Um Estudo dos *Frameworks* Java Mais Populares sob a Ótica da Qualidade de Testes

Altino Alves Júnior[1], **Letícia de Souza Meireles**[1]

[1]Instituto de Ciências Exatas e Informática - PUC Minas
Rua Cláudio Manoel, 1.162 - 30140-100 - Belo Horizonte, MG – Brasil

{aajunior,leticia.meireles.124503}@sga.pucminas.br

Abstract. *Frameworks are widely used by developers to facilitate software development. Like any other code, they also need to be tested and quality assured. Therefore, they may suffer from the presence of test smells, which are a sign of poor test design and low quality. Little is discussed about how the quality of test codes influences on the framework's use. Thus, this study seeks to understand how the quality of the test suite frameworks can influence the emergence of problems for its users. For this, this work performs a descriptive study that explores the Java projects of frameworks available on GitHub. After the analysis, it was observed that as new versions are released, the frameworks increase the amount of test smells, making this issue a threat to the quality of the systems. Also, a greater recurrence of the test smells Assertion Roulette and Lazy Test was identified, while the less recurrent ones are Empty Test and Constructor Initialization.*
Keywords: *test smells, software quality, frameworks, tests*

Resumo. *Frameworks são amplamente utilizados por desenvolvedores para facilitar o desenvolvimento de software. Como qualquer outro código, eles também precisam ser testados e ter sua qualidade garantida. Sendo assim, podem sofrer com a presença de test smells, que são um sinal de design ruim e baixa qualidade do teste. Com este problema em questão, este trabalho realiza um estudo descritivo que explora os projetos de frameworks Java disponíveis no GitHub sob a ótica dos test smells. Após a análise, observou-se que ao decorrer dos lançamentos de novas versões, os frameworks aumentam relativamente a quantidade de test smells, tornando-se esta questão uma ameaça para a qualidade dos sistemas. Também, identificou-se maior recorrência dos test smells Assertion Roulette e Lazy Test, enquanto os menos recorrentes são Empty Test e Constructor Initialization.*
Palavras-chave *test smells, qualidade de software, frameworks, testes*

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

Orientador de conteúdo (TCC I): Cleiton Tavares - cleitontavares@pucminas.br
Orientadora de conteúdo (TCC I): Simone de Assis - simone@pucminas.br
Orientador acadêmico (TCC I): Laerte Xavier - laertexavier@pucminas.br
Orientador do TCC II: Cleiton Tavares - cleitontavares@pucminas.br

Belo Horizonte, 14 de maio de 2023.

1. Introdução

A utilização de *frameworks* tem o objetivo de auxiliar o desenvolvedor a criar *software* mais rapidamente [Gajewski and Zabierowski 2019]. *Frameworks* dão suporte ao reúso de código, pois fornecem uma moldura de código para a aplicação [Pressman and Maxim 2021]. Assim como qualquer outro programa, o código-fonte de *frameworks* precisa ter sua qualidade validada por testes, com a função de garantir a estabilidade do código sob condições extremas [Wu et al. 2022]. O teste de *software* é um processo projetado para verificar se o código executa o que foi determinado e, por outro lado, não faça o que não foi estipulado. Assim como o código, testes são suscetíveis à qualidade de escrita e podem possuir *test smells*, definidos como um sinal de *design* ruim e má implementação ou escrita do teste [Wu et al. 2022]. Uma ferramenta que pode ser utilizada para saber se *test smells* estão afetando o uso do *framework* são as *issues*, que permitem a comunicação e acompanhamento de um trabalho entre desenvolvedores e usuários de um projeto [GitHub 2022, Pipinellis 2015]

Geralmente, nota-se que ao longo do tempo, a qualidade dos códigos de testes pode deteriorar, podendo ocasionar a introdução de *test smells* [Greiler et al. 2013]. Alguns trabalhos estudam a relação dos *test smells* com a qualidade do *software*, como, por exemplo, a eliminação de *test smells* em projetos *open-souce* [Wu et al. 2022]. Apesar disso, ainda não está claro na literatura estudos que identificam a presença de *test smells* no código-fonte de teste de *frameworks* e como isso pode impactar o código de quem utiliza os utiliza. Problemas como tempo desnecessário procurando por um *bug* em seu código, sendo que o problema está no *framework*, motivam a conduzir este estudo. Considerando que *frameworks* podem ser a base estrutural de aplicações e que os *test smells* indicam a presença de testes de baixa qualidade, o problema a ser tratado por este trabalho é: **como a qualidade da suíte de testes dos *frameworks* pode influenciar no surgimento de problemas em seu código-fonte?**

A resolução desse problema é de crescente importância, uma vez que busca entender como os *test smells* influenciam a qualidade dos códigos de testes e como isso afeta o desenvolvimento final de produtos de *software* quando há utilização de *frameworks*. Jorge et al. (2021) revelam que dos onze projetos *open-source* analisados no estudo, os quais são desenvolvidos na linguagem JavaScript, todos contam com *test smells*, sendo respectivamente de 33% a 100% dos conjuntos de testes infectados. Projetos que evoluem e dispõe de constante colaboração da comunidade estão mais suscetíveis a incidência de *test smells*, como por exemplo React e Babel [Jorge et al. 2021]. Ademais, medir o impacto de *test smells* na qualidade do código pode prover um guia para refatorar e realizar manutenção do código [Wu et al. 2022].

Assim, **o objetivo geral deste trabalho é analisar se o surgimento de problemas no código-fonte estão vinculados à qualidade da suíte de testes dos *frameworks***. O presente estudo tem como objetivos específicos responder as seguintes questões: 1) Os *frameworks* apresentam menos *test smells* ao decorrer dos lançamentos de novas versões? 2) Quais *frameworks* tendem a ter menos *test smells*? 3) Quais são os *test smells* mais e menos recorrentes nos códigos de testes dos *frameworks*? 4) As *issues* de *bugs* refletem relação com a qualidade dos códigos de teste?.

Como resultado foi encontrado que a quantidade de *test smells* aumentou no decorrer dos lançamentos de novas versões e, ainda, que os *frameworks* com menos *test smells*

são *darmiel/eeee, OptiJava/Optilog-Client, victordiaz/PHONK, spring-projects/spring-vault* e *cofcool/chaos-server*. Os tipos de *test smells* com maior ocorrência foram o *Assertion Roulette* e o *Lazy Test*. Além disso, não foi possível confirmar que o surgimento de problemas para o desenvolvedor durante a utilização do *framework* está vinculada com a qualidade da suíte de testes.

O presente trabalho é organizado em sete seções. A Seção 2 apresenta o referencial teórico e os principais conceitos técnicos abordados neste trabalho. A Seção 3 apresenta os trabalhos relacionados. A Seção 4 detalha a metodologia utilizada, os materiais e métodos. A seção 5 especifica os resultados encontrados através da caracterização dos dados e da discussão dos mesmos. A Seção 6 apresenta as ameaças à validade e, por fim, a Seção 7 apresenta a conclusão e sugestões de trabalhos futuros.

2. Fundamentação Teórica

Esta seção detalha os principais conceitos técnicos abordados neste trabalho, respectivamente: Testes de *Software*, Qualidade de Testes, de *Test Smell* e Desenvolvimento Dirigido a *Frameworks*.

2.1. Testes de *Software*

Teste de *software* é uma forma sistemática de tentar buscar defeitos em um sistema [Haiderzai and Khattab 2019]. Nesse contexto, testes têm como finalidade verificar se um programa realiza o que foi projetado para fazer e também descobrir problemas do *software* antes de ser utilizado [Pressman and Maxim 2021]. Quando um programa é testado, dados hipotéticos são utilizados e os resultados dos testes são analisados para verificar se há anomalias, erros ou informações sobre atributos não funcionais do programa. [Pressman and Maxim 2021].

A necessidade de criação de casos de teste torna-se cada vez mais reconhecida como fator para ser garantida a qualidade do *software*. Desse modo, maior atenção deve ser dada à qualidade de *software*, o que contribuirá para serem levantados requisitos abrangentes sobre desempenho e funcionalidades, sem que seja esquecida a importância da estabilidade e da segurança. Testar o *software* é um elemento chave para o processo de garantia da qualidade [Virgínio et al. 2019]. Todos esses requisitos podem ser garantidos e testados por testes de *software*, o que, por si só, evidencia sua importância [Zhao et al. 2021].

Vale destacar que há cinco tipos de testes: unitário, de integração, de sistemas, de regressão e de aceitação [Hooda and Chhillar 2015]. O teste unitário é o que testa a menor parte do código, enquanto o teste de integração visa testar a comunicação entre vários módulos do programa para garantir que o fluxo entre os componentes esteja correto. Testes de sistema servem para garantir que o sistema inteiro se comporta e funciona conforme esperado. Testes de regressão possuem o objetivo de confirmar que o programa não quebrou após consertos de *bugs*. Já testes de aceitação, tem o objetivo de coletar *feedbacks* do cliente final do *software* e saber se o mesmo consegue utilizar as funcionalidades [Hooda and Chhillar 2015].

