Identificação e Caracterização de *Test Smells* em JavaScript

Andrew Costa Silva[1]

[1]Instituto de Ciências Exatas e Informática - Pontifícia Universidade Católica de Minas Gerais (PUC Minas) – Belo Horizonte, MG – Brazil

andrew.costa@pucminas.br

Abstract. *The appearance of code smells in test cases is called test smells. The literature presents some static analysis tools for detecting test smells, however, these tools have a focus on statically typed languages. Although, even with the popularity of JavaScript, there are few works that define test smells of this language. Therefore, this paper proposes 3 new types of test smells in JavaScript projects, develops a new tool that identifies 8 types of test smells for JavaScript, and explores 977 JavaScript projects available on GitHub in an attempt to locate the most common test smells. After the analysis, the most frequent types of test smells among the projects were Magic Test, Conditional Test Logic, Unused Imports and Duplicate Asserts.*

Resumo. *O aparecimento de code smells em casos de teste é denominada por test smells. A literatura apresenta ferramentas de análise estática para a detecção de test smells, todavia, essas ferramentas possuem um foco em linguagens estaticamente tipadas. No entanto, mesmo com a popularidade do JavaScript, são escassos os trabalhos que definem test smells dessa linguagem. Portanto, este trabalho propõe 3 novos tipos de test smells em projetos JavaScript, desenvolve uma nova ferramenta de identificação de 8 tipos de test smells para JavaScript e explora 977 projetos JavaScript disponíveis no GitHub em busca de localizar os test smells mais frequentes. Após a análise, os tipos de test smells mais frequentes entre os projetos foram: Magic Test, Conditional Test Logic, Unused Imports e Duplicate Asserts.*

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): Cleiton Silva Tavares — cleitontavares@pucminas.br

Belo Horizonte, 7 de Dezembro de 2022.

1. Introdução

Code smells referem-se às más decisões de implementação, tanto ao nível de *design* quanto de código de um *software* [Fowler 1999]. Essas decisões, quando mal executadas, podem afetar o processo de manutenção. Analogamente, a atividade de construção de um caso de teste unitário pode estar sujeita ao *code smell*. Nesse contexto, o aparecimento desse elemento é chamado por *test smells*. Além disso, similar aos *code smells*, os *test smells* podem ser mitigados por um processo de refatoração [Van Deursen et al. 2001].

O primeiro trabalho definiu o termo *tests smell* e descreveu 11 tipos [Van Deursen et al. 2001]. Consequentemente, ao longo do tempo outros autores foram caracterizando novos *test smells* para diferentes tipos de projetos e de linguagens, formando-se um catálogo de 139 tipos [Garousi and Küçük 2018]. Todavia, observa-se um enfoque em linguagens estaticamente tipadas. No entanto, ao explorar o campo de linguagens dinamicamente tipadas, observa-se a presença de *test smells* específicos de linguagem. Wang et al. (2021) criaram um *plugin* para PyCharm, visando identificar *test smells* específicos da linguagem Python. Similarmente, a linguagem JavaScript compartilha essa característica de linguagem dinamicamente tipada. Não obstante, mesmo com a popularidade da linguagem[1] e dos diversos *frameworks* de testes disponíveis, ainda são escassos trabalhos dedicados à **identificação e detecção de *test smells* para a linguagem JavaScript**.

Nesse contexto, o tema deste trabalho é relevante devido às características da linguagem do JavaScript. Visto que os trabalhos relacionados enfocam em linguagens estaticamente tipadas, enquanto o JavaScript é uma linguagem dinamicamente tipada. Além disso, possui alta popularidade no GitHub [Forsgren et al. 2021]. Portanto, justifica-se a necessidade de análise da qualidade do *design* dos casos de teste para projetos em JavaScript.

O objetivo geral deste trabalho é **identificar *test smells* implementados na linguagem JavaScript**. Portanto, definem-se os seguintes objetivos específicos: i) identificar a presença de *test smells* específicos para a linguagem Javascript; ii) criar uma ferramenta para detecção automatizada de *test smells* para a linguagem Javascript; iii) ranquear principais tipos de *test smells* presentes em projetos Javascript disponíveis no GitHub; iv) avaliar a ferramenta proposta em projetos disponíveis no GitHub.

Este trabalho propôs três contribuições para o meio acadêmico. Primeiro, foram caracterizados de 3 tipos de *test smells* na linguagem JavaScript. Em seguida, disponibilizou-se uma ferramenta de detecção de 8 tipos de *test smells* para JavaScript. Por fim, realizou-se uma análise da difusão dos *test smells* mais frequentes em 977 projetos em JavaScript.

Este trabalho está organizado em sete seções: a Seção 2 apresenta os conceitos relacionados ao trabalho. A Seção 3 discute os principais trabalhos relacionados ao tema *test smells*. A Seção 4 apresenta os materiais e métodos utilizados para o desenvolvimento. A Seção 5 apresenta os resultados obtidos. Na Seção 6 os resultados são discutidos e comparados com trabalhos relacionados. Por fim, na Seção 7 são apresentadas as conclusões e trabalhos futuros.

2. Referencial Teórico

Nesta seção são apresentados os conceitos necessários para a compreensão do trabalho. Inicialmente, é apresentado o conceito de testes unitários. Em sequência, apresenta-se sobre a linguagem JavaScript. Posteriormente, apresenta-se o Jest, um *framework* de execução de testes automatizados para JavaScript. Por fim, é apresentada a definição de *test smells* segundo a literatura.

[1] https://survey.stackoverflow.co/2022/#technology-most-popular-technologies

2.1. Teste Unitários

Os testes unitários têm por objetivo realizar o "teste da lógica interna de processamento e as estruturas de dados nos limites de um componente" [Pressman 2009]. Essas tarefas podem ser executadas tanto manualmente quanto automatizada. Em relação à execução de testes automatizados, ferramentas de apoio oferecidas pelas linguagens de programação e *frameworks* contribuem nessas atividades. Por fim, esses artefatos produzidos são armazenados em conjunto com código-fonte nas ferramentas de revisão de código [Van Deursen et al. 2001]. Consequentemente, algumas métricas relacionadas aos testes unitários começaram a ser exploradas pela literatura, como cobertura de código, *mutation score* e *test smells* [Kaczanowski 2012].

2.2. *Test Smells*

O uso de anti-padrões no ambiente de testes unitários é chamado por *test smells* [Garousi and Küçük 2018]. Esse sintoma pode ser utilizado para a avaliação de métricas de qualidade do código e esforço de manutenção [Van Deursen et al. 2001]. Assim sendo, diversos autores buscaram definir e caracterizar esses elementos, assim criando-se um catálogo de *test smells*. Desse modo, a literatura iniciou de um catálogo de 11 tipos de *test smells* [Van Deursen et al. 2001] e que cresceu ao longo do tempo para 139 tipos [Garousi and Küçük 2018]. Além disso, existem trabalhos que buscam expandir esse catálogo, identificando a presença de *test smells* específicos de linguagens e *frameworks* de testes automatizados [Wang et al. 2021, Fernandes et al. 2021].

