projeto para proporcionar a diversidade dos dados. A popularidade é medida pelo número de estrelas acima de 100, que corresponde ao número de usuários do GitHub que manifestaram interesse no projeto [Xue et al. 2019]. São selecionados apenas repositórios que contenham Java como a linguagem principal, visto que é uma das linguagens de programação mais populares [Silveira et al. 2021]. Além disso, são selecionadas apenas PR que já tenham passado pelo processo de *merge*, priorizando as mais recentes e com alteração em pelo menos um arquivo de extensão .java. Ainda é realizado um filtro manual dos repositórios para remover projetos pessoais, de documentações ou tutoriais.

A coleta das informações é realizada por meio da biblioteca Axios[4] devido à familiaridade da autora. Os dados são armazenados em arquivos de formato de Valores Separados por Vírgula (CSV, do inglês *Comma-Separated Values*), com o auxílio da biblioteca CSV Writer[5]. É observada uma limitação de detecção das regras de proteção de *branch* vigentes durante o *merge* das PR por meio da API do GitHub, pois, para não administradores dos repositórios, apenas as regras atuais de cada *branch* são acessíveis. Sendo assim, como forma alternativa e segura para coleta das regras seguidas pelas PR durante o *merge*, assume-se que uma PR segue determinada política, caso sejam encontradas em seus metadados as seguintes características [GitHub 2022a]:

1. Política de exigência de revisão de código: é verificado se a PR contém pelo menos uma revisão de código aprovada antes do *merge*;
2. Política de quantidade mínima de aprovações necessárias: é buscada a quantidade de aprovações da PR realizadas antes do *merge*;
3. Política de exigência de fechamento de todos os comentários realizados pelos revisores: é verificado se todos os comentários de revisão criados antes do *merge* da PR estão com o status *resolved*.

Ao fim da primeira coleta de dados, é observada uma predominância de PR que não seguem a política de exigência de revisão, e poucos PR que possuem dois ou mais

[4]Disponível em: `https://axios-http.com`. Último acesso: 15/05/2022.

[5]Disponível em: `https://www.npmjs.com/package/csv-writer`. Último acesso: 15/05/2022.

revisores. Além disso, nota-se que a maior parte das PR coletadas não possuem nenhum comentário de revisão e, portanto, não é possível inferir se o repositório segue a política de exigência de fechamentos dos comentários. Sendo assim, são realizados ajustes no *script* e adicionados novos filtros para coletar apenas PR que possuam pelo menos um comentário de revisão. Além disso, essas PR são distribuídas em 6 arquivos diferentes, representando grupos de combinações das políticas para possibilitar a coleta de dados de forma balanceada, sendo eles:

- Grupo 1: PR que não seguem nenhuma das políticas;
- Grupo 2: PR que seguem a política de exigência de revisão de código e a política de quantidade mínima de aprovações igual a 1;
- Grupo 3: PR que seguem a política de exigência de revisão de código e a política de quantidade mínima de aprovações maior ou igual a 2;
- Grupo 4: PR que seguem a política de exigência de revisão de código, política de quantidade mínima de aprovações igual a 1, e política de exigência de fechamento dos comentários de revisão;
- Grupo 5: PR que seguem a política de exigência de revisão de código, política de quantidade mínima de aprovações maior ou igual a 2, e política de exigência de fechamento dos comentários de revisão;
- Grupo 6: PR que seguem apenas a política de exigência de fechamento dos comentários de revisão;

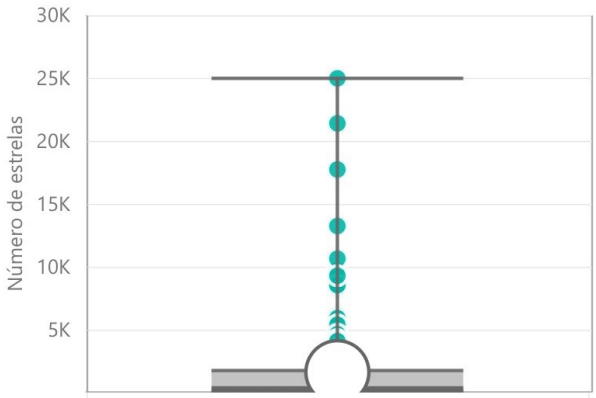

Figura 2. Quantidade de estrelas por repositórios

Ainda é observado que parte das PR coletadas inicialmente, são descartadas posteriormente devido a limitações na etapa de análise da qualidade do código. Considerando isso, é necessário coletar uma quantidade maior de PR; no entanto, por uma limitação de busca aninhada da API GraphQL do GitHub, é necessário dividir a coleta em duas etapas: a primeira, para coleta dos repositórios; e a segunda, para coleta das PR. Com base no nome dos repositórios coletados na segunda versão do *script*, são extraídas 55 palavras-chave que indicam a possibilidade de ser um repositório pessoal, de curso ou documentação, como *book*, *tutorial*, *guide* e *introduction*. É contabilizado um total de 2.973 repositórios após o filtro com base nas palavras-chave. Desses, apenas 207 passaram para a etapa de análise da qualidade de código ou foram bem-sucedidos nela. A Figura 2 apresenta um *boxplot* com o número de estrelas dos repositórios após terem sido removidos os *outliers*. Observa-se que o número médio de estrelas é 1.674, e a mediana 587.

