UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
UNIDADE ACADÊMICA DE SISTEMAS E COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

DALTON NICODEMOS JORGE

UMA INVESTIGAÇÃO SOBRE TEST SMELLS EM CÓDIGOS DE TESTES JAVASCRIPT

CAMPINA GRANDE - PB
2024

Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Uma Investigação sobre Test Smells em Códigos de Testes JavaScript

Dalton Nicodemos Jorge

Tese submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Doutor em Ciência da Computação.

Área de Concentração: Ciência da computação
Linha de Pesquisa: Engenharia de Software

Patricia D. L. Machado e Wilkerson L. Andrade
(Orientadores)

Campina Grande, Paraíba, Brasil

J82i Jorge, Dalton Nicodemos.
Uma investigação sobre *Test Smells* em códigos de testes JavaScript / Dalton Nicodemos Jorge. – Campina Grande, 2024.
143 f. : il. color.

Tese (Doutorado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2024.
"Orientação: Profa. Dra. Patrícia Duarte de Lima Machado, Prof. Dr. Wilkerson de Lucena Andrade".
Referências.

1. Engenharia de Software. 2. Teste de Software – Desenvolvimento de Sistemas. 3. Linguagem JavaScript. 4. Estudos de *Test Smells* – Linguagem JavaScript. I. Machado, Patrícia Duarte de Lima. II. Andrade, Wilkerson de Lucena. III. Título.

CDU 004.41(043)

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECÁRIA SEVERINA SUELI DA SILVA OLIVEIRA CRB-15/225

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
POS-GRADUACAO EM CIENCIA DA COMPUTACAO
Rua Aprígio Veloso, 882, Edifício Telmo Silva de Araújo, Bloco CG1, - Bairro Universitário, Campina Grande/PB, CEP 58429-900
Telefone: 2101-1122 - (83) 2101-1123 - (83) 2101-1124
Site: http://computacao.ufcg.edu.br - E-mail: secretaria-copin@computacao.ufcg.edu.br / copin@copin.ufcg.edu.br

FOLHA DE ASSINATURA PARA TESES E DISSERTAÇÕES

DALTON NICODEMOS JORGE

UMA INVESTIGAÇÃO SOBRE TEST SMELLS EM CÓDIGOS DE TESTE JAVASCRIPT

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação como pré-requisito para obtenção do título de Doutor em Ciência da Computação.

Aprovada em: 15/09/2023

Profa. Dra. PATRICIA DUARTE DE LIMA MACHADO, UFCG, Orientadora

Prof. Dr. WILKERSON DE LUCENA ANDRADE, UFCG, Orientador

Profa. Dra. MELINA MONGIOVI BRITO LIRA, UFCG, Examinadora Interna

Profa. Dra. SABRINA DE FIGUEIRÊDO SOUTO, UEPB, Examinadora Interna

Prof. Dr. IVAN DO CARMO MACHADO, UFBA, Examinador Externo

Prof. Dr. LEOPOLDO MOTTA TEIXEIRA, UFPE, Examinador Externo

Documento assinado eletronicamente por **Ivan do Carmo Machado**, **Usuário Externo**, em 15/09/2023, às 15:54, conforme horário oficial de Brasília, com fundamento no art. 8°, caput, da Portaria SEI n° 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **PATRICIA DUARTE DE LIMA MACHADO**, **PROFESSOR(A) DO MAGISTERIO SUPERIOR**, em 15/09/2023, às 21:49, conforme horário oficial de Brasília, com fundamento no art. 8°, caput, da Portaria SEI n° 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **WILKERSON DE LUCENA ANDRADE**, **PROFESSOR(A) DO MAGISTERIO SUPERIOR**, em 15/09/2023, às 22:33, conforme horário oficial de Brasília, com fundamento no art. 8°, caput, da Portaria SEI n° 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **Leopoldo Motta Teixeira**, **Usuário Externo**, em 15/09/2023, às 23:26, conforme horário oficial de Brasília, com fundamento no art. 8°, caput, da Portaria SEI n° 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **MELINA MONGIOVI CUNHA LIMA SABINO**, **COORDENADOR DE POS-GRADUACAO**, em 16/09/2023, às 12:01, conforme horário oficial de Brasília, com fundamento no art. 8°, caput, da Portaria SEI n° 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **Sabrina de Figueiredo Souto**, **Usuário Externo**, em 18/09/2023, às 09:12, conforme horário oficial de Brasília, com fundamento no art. 8°, caput, da Portaria SEI n° 002, de 25 de outubro de 2018.

A autenticidade deste documento pode ser conferida no site https://sei.ufcg.edu.br/autenticidade, informando o código verificador **3785483** e o código CRC **EC37F5A5**.

Resumo

Teste de *software* é uma atividade imprescindível no desenvolvimento de sistemas, pois minimiza a possibilidade de manifestar problemas e maximiza as chances do código de produção estar conforme suas especificações. No entanto, assim como uma má escolha de design (*bad smell*) afeta o código de produção, os testes também estão suscetíveis ao mesmo tipo de problema. Sendo assim, um teste com indicação de possíveis defeitos no *design* (*test smell*) pode ocasionar adversidades que vão desde a legibilidade do código, dificultando a manutenção, até mesmo a estar suscetível a fornecer falsos positivos ou simplesmente parar de funcionar. De fato, *test smells* são anti-padrões adotados na implementação do código de teste. Esses *smells* podem ter variações ou desencadear o surgimento de novos tipos, dependendo da linguagem ou tecnologia utilizada para a escrita dos testes.

Em paralelo, a linguagem JavaScript tem se desenvolvido ao longo de décadas desde sua criação e uma notável evolução desde 2015 com a especificação ECMAScript 6 (ES6). JavaScript é uma linguagem de alto nível, que inclui as seguintes particularidades: ser multi-paradigma, interpretada, utilizar tipagem dinâmica, possuir uma orientação a objeto baseada em protótipos e até mesmo funções de primeira classe. Diante disso, focamos nosso estudo de *smells* em *JavaScript*, pelo fato dela ser uma linguagem em constante evolução e largamente utilizada para desenvolver software em diversas plataformas, tais como: servidores, *desktop*, linha de comando, sistemas móveis e internet das coisas.

Em síntese, o objetivo deste trabalho é investigar a ocorrência e a severidade de *test smells* clássicos e específicos na linguagem *JavaScript*, e verificar se estes anti-padrões podem estar correlacionados e com as métricas de qualidade do código de teste. Para alcançar este objetivo, inicialmente selecionamos 15 *smells* clássicos da literatura e um inerente à linguagem, para avaliar a ocorrência