2.2. Qualidade e *Test Smells*

Testar o *software* é uma atividade chave para garantir a qualidade de *software* em circunstâncias complexas, acarretando a criação de muitos códigos de testes. Esses testes

são produzidos manualmente e são difíceis de serem mantidos. Desenvolvedores tendem a ignorar a manutenção do código de testes, resultando em falta de refatoração adequada dos planos de testes e, consequentemente, o código de teste é mais suscetível à problemas de qualidade [Wu et al. 2022].

Tendo em vista o código de testes, uma das formas de identificar a qualidade do código de teste construído é através da detecção de *test smells*, definido como um sinal de *design* ruim e má implementação dos testes [Wu et al. 2022]. Os *test smells* podem ser detectados por heurísticas e regras pré-definidas [Wu et al. 2022]. A presença de *test smells* pode causar muitos problemas e provocar graves custos para empresas de *software*. Em 2016, um *post* no *blog* oficial da Google sobre testes mencionou que quase 16% dos testes da Google, ou seja, um em cada sete testes escritos pela empresa, ocasionalmente falham por motivos não relacionados à mudança no código de produção ou no código de testes [Garousi et al. 2018].

Os *test smells* possuem similaridades entre seus vários tipos. Eles podem ser classificados em uma das seguintes categorias: 1) relacionados à execução e comportamento do teste; 2) relacionados à semântica ou lógica; 3) relacionados aos passos (*setup*, exercitar, verificar e derrubar); 4) relacionados às dependências e 5) relacionados à duplicação de código [Garousi et al. 2018]. Garousi et al. (2019) apresenta um catálogo contendo 86 tipos de *test smells*, um exemplo é o *Mystery Guest*. Este ocorre quando um método de teste utiliza recursos externos [Wu et al. 2022]. Todavia, neste trabalho serão analisados 21 tipos de *test smells*, como por exemplo o *Assertion Roulette*. Sua ocorrência sucede-se quando um método de teste verifica vários métodos do objeto testado, tornando os testes mais dependentes um do outro e de difícil manutenção [Wu et al. 2022]. A Tabela 8 do Apêndice A contém um descritivo de cada um dos tipos de *test smells* analisados.

2.3. Desenvolvimento Dirigido a *Frameworks*

Dentro das disciplinas de Engenharia de Software, a construção de *software* é uma das mais complexas [Edwin 2014]. A maioria dos sistemas criados implementam, em partes, algo que já foi construído e tendem a seguir modelos arquiteturais conhecidos [Edwin 2014]. Isso ocorre através da utilização de *frameworks* como Spring-Boot, Spring-Framework e Restlet-Framework-Java, que proporcionam uma moldura de arquitetura para o *software* e possibilita o reúso de classes, seja de forma direta ou através da herança e do polimorfismo [Sommerville 2018]. Um fator benéfico na utilização de *frameworks* é a simplificação do ambiente de desenvolvimento de *software*, o que permite que desenvolvedores concentrem seus esforços nos requisitos do projeto, ao invés de lidar com bibliotecas e funções repetitivas [Edwin 2014].

3. Trabalhos Relacionados

Os trabalhos relacionados apresentados nesta seção abrangem os contextos da qualidade de *software* baseados nos testes, onde se explora essa vertente através da detecção e análise de *test smells*. Também se relacionam a este estudo, trabalhos focados em ferramentas para detecção de *test smells*, bem como o uso de *smells* para identificar problemas relacionado à qualidade das suítes de testes.

Jorge et al. (2021) propõem um estudo empírico para analisar a ocorrência de *test smells* em projetos de *software*, bem como compreender se sua presença está cor-

relacionada com a qualidade dos códigos de testes. Os autores desenvolveram a ferramenta STEEL para detecção dos *smells*, utilizando-a nas suítes de testes de 11 projetos *open-source* desenvolvidos na linguagem de programação JavaScript. Observa-se que, os projetos analisados contam com cerca de 33% a 100% dos conjuntos de testes infectados. Além disso, os *smells* mais frequentes são *Duplicate Assert*, *Magic Number Test*, *Unknown Test* e *Conditional Test Logic*.

Kim (2020) propõe um estudo relacionado a manutenção de *test smells*. Para isso, o autor selecionou 88 sistemas *open-source*, desenvolvidos na linguagem de programação Java, que possuiam mais de 1.000 casos de testes e utilização em ambientes comerciais. Dessa forma, foram quantificados alguns tipos de *test smells*, os quais foram adicionados e removidos durante toda evolução dos sistemas, visando compreender qualitativamente a motivação e maneira como o desenvolvedor os mitigam. Em relação aos resultados, observa-se que, os *smells Assertion Roulette*, *Conditional Test Logic* e *Unknown Test* possuem maior taxa de edição. Além disso, a adição e melhorias de funcionalidades motivam a refatoração do teste, porém os *test smells* associados persistem.

Aljedaani et al. (2021) analisam 22 ferramentas para detecção de *test smells*. Os autores realizaram uma comparação dos tipos de *test smells* identificados por cada ferramenta, bem como as estratégias de detecção. Nos resultados, nota-se que as ferramentas detectam *test smells* em suítes de testes nas linguagens de programação Java, Scala, Smalltalk e C++. Além disso, é demonstrado que grande parte das ferramentas sobrepõe-se na detecção de tipos específicos de *smells*, como o *General Fixture*, por exemplo. Dessa forma, o estudo atua como uma fonte única para seleção da ferramenta apropriada para sua necessidade, além de ser um guia para o desenvolvimento de futuras ferramentas.

Virgínio et al. (2019) investigam as correlações e influência dos *test smells* na cobertura do teste. Para isso, os autores introduziram o JNose, uma ferramenta responsável pela detecção de *test smells*, analisando, assim, 11 projetos *open-source* desenvolvidos na linguagem de programação Java, que utilizam Maven e tenham testes unitários implementados usando o JUnit. Dessa maneira, são detectados 21 tipos de *test smells* e 10 métricas de cobertura de testes. Em relação aos resultados, nota-se que o *test smell Lazy Test* foi o de maior recorrência em todos os projetos.

Camara et al. (2021) sugerem um estudo para analisar uma maneira diferente de prever *flaky tests* a partir de *test smells* em testes de regressão. Para realização do estudo, foram treinados modelos baseados em duas bases de dados: uma com *test cases* de um mesmo projeto e outra de projetos distintos. Assim, cada modelo executava um algoritmo diferente do executado pelos outros modelos. Como resultado foi identificado que os *test smells Sleep Test* e *Constructor Initialization* estão mais correlacionados a ocorrência de *flaky tests*. Além disso, os autores comprovaram que a *performance* de modelos baseados em vocabulário são melhores em relação aos modelos baseados em *test smells*. Contudo, no cenário com base de dados de diferentes repositórios, modelos baseados em *test smells* demonstraram maior eficácia.

Embora nem todos os estudos analisam *frameworks*, os trabalhos relacionados apresentam abordagens relevantes ao presente estudo. Os trabalhos propostos por Jorge et al. (2021), Kim (2021) e Virgínio et al. (2019) se relacionam ao presente artigo, trazendo perspectivas e objetivos semelhantes de analisar a qualidade do *software* baseando-se em

test smells, bem como sua incidência. O estudo de Aljedaani et al. (2021) se assemelha sendo um guia para o desenvolvimento da metodologia além da seleção das ferramentas adequadas para detecção de *test smells*. Por fim, Camara et al. (2021) retrata problemas no código que podem ser relacionados e detectados através da presença de *test smells*, além de trazer modelos adequados para detecção em diferentes cenários.

4. Materiais e Métodos

Este estudo propõe comparar e apresentar os *frameworks* mais populares para projetos na linguagem de programação Java no âmbito da qualidade de testes. Esta pesquisa é um estudo descritivo, já que estuda detalhadamente os *test smells* nestas ferramentas através da coleta de dados, análise e interpretação dos mesmas. Além disso, também é um estudo quantitativo, uma vez que através de métricas busca-se identificar a influência de *test smells* no surgimento de problemas para os usuários do *framework*. Ainda, realiza-se uma análise das *issues* que relatam *bugs* e problemas para verificar a relação entre a qualidade dos testes dos *frameworks* e o surgimento dos *bugs*. No restante dessa seção, são apresentados os métodos aplicados, ferramentas empregadas e etapas necessárias para a execução do estudo proposto.