Para cada um dos repositórios coletados, são buscadas as PR aplicando os mesmo filtros da primeira tentativa de coleta, no entanto, sem limitar a quantidade por repositório, para alcance de uma maior quantidade de dados do que os coletados inicialmente. As informações registradas das PR são: URL de acesso, nome e número de estrelas do repositório, identificador e nome da PR, identificador do *commit* gerado após o *merge* da PR, quantidade de linhas adicionadas e removidas, quantidade de linhas alteradas, além das informações referentes às regras de proteção de *branch*, citadas anteriormente. Para a metodologia proposta, apenas PR que acrescentem um único *commit* no *branch* de origem devem ser consideradas; logo, as PR que não possuíam o identificador do *commit* gerado após o *merge* são descartadas. Ao todo, são coletadas cerca de 20.000 PR, porém, a maior parte foi filtrada durante a etapa de análise da qualidade do código, ou não obtiveram sucesso durante esse processo. A Figura 3 apresenta a distribuição das 2.180 PR que chegaram ao final da análise nos 6 agrupamentos de políticas definidos anteriormente.

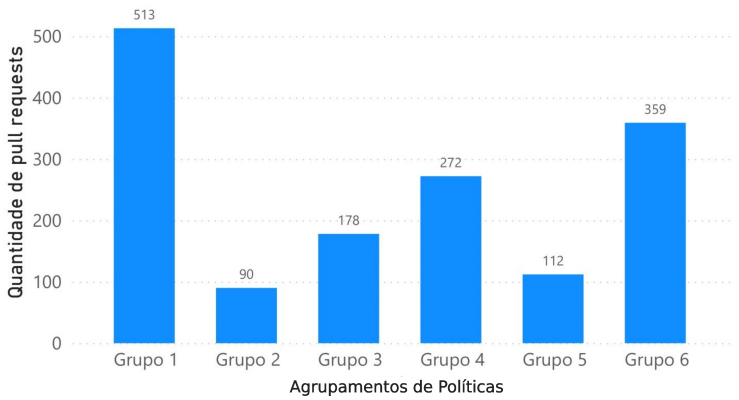

Figura 3. Quantidade de PR por agrupamento de políticas

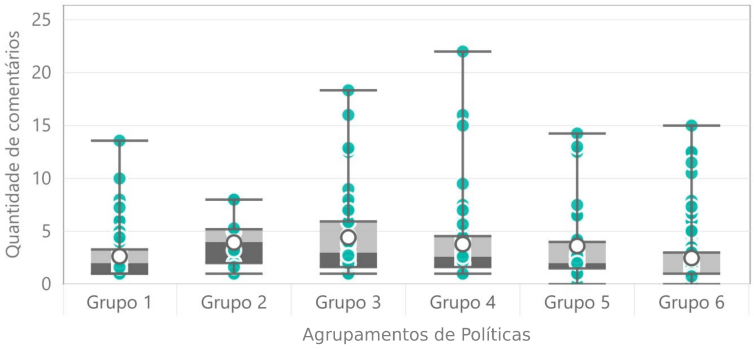

Figura 4. Quantidade de comentários por agrupamento de PR

É possível observar pela Figura 3 que a maior parte das PR coletadas não seguem nenhuma das políticas, ou seguem apenas a política de exigência de resolução dos comentários. A Figura 4 mostra que os grupos 4, 5 e 6, que seguem a política de resolução dos comentários, têm as menores médias e medianas do número de comentários por PR, logo após o Grupo 1.

A Figura 5 mostra que a média de linhas alteradas de cada um dos grupos está entre 154 do Grupo 6, e 253 do Grupo 3; e a mediana entre 251 do Grupo 2, e 89 do Grupo 5.