Ademais, o catálogo de *test smells* pode ser agrupado conforme a relação dos elementos dos casos de teste [Meszaros 2007]. Pode-se citar: i) a relação ao código, identificados pela análise direta do código; ii) o comportamento, detectados durante a execução do caso de teste. Além disso, os *test smells* podem ser associados a categorias de semântica/lógica, *design*, passo do teste (ex.: *setup* e *assertions*), *mock* e *stub*, associação com código de produção e de dependência [Garousi et al. 2019]. Por outro lado, sob a perspectiva dos desenvolvedores, existe uma problemática na percepção de *test smells*. Além disso, sob a perspectiva de origem, os *test smells* não surgem pela degradação da arquitetura, mas sim inseridos nas primeiras implementações [Tufano et al. 2016].

2.3. JavaScript

O JavaScript é uma linguagem interpretada, baseada em protótipos, multi-paradigma e dinâmica. Criado pela empresa Netscape em 1995, visa gerar conteúdos dinâmicos em *Hyper Text Markup Language* (HTML) de páginas da *web* [Flanagan 2011]. No entanto, não é uma linguagem restrita a navegadores. O Node.js[2], um ambiente de execução JavaScript, permite a criação de *Application Programming Interface* (API) e flexibiliza o gerenciamento de pacotes em projetos JavaScript via *Node Package Manager* (NPM). Além disso, destaca-se a popularidade da linguagem no GitHub [Forsgren et al. 2021]. Ademais, a linguagem possui alta flexibilidade em relação ao número de *frameworks* e bibliotecas disponíveis. Por exemplo, para a criação de interfaces *web*: ReactJS, Vue e Angular; para testes automatizados podem ser utilizados Mocha, Jest, Jasmine, entre outros.

[2]https://nodejs.org/

2.4. Jest

Criado pela empresa Meta, o Jest é um *framework* de testes unitários de código aberto possuindo suporte a projetos construídos em Babel, TypeScript, Node.js, React, Angular, Vue.js e Svelte [Jestjs.io 2022]. Ademais, o Jest possui como características, popularidade [Greif 2022], uso em *websites* como Facebook, Twitter e Spotify. Além disso, ao aplicar uma pesquisa sobre o estado do JavaScript em 2018 sob a perspectiva dos desenvolvedores, Greif (2018) identificou os seguintes fatores que levam ao *framework* a ser utilizado novamente: boa documentação, baixa curva de aprendizagem, padrão e estilo de programação elegante.

3. Trabalhos Relacionados

Nesta seção apresentam-se os trabalhos relacionados a *test smells* que contribuíram com ideias, conceitos e métodos para a execução desse trabalho. Na primeira seção apresentam-se trabalhos na literatura que envolvem o catálogo de *test smells*. Na segunda seção, apresentam-se trabalhos que contribuíram com ferramentas de detecção de *test smells*.

3.1. Catálogo de *Test Smells*

Van Deursen et al. (2001) analisaram os casos de teste do projeto DocGen. Durante o processo de refatoração, os autores definiram o termo após identificar o uso de uma solução de *design* simplificada. Assim, foram definidos 11 tipos de *test smells* e 6 métodos de refatoração. Semelhantemente, este trabalho se baseia em alguns tipos de *test smells* definidos por esses autores.

Sob outra perspectiva de linguagens dinamicamente tipadas, métodos de detecção e tipos de *test smells* foram mapeados em projetos em Python. Após a análise de 90 projetos em Python, foram adaptados para a linguagem 20 tipos de *test smells* pré-existentes na literatura, e 4 novos foram acrescentados ao catálogo de *test smells*. Além disso, após a caracterização dos novos tipos, realizou-se um questionário em que foi identificada a relação desses *test smells* e as preocupações ao se deparar com algum deles em casos de teste [Fernandes et al. 2021]. De maneira similar, este trabalho utiliza algumas das novas definições elaboradas e adaptadas ao JavaScript.

3.2. Ferramentas de Detecção de *Test Smells*

Virgínio et al. (2020) propuseram a JNose para a detecção de *test smells* em projetos em JUnit do Java. A ferramenta utiliza como apoio RefActorIng test Design Errors (RAIDE), um *plugin* para Eclipse para a identificação de *test smells*, e TSVizzEvolution para a visualização da evolução dos *test smells* conforme novos *commits* eram criados. Para avaliar a ferramenta criada, os autores realizaram a análise da biblioteca commons-io da Apache Commons IO. Após a análise do projeto, os *test smells Exception Catching Throwing*, *Assertion Roulette* e o *Eager Test* foram mais frequentes. Além disso, por meio da análise da evolução dos *test smells* proposta pela ferramenta, observou-se um crescimento da quantidade ao longo do tempo, indicando que desenvolvedores não dedicaram a corrigi-los. Este trabalho compartilha as técnicas de detecção de *test smells*, no entanto, adaptadas para as especificidades do JavaScript.

Wang et al. (2021) propuseram a PyNose, uma ferramenta de detecção de *test smells* em projetos Python. Foram adaptados 17 *test smells* presentes na literatura para

a linguagem. O conjunto de dados do trabalho originou-se do GHTorrent com processamento do PGA-create. Filtrou-se por projetos Python com ao menos 50 estrelas, 10 contribuidores, 1.000 *commits*, data de modificação de até um ano, sem *fork* e com alterações em arquivos de teste, assim selecionando-se 248 projetos. Os autores utilizaram a ferramenta de Golubev et al. (2021), PythonChangeMiner que se baseou no algoritmo de Nguyen et al. (2019) para identificação de padrões em código Java. Cerca de 84% dos testes haviam ao menos um *test smell*. Além disso, registrou-se um novo *test smell* específico *Suboptimal Assert*. Apesar de linguagens diferentes, Python e JavaScript compartilham a característica de linguagem dinamicamente tipada. Portanto, este trabalho utiliza parte dos critérios de seleção dos repositórios e parte da lista de *test smells* elaborada pelo autor adaptando-se ao JavaScript.

Jorge et al. (2021) propuseram a Steel, uma ferramenta de detecção de 15 tipos de *test smells* da literatura adaptados ao JavaScript. Selecionou-se 11 repositórios JavaScript e TypeScript ativos no GitHub, que utilizavam *assert* do Node.js, Chai ou Jest, e com mais de 5000 estrelas. Os resultados obtidos indicaram que de 33% até 100% do código das suítes de testes continham *test smells*. Os *test smells* mais frequentes dos repositórios foram *Duplicate Assert*, *Magic Number Test*, *Unknown Test* e *Conditional Test Logic*. Além de uma forte correlação entre *Assertion Roulette* e *Redundant Assertion*. Os *test smells Conditional Test Logic*, *Magic Number Test* possuíam alta correlação com as métricas de qualidade de código. Este trabalho utiliza a ferramenta proposta pelos autores para detecção de *test smells*, além disso, complementa os resultados ao explorar uma maior quantidade de projetos e identifica novos tipos de *test smells* específicos da linguagem.