4.1. Procedimentos

O estudo proposto é dividido em quatro estágios. O primeiro consiste na seleção de repositórios dos *frameworks* desenvolvidos na linguagem de programação Java. O segundo corresponde à análise da suíte de testes do código-fonte dos *frameworks* selecionados, identificando a provável existência de *test smells*. O terceiro consiste na coleta das *issues* dos repositórios selecionados na etapa anterior. Por fim, o quarto contempla a compilação e análise estatística de todos os dados coletados previamente.

A coleta de repositórios e suas *issues* ocorrem através da API GraphQL do GitHub[1]. Compreende-se a Interface de Programação de Aplicação (API, do inglês *Application Program Interface*) como uma área de interação de um *software*, proporcionando a comunicação e integração com outros sistemas, viabilizando o acesso aos recursos e dados, dispensando o conhecimento aprofundado dos detalhes técnicos da implementação [Cabral et al. 2022]. Dessa forma, a API GraphQL do GitHub proporciona maior flexibilidade e capacidade de definição dos dados desejados. Assim, por meio de especificações, é possível escolher e obter dados entre os diversos recursos disponíveis na plataforma do GitHub – repositórios, *issues* e *pull requests*, por exemplo – substituindo as diversas integrações às APIs REST [GitHub 2022].

Seleção de Repositórios

A primeira coleta de dados deste trabalho é de repositórios de projetos de *frameworks* desenvolvidos na linguagem de programação Java e hospedados no GitHub. Em conjunto com a API GraphQL do GitHub[2], para filtrar os repositórios, utiliza-se também o GitHub Topics[3]. A ferramenta GitHub Topics permite a administradores de repositórios

[1]https://docs.github.com/en/graphql
[2]https://docs.github.com/en/graphql
[3]https://github.com/topics

no *GitHub* adicionarem rótulos por meio de palavras-chave para identificar projetos baseados em temas específicos, como por exemplo a linguagem de programação e/ ou *framework* [AlMarzouq et al. 2020]. Também possibilita que usuários da plataforma busquem repositórios para colaboração, seja através do site da plataforma e/ ou API.

Neste trabalho, foram definidos seis critérios para seleção de repositórios a serem analisados e um *script* foi desenvolvido para filtrar os repositórios a partir dos critérios definidos. A Tabela 1 apresenta os critérios de seleção aplicados. Os dois primeiros critérios são ter Java como linguagem de programação principal de desenvolvimento, o que resultou em 2.913.136 repositórios, e contemplar o tópico *"framework"*, o que resultou em 923 repositórios. O terceiro critério é a soma de *issues* abertas e fechadas ser maior que zero. Por fim, foram definidos como quarto e quinto critérios, o número de *tags* ser maior que 2 e a data mínima de atualização a partir 01/01/2023.

Além disso, os repositórios foram ordenados por quantidade de estrelas, recurso originalmente do inglês *Most Stars*, ou seja, ordenou-se a partir do maior número de estrelas para o menor, sendo a quantidade mínima de estrelas igual a um. As estrelas são um recurso do GitHub que permitem aos usuários manifestarem interesse ou satisfação com um repositório hospedado, através da marcação de uma estrela. Dessa forma, as estrelas são classificadas como um fator de popularidade [Borges et al. 2016]. Quantitativamente, através da busca no GitHub com os critérios estabelecidos inicialmente, obtém-se um total de 923 repositórios. A Tabela 1 apresenta detalhadamente os valores de projetos encontrados para cada um dos critérios de seleção, bem como para quando se estabelece uma quantidade mínima de estrelas.

Para coleta e armazenamento dos dados, emprega-se um *script*. Entende-se *script* como um conjunto de instruções lógicas com etapas bem definidas, executando funções para manipular, automatizar e personalizar tarefas sistêmicas. Inicialmente, o *script* comunica-se com a API GraphQL do GitHub, realizando a busca de repositórios conforme os critérios apresentados anteriormente. Para cada um dos repositórios encontrados, realiza-se o *clone*, ou seja, o armazenamento local do repositório, o que viabiliza a identificação de arquivos de teste. Para cada repositório armazenado localmente na etapa prévia, itera-se entre os arquivos existentes no diretório, buscando por arquivos cuja nomenclatura inclua a palavra *Test* em seu início ou fim.

Caso sejam identificados arquivos com a nomenclatura especificada, por padrão da linguagem de programação Java, este é considerado como um arquivo/classe referente a implementação de testes. Logo, em cada arquivo de teste, valida-se a utilização da biblioteca de testes unitários JUnit[4], buscando-se pelo *import* da biblioteca no conteúdo do arquivo com uma das seguintes expressões: *"import org.junit.jupiter.api.Test;"*, *"import org.junit"* ou *"import org.junit.jupiter.api."*, associado também a *annotation @Test*.

Além dos critérios estabelecidos acima, para a filtragem dos repositórios também é levada em consideração a quantidade de *tags*. *Tags* são semelhantes a uma referência no Git e apontam para versões ou *releases* específicas de um projeto [GitHub 2022]. No presente estudo, as *tags* são utilizadas para demarcar *releases* do *software*. Sendo assim, foram considerados apenas repositórios com duas ou mais *tags*, visto que valores inferiores (zero ou uma *tag*) não são passíveis de análise histórica, por não possuírem versões

[4]https://junit.org/junit5/

suficientes para compraração. Por fim, são considerados aptos para análise projetos que foram atualizados ou tiveram *commits* a partir do dia 01/01/2023, uma vez que repositórios não atualizados recentemente, podem não ser utilizados e, portanto, serem pouco relevantes para o estudo. Dessa forma, conta-se com 100 repositórios que atendem aos critérios de seleção e possuem testes implementados com a biblioteca JUnit, conforme características apresentadas na Tabela 9 do Apêndice B.

Tabela 1. Quantidade total de repositórios por critério de seleção

Repositórios		
Identificador	**Critério de seleção**	**Quantidade**
C1	Linguagem de programação Java	2.913.136
C2	C1 + tópico *"framework"*	923
C3	C2 + *issues* habilitadas	905
C4	C3 + *issues* fechadas >0	308
C5	C4 + testes	227
C6	C5 + *tags* >2 + data atualização >= 01/01/2023	100

Análise da suíte de testes do código-fonte

Para cada um dos repositórios de projeto de *frameworks* selecionados, realiza-se uma análise da qualidade da suíte de testes, verificando a provável existência de *test smells*. Para isso, processa-se os testes unitários em duas ferramentas JNose[5] e Test Smell Detector[6], conforme apresentado a seguir. Foram analisados 100 repositórios, sendo que, dos repositórios selecionados anteriormente, somente os que possuem testes em JUnit foram devidamente analisados, já que as duas ferramentas suportam somente esta biblioteca de testes.

O JNose é uma ferramenta que automatiza a detecção de *test smells* em classes de testes, sendo capaz de identificar 21 tipos diferentes de *test smells* [Virgínio et al. 2019]. Neste trabalho, utiliza-se a ferramenta para analisar individualmente cada um dos arquivos de teste, relacionando com a classe do código de produção. Vale ressaltar que o JNose também coleta os códigos de todas as versões de *releases*, com base nas *tags*, liberadas pelo administrador do repositório. Sendo assim, com o uso da ferramenta são obtidos os números, tipos e ocorrências de *test smells* em cada arquivo de teste por versão de cada *release* dos *frameworks*, sendo cada *release* demarcada por uma *tag* específica. Para viabilizar o uso dessa ferramenta, foi necessário realizar algumas adequações em seu código, considerando a atualização de algumas bibliotecas de uso interno da mesma.

O Test Smell Detector, também conhecido como tsDetect, é uma ferramenta *open-source* para detecção automatizada de *test smells* em códigos de testes de projetos desenvolvidos na linguagem de programação Java [Aljedaani et al. 2021]. Através de códigos de testes unitários e classes de produção em seu funcionamento, conta-se com a geração de Árvores de Sintaxe Abstastra (ASTs, do inglês, *Abstract Syntax Tree*), assim, identificando padrões e práticas ruins de testes utilizando regras de detecção [Peruma et al. 2020]. Ademais, vale ressaltar que, a ferramenta é compatível somente com testes unitários implementados com o *framework* de testes JUnit.

Em relação a implementação e execução, o tsDetect conta a execução em linha de comando, através de um arquivo executável no formato *jar*. Assim, detectando 21

[5]https://jnosetest.github.io/

[6]https://testsmells.org/

tipos de *test smells*. Para cada projeto analisado, é utilizado como entrada um arquivo no formato *csv*, onde, cada linha contém o nome da aplicação, caminho do arquivo de teste e caminho do arquivo de produção relativo. Inicialmente, para geração do arquivo de entrada, foi tentada a combinação de ferramentas TestFileDetector e TestFileMapping, desenvolvidas pelos criadores do tsDetect para detecção dos arquivos de testes e detecção de arquivos de produção, porém não se obteve sucesso. Sendo assim, implantou-se um *script* responsável por detectar todos os arquivos de teste e produção, gerando um arquivo de entrada no formato *csv* e padrão requerido pela aplicação.