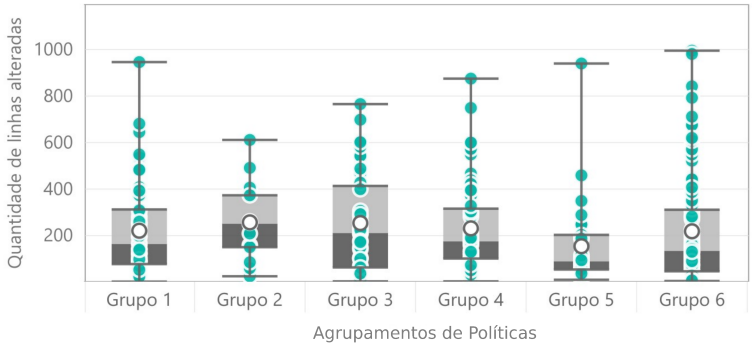

Figura 5. Número de linhas alteradas por agrupamento de PR

O desvio padrão das médias de cada grupo é de 41,5 linhas e, removendo o Grupo 5 que obteve maior diferença na média, o desvio padrão cai para apenas 21 linhas. O desvio padrão das medianas, por sua vez, é de 57 linhas. Sendo assim, pode-se considerar que os grupos possuem características similares quanto a quantidade de linhas alteradas, com exceção do Grupo 5, que apresenta a menor média e mediana.

4.2. Análise da Qualidade de Código

Para identificação de novos CQI introduzidos pelas alterações de uma PR, é necessário analisar duas versões do código: com e sem as alterações propostas. Sendo assim, para acessar a versão de código contendo as alterações da PR, é utilizado o comando *git reset –hard <commit>* na *branch* de *merge* da PR. Para acessar um *commit* anterior, para visualizar a versão de código sem as alterações da PR, é executado o comando *git reset –hard <commit>∼1*. Para cada uma dessas versões de código, é executado o comando *git clean -nfxd* para limpar qualquer resquício de arquivos sem rastreamento, e o comando *mvn compile sonar:sonar* no diretório raiz do repositório para iniciar a análise de código pelo SonarQube.

Após a execução com sucesso da ferramenta de análise estática de código, é buscado o status do relatório daquela análise por meio da API do SonarQube durante um período de tempo, até que o status seja alterado para *success*. Logo em seguida, as métricas da análise são acessadas também por meio da API da ferramenta e armazenadas em arquivos CSV. A cada análise, a ferramenta SonarQube detecta os CQI, que podem ser classificados como vulnerabilidade, *bug* ou *code smell* [Lu et al. 2016, Xue et al. 2019]. Cada CQI pode receber uma das cinco gravidades [SonarSource 2022]:

- BLOCKER: *bug* com alta probabilidade de impactar o comportamento da aplicação e deve ser corrigido imediatamente.
- CRITICAL: *bug* com média probabilidade de impactar o comportamento da aplicação e deve ser revisado imediatamente.
- MAJOR: falha de qualidade com probabilidade de causar alto impacto na produtividade do desenvolvedor.
- MINOR: falha de qualidade com probabilidade de causar médio impacto na produtividade do desenvolvedor.
- INFO: apenas um achado, que não representa uma falha de qualidade ou um *bug*.

As métricas de qualidade armazenadas são a quantidade de vulnerabilidades, *bugs* ou *code smell*, e a quantidade de CQI de cada uma das cinco gravidades. Para cada gravidade de problemas, é calculada a métrica Densidade de Problemas de Qualidade de Código (CQID, do inglês *Code Quality Issue Density*), que representa o número de CQI introduzidos por linha de código alterada [Xue et al. 2019]. No Apêndice A, é apresentado como as métricas de CQI são detectadas pelo SonarQube.

5. Resultados e Análises

Nesta seção, são apresentados os resultados obtidos e as análises realizadas neste estudo. Os valores de CQID encontrados variaram entre zero, maior e menor que zero, representando PR que não apresentaram variação, adicionaram ou diminuíram a quantidade de CQI no código, respectivamente. Dessa forma, os resultados foram analisados considerando dois grupos de PR: as que incluíram e as que mantiveram ou diminuíram a quantidade de CQI.

5.1. Preparação e Caracterização dos Dados

É possível que as PR de um mesmo repositório sigam diferentes combinações de políticas de proteção de *branch*. Além disso, ao seguir a política de exigência de revisão de código, a PR obrigatoriamente segue também a política de quantidade mínima de aprovações. Logo, para análise dos resultados, as PR foram separadas em novos agrupamentos, considerando apenas a política de exigência de revisão de código e a política de exigência de fechamento de todos os comentários. Os novos agrupamentos foram: 1) PR sem nenhuma política; 2) PR com uma única política; 3) PR com duas políticas. Dessa forma, para cada um desses grupos foi criado um arquivo CSV para armazenamento dos dados identificadores da PR, bem como o grupo que pertence e o respectivo cálculo de CQID. Para visualização de dados, foi utilizada a ferramenta Power BI[6] por possuir licença gratuita e ser desenvolvida pela Microsoft, empresa reconhecida pela Gartner como líder em plataformas de análise e inteligência de negócios [Kronz et al. 2022]. Já a análise de dados e testes estatísticos foram realizados por meio do *software* estatístico de código aberto Jamovi[7] devido a sua gratuidade e suporte aos cálculos estatísticos usados neste trabalho.