4. Materiais e Métodos

Este trabalho utiliza a metodologia exploratória de análise quantitativa. Justifica-se esse método ao trabalho devido se tratar de um estudo de origem de exploratória, que realiza uma análise do objeto em seu ambiente natural e a partir disso, são construídos novos conhecimentos sobre o objeto observado [Wohlin et al. 2012].

Nesta seção apresentam-se os materiais e métodos utilizados para a execução deste trabalho. São apresentados os critérios e ferramentas utilizadas para a construção do conjunto de dados, identificação de *test smells* e algoritmos. Por fim, são apresentados os métodos de avaliação dos resultados.

4.1. Conjunto de Dados

Para ranquear os principais tipos de *test smells* em projetos JavaScript no GitHub, foram utilizados os serviços de GraphQL, REST e um *script web crawler* criado especificamente para esse trabalho. Na Figura 1, são apresentadas as quatro etapas definidas para seleção dos repositórios analisados e a quantidade restante entre cada etapa.

Na primeira etapa, utilizou-se o serviço GraphQL do GitHub como ferramenta de consulta de repositórios candidatos. Foram definidos três critérios de seleção dos repositórios: i) linguagem em JavaScript; ii) possuir pelo menos 500 estrelas; iii) não ter como origem um *fork* de um outro projeto. Em vista disso, repositórios brinquedos ou *forks* de outros projetos não foram incluídos na base de dados [Wang et al. 2021]. Por meio desses critérios, foram selecionados 11.865 repositórios.

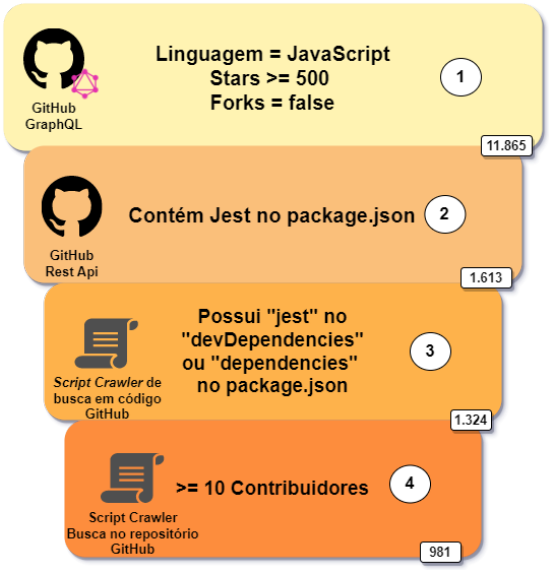

Figura 1. Seleção de repositórios candidatos

Na segunda etapa, para atingir os objetivos específicos, foi selecionado o *framework* Jest como o critério de avaliação dos casos de teste em JavaScript. Para isso, foram selecionados apenas repositórios com dependência do *framework* Jest, realizado com apoio do serviço de busca em código do GitHub[3]. A ferramenta possibilitou a identificação de repositórios que utilizam "jest" no arquivo "package.json" localizado na raiz do repositório, delimitando-se assim em 1.613 repositórios.

Ao selecionar amostras geradas pela etapa anterior, foram localizados repositórios que não utilizavam o *framework* com dependência. Portanto, na terceira etapa utilizou-se como alternativa o recurso de visualização em texto de arquivos no GitHub. Essa ferramenta de exibição serviu de insumo para o processo de filtragem de dependência do Jest no arquivo "package.json". Assim sendo, foi desenvolvida uma ferramenta customizada de *web crawling* em Python que interpreta o retorno em texto e o converte em um dicionário, permitindo assim acessar as propriedades "devDependencies" e "dependencies". Justifica-se a busca nessas propriedades, pois em projetos JavaScript as dependências são referenciadas nesse arquivo. Portanto, buscou-se nessas propriedades o valor "jest". Após a aplicação desse filtro, foram mantidos 1.324 repositórios.

Por fim, na quarta etapa, aplicou-se um novo filtro para selecionar repositórios com um número igual ou superior a 10 contribuidores com o intuito de selecionar um conjunto menor de dados recomendado pela literatura e utilizado em trabalhos relacionados [Kalliamvakou et al. 2014, Wang et al. 2021]. No entanto, não foi localizada uma interface de programação para a consulta dessa informação. Contudo, na página de cada repositório no GitHub é exibido um contador de número de contribuidores. Desse modo, adicionou-se na ferramenta de *web crawling* uma funcionalidade de busca no conteúdo de cada página dos repositórios, assim restando 981 repositórios armazenados em um arquivo no formato de Valores Separados por Vírgula (CSV, do inglês *Comma Separated Values*).

[3]https://docs.github.com/en/rest/search

4.2. Fluxo de Atividades

A Figura 2 apresenta o processo de análise dos repositórios composto por 4 etapas. O primeiro passo inicia-se com a lista de repositórios em uma lista no formato CSV elaborados conforme descritos na Seção 4.1. No segundo passo executou-se um *script* para a leitura do CSV para que fosse realizada a clonagem dos repositórios. Em sequência, para cada repositório clonado foram executadas duas ferramentas de detecção de *test smells*: i) a Steel; ii) a ferramenta de detecção criada neste trabalho.

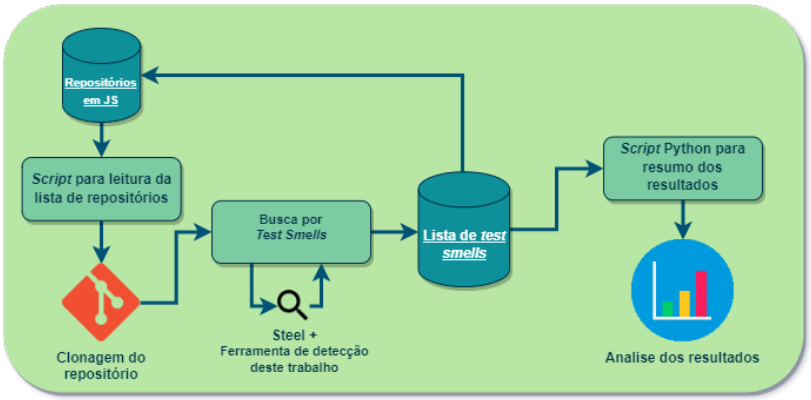

Figura 2. Processo de análise de repositórios

Após a execução do passo anterior, ambas ferramentas geraram uma lista de *test smells* em vários arquivos e diretórios. Assim sendo, no terceiro passo os resultados foram agregados em apenas um arquivo, para isso, executou-se um *script* em Python que acessou cada diretório e gerou um novo arquivo contendo apenas a quantidade de *test smells* de cada repositório, que por fim foi agregado em um único arquivo final para a análise dos resultados obtidos.