Coleta de *issues*

A partir dos repositórios selecionados, coletou-se as *issues* referentes ao relato de *bugs* e falhas. No GitHub, as *issues* permitem a comunicação e acompanhamento de um trabalho entre desenvolvedores e usuários de um projeto, centralizando informações. É um local no GitHub para reportar *bugs*, falhas e novas melhorias [GitHub 2022, Pipinellis 2015]. Assim, as *issues* são uma forma de trazer a perspectiva dos usuários a respeito de problemas enfrentados com os *frameworks*. Para identificação de *issues* referente a *bugs*, foram definidos alguns termos como sendo similares a esta ocorrência, sendo eles: *bug*, *bugfix*, *fail*, *failure*, *fault* ou *error*. Dessa forma, através da API GraphQL do GitHub, selecionou-se as *issues* com *status open* ou *closed*, além do corpo de descrição da *issue* (*BodyText*) contendo um dos termos referente a *bugs*. A motivação para uso do corpo de descrição se dá pela ausência da padronização de nomenclatura para reporte de *bugs* ou falhas adotada pelos repositórios analisados nos demais campos disponíveis, como por exemplo, a *label*.

4.2. Métricas e Avaliação

As métricas criadas para auxiliar a responder as perguntas levantadas foram escolhidas conforme os critérios apresentados nesta seção, conforme apresentado abaixo:

- **Contagem individual de *test smells*:** Número total de cada tipo de *test smell* por projeto.
- **Número de *test smells* por versão:** é computado o número total de *test smells* por projeto e versão. Essa métrica visa esclarecer se no decorrer dos lançamentos de novas versões, como o total de *test smells* variou, ou seja, se aumentou ou diminuiu.
- **Variação de *bugs* e *test smells*:** Essa métrica visa compreender a razão entre a quantidade de *bugs* reportados e a variação da quantidade de *test smells* entre uma versão e outra.

5. Resultados

Nesta seção, apresenta-se os resultados obtidos para o estudo proposto. Na Seção 5.1 é apresentada a caracterização dos dados analisados. Na Seção 5.2 são apresentados os resultados, respondendo as quatro perguntas levantadas como forma de atingir o objetivo geral. Na seção 5.3 é trazida uma sumarização dos resultados encontrados.

5.1. Caracterização dos dados

Conforme apresentado na Seção 4.1, selecionaram-se 100 repositórios de projetos referentes a *frameworks* Java. Contudo, durante a análise da suíte de testes com as ferramentas

de detecção de *test smells*, JNose e tsDetect, foram descartados 13 repositórios, pois os arquivos de testes unitários presentes não possuíam associação direta a uma classe (arquivo) de produção, impedindo a análise destes pelas ferramentas, restando assim, 87 repositórios. Vale ressaltar que, neste estudo propõe-se realizar uma análise evolutiva dos *test smells*, para isso, utilizou-se as *tags*.

Assim, considerou-se repositórios com o mínimo de duas *tags*, analisando todas as versões quando o número de *tags* é entre dois e quatro. Para repositórios com valores superiores a quatro *tags*, considera-se 3 versões, respectivamente: a primeira versão, uma intermediária – total de versões dividido por 2; caso o total de versões seja número ímpar, faz-se arredondamento do resultado – e a mais recente (lançada até o dia 20/04/2023). A análise de três versões sucede-se para avaliar 3 momentos na evolução do projeto, respectivamente: início, meio e atualmente. Enquanto que, para repositório com duas a quatro *tags*, foi analisado a totalidade de versões pela dificuldade de particionamento, não sendo viável determinar uma versão intermediária.

Após a análise dos resultado dos repositórios com as JNose e tsDetect, algumas observações podem ser consideradas em relação aos resultados. Em termos gerais, observa-se um total de 435.986 *test smells* encontrados pela ferramenta JNose, bem como 305.365 pela ferramenta tsDetect. Acerca das versões analisadas dos repositórios, nota-se que 94,32% (82) dos repositórios possuem cinco ou mais *tags*, analisando-se então três versões. Além disso, para os demais repositórios (5), 5,68%, todas as versões foram consideradas.

Tabela 2. Resumo estatísticos dos repositórios analisados

Test Smell	*JNose*					*tsDetect*				
	min	*max*	$\bar{x}$	*Md*	Σ	*min*	*max*	$\bar{x}$	*Md*	Σ
Assertion Roulette	0	1.751	17,01	5,00	252.970	0	140	2,122	0,00	37.862
Conditional Test Logic	0	124	0,60	0,00	8.852	0	78	0,245	0,00	4.364
Constructor Initialization	0	2	0,04	0,00	522	0	3	0,016	0,00	280
Default Test	0	0	0,00	0,00	0	0	0	0,00	0,00	0
Dependent Test	0	0	0,00	0,00	0	0	0	0,00	0,00	0
Duplicate Assert	0	311	1,50	0,00	22.267	0	87	0,514	0,00	9.174
Eager Test	0	173	2,84	1,00	42.242	0	266	2,243	0,00	40.018
Empty Test	0	17	0,03	0,00	473	0	17	0,035	0,00	622
Exception Catching Throwing	0	119	0,69	0,00	10.271	0	337	2,422	0,00	43.222
General Fixture	0	23	0,39	0,00	5.868	0	424	1,425	0,00	25.422
Ignored Test	0	113	0,66	0,00	9.762	0	113	0,459	0,00	8.192
Lazy Test	0	1.360	3,74	1,00	55.559	0	1.084	9,204	0,00	164.249
Magic Number Test	0	405	3,39	0,00	50.439	0	424	6,072	3,00	108.346
Mystery Guest	0	72	0,07	0,00	1.076	0	72	0,055	0,00	989
Print Statement	0	170	0,15	0,00	2.235	0	79	0,064	0,00	1.151
Redundant Assertion	0	66	0,09	0,00	1.323	0	19	0,018	0,00	316
Resource Optimism	0	72	0,07	0,00	1.074	0	72	0,046	0,00	821
Sensitive Equality	0	167	0,26	0,00	3.874	0	54	0,124	0,00	2.215
Sleepy Test	0	41	0,05	0,00	809	0	16	0,024	0,00	437
Unknown Test	0	266	1,27	0,00	18.864	0	260	1,130	0,00	20.156
Verbose Test	0	73	0,60	0,00	8.970	0	0	0,00	0,00	0
Total					**435.986**					**305.365**

A Tabela 2 apresenta um resumo estatístico dos resultados obtidos com as ferramentas. Na primeira coluna, apresentam-se os *test smells* detectados pelas ferramentas.

Na segunda e terceira colunas, são apresentados os dados identificados de todos os repositórios por cada ferramenta. Finalmente, esses dados são subdivididos e apresentados valores estatísticos em cinco sub colunas, tendo-se: os valores mínimos (*min*), máximos (*max*), a média da amostra ($\bar{x}$), mediana (*Md*) e por fim, a somatória (Σ).

5.2. Apresentação dos Resultados

Esta seção apresenta a discussão dos resultados para responder cada uma das questões propostas no objetivo do presente estudo.

Os *frameworks* apresentam menos *test smells* ao decorrer dos lançamentos de novas versões?

Para responder essa questão, considera-se o total de *test smells* identificados pelas ferramentas para cada versão. Ao relacionar os *test smells* com as versões coletadas dos repositórios, conforme apresentado na Tabela 3, observa-se que nos resultados obtidos pelas ferramentas, maior parte dos *smells* estão distribuídos na última versão dos repositórios, sendo que o JNose detectou 51,95% (226.497) e o tsDetect detectou 54,19% (165.479). Por outro lado, as segunda e terceira versões apresentam menores percentuais, sendo importante destacar que somente cinco repositórios tiveram a segunda e terceira versões analisadas por apenas possuírem entre 2 e 4 versões. Sendo assim, considera-se que a primeira versão para ambas ferramentas apresentam menores percentuais de *test smells*, 12,62% (48.624) para o JNose e 7,18% (17.289) tsDetect. Conforme apresentado no Apêndice 11, nota-se o total de cada tipo de *test smell* por versão. Dessa forma, é possível observar que ao decorrer dos lançamentos de novas versões, os *frameworks* apresentam mais *test smells*.

Tabela 3. Total de test smells detectados por versão

Ferramenta	***Versão***				
	Primeira	**Segunda**	**Terceira**	**Intermediária**	**Última**
JNose	48.624	0	76	160.789	226.497
tsDetect	17.289	0	63	122.534	165.479

Quais *Frameworks* tendem a ter menos *test smells*?