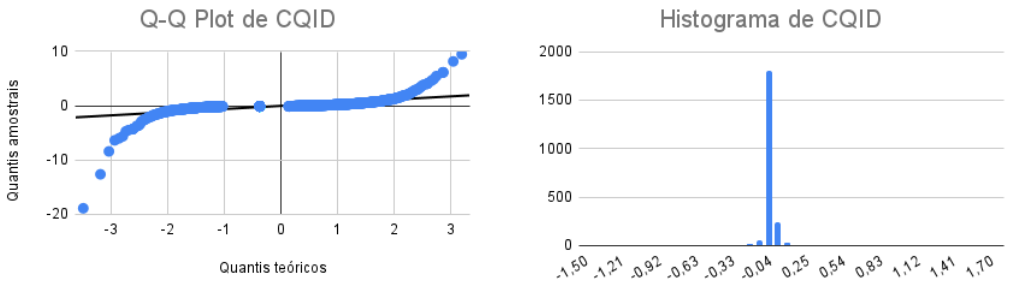

Figura 6. Métodos gráficos para verificar a normalidade dos dados de CQID

A Figura 6 mostra que, assim como no trabalho de Lu et al. (2016), os dados de CQID deste trabalho não seguem uma distribuição normal, visto que, no gráfico Q-Q Plot

[6]Disponível em: `https://powerbi.microsoft.com`. Último acesso: 15/05/2022.
[7]Disponível em: `https://www.jamovi.org`. Último acesso: 15/05/2022.

apresentado, os dados não se concentram em torno da reta de tendência e, no gráfico de histograma, os dados se concentram em um único grupo e se desviam da normalidade. Portanto, utilizou-se o teste não paramétrico de Kruskal-Wallis, usado na comparação de três ou mais amostras independentes. Dessa forma, foi possível examinar a existência de diferença de CQID entre pelo menos dois dos três grupos definidos. Para o caso de observação de diferença entre os grupos, foi realizado o teste U de Mann-Whitney com os pares de grupos para identificar quais são diferentes entre si.

5.2. Análise do Aumento de CQI

Considerando apenas as PR que tiveram um aumento de CQI, foi realizado o teste de Kruskal-Wallis para os três agrupamentos de PR, definidos com base na quantidade de políticas seguidas (zero, uma ou duas). Foi obtido um *p-value* igual a 0.046, que representa o nível de significância estatística [Patil and Yaligar 2017]. Dessa forma, pode-se afirmar que há diferença estatisticamente significativa entre algum dos três grupos analisados, considerando um nível de significância de 95%. Executando o teste estatístico de Mann-Whitney com o mesmo nível de significância entre os pares de grupos, foi identificada diferença de CQID entre dois deles: os que não seguiam nenhuma política com os que seguiam uma política, e os que não seguiam nenhuma política com os que seguiam duas políticas. Porém, entre o grupo que seguia uma política e o que seguia duas políticas não foi encontrada diferença significativa. Dessa forma, foi executado o teste Mann-Whitney considerando apenas dois grupos: os que não seguiam nenhuma política e os que seguiam uma ou mais políticas. Para este último teste, foi encontrado um *p-value* igual a 0.01, que demonstra diferença significativa entre os grupos.

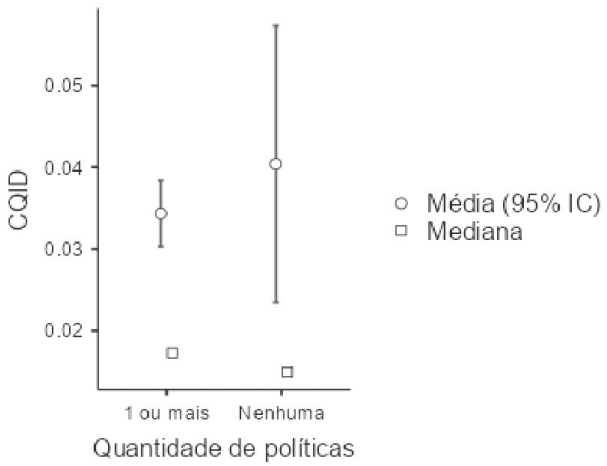

Figura 7. Boxplot do aumento de CQID por quantidade de políticas

Pela Figura 7 é possível observar que PR que não seguiam nenhuma política apresentaram valor de média maior do que os grupos que seguiam pelo menos uma política. A mediana do grupo que não seguia nenhuma política foi de 0.014, e a do grupo que seguia pelo menos uma política foi de 0.017. Apesar da mediana do grupo que seguia uma ou mais políticas ser pouco maior, o desvio padrão desse grupo foi de apenas 0.05, enquanto o desvio do grupo que não seguia nenhuma política foi de 0.13, indicando maior variação de dados e apresentando os maiores valores de CQID. Como mostrado na Figura 8, dos cinco níveis de gravidade de CQID coletados pelo SonarQube, três tiveram média mais