4.3. Identificação de *Test Smells*

A identificação de *test smells* deste trabalho se baseia na categoria de detecção por regras e pela identificação de padrões específicos nos casos de teste. No modelo de detecção por regras, definem-se heurísticas de detecção de *test smells*. Estas heurísticas, são utilizadas para avaliar possíveis *smells* no código por meio da análise da *Abstract Syntax Tree* (AST) [Aljedaani et al. 2021].

Para efetivação do objetivo específico de identificação dos *test smells* mais frequentes, foram analisados no total 21 tipos de *test smells*. Dessa quantidade, 18 são descritos na literatura e apresentados nesta seção, e 3 novos são definidos na Seção 5.1 deste trabalho. Sendo assim, os *test smells* caracterizados na literatura tem como origem uma ferramenta já existente para JavaScript (13 *test smells*) e 5 *test smells* foram adaptados de outras linguagens para a ferramenta construída neste trabalho. Portanto, uma das ferramentas de apoio utilizadas para a detecção de *test smells* para JavaScript é a Steel [Jorge et al. 2021]. Apesar da ferramenta detectar 15 tipos presentes na literatura e adaptados ao contexto do JavaScript. Os *test smells Eager Test* e *Lazy Test* não serão analisados neste trabalho, pois demandam a seleção manual dos arquivos testados. Desse modo, foram analisados 13 tipos de *test smells* detectados pela Steel enumerados a seguir:

1. *Assertion Roulette* (AR): sequência de métodos *assert* sem a presença de explicação sobre o cenário avaliado
2. *Conditional Test Logic* (CT): uso de condicionais (*if-else* e instruções de repetição)
3. *Duplicate Assert* (DA): uso de uma instrução de *assert* que utiliza os mesmo parâmetros
4. *Empty Test* (ET): casos de teste que não possuem uma implementação definida geralmente em um bloco vazio
5. *Exception Handling* (EH): o caso de teste possui instruções de *try*, *catch* e *throw*
6. *Ignored Test* (IT): o caso de teste não é executado devido habilitar o uso da propriedade ".skip"
7. *Sleepy Test* (ST): o caso de teste utiliza métodos que pausam a execução em uma *thread* o que pode levar a resultados inesperados
8. *Magic Number Test* (MN): uso de valores literais para avaliação dos casos de teste
9. *Redundant Assertion* (RA): os parâmetros utilizados no método de *assert* são iguais de modo que o teste seja ineficaz na detecção de falhas
10. *Redundant Print* (RP): uso de métodos de saída como console para verificar o resultado do teste
11. *Mystery Test* (MT): a suíte de teste possui dependências com recursos externos impactando negativamente o resultado e o desempenho do teste
12. *Unknown Test* (UT): ausência de instrução de *assert* no caso de teste
13. *Resource Optimism* (RO): o método de teste assume que um recurso externo existe

Contudo, para expandir o número de *test smells* analisados neste trabalho , foram explorados trabalhos relacionados a outras duas linguagens. A primeira linguagem a ser analisada foi o Python por se tratar de uma linguagem dinamicamente tipada compartilhando os desafios de detecção. Após a exploração de trabalhos que envolveram a construção de ferramentas de detecção foram adaptados mais 4 tipos de *test smells* implementados para Python [Wang et al. 2021, Fernandes et al. 2022]:

14. *Suboptimal Assert* (SA): uso de funções *assert* do *framework* inadequadas para o caso de teste analisado
15. *Verifying in Setup Method* (VSM): realização de checagens nos métodos de configuração do *framework*
16. *Non-Functional Statement* (NFS): uso de uma instrução não funcional, por exemplo, `pass` do Python ou uma função de escopo vazio no JavaScript
17. *Verbose Test* (VT): um caso de teste que possui um alto número de instruções

Além disso, existem *test smells* do catálogo que não possuem restrições para a adaptação ao JavaScript. Desse modo, realizou-se uma análise de um trabalho anterior relacionado a linguagem TTCN-3 [Neukirchen et al. 2008]. Nesse trabalho foi definido um catálogo de 37 *test smells* em casos de testes dessa linguagem. Após análise do trabalho, foi selecionado apenas um tipo de *test smell* para inclusão na ferramenta de detecção:

18. *Unused Imports* (UI): importação de um módulo não utilizado no código.

Embora as caraterísticas do JavaScript tragam um desafio para a caracterização de novos tipos de *test smells*, a literatura de *code smells* apresenta algumas soluções para a análise do problema. No cenário de trabalhos relacionados de intensificação de *code smells*, pode-se basear na análise do comportamento da linguagem [Saboury et al. 2017].

Como também, pode-se revisar as recomendações da comunidade sobre o uso de anti-padrões de código [Fard and Mesbah 2013]. Sob o mesmo ponto de vista, os *test smells* podem ser identificados pelas características dos métodos de *assert* do *framework* de teste [Wang et al. 2021].

Assim sendo, neste trabalho foram utilizadas três estratégias de análise para a caracterização de novos *test smells*. A primeira estratégia se baseou na verificação de recomendações de uso baseado na documentação[4] do *framework* Jest. Na segunda estratégia, analisou-se o Javascript Testing Best Practices[5], um guia criado pela comunidade de boas práticas para testagem em Node.js e JavaScript. Por fim, na terceira etapa examinou-se o Jest Eslint[6], uma ferramenta de *lint* para testes em JavaScript com *framework* Jest.

4.4. Avaliação dos Resultados

Com o intuito de avaliar a capacidade de identificação de *test smells* pela ferramenta construída, o desenvolvimento baseou-se no uso do Desenvolvimento Dirigido por Testes (TDD, do inglês *Test Driven Development*). Desse modo, em cada caso de teste foram avaliados trechos casos de testes com *test smells* e a quantidade de *test smells* esperada. Além disso, avaliou-se a capacidade da ferramenta analisar projetos que utilizavam *syntax plugins* disponíveis para a Babel, um compilador para JavaScript[7].

5. Resultados

Nesta seção são apresentados os resultados alcançados para o estudo proposto. Na Seção 5.1 detalham-se os *test smells* identificados, suas características e as possíveis ações para sua correção. Em sequência, na Seção 5.2, é apresentada a ferramenta desenvolvida neste trabalho. Posteriormente, na Seção 5.3 são apresentados os *test smells* mais frequentes encontrados em projetos disponíveis no GitHub.

5.1. Caracterização de *Test Smells* em JavaScript

A caracterização de *test smells* utilizada neste trabalho foi baseada em três estratégias de análise descritas na Seção 4.3. Com estas estratégias foram propostos três novos tipos de *test smells*, descritos a seguir.

1. *Identical Test Description* (ID)

O Jest oferece os métodos para organização das suítes (`describe()`) e casos de teste (`it()`) [Mulders 2022]. No Algoritmo 1, é apresentado um exemplo de um caso de teste em que o parâmetro "name" está com o valor de argumento "did not rain". Em alguns cenários, pode-se utilizar esse parâmetro para explicitar o cenário avaliado no teste[8]. Além disso, é possível observar na Figura 3 disponível no Apêndice A, que em caso de falha no teste o parâmetro é utilizado na construção da mensagem de erro.