Para responder essa questão de pesquisa, para cada repositório, calcula-se a quantidade total de *test smells* por versão. Após, ordenando de forma decrescente com base no total *test smells* para todas as versões.

As Tabelas 4 e 5 apresentam os dez projetos de *frameworks* que contém as menores quantidades de *test smells*, segundo os resultados obtidos com as ferramentas JNose e tsDetect, respectivamente. Na primeira coluna, apresentam-se os nomes do repositórios. As colunas 2, 3 e 4 apresenta as quantidades de *test smells* identificados a cada versão, respectivamente: primeira, intermediária e última. Comparando os resultados presente nas tabelas, nota-se cinco projetos em comum e, embora contenham resultados diferentes na quantidade de *test smells* identificados por versão, ambas detectaram os mesmos. O *ranking* de repositórios com as menores quantidades de *test smells* para as ferramentas são respectivamente: *darmiel/eeee*, *OptiJava/Optilog-Client*, *victordiaz/PHONK*, *spring-projects/spring-vault* e *cofcool/chaos-server*.

Tabela 4. Top 10 repositórios com menor número de *test smells* – JNose

	Primeira	Intermediária	Última
darmiel/eeee	0	0	0
OptiJava/Optilog-Client	0	0	6
victordiaz/PHONK	0	0	6
spring-projects/spring-vault	1	5	5
cofcool/chaos-server	0	0	13
spring-projects/spring-data-commons	8	8	8
spring-projects/spring-data-rest	0	22	24
AtomGraph/Core	0	3	49
TechEmpower/FrameworkBenchmarks	0	55	3
ibissource/iaf	0	0	58
Total	9	93	172

Tabela 5. Top 10 repositórios com menor número de test smells – tsDetect

	Primeira	Intermediária	Última
cofcool/chaos-server	0	0	0
darmiel/eeee	0	0	4
OptiJava/Optilog-Client	0	0	7
spring-projects/spring-vault	2	5	0
victordiaz/PHONK	0	0	10
spring-projects/spring-data-relational	2	0	15
han-yaeger/yaeger	13	1	3
ChameleonFramework/Chameleon	0	0	21
libgdx/gdx-ai	0	5	20
Bethibande/JWebAPI	0	15	11
Total	17	26	91

Ademais, identificou-se dois projetos que não houveram detecção de *test smells* por uma das ferramentas, porém houve a detecção pela outra. Em relação aos resultados obtidos com o JNose, observa-se que o projeto *darmiel/eeee* não conta com a detecção de *test smells* para nenhuma de suas versões. Diferindo dos dados obtidos com o tsDetect, que identificou quatro *test smells* na última versão. O mesmo ocorre com o projeto *cofcool/chaos-server*, onde, não há a detecção de *test smells* com o tsDetect, mas há a detecção de 13 *smells* com o JNose. Portanto, tendo em vista os dados analisados, observa-se que entre os dez repositórios apresentados em cada ferramenta, pode ser ressaltado que a maior parte dos *test smells* se concentram na última versão.

Quais são os *test smells* mais e menos recorrentes nos códigos de testes dos *frameworks*?

Para responder essa questão de pesquisa, para cada repositório, calcula-se a quantidade total de cada tipo de *test smell*. Após, selecionando-se os repositórios com base nos menores e maiores valores do total de *test smell*. Vale destacar que, para o menores valores filtrou-se aqueles com valores maiores que 0.

Na Tabela 6 demonstram-se os repositórios com maior quantidade de *test smells* por tipo. Na primeira coluna, dispõem-se cada tipo de *test smell*. As colunas 2 e 3 foram dividas em subcolunas, representando o nome do *framework* e a quantidade de *test smells* contida (Qtd). Na perspectiva de recorrência, conforme apresentado na coluna (*min*) da Tabela 2, é possível perceber que não há a presença de nenhum tipo de *test smell* em pelo menos um.

Nos resultados obtidos com a ferramenta JNose, em relação a maior recorrência destaca-se o tipo *Assertion Roulette* representando 58,02% (252.970) do total de *test smells* detectados pela ferramenta. Além de menor recorrência do tipo *Empty Test*, correpondente a 0,10% (473). Por outro lado, observa-se que, na ferramenta tsDetect evidencia-se maior recorrência do *test smell* do tipo *Lazy Test*, correspondente a 53,78% (164.249) do total de *test smells*. Enquanto menor recorrência do tipo *Constructor Initialization* com 0,09% (280) do total detectado pela ferramenta.

Além disso, observa-se que para os *test smells* mais recorrentes, ou seja, *Assertion Roulette* e *Lazy Test*, o projeto *halogenOS/android_frameworks_base* detêm os maiores valores para os dois tipos, respectivamente: 57.338 para o JNose e 51.593 tsDetect. No entanto, analisando-se os *test smells* menos recorrentes, assim, em relação ao tipo *Empty Test* o projeto *pumbas600/Halpbot-v1* possui o menor valor: 1. Enquanto para o tipo

Constructor Initialization, destaca-se o *AtomGraph/Processor* com 1.

Contudo, observou-se também dois casos especiais. O primeiro, é possível observar que não há detecção de dois tipos de *test smells* em ambas ferramentas, respectivamente: *Default Test* e *Dependent Test*. Ou seja, não foi detectado nenhuma ocorrência desse tipo de *test smell* em nenhum dos projetos e versões analisadas. O segundo caso, está relacionado ao *test smell Verbose Test*, o qual foi detectado somente pelo JNose, não havendo ocorrência em nenhum dos repositórios analisados com o tsDetect.

Tabela 6. Repositórios com maior e menores quantidades de test smells por tipo

Test Smell	*JNose*				*tsDetect*			
	Maior Recorrência		**Menor Recorrência**		**Maior Recorrência**		**Menor Recorrência**	
	Repositório	**Qtd**	**Repositório**	**Qtd**	**Repositório**	**Qtd**	**Repositório**	**Qtd**
Assertion Roulette	*halogenOS/android_frameworks_base*	57.338	*victordiaz/PHONK*	1	*halogenOS/android_frameworks_base*	10.698	*International-Data-Spaces-Association/IDS-Messaging-Services*	2
Conditional Test Logic	*halogenOS/android_frameworks_base*	3.429	*DevNatan/inventory-framework*	2	*halogenOS/android_frameworks_base*	1.677	*DevNatan/inventory-framework*	1
Constructor Initialization	*halogenOS/android_frameworks_base*	249	*TechEmpower/FrameworkBenchmarks*	1	*freedomotic/freedomotic*	47	*AtomGraph/Processor*	1
Default Test	Não detectado		Não detectado		Não detectado		Não detectado	
Dependent Test	Não detectado		Não detectado		Não detectado		Não detectado	
Duplicate Assert	*halogenOS/android_frameworks_base*	5.265	*ChillyCheesy/Modulo*	1	*halogenOS/android_frameworks_base*	3.146	*International-Data-Spaces-Association/IDS-Messaging-Services*	1
Eager Test	*halogenOS/android_frameworks_base*	10.570	*victordiaz/PHONK*	1	*halogenOS/android_frameworks_base*	12.107	*OptiJava/Optilog-Client*	1
Empty Test	*dlsc-software-consulting-gmbh/PreferencesFX*	256	*pumbas600/Halpbot-v1*	1	dlsc-software-consulting-gmbh/PreferencesFX	256	*DYStack/chunjun*	1
Exception Catching Throwing	*rife2/rife2*	2.753	*alibaba/transmittable-thread-local*	1	*spring-projects/spring-framework*	13.562	*spring-projects/spring-vault*	1
General Fixture	*halogenOS/android_frameworks_base*	3.725	*freedomotic/freedomotic*	1	*halogenOS/android_frameworks_base*	18.178	*spring-projects/spring-boot*	1
Ignored Test	*rife2/rife2*	3.706	*AtomGraph/Core*	1	*apache/dubbo*	3.188	*AtomGraph/Core*	1
Lazy Test	*spring-projects/spring-framework*	20.499	*omnetpp/omnetpp*	1	*halogenOS/android_frameworks_base*	51.593	*International-Data-Spaces-Association/IDS-Messaging-Services*	2
Magic Number Test	*halogenOS/android_frameworks_base*	15.372	*victordiaz/PHONK*	1	*halogenOS/android_frameworks_base*	32.561	*OptiJava/Optilog-Client*	2
Mystery Guest	*halogenOS/android_frameworks_base*	518	*lutece-platform/lutece-core*	1	*halogenOS/android_frameworks_base*	464	*ChillyCheesy/Modulo*	1
Print Statement	jencisopy/JavaBeanStack	803	*TechEmpower/FrameworkBenchmarks*	1	jencisopy/JavaBeanStack	413	*TechEmpower/FrameworkBenchmarks*	1
Redundant Assertion	*rife2/rife2*	680	*AlmasB/FXGL*	1	*spring-projects/spring-framework*	125	*AlmasB/FXGL*	1
Resource Optimism	*halogenOS/android_frameworks_base*	474	*lutece-platform/lutece-core*	1	*halogenOS/android_frameworks_base*	450	*lutece-platform/lutece-core*	1
Sensitive Equality	*spring-projects/spring-framework*	892	*Flash3388/FlashLib*	1	*spring-projects/spring-framework*	803	*Ravaelles/Atlantis*	1
Sleepy Test	*halogenOS/android_frameworks_base*	216	*spring-projects/spring-data-neo4j*	1	*halogenOS/android_frameworks_base*	143	jencisopy/JavaBeanStack	1
Unknown Test	*halogenOS/android_frameworks_base*	6.767	*AlmasB/FXGL*	1	*halogenOS/android_frameworks_base*	7.332	*AlmasB/FXGL*	1
Verbose Test	*halogenOS/android_frameworks_base*	2.905	*alibaba/transmittable-thread-local*	1	Não detectado		Não detectado	

As *issues* de *bugs* refletem relação com a qualidade dos códigos de teste?