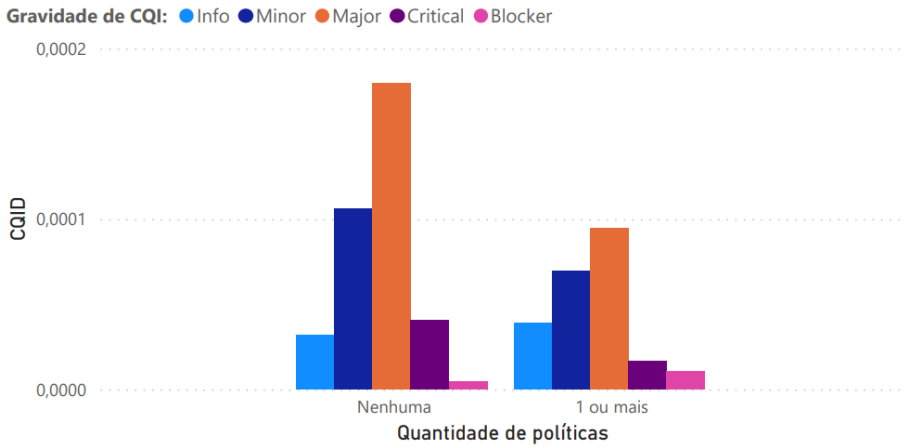

Figura 8. Média do aumento de CQID por quantidade de políticas e gravidade de CQI

alta no grupo de PR que não seguiam nenhuma política, sendo elas *minor*, *major* e *critical*. Além disso, realizando o teste estatístico de Mann-Whitney novamente para os valores de CQID de CQI do tipo *code smell* e *bug* individualmente, foram encontrados valores de p iguais a 0.01 e 0.8, respectivamente. Portanto, considerando o nível de significância de 0.05 é possível afirmar que há diferença estatisticamente significativa entre os valores de CQID positivos para *code smell* entre os diferentes grupos de PR, mas não para *bug*. Os valores de CQID do tipo vulnerabilidade foram descartados nesta análise devido à baixa quantidade de PR que tiveram aumento nesse tipo de CQI (menos de 5).

5.3. Análise da Diminuição ou Conservação de CQI

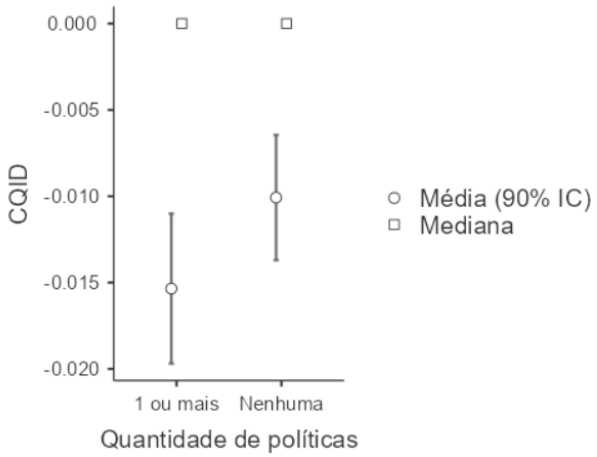

Figura 9. Boxplot da diminuição ou conservação de CQID por quantidade de política

Assim como na análise do aumento de CQID, as PR que apresentaram diminuição ou conservação de CQID foram reagrupados considerando apenas dois grupos: o grupo contendo as PR que não seguiam nenhuma política, e o grupo contendo as PR que seguiam uma ou mais políticas. Para esse agrupamento, o teste de Mann-Whitney apresentou um *p-value* igual a 0.089 que, diminuindo o nível de significância para 90%, pode-se

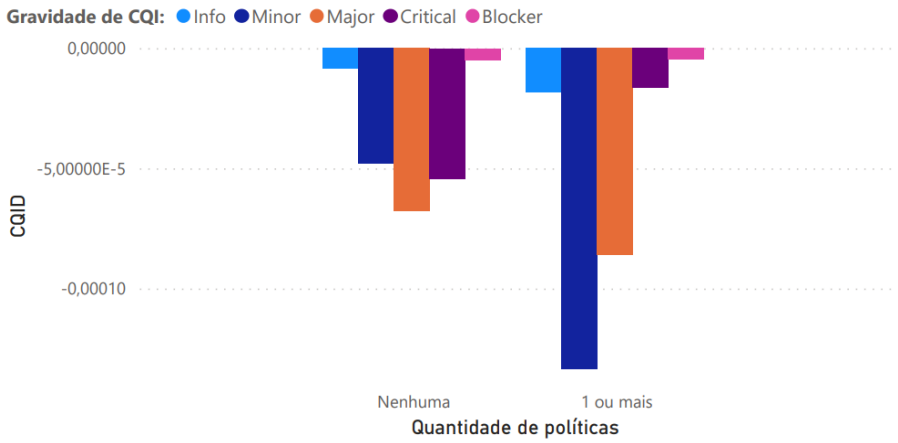

Figura 10. Média da diminuição de CQID por quantidade de política e gravidade de CQI