[4]https://jestjs.io/docs/getting-started
[5]https://github.com/goldbergyoni/javascript-testing-best-practices
[6]https://github.com/jest-community/eslint-plugin-jest
[7]https://babeljs.io/
[8]https://github.com/jest-community/eslint-plugin-jest/blob/main/docs/rules/no-identical-title.md

```
// it( name, fn, timeout)
it("did not rain", () => {
    expect(inchesOfRain()).toBe(0);
});
```

Algoritmo 1. Descrição de um caso de teste no Jest

No entanto, mesmo que desenvolvedores repitam o valor do parâmetro em um mesmo arquivo e em um mesmo escopo, a ferramenta não impedirá e nem notificará o usuário. No Algoritmo 2, é apresentado um exemplo desse *test smell*, no qual se realiza uma validação de uma função `normalizeName`. Neste caso de teste, espera-se que esta função faça a normalização de uma *string* de entrada e retorne-a substituindo seus caracteres especiais e acentos. No entanto, ao olhar a implementação interna é possível observar que são avaliados dois cenários diferentes. O primeiro avalia com uma entrada "Noël and Renée" e espera que substitua apenas as acentuações. E no segundo, além da acentuação, é avaliada o mapeamento do caractere "&" para "and". No Algoritmo 3, é apresentada a proposta de solução para esse *test smell*, no qual a descrição genérica é substituída por uma em que apresenta o contexto do cenário avaliado por cada teste. Portanto, para realizar a detecção deste *test smell* na ferramenta, foi considerada a repetição da descrição de dois ou mais `it` em um escopo de um bloco *describe*.

```
describe("normalize name", () => {
  it("should normalize characters from name", () => {
        expect(normalizeName("Noel and Renée")).toBe("Noel and Renee");
  });

  it("should normalize characters from name", () => {
        expect(normalizeName("Noel & Renée")).toBe("Noel and Renee");
  });
});
```

Algoritmo 2. *Identical Test Description*

```
describe("normalize name", () => {
    it("should normalize accents from name", () => {
        expect(normalizeName("Noel and Renée")).toBe("Noel and Renee");
    });

    it("should normalize special characters from name", () => {
        expect(normalizeName("Noel & Renee")).toBe("Noel and Renee");
    });
});
```

Algoritmo 3. Correção para *Identical Test Description*

2. *Only Test* (OT)

Em determinadas situações um desenvolvedor pode necessitar realizar o *debug* de um arquivo composto por vários casos de testes. Neste cenário, o Jest oferece a propriedade `.only` que pode ser acrescentada na suíte de testes (`describe.only`) ou em um caso de teste (`it.only`) [Jestjs.io 2022]. No Algoritmo 4, é apresentado um trecho de código com o `only` habilitado no primeiro `describe` com o nome "my beverage". Neste cenário, será executado apenas o teste marcado com essa propriedade, ignorando os outros testes restantes do arquivo.

Contudo, ao deixar esse dispositivo habilitado sem um motivo definido, possivelmente influenciará tanto na compreensão do caso de teste quanto na qualidade do código

testado. Dado que semelhante ao *test smell Ignored Test*, o uso do `.only()` impacta a execução da suíte de testes e consequentemente a detecção de falhas. Diante disso, ao permitir que um trecho de código com o `.only()` ativo seja incluído nas ferramentas de controle de versão, pode fazer que os outros testes sejam ignorados. Além disso, ao manter essa funcionalidade ativa, podem surgir dúvidas em relação ao porquê do mesmo estar habilitado, e a relação de importância dos outros testes na suíte. Desse modo, para realizar a detecção deste *test smell*, considerou o uso dessa propriedade na suíte e no caso de teste.

```
describe.only("my beverage", () => {
    test("is delicious", () => {
        expect(myBeverage.delicious).toBeTruthy();
    });

    test("is not sour", () => {
        expect(myBeverage.sour).toBeFalsy();
    });
});

describe("my other beverage", () => {
    // ... will be skipped
    expect(myBeverage.sweet).toBeFalsy();
});
```

Algoritmo 4. ***Only Test***

3. *Complex Snapshot* (CS)

O Jest também possibilita o teste de interface de usuário por meio de *snapshots*. Essa funcionalidade realiza a comparação entre a referência de um componente React com o caso de teste atual. Além disso, essa funcionalidade pode ser utilizada tanto em arquivos externos (`toMatchSnapshot`) do teste quanto internos (`toMatchInlineSnapshot`). Consequentemente, essa funcionalidade tem como propósito evitar a realização de uma checagem manual do comportamento do componente. Todavia, o teste de *snapshot* não indica se o resultado está correto ou não, apenas indica que existe uma diferença entre a saída do teste e o *snapshot* atual [Jestjs.io 2022]. No Algoritmo 5, é apresentado um exemplo de um *Complex Snapshot*.

```
exports[`Should render form in read-only mode 1`] = `
<form  className="Form">
  <h2>
    Read-only form
  </h2>
  <label>
    Name
    <input
      onChange={[Function]}
      readOnly={true}
      type="text"
      value="Test name"
    />
  </label>
  ... 50 lines ...
</form>
`;
```

Algoritmo 5. ***Complex Snapshot***

Portanto, quando um teste de *snapshot* falha, é preciso inspecionar o motivo da falha. Visto que o componente testado pode ter sido modificado por um *bug* introduzido ou

sua referência do *snapshot* pode estar desatualizada em relação à versão atual da interface [Nakazawa 2016]. Assim sendo, essa análise pode se tornar uma tarefa complexa, dado que é preciso inspecionar o *snapshot* para identificar o motivo da falha do teste. Portanto, a resolução deste *test smell* envolve a prática de redução do tamanho dos *snapshots*. Desse modo, uma sugestão de solução deste *test smell* é a redução dos componentes simultâneos testados, de modo que os *snapshots* serão mais específicos e menos frágeis a mudanças de interface[9]. Além disso, a preferência pelo uso `toMatchInlineSnapshot` é uma prática recomendada. Desse modo, definiu-se como limiar de detecção o número superior a 50 linhas[10].

5.2. Ferramenta de Identificação de *Test Smells*

A ferramenta construída neste trabalho realiza a detecção de 8 tipos de *test smells*, sendo 3 descritos na Seção 5.1 e 5 tipos adaptados ao contexto do JavaScript, com base na caracterização por outras ferramentas e linguagens, conforme apresentado na Seção 4.3. Na Tabela 5 disponibilizada no Apêndice B, é apresentado uma lista comparativa das ferramentas e de seus respectivos *test smells* detectados em relação a nova ferramenta proposta neste trabalho.

Assim sendo, ao comparar os tipos detectados entre as ferramentas de detecção de *test smells* disponíveis para Python e outras linguagens, foram implementados 5 tipos não mapeados pela ferramenta Steel. Os tipos *Non-Functional Statement*, *Suboptimal Assert*, *Verifying in Setup Method*, *Verbose Test*, são relacionados a ferramentas de detecção em Python [Wang et al. 2021, Fernandes et al. 2022]. E o tipo *Unused Imports*, pertence à linguagem TTCN [Neukirchen et al. 2008], adaptado ao cenário de importação de módulos e pacotes do JavaScript. Além disso, os 3 tipos de *test smells* restantes *Only Usage*, *Complex Snapshots* e *Identical Test Description* foram implementados conforme as regras definidas na Seção 5.1.