Para responder essa questão de pesquisa, considerou-se o total de *issues* por versão, respectivamente classificadas como pertencentes a todas *issues* e de *bugs*. Compreende-se uma *issue* pertence a versão quando, sua data de abertura para *issues* com status *open* compreende-se entre data de início e publicação da versão, enquanto para *issues* com status *closed*, a data de encerramento.

No que se diz respeito às *issues*, para os 87 repositórios dos *frameworks* analisados neste estudo, identificou-se um total de 117.176 *issues* em todas as suas versões. As quais distribuem-se da seguinte maneira: 106.835 *issues* fechadas (*Closed Issues*) e 10.341 *issues* abertas (*Open Issues*). Conforme abordado na Subseção 4.1, um dos critérios para seleção das *issues* é a presença de palavras-chave referentes ao reporte de *bugs* e falhas no corpo de descrição (*BodyText*). Sendo assim, do número total de *issues* foram analisadas 10.482 *issues*, sendo 9.877 *isssues* fechadas e 605 abertas. Vale ressaltar que, as *issues* estão dispostas nas três versões utilizadas para análise.

A Tabela 7 apresenta o número de *issues* pelas versões analisadas. A primeira coluna apresenta as versões, já a segunda coluna, o número total de *issues* fechadas. A terceira coluna, o número de *issues* abertas. Por fim, a quarta, o número de *issues* analisadas. É importante destacar que, as colunas dois, três e quatro foram subdivididas em subcolunas de *issues* de bugs (*Issues* de *Bugs*) e o número total de *issues* (Todas as Issues).

Então, para as 10.482 *issues* analisadas, é possível notar que 85,21% (8.932) das *issues* estão alocadas na primeira versão dos repositórios. Havendo uma redução de 89,36% entre a primeira versão (8.932) e a versão intermediária (959), aumentando novamente em 17,93% entre a intermediária (959) e última versões (1.131). Além disso, observa-se que o número total de *issues* das versões analisadas é igual ao número de *issues* classificadas como *bug*.

Apesar de haver um aumento do número de *test smells*, conforme Tabela 3, e de

issues abertas ao decorrer das versões, conforme Tabela 7, não é possível afirmar que o aumento do número de *issues* é uma consequência da qualidade do código de testes. Isso se deve ao fato de que a motivação para a abertura de *bugs* pode ser relacionada com problemas semânticos, regras de negócio, segurança e de memória [Li et al. 2006]. Seria necessária uma análise qualitativa das *issues* para entender se o motivo da abertura delas é relacionada com alguma falha da suíte de testes ou não.

Tabela 7. Distribuição de issues por versão

Versões	*Issues* Fechadas	*Issues* Abertas	Número Total de *Issues*
Primeira	8.251	141	8.392
Intermediária	852	107	959
Última	774	357	1131
Total	**9.877**	**605**	**10.482**

5.3. Sumarização dos Resultados

A presente seção tem como objetivo sintetizar os resultados apresentados e analisados na Subseção 5.2, respondendo diretamente as perguntas.

Os *frameworks* apresentam menos *test smells* ao decorrer dos lançamentos de novas versões? De acordo com as versões avaliadas, é possível observar que ao decorrer dos lançamentos de novas versões os *frameworks* apresentam mais *test smells*.

Quais *Frameworks* tendem a ter menos *test smells*? Os *frameworks* que tendem a ter menos *test smells* são: *darmiel/eeee*, *OptiJava/Optilog-Client*, *victordiaz/PHONK*, *spring-projects/spring-vault* e *cofcool/chaos-server*.

Quais são os *test smells* mais e menos recorrentes nos códigos de testes dos *frameworks*? Os tipos de *test smells* mais recorrentes são *Assertion Roulette* e *Lazy Test*. Enquanto os menos recorrentes são *Empty Test* e *Constructor Initialization*.

As *issues* de *bugs* refletem relação com a qualidade dos códigos de teste? Não foi possível estabelecer uma relação entre as *issues* de *bugs* e a qualidade de código de testes. Não é possível afirmar que o aumento ou diminuição do número de *issues* é uma consequência da qualidade do código de testes. Isso se deve ao fato de que a motivação para a abertura de *bugs* pode ser relacionada com outros problemas, como por exemplo: semântica e segurança.

6. Ameaças à validade

Nesta seção apresenta-se as ameaças à validade, bem como estratégias para sua mitigação. Em relação à validade de construção, têm-se a maneira como foram coletados os *test smells*, podendo gerar os vieses no momento de análise de resultados. Para mitigação, utilizou-se duas ferramentas de detecção de *test smells*: tsDetect e JNose.

Quanto a ameaça à validade interna, nota-se problemas enfrentados com as ferramentas JNose e tsDetect. No que diz respeito à primeira, por estar desatualizada, não conseguia ler arquivos de todas as versões mais novas do Java. Portanto, a ferramenta teve que ser modificada para abarcar as novas versões das bibliotecas internas, e, ao atualizá-las, observou-se pequenas incompatibilidades com alguns objetos. No entanto, embora tenham sido feitas modificações em estruturas de dados e alguns parâmetros, mitigou-se

o risco ao verificar após a atualização que toda a suíte de testes da ferramenta continuava a ser executada com sucesso.

Ainda nas ameaças à validade interna, em relação ao tsDetect, observou-se alguns problemas com as ferramentas complementares TestFileDetector e TestFileMapping que são executadas para mapear os arquivos de teste e produção dos projetos. Porém, para repositórios que contém uma grande quantidade de arquivos, lida-se com erros durante sua execução. Dessa forma, mitigou-se criando um *script* em Python que realiza a mesma lógica implementada nas ferramentas para gerar o arquivo de entrada com os dados e formatos necessários.

Por fim, em relação à validade externa, relaciona-se a dificuldade de generalizar todos os *frameworks* Java. Para mitigar, avaliou-se todos os repositórios públicos disponíveis de *frameworks* Java que contam com casos de testes implementados, bem como, possuem atributos que comprovam atualizações recentes e portanto sua relevância.

7. Conclusão

Neste estudo foram identificados e caracterizados os *frameworks* Java na perspectiva da qualidade de testes baseada em *test smells*. Com o intuito de identificar se o surgimento de problemas para o desenvolvedor durante o reúso de *frameworks* estão vinculados à qualidade da suíte de testes de tais ferramentas. Para isso, foram analisados a suíte de teste de 87 repositórios disponíveis no GitHub.

Como resultado, foi identificado que ao decorrer dos lançamentos de novas versões, os *frameworks* aumentam relativamente a quantidade de *test smells*, tornando-se esta questão uma ameaça para a qualidade dos sistemas. Adiante, observou-se que os *frameworks* comuns em ambas ferramentas usadas para análise que têm incidência a menor *test smells* foram o *darmiel/eeee*, *OptiJava/Optilog-Client*, *victordiaz/PHONK*, *spring-projects/spring-vault* e *cofcool/chaos-server*. Também foi encontrado uma maior recorrência dos *test smells* do tipo *Assertion Roulette* e *Lazy Test*. Enquanto, respectivamente, o primeiro ocorre quando um método de teste possui múltiplas asserções não documentadas, dificultando a leitura, compreensão e manutenção do teste; já o segundo ocorre quando métodos de teste diferentes utilizam o mesmo produtor de objetos, ou seja, reutilizam-se métodos auxiliares ao teste em diferentes testes.