considerar diferença estatística entre os grupos. A Figura 9 mostra que a mediana dos dois grupos foram iguais, porém, a média do grupo de PR que seguia pelo menos uma política foi menor que o grupo que não seguia nenhuma política, sendo, respectivamente, -0.0154 e -0.0101. A porcentagem de PR nas quais não foram observadas alterações na quantidade de CQI para o grupo que não seguia nenhuma política foi de 44%, e para o grupo que seguia pelo menos uma política foi de 38%, indicando uma tendência maior de diminuição de CQI nesse último grupo. Combinado a isso, ainda é possível observar na Figura 10 que o grupo de PR que seguia uma ou mais políticas teve menor média de CQID em três dos cinco níveis de gravidade de CQI, sendo: *info*, *minor* e *major*.

6. Discussão e Ameaças à Validade

Com base nos resultados encontrados, pode-se inferir que PR que seguem pelo menos uma das políticas de proteção de *branch* tratadas neste artigo tendem a incluir menos CQI no código do que PR que não seguem nenhuma política; em especial, para as gravidades *minor*, *major* e *critical*. Considerando os três tipos de CQI buscados pelo SonarQube e as análises estatísticas realizadas para cada um deles, é possível afirmar que há menor chance de incluir *code smells* ao código com o uso das políticas de proteção de *branch*. Porém, não é possível inferir nenhum impacto do uso das políticas em relação a CQI do tipo *bug* ou vulnerabilidade. Além disso, os resultados sugerem que PR que seguem pelo menos uma das políticas aumentem a chance de diminuição de CQI, principalmente para os classificados entre os três primeiros níveis de gravidades (*info*, *minor* e *major*). No entanto, não foi notada diferença significativa nos valores de CQID entre o grupo de PR que seguiam apenas uma política do grupo que seguiam duas política s.

Sendo assim, recomenda-se que as organizações do GitHub que utilizam o processo de revisão de código incorporado à ferramenta sigam pelo menos uma das políticas de proteção de *branch*. Dessa forma, é possível garantir que alterações ou novas implementações no código estejam menos sujeitas à inserção de problemas de qualidade, em especial no que diz respeito à manutenibilidade do código. Considerando os aspectos de segurança e confiabilidade, representados por meio dos CQI de vulnerabilidade e *bug*, respectivamente, entende-se que apenas o uso das políticas de proteção não é suficiente para evitá-los. Portanto, sugere-se que o uso das políticas combinado com outras ações

durante o processo de desenvolvimento de software impacte mais significativamente esses atributos da qualidade de código, como a adesão de outras políticas de proteção de *branch* não abordadas neste trabalho, ou a integração com ferramentas adicionais de revisão.

Este trabalho apresenta algumas ameaças à validade, sendo possíveis riscos durante o planejamento e execução do estudo [Claes Wohlin 2012]. Como ameaças à validade interna, pode-se citar os falso positivos, que são uma questão comum em ferramentas de análise estática de código, em que a análise apresenta um problema de qualidade que, na verdade, não existe. Como validade externa, este estudo contempla apenas projetos escritos na linguagem Java; logo, as conclusões apresentadas não devem ser estendidas para outras linguagens de programação. Deve-se considerar ainda que, por limitações da coleta de dados, as políticas de proteção de *branch* tratadas neste trabalho refere-se a um mecanismo de verificação realizada durante a revisão de código de uma PR, que se baseia nas regras de proteção de *branch* do GitHub, portanto, não necessariamente são regras configuradas nos repositórios analisados.

Ademais, a metodologia inicialmente proposta para realização deste estudo apresentou limitações devido às características dos projetos analisados e do método de coleta dos dados das PR. Dessa forma, foi necessário repensar a estratégia de análise de qualidade e o método de coleta para viabilizar este trabalho. Portanto, recomenda-se que para trabalhos que se estendam deste estudo, estas questões sejam consideradas durante ajustes na metodologia que facilitem a execução da pesquisa, como a escolha da linguagem dos repositórios.