5.3. Frequência dos *Test Smells*

Conforme os critérios definidos na Seção 4.1, foram selecionados 981 repositórios. No entanto, 4 repositórios não conseguiram ser clonados, restando-se 977 projetos. Desses repositórios analisados, 34 não continham arquivos de teste detectados, além disso, o somatório dos repositórios que possuíam arquivos de teste foi de 39.036 arquivos. Na Tabela 1, é apresentado um resumo estatístico dos repositórios analisados. Na primeira coluna apresentam-se as métricas avaliadas. Na segunda coluna, apresentam-se a média dos valores ($\bar{x}$). Na terceira coluna, o desvio padrão (s). Na quarta coluna, os valores mínimos (min). Na quinta, o primeiro quartil (Q_1). Na sexta coluna, o segundo quartil (Q_2). Na sétima coluna, o terceiro quartil (Q_3). E por fim, na oitava coluna, os valores máximos de cada métrica (max).

[9]https://github.com/goldbergyoni/javascript-testing-best-practices#-%EF%B8%8F-18-if-needed-use-only-short–inline-snapshots

[10]https://github.com/jest-community/eslint-plugin-jest/blob/main/docs/rules/no-large-snapshots.md

Tabela 1. Informações gerais sobre os repositórios analisados

Medida	$\bar{x}$	s	min	Q_1	Q_2	Q_3	max
nº de contribuidores	80,44	216,04	10,0	18,0	32,0	65,0	3.917,0
nº de estrelas	4.325,25	9.809,10	500,0	854,0	1.637,0	3.937,0	196.928,0
nº de arq. testes	54,19	241,18	0,0	3,0	11,0	37,0	5.257,0
arq. com *smells*	36,38	113,72	0,0	2,0	9,0	27,0	1.520,0
nº de casos de teste	213,42	656,31	0,0	15,0	53,0	151,0	10.951,0
nº de suítes de teste	66,55	259,67	0,0	2,0	11,0	39,0	4.411,0

Em sequência, esses repositórios foram analisados com duas ferramentas de detecção de *test smells* para JavaScript: Steel e a ferramenta criada neste trabalho. Na Tabela 2 é apresentado uma análise do resumo estatístico dos 977 repositórios, e a relação de cada com os 21 tipos de *test smells* analisados. A organização das colunas dessa tabela é similar a Tabela 1, porém ao final é acrescentada uma nova coluna (Σ), contendo o somatório da quantidade total de *test smells* detectada para cada tipo. Na Tabela 3 se concentra nos 10 repositórios com maior número de estrelas e apresenta a quantidade de cada um dos 21 tipos de *test smells* analisados e a quantidade total encontrada. Por fim, na Tabela 4 destacam-se os repositórios com maior quantidade de *test smells* de cada tipo. Desse modo, na primeira coluna são listados os tipos de *test smells*, na segunda o nome do repositório e na terceira coluna a quantidade de *test smell* encontrada. Possibilitando identificar *outliers* que tiveram maior impacto na quantidade total de *test smells*.

Tabela 2. Resumo estatístico dos repositórios analisados

Nome *Test Smell*	$\bar{x}$	s	min	Q_1	Q_2	Q_3	max	Σ
Assertion Roulette	5,80	71,79	0,0	0,0	0,0	0,0	1.528,0	5667,0
Complex Snapshots	0,18	2,14	0,0	0,0	0,0	0,0	57,0	180,0
Conditional Test Logic	48,61	299,10	0,0	0,0	3,0	15,0	6.005,0	47.497,0
Duplicate Asserts	31,17	235,27	0,0	0,0	0,0	8,0	5.982,0	30.458,0
Empty Test	0,35	3,63	0,0	0,0	0,0	0,0	78,0	345,0
Exception Handling	4,62	24,96	0,0	0,0	0,0	1,0	573,0	4.515,0
Identical Test Description	14,14	59,87	0,0	0,0	0,0	6,0	938,0	13.819,0
Ignored Test	0,89	4,24	0,0	0,0	0,0	0,0	79,0	879,0
Magic Test	61,09	320,27	0,0	0,0	6,0	31,0	8.076,0	89.694,0
Mystery Test	1,08	5,92	0,0	0,0	0,0	0,0	110,0	1.058,0
Non Functional Statement	12,90	63,80	0,0	0,0	0,0	5,0	1.025,0	12.605,0
Only Test	0,03	0,35	0,0	0,0	0,0	0,0	6,0	37,0
Redundant Assertion	0,71	8,72	0,0	0,0	0,0	0,0	177,0	696,0
Redundant Print	2,55	15,25	0,0	0,0	0,0	0,0	403,0	2.501,0
Resource Optimism	0,01	0,31	0,0	0,0	0,0	0,0	9,0	14,0
Sleepy Test	1,33	6,83	0,0	0,0	0,0	0,0	127,0	1.302,0
Suboptimal Assert	17,53	84,21	0,0	0,0	0,0	7,0	1.543,0	17.128,0
Unknown Test	30,39	113,35	0,0	0,0	2,0	15,0	1.501,0	29.696,0
Unused Imports	36,95	169,28	0,0	0,0	3,0	16,0	3.058,0	36.107,0
Verbose Test	15,48	145,24	0,0	0,0	1,0	6,0	4.349,0	15.126,0
Verifying In Setup Method	0,07	1,11	0,0	0,0	0,0	0,0	33,0	76,0

6. Discussão dos Resultados

Após a análise dos repositórios, algumas observações foram realizadas em relação aos resultados. É possível observar na coluna (min) da Tabela 2, que nenhum tipo de *test smell* esteve presente em todos os repositórios. Por outro lado, 37 projetos (3,78% do total) estavam livres de *test smells*. Além disso, ao comparar o resultado deste trabalho com o trabalho relacionado de Jorge et al. (2021), ambos identificamos 3 tipos de *test smells* mais frequentes: *Duplicate Assert*, *Magic Number Test* e *Conditional Test Logic*, porém em ordem diferente. Além disso, neste trabalho é analisado o tipo *Unused Import*

não analisado no trabalho relacionado, sendo o terceiro tipo de *test smell* mais frequente. É preciso ressaltar que o trabalho relacionado analisou uma quantidade menor de repositórios (11), além de analisar repositórios com outros *frameworks* de teste além do Jest (Chai e o módulo *assert* do Node.js).