Embora não tenha sido possível confirmar se o surgimento de problemas para o desenvolvedor durante o reúso de *frameworks* estão vinculados à qualidade da suíte de testes de tais ferramentas, foram encontrados diversos pontos relevantes que contribuem para analisar o estado atual das ferramentas analisadas. Em conjunto, foram observados pontos interessantes passíveis de investigação em trabalhos futuros, por exemplo: 1) analisar como o aumento dos *test smells* impacta questões como a manutenção *frameworks* 2) analisar qualitativamente as *issues* de *bugs* e falhas, entendendo se há uma relação com problemas na suíte de testes 3) realizar a intersecção entre os resultados obtidos pelo tsDetect e JNose.

Pacote de Replicação

O pacote de replicação deste trabalho encontra-se disponível em:

`https://github.com/ICEI-PUC-Minas-PPLES-TI/plf-es-2022-2-tcci-5308100-pes-altino-alves-e-leticia-meireles`

Referências

Aljedaani, W., Peruma, A., Aljohani, A., Alotaibi, M., Mkaouer, M. W., Ouni, A., Newman, C. D., Ghallab, A., and Ludi, S. (2021). Test smell detection tools: A systematic mapping study. *Evaluation and Assessment in Software Engineering*, pages 170–180.

AlMarzouq, M., AlZaidan, A., and AlDallal, J. (2020). Mining github for research and education: challenges and opportunities. *International Journal of Web Information Systems*.

Borges, H., Hora, A., and Valente, M. T. (2016). Understanding the factors that impact the popularity of github repositories. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 334–344.

Cabral, P. et al. (2022). *Aplicativo Android para encontrar o ponto de entrega voluntária mais próximo em Rio Verde e Goiás*. Instituto Federal Goiano.

Camara, B., Silva, M., Endo, A., and Vergilio, S. (2021). On the use of test smells for prediction of flaky tests. In *Brazilian Symposium on Systematic and Automated Software Testing*, pages 46–54.

Edwin, N. M. (2014). Software frameworks, architectural and design patterns. *Journal of Software Engineering and Applications*, 2014.

Gajewski, M. and Zabierowski, W. (2019). Analysis and comparison of the spring framework and play framework performance, used to create web applications in java. In *2019 IEEE XVth International Conference on the Perspective Technologies and Methods in MEMS Design (MEMSTECH)*, pages 170–173.

Garousi, V., Kucuk, B., and Felderer, M. (2018). What we know about smells in software test code. *IEEE Software*, 36(3):61–73.

Garousi, V., Kucuk, B., and Felderer, M. (2019). What we know about smells in software test code. *IEEE Software*, 36(3):61–73.

GitHub (2022). Github documentation. Disponível em: https://docs.github.com/pt Último acesso em 14/05/2023.

Greiler, M., Zaidman, A., Van Deursen, A., and Storey, M.-A. (2013). Strategies for avoiding text fixture smells during software evolution. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 387–396. IEEE.

Haiderzai, M. D. and Khattab, M. I. (2019). How software testing impact the quality of software systems? *IJECS*, 1(2):05–09.

Hooda, I. and Chhillar, R. S. (2015). Software test process, testing types and techniques. *International Journal of Computer Applications*, 111(13).

Jorge, D., Machado, P., and Andrade, W. (2021). Investigating test smells in javascript test code. In *Brazilian Symposium on Systematic and Automated Software Testing*, pages 36–45.

Kim, D. J. (2020). An empirical study on the evolution of test smell. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 149–151. IEEE.

Li, Z., Tan, L., Wang, X., Lu, S., Zhou, Y., and Zhai, C. (2006). Have things changed now? an empirical study of bug characteristics in modern open source software. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, ASID '06, page 25–33, New York, NY, USA. Association for Computing Machinery.

Peruma, A., Almalki, K., Newman, C. D., Mkaouer, M. W., Ouni, A., and Palomba, F. (2020). Tsdetect: An open source test smells detection tool. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 1650–1654.

Pipinellis, A. (2015). *GitHub essentials*, volume 2. Packt Publishing.

Pressman, R. S. and Maxim, B. R. (2021). *Engenharia de software-9*. McGraw Hill Brasil.

Sommerville, I. (2018). *Software Engineering*. Pearson, 10 edition.

Virgínio, T., Santana, R., Martins, L. A., Soares, L. R., Costa, H., and Machado, I. (2019). On the influence of test smells on test coverage. In *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*, pages 467–471.

Wu, H., Yin, R., Gao, J., Huang, Z., and Huang, H. (2022). To what extent can code quality be improved by eliminating test smells? In *2022 International Conference on Code Quality (ICCQ)*, pages 19–26.

Zhao, Y., Hu, Y., and Gong, J. (2021). Research on international standardization of software quality and software testing. In *2021 IEEE/ACIS 20th International Fall Conference on Computer and Information Science (ICIS Fall)*, pages 56–62.

A. Apêndice - *Test Smells*

Neste apêndice, introduz-se sobre os *test smells*. Assim, na Tabela 8 descreve os 21 tipos de *test smells* analisados neste trabalho.

Tabela 8. Descrição de cada *test smell*

Test Smell	Descrição
Assertion Roulette	Ocorre quando o método contém mais de uma instrução de asserção sem nenhuma explicação ou mensagem.
Conditional Test Logic	Um método de teste que contém uma ou mais instruções de controle como: *if*, *switch*, expressão condicional, *for*, *foreach* e *while*.
Constructor Initialization	Acontece quando uma classe de teste usa o construtor dela para inicializar campos em vez de usar o método de configuração de teste.
Default Test	Acontece em uma classe que contém o teste genérico de exemplo gerado pela suíte de desenvolvimento.
Dependent Test	Se trata do teste que só é executado na execução bem-sucedida de outros testes.
Duplicate Assert	Um método de teste que contém mais de uma instrução de asserção com os mesmos parâmetros.
Eager Test	Ocorre quando um método de teste invoca mais de um método de produção.
Empty Test	Acontece quando um teste não possui nenhuma instrução executável.
Exception Catching Throwing	Ocorre quando um método de teste usa o bloco *try-catch* para em vez da verificação da suíte de teste como o JUnit *assert*.
General Fixture	Decorre quando algum atributo de uma classe é instanciado no método de inicialização do teste mas não é utilizado.
Ignored Test	Acontece quando um teste é ignorado (usando a anotação *@ignore*, por exemplo).
Lazy Test	Ocorre quando vários métodos de teste utilizam o mesmo método de produção.
Magic Number Test	Um método de asserção que contém um literal numérico como argumento.
Mystery Guest	Se trata do método de teste que contém instâncias de objetos de arquivos e classes de bancos de dados.
Print Statement	Contém declarações de impressão do resultado do teste.
Redundant Assertion	Acontece no teste que possui uma declaração de asserção que é permanentemente verdadeira ou falsa.
Resource Optimism	Se trata do teste que faz uma suposição sobre a existência de recursos externos, sem verificá-los.
Sensitive Equality	Ocorre quando uma asserção tem uma verificação de igualdade usando o método *toString*.
Sleepy Test	Um método de teste que invoca o método *"Thread.sleep()"*.
Unknown Test	Um método de teste que não contém uma única instrução de asserção e o parâmetro de anotação *"@Test"*.
Verbose Test	Código de teste que é complexo (muitas instruções e fluxos) e não limpo.

B. Apêndice - Dados Coletados no GitHub

Neste apêndice, apresenta-se dados coletados e obtidos através da API GraphQL do GitHub. Assim, Tabela 9 apresenta dados dos 100 repositórios coletados e analisados com

as ferramentas de detecção *test smells*, respectivamente: nome e dono (*owner*), número de estrelas, quantidade *issues* abertas, fechadas, *forks*, *releases* e *tags*. Vale destacar que, os repositórios estão ordenados pelo número de estrelas.