7. Conclusão e Trabalhos Futuros

Neste estudo, foi investigado o impacto do uso de políticas de proteção de *branch* oferecidas pelo GitHub na qualidade de código de 2.180 projetos OSS. Foi descoberto que PR de repositórios que seguem pelo menos uma das políticas de proteção propostas evita significativamente a adição de novos CQI ao código em comparação aos repositórios que não seguem nenhuma das políticas. Além disso, é possível que a adoção de pelo menos uma das políticas ajude a diminuir os CQI já existentes no código. Este trabalho mostra que as gravidades de CQI mais encontradas foram *major* e *minor*, respectivamente. Portanto, complementam-se os resultados encontrados no trabalho de Lu et al. (2016), em que, entre os grupos de contribuintes principais e casuais de repositórios de código aberto, as gravidades de CQI mais encontradas também foram *major* e *minor*. Assim como no trabalho de Lu et al. (2016) a ocorrência de CQI dessas gravidades são diminuídas em PR de contribuintes principais, elas também são diminuídas em PR que seguem pelo menos uma das políticas de proteção de *branch* apontadas neste trabalho.

Entre os tipos de CQI analisados, apenas os *code smells* tiveram diminuição significativa entre o grupo de PR que seguiam uma ou nenhuma política. Portanto, é possível afirmar que o uso das políticas de proteção de *branch* auxiliam na manutenibilidade de código, mas não é possível inferir o mesmo para segurança e confiabilidade. No entanto, assim como apresentado no trabalho de Khoshnoud et al. (2022), frequentemente *bugs* não são percebidos durante a revisão de código. Logo, como trabalho futuro, sugere-se a busca de mais PR, considerando diferentes linguagens de programação e outras políticas de proteção de *branch*, para verificar mais a fundo se as políticas tendem a impactar na ocorrência de *bugs* e vulnerabilidades, afetando a confiabilidade e segurança do software,

respectivamente.

Pacote de Replicação

O pacote de replicação deste trabalho encontra-se disponível em: `https://github.com/ICEI-PUC-Minas-PPLES-TI/plf-es-2022-1-tcci-5308100-pes-camila-campos`

Referências

Al Dallal, J. (2013). Object-oriented class maintainability prediction using internal quality attributes. *Information and Software Technology*, 55(11):2028–2048.

Atlassian (2022). Git merge. https://www.atlassian.com/git/tutorials/using-branches/git-merge.

Bosu, A., Carver, J. C., Bird, C., Orbeck, J., and Chockley, C. (2017). Process aspects and social dynamics of contemporary code review: Insights from open source development and industrial practice at microsoft. *IEEE Transactions on Software Engineering*, 43(1):56–75.

Claes Wohlin, Per Runeson, M. H. M. C. O. B. R. A. W. (2012). *Experimentation in Software Engineering*. Springer Berlin, Heidelberg, 1 edition.

GitHub (2019). The state of the octoverse. https://octoverse.github.com/2019. Último acesso: 10/06/2022.

GitHub (2020). The state of the octoverse. http://oss.x-lab.info/github-insight-report-2020-en.pdf. Último acesso: 10/06/2022.

GitHub (2022a). About protected branches. https://docs.github.com/en/enterprise-server@3.4/repositories/configuring-branches-and-merges-in-your-repository/defining-the-mergeability-of-pull-requests/about-protected-branches. Último acesso: 06/09/2022.

GitHub (2022b). About pull requests. https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests. Último acesso: 10/06/2022.

Han, D., Ragkhitwetsagul, C., Krinke, J., Paixao, M., and Rosa, G. (2020). Does code review really remove coding convention violations? In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 43–53.

IEC, I. (2011). *25010: 2011-Systems and Software Engineering-Systems and software Quality Requirements and Evaluation (SQuaRE)-System and Software Quality Models*. ISO/IEC JTC 1/SC 7 Software and systems engineering.

ISO/IEC (2014). *ISO/IEC 25000: 2014. Systems and Software Engineering—Systems and Software Quality Requirements and Evaluation (SQuaRE)—Guide to SQuaRE*. ISO/IEC JTC 1/SC 7 Software and systems engineering.

Jabangwe, R., Börstler, J., Šmite, D., and Wohlin, C. (2015). Empirical evidence on the link between object-oriented measures and external quality attributes: a systematic literature review. *Empirical Software Engineering*, 20(3):640–693.

Khoshnoud, F., Rezaei Nasab, A., Toudeji, Z., and Sami, A. (2022). Which bugs are missed in code reviews: An empirical study on smartshark dataset. In *19th International Conference on Mining Software Repositories (MSR 2022 Challenge Track)*.

Kononenko, O., Baysal, O., Guerrouj, L., Cao, Y., and Godfrey, M. W. (2015). Investigating code review quality: Do people and participation matter? In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 111–120.

Kronz, A., andJulian Sun, K. S., Pidsley, D., and Ganeshan, A. (2022). Magic quadrant for analytics and business intelligence platforms. https://www.gartner.com/doc/reprints?id=1-292LEME3&ct=220209&st=sb.

Kudrjavets, G., Kumar, A., Nagappan, N., and Rastogi, A. (2022). Mining code review data to understand waiting times between acceptance and merging: An empirical analysis. *arXiv preprint arXiv:2203.05048*.