Tabela 3. Quantidade de *test smells* por projetos populares

	AR	CS	CTL	DA	ET	EH	ID	IT	MT	MG	NFS	OT	RA	RP	RO	ST	SA	UT	UI	VT	VSM
react	0	14	2.621	2.284	1	573	143	9	1.181	0	483	6	0	403	0	22	648	1.501	62	884	4
create-react-app	0	0	20	2	2	7	0	0	29	21	3	0	0	0	0	0	5	47	126	3	0
json-server	0	0	5	0	0	0	21	0	13	1	0	0	0	0	0	3	0	96	0	3	0
webpack	0	3	1.493	278	78	110	20	12	712	110	261	0	0	5	0	39	197	352	552	71	0
tailwindcss	0	0	19	4	0	2	5	1	61	17	5	1	0	0	0	0	55	510	119	5	0
gatsby	0	6	553	2	0	0	0	58	0	0	80	0	0	8	0	0	236	19	272	112	0
mermaid	0	0	37	82	2	17	29	5	548	1	0	0	0	0	0	0	434	105	35	64	0
strapi	0	57	125	148	0	72	170	6	1.220	0	103	0	0	19	0	4	146	307	884	81	0
github-readme-stats	0	1	1	6	0	1	0	0	3	0	0	0	0	1	0	0	2	40	15	3	0
prettier	0	0	1.056	2	0	1	20	3	40	0	1.025	5	0	9	0	2	4	15	251	7	0
Total	0	111	5.930	2.808	83	783	408	95	3.807	150	1.960	12	0	445	0	70	1.727	2.992	2.316	1.233	4

AR: *Assertion Roulette* **IT**: *Ignored Test* **RO**: *Resource Optimism*
CS: *Complex Snapshots* **MT**: *Magic Test* **ST**: *Sleepy Test*
CTL: *Conditional Test Logic* **MG**: *Mystery Guest* **SA**: *Suboptimal Assert*
DA: *Duplicate Asserts* **NFS**: *Non Functional Statement* **UI**: *Unused Imports*
ET: *Empty Test* **OT**: *Only Test* **UT**: *Unknown Test*
EH: *Exception Handling* **RA**: *Redundant Assertion* **VT**: *Verbose Test*
ID: *Identical Test Description* **RP**: *Redundant Print* **VSM**: *Verifying in Setup Method*

Tabela 4. Projetos com maior quantidade de cada tipo

Tipo de *Test Smell*	Repositório	Quantidade
Assertion Roulette	DevExpress/DevExtreme	1.528
Complex Snapshots	strapi/strapi	57
Conditional Test Logic	DevExpress/DevExtreme	6.005
Duplicate Asserts	diegomura/react-pdf	5.982
Empty Test	webpack/webpack	78
Exception Handling	facebook/react	573
Identical Test Description	chrisleekr/binance-trading-bot	938
Ignored Test	TheOdinProject/javascript-exercises	79
Magic Test	diegomura/react-pdf	8.076
Mystery Guest	webpack/webpack	110
Non Functional Statement	prettier/prettier	1.025
Only Test	facebook/react	6
Redundant Assertion	pugjs/pug	177
Redundant Print	facebook/react	403
Resource Optimism	indexzero/nconf	9
Sleepy Test	CartoDB/cartodb	127
Suboptimal Assert	bootstrap-vue/bootstrap-vue	1.543
Unknown Test	facebook/react	1.501
Unused Imports	Automattic/wp-calypso	3.058
Verbose Test	DevExpress/DevExtreme	4.349
Verifying In Setup Method	CartoDB/cartodb	33

Ademais, na Tabela 3, é possível perceber que os três *test smells* caracterizados: *Complex Snapshots*, *Identical Test Description* e *Only Test*, apareceram em 9 dos 10 repositórios. Além disso, nota-se que os *test smells* mais frequentes dos repositórios populares foram *Conditional Test Logic*, *Magic Test*, *Duplicate Asserts* e *Unknown Test*. Destaca-se

também que na lista dos 10 projetos mais populares, encontra-se o "facebook/react". Pertencente a empresa criadora do Jest (Meta), é o repositório que apresenta o maior número de *Suboptimal Assert*. Além disso, na Tabela 4 esse repositório obteve a maior quantidade de *test smells* em 4 tipos: *Exception Handling*, *Only Test*, *Redundant Print* e *Unknown Test*. Por fim, dentre os *test smells* específicos apresentados, o *Identical Test Description* foi o mais recorrente, estando presente em 482 projetos explorados (49,33% do total).

Ademais, observou-se que a ferramenta possui maior resiliência em coletar métricas que em relação a Steel. Pois, a ferramenta desenvolvida neste trabalho tratou de detectar múltiplos *plugins* interpretadores e de sintaxe para JavaScript. Desse modo, foi possível executar em vários repositórios no GitHub com diferentes tipos de *plugins* de sintaxe. Após observar essa diferença, transferiu-se a configuração usada neste trabalho para a Steel, para que ambas conseguissem analisar o maior número de repositórios disponíveis.

Por fim, destacam-se duas ameaças à validade deste trabalho. A primeira ameaça, é a definição dos novos *test smells* caracterizados para a linguagem JavaScript. Pois, mesmo que o trabalho tenha se baseado em guias e ferramentas relevantes e utilizadas pela comunidade, existe a necessidade de investigação em trabalhos futuros quanto à sua relevância e ao esforço necessário para refatoração. A segunda ameaça é que os resultados obtidos não podem ser considerados universais para todos os projetos em JavaScript, pois o conjunto de dados foi selecionado com base em um recorte de critérios como o número de estrelas, de contribuidores e o uso do Jest como *framework* de testes.

7. Conclusão

Neste trabalho foram identificados e caracterizados os *test smells* presentes em projetos na linguagem JavaScript. Foram definidos três novos tipos de *test smells*: *Complex Snapshots*, *Identical Test Description* e *Only Test*. Além disso, desenvolveu-se uma ferramenta de detecção de 8 tipos de *test smells*. Desses, 3 tipos são exclusivos do JavaScript sendo caracterizados neste trabalho. E os 5 restantes, foram adaptados conforme as definições apresentadas na literatura [Wang et al. 2021, Fernandes et al. 2021, Neukirchen et al. 2008].

Em sequência, foram ranqueados os tipos de *test smells* mais recorrentes entre 977 projetos em JavaScript disponíveis no GitHub. Essa análise foi realizada com o apoio de duas ferramentas de identificação automatizada, a ferramenta elaborada neste trabalho e a Steel, realizando-se a análise de 21 tipos de *test smells*. Desses repositórios analisados, os *test smells* mais frequentes foram *Magic Test*, *Conditional Test Logic*, *Unused Imports* e *Duplicate Asserts*. Em trabalhos futuros, cogita-se integrar essa ferramenta em ambientes de desenvolvimento integrado, como também desenvolver um *plugin* para o uso em sistemas de integração contínua. Além disso, seria interessante realizar uma análise dos impactos desses *test smells* sobre as perspectivas dos desenvolvedores, como a gravidade de cada tipo e o esforço necessário para refatoração.

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-andrew-costa`

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. In *Evaluation and Assessment in Software Engineering*, EASE 2021, page 170–180, New York, NY, USA. Association for Computing Machinery.