Tabela 9. Informações dos repositórios coletados e analisados

nome/dono (*name/owner*)	Estrelas	*Issues* Abertas	*Issues* Fechadas	*Forks*	*Releases*	*Tags*
spring-projects/spring-boot	66712	612	28467	38437	166	267
spring-projects/spring-framework	51525	1250	22091	35962	255	258
apache/dubbo	38768	742	5338	25815	81	125
libgdx/libgdx	21364	211	3389	6414	9	52
code4craft/webmagic	10794	310	598	4124	23	27
spring-projects/spring-security	7617	815	9962	5415	137	264
TechEmpower/FrameworkBenchmarks	6911	108	1124	1846	15	19
alibaba/transmittable-thread-local	6321	18	227	1561	55	60
DTStack/chunjun	3572	228	816	1604	30	36
AlmasB/FXGL	3119	111	833	410	56	56
xiaojinzi123/Component	2682	5	110	284	97	127
spring-projects/spring-data-jpa	2633	168	2165	1263	84	279
spring-projects/spring-data-redis	1575	137	1717	1046	84	262
spring-projects/spring-data-mongodb	1504	247	2967	1015	84	279
codenameone/CodenameOne	1466	558	2836	366	56	114
cuba-platform/cuba	1309	581	2494	227	0	191
apache/inlong	1140	71	4139	410	8	19
libgdx/gdx-ai	1095	22	59	232	0	7
funkygao/cp-ddd-framework	931	7	31	238	6	6
zebrunner/carina	913	18	818	201	90	107
spring-projects/spring-data-rest	853	733	1208	524	84	256
apple/servicetalk	829	87	172	153	74	104
spring-projects/spring-data-neo4j	781	25	2252	611	84	295
dromara/sureness	778	11	35	154	23	30
junkdog/artemis-odb	740	30	495	103	25	54
authorjapps/zerocode	722	86	218	288	53	89
spring-projects/spring-data-commons	688	162	2032	617	85	289
spring-projects/spring-data-relational	656	152	946	293	82	153
activej/activej	637	25	148	61	0	42
restlet/restlet-framework-java	636	693	518	274	0	93
dlsc-software-consulting-gmbh/PreferencesFX	559	30	60	63	13	30
aikar/commands	454	56	141	133	0	3
omnetpp/omnetpp	451	226	719	125	45	99
victordiaz/PHONK	420	40	55	25	17	19
iohao/ioGame	398	35	60	94	9	12
freedomotic/freedomotic	392	63	128	492	5	10
mcdcorp/opentest	392	33	484	100	25	31
spring-projects/spring-data-cassandra	351	35	1140	302	84	243
jmix-framework/jmix	345	594	898	82	16	36
paypal/SeLion	267	23	142	242	5	5
spring-projects/spring-vault	246	13	643	167	12	49
odpf/dagger	239	21	40	37	25	25
rife2/rife2	170	2	7	12	34	59
isxcode/spring-oxygen	143	6	47	24	9	9
temporalio/sdk-java	141	223	346	95	49	50
sarl/sarl	126	66	973	43	51	54
lsfusion/platform	94	217	355	19	0	32
Telenav/kivakit	81	6	27	10	0	31
TAKETODAY/today-infrastructure	78	17	129	13	12	14
ibissource/iaf	72	231	1312	68	37	231
DevNatan/inventory-framework	60	39	107	14	16	22
spring-projects/spring-data-ldap	57	11	356	49	84	198
AtomGraph/Processor	55	4	24	6	0	10
ISID/iPLAss	55	57	788	22	82	82
Ravaelles/Atlantis	54	3	11	10	4	43
lutece-platform/lutece-core	54	3	3	40	0	46
Ravaelles/Atlantis	54	3	11	10	4	4
kaiso/relmongo	50	1	45	8	27	27
GamerCoder215/MobChip	49	0	10	4	18	18
tonivade/resp-server	45	2	9	14	11	27
spring-projects/spring-boot-data-geode	41	19	86	41	17	133
AtomGraph/Core	38	3	19	6	0	61
freya022/BotCommands	37	0	2	2	22	25
SDA-SE/sda-dropwizard-commons	33	3	20	7	929	967
scireum/sirius-kernel	32	0	1	19	165	384
DarkPhoenixs/message-queue-client-framework	29	1	1	22	29	29
ChameleonFramework/Chameleon	21	13	21	4	8	9
Coreoz/Plume	21	2	1	7	7	24
spring-projects/spring-session-data-geode	18	13	27	22	7	91
joafalves/pixel-community	18	1	17	3	12	12
snf4j/snf4j	18	0	13	3	11	11
Huazie/flea-framework	18	3	3	2	5	5
spring-projects/spring-test-data-geode	17	5	2	19	9	59
xdev-software/csapi	15	0	6	0	2	2
codingchili/chili-core	13	10	265	4	52	59
han-yaeger/yaeger	12	18	200	24	15	15
jeakfrw/jeak-framework	11	26	38	3	18	19
International-Data-Spaces-Association/IDS-Messaging-Services	11	3	127	4	25	25
darmiel/eeee	10	2	7	1	10	11
OptiJava/Optilog-Client	9	0	1	2	11	11
Flash3388/FlashLib	9	12	35	1	12	22
fjalvingh/domui	9	23	25	7	0	2
cofcool/chaos-server	8	0	6	2	9	10
CrissNamon/aide	5	1	2	0	4	
halogenOS/android_frameworks_base	5	0	1	9	0	11
ChillyCheesy/Modulo	5	5	4	1	5	7
strolch-li/strolch	5	1	2	5	13	156
numen06/JBM	5	0	1	8	3	5
jencisopy/JavaBeanStack	5	0	1	6	6	7
zebrunner/carina-cucumber	5	3	30	9	15	17
CrissNamon/aide	5	1	2	0	4	4
phax/ph-oton	4	2	9	4	37	52
Arpit-Shah/Artos	4	0	10	9	19	21
ArcanePlugins/ArcaneFramework	3	2	9	1	7	7
zebrunner/carina-commons	3	0	2	0	3	3
SnapGames/minimal	3	2	4	0	2	2
Koboo/en2do	2	0	2	1	6	6
Bethibande/JWebAPI	2	6	29	0	3	3
teverett/Pragmatach	2	31	9	1	0	36
CrissNamon/progressive	1	0	1	0	18	20
pumbas600/Halpbot-v1	1	5	9	0	2	2

C. Apêndice - Dados obtidos com as ferramentas de detecção de *test smells*

Neste apêndice, apresenta-se os dados obtidos com as ferramentas de detecção de *test smells*, respectivamente: tsDetect e JNose. A Tabela 10 demonstra os tipos de *test smells* detectados por cada ferramenta. Enquanto a Tabela 11 demonstra o total de cada tipo de *test smell* detectado por versão e ferramenta.

Tabela 10. Relação de *test smells* detectados por ferramenta

Test Smell	*Ferramenta*	
	JNose	tsDetect
Assertion Roulette	X	X
Conditional Test Logic	X	X
Constructor Initialization	X	X
Default Test	X	X
Dependent Test	X	X
Duplicate Assert	X	X
Eager Test	X	X
Empty Test	X	X
Exception Catching Throwing	X	X
General Fixture	X	X
Ignored Test	X	X
Lazy Test	X	X
Magic Number Test	X	X
Mystery Guest	X	X
Print Statement	X	X
Redundant Assertion	X	X
Resource Optimism	X	X
Sensitive Equality	X	X
Sleepy Test	X	X
Unknown Test	X	X
Verbose Test	X	X

Tabela 11. Total de *test smells* por tipo e versão

Test Smell	*JNose Test*					*tsDetect*				
	Primeira (Σ)	Segunda (Σ)	Terceira (Σ)	Intermediária (Σ)	Última (Σ)	Primeira (Σ)	Segunda (Σ)	Terceira (Σ)	Intermediária (Σ)	Última (Σ)
Assertion Roulette	27.367	0	28	92.611	132.964	2.411	0	3	15.485	19.963
Conditional Test Logic	1.074	0	11	3.034	4.733	309	0	5	1.590	2.460
Constructor Initialization	130	0	0	188	204	33	0	0	121	126
Default Test	0	0	0	0	0	0	0	0	0	0
Dependent Test	0	0	0	0	0	0	0	0	0	0
Duplicate Assert	2.621	0	3	7.433	12.210	613	0	2	3.551	5.008
Eager Test	4.508	0	16	15.795	21.923	2.231	0	16	16.503	21.268
Empty Test	139	0	0	173	161	141	0	0	313	168
Exception Catching Throwing	1.790	0	4	4.121	4.356	2.421	0	4	18.357	22.440
General Fixture	613	0	0	2.214	3.041	868	0	0	9.605	14.949
Ignored Test	269	0	0	2.732	6.761	207	0	0	1.919	6.066
Lazy Test	3.875	0	3	22.359	29.322	11.265	0	11	66.750	86.223
Magic Number Test	6.623	0	5	20.545	23.266	6.199	0	26	44.060	58.061
Mystery Guest	81	0	0	419	576	43	0	0	366	580
Print Statement	320	0	3	566	1.347	144	0	1	317	689
Redundant Assertion	300	0	0	495	528	48	0	0	110	158
Resource Optimism	84	0	0	400	590	43	0	0	342	436
Sensitive Equality	396	0	0	1.476	2.002	158	0	0	888	1.169
Sleepy Test	83	0	0	354	372	28	0	0	210	189
Unknown Test	1.913	0	6	7.112	9.833	1.262	0	6	8.119	10.769
Verbose Test	1.101	0	5	3.030	4.834	0	0	0	0	0
Total	**48.624**	**0**	**76**	**160.789**	**226.497**	**17.289**	**0**	**63**	**122.534**	**165.479**