Li, W. and Henry, S. (1993). Object-oriented metrics that predict maintainability. *Journal of systems and software*, 23(2):111–122.

Lu, Y., Mao, X., Li, Z., Zhang, Y., Wang, T., and Yin, G. (2016). Does the role matter? an investigation of the code quality of casual contributors in github. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, pages 49–56.

McIntosh, S., Kamei, Y., Adams, B., and Hassan, A. E. (2014). The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Proceedings of the 11th working conference on mining software repositories*, pages 192–201.

McIntosh, S., Kamei, Y., Adams, B., and Hassan, A. E. (2016). An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, 21(5):2146–2189.

Patil, N. S. and Yaligar, M. F. (2017). Analysis of linear relation between p-value and co relational value using r programming. In *2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*, pages 983–987.

Radigan, D. (2022). Branching strategy: A path to greatness. https://www.atlassian.com/agile/software-development/branching.

S.A, S. (2022). Sonarqube documentation. https://docs.sonarqube.org/9.6/.

Sadowski, C., Söderberg, E., Church, L., Sipko, M., and Bacchelli, A. (2018). Modern code review: A case study at google. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 181–190.

Santos, D., Resende, A., Junior, P. A., and Costa, H. (2016). Attributes and metrics of internal quality that impact the external quality of object-oriented software: A systematic literature review. In *2016 XLII Latin American Computing Conference (CLEI)*, pages 1–12.

Sedhain, A. and Kuttal, S. K. (2022). Information seeking behavior for bugs on github: An information foraging perspective. In *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 1–3.

Shimagaki, J., Kamei, Y., Mcintosh, S., Hassan, A. E., and Ubayashi, N. (2016). A study of the quality-impacting practices of modern code review at sony mobile. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 212–221.

Silveira, P., Mannan, U. A., Almeida, E. S., Nagappan, N., Lo, D., Singh Kochhar, P., Gao, C., and Ahmed, I. (2021). A deep dive into the impact of covid-19 on software development. *IEEE Transactions on Software Engineering*, pages 1–1.

SonarSource (2022). Issues. https://docs.sonarqube.org/latest/user-guide/issues. Último acesso: 10/06/2022.

Thongtanunam, P. and Hassan, A. E. (2021). Review dynamics and their impact on software quality. *IEEE Transactions on Software Engineering*, 47(12):2698–2712.

Tonella, P. and Abebe, S. L. (2008). Code quality from the programmer's perspective. In *Proceedings of Science, XII Advanced Computing and Analysis Techniques in Physics Research, Erice, Italy*, volume 5, page 153.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M., Regnell, B., and Wesslén, A. (2012). *Experimentation in Software Engineering*. Computer Science. Springer Berlin Heidelberg.

Xue, J., Mao, X., Lu, Y., Yu, Y., and Wang, S. (2019). History-driven fix for code quality issues. *IEEE Access*, 7:111637–111648.

Zou, W., Zhang, W., Xia, X., Holmes, R., and Chen, Z. (2019). Branch use in practice: A large-scale empirical study of 2,923 projects on github. In *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*, pages 306–317.

A. Apêndice A. Detecção das Métricas de CQI pelo SonarQube

Neste apêndice é apresentado como as métricas de CQI são detectadas pelo SonarQube. Essa ferramenta de análise de código é composta por quatro elementos: analisadores, servidores, *plugins* instalados no servidor e base de dados. Os analisadores são os responsáveis pela análise do código, executada linha a linha, tendo como base algumas regras padrões para a linguagem do código analisado. A ferramenta ainda permite a criação de um conjunto de regras personalizadas [S.A 2022]. Os CQI identificados pela ferramenta são:

1. *Bug*: problema de codificação que pode levar a um erro ou comportamento inesperado em tempo de execução. Algumas das regras padrões utilizadas para detecção desse CQI são: 1) as recursões e iterações não devem ser infinitas; 2) as expressões regulares devem ser sintaticamente válidas; 3) o valor zero não deve ser um possível denominador.
2. Vulnerabilidade: um ponto em que o código está aberto a ataques. Exemplos de regras utilizadas pelo SonarQube para interpretação de uma linha de código como vulnerável são: 1) as consultas de banco de dados não devem ser vulneráveis a ataques de injeção; 2) os analisadores de Linguagem de Marcação Estendida (XML, do inglês *Extensible Markup Language*) não devem ser vulneráveis a ataques de Entidade Externa XML (XXE, do inglês *XML external entity*); 3) as classes não devem ser carregadas dinamicamente.

3. *Code smell*: problema de código que o torna confuso e difícil de manter. A correção desse problema é considerada como uma otimização de código. Exemplos de regras utilizadas para detecção desse problema são: 1) os retornos dos métodos não devem ser invariáveis; 2) futuras palavras-chave não devem ser usadas como nomes; 3) a complexidade cognitiva dos métodos não deve ser alta.