Fard, A. M. and Mesbah, A. (2013). Jsnose: Detecting javascript code smells. In *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 116–125.

Fernandes, D., Machado, I., and Maciel, R. (2021). Handling test smells in python: Results from a mixed-method study. In *Proceedings of the XXXV Brazilian Symposium on Software Engineering*, SBES '21, page 84–89, New York, NY, USA. Association for Computing Machinery.

Fernandes, D., Machado, I., and Maciel, R. (2022). Tempy: Test smell detector for python. In *Proceedings of the XXXVI Brazilian Symposium on Software Engineering*, SBES '22, page 214–219, New York, NY, USA. Association for Computing Machinery.

Flanagan, D. (2011). *JavaScript: The Definitive Guide*. Definitive Guides. O'Reilly Media.

Forsgren, N., Kalliamvakou, E., and Noland, A. (2021). The state of the octoverse: The state of the octoverse explores a year of change with new deep dives into writing code faster, creating documentation and how we build sustainable communities on github.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA.

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

Garousi, V. and Küçük, B. (2018). Smells in software test code: A survey of knowledge in industry and academia. *Journal of Systems and Software*, 138:52–81.

Golubev, Y., Li, J., Bushev, V., Bryksin, T., and Ahmed, I. (2021). Changes from the trenches: Should we automate them?

Greif, S. (2018). The state of javascript 2018: Testing - jest. `https://2018.stateofjs.com/testing/jest/`. Online; Acesso em 30 Novembro 2022.

Greif, S. (2022). The state of js 2021. `https://2021.stateofjs.com/en-US/libraries/testing`. Online; Acesso em 30 Novembro 2022.

Jestjs.io (2022). Jestjs.io. `https://jestjs.io`. Online; Acesso em 30 Novembro 2022.

Jorge, D., Machado, P., and Andrade, W. (2021). Investigating test smells in javascript test code. In *Brazilian Symposium on Systematic and Automated Software Testing*, SAST'21, page 36–45, New York, NY, USA. Association for Computing Machinery.

Kaczanowski, T. (2012). *Practical Unit Testing with TestNG and Mockito*. Tomasz Kaczanowski, POL.

Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D. M., and Damian, D. (2014). The promises and perils of mining github. In *Proceedings of the 11th working conference on mining software repositories*, pages 92–101.

Meszaros, G. (2007). *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley Signature Series. Addison-Wesley, Upper Saddle River, NJ.

Mulders, M. (2022). Jest testing tutorial: 5 easy steps. `https://www.testim.io/blog/jest-testing-a-helpful-introductory-tutorial/`. Online; Acesso em 30 Novembro 2022.

Nakazawa, C. (2016). Jest 14.0: React tree snapshot testing. `https://jestjs.io/blog/2016/07/27/jest-14`. Online; Acesso em 30 Novembro 2022.

Neukirchen, H., Zeiss, B., and Grabowski, J. (2008). An approach to quality engineering of ttcn-3 test specifications. *International Journal on Software Tools for Technology Transfer*, 10(4):309–326.

Nguyen, H. A., Nguyen, T. N., Dig, D., Nguyen, S., Tran, H., and Hilton, M. (2019). Graph-based mining of in-the-wild, fine-grained, semantic code change patterns. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 819–830.

Pressman, R. (2009). *Software Engineering: A Practitioner's Approach*. McGraw-Hill, Inc., USA, 7 edition.

Saboury, A., Musavi, P., Khomh, F., and Antoniol, G. (2017). An empirical study of code smells in javascript projects. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 294–305.

Spadini, D., Schvarcbacher, M., Oprescu, A.-M., Bruntink, M., and Bacchelli, A. (2020). Investigating severity thresholds for test smells. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR '20, page 311–321, New York, NY, USA. Association for Computing Machinery.

Tufano, M., Palomba, F., Bavota, G., Di Penta, M., Oliveto, R., De Lucia, A., and Poshyvanyk, D. (2016). An empirical investigation into the nature of test smells. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, page 4–15, New York, NY, USA. Association for Computing Machinery.

Van Deursen, A., Moonen, L., Van Den Bergh, A., and Kok, G. (2001). Refactoring test code. In *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*, pages 92–95. Citeseer.

Virgínio, T., Martins, L., Rocha, L., Santana, R., Cruz, A., Costa, H., and Machado, I. (2020). Jnose: Java test smell detector. In *Proceedings of the 34th Brazilian Symposium on Software Engineering*, SBES '20, page 564–569, New York, NY, USA. Association for Computing Machinery.

Wang, T., Golubev, Y., Smirnov, O., Li, J., Bryksin, T., and Ahmed, I. (2021). Pynose: A test smell detector for python. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 593–605. IEEE.

Wohlin, C., Runeson, P., Hst, M., Ohlsson, M. C., Regnell, B., and Wessln, A. (2012). *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated.

Apêndice A Figuras

Neste apêndice são apresentadas as figuras utilizadas neste trabalho.

Figura 3. Mensagem de falha em um caso de teste no Jest

Apêndice B Tabelas

Nesta seção do Apêndice, apresenta-se a tabela comparativa entre as ferramentas de detecção de *test smells* utilizada na Seção 5.2. Para cada tipo de *test smell*, são apresentados a relação entre as ferramentas de detecção de trabalhos relacionados ao Python: PyNose e Tempy, como também ao JavaScript: Steel e a ferramenta desenvolvida neste trabalho. Desse modo, as colunas assinaladas com um "x" indicam o suporte para a detecção do respectivo *test smell*.

Tabela 5. Comparação de detecção de *test smells* entre ferramentas

Test Smells	PyNose	Tempy	Steel	Este Trabalho
Assertion Roulette	x		x	
Complex Snapshots				x
Conditional Test Logic	x	x	x	
Constructor Initialization	x			
Default Test	x			
Duplicate Assertion	x		x	
Eager Test	x		x[1]	
Empty Test	x		x	
Exception Handling	x	x	x	

[1] Não analisados no trabalho.
[2] Considerou-se o uso de escopo vazio https://stackoverflow.com/a/39373292
[3] Com base em um limiar de 13 instruções [Spadini et al. 2020]

Tabela 5. Comparação de detecção de *test smells* entre ferramentas

General Fixture	x			
Identical Test Description				x
Ignored Test	x		x	
Lack of Cohesion of Test Cases	x			
Lazy Test			x[1]	
Magic Number	x		x	
Mystery Guest			x	
Non-Functional Statement[2]		x		x
Only Test				x
Programming Paradigms Blend		x		
Redundant Assertion	x		x	
Redundant Print		x	x	
Resource Optimism			x	
Sleepy Test	x	x	x	
Suboptimal Assert	x			x
Verifying in Setup Method		x		x
Verbose Test[3]		x		x
Test Maverick	x			
Unknown Test	x	x	x	
Unused Imports				x

[1] Não analisados no trabalho.
[2] Considerou-se o uso de escopo vazio https://stackoverflow.com/a/39373292
[3] Com base em um limiar de 13 instruções [Spadini et al. 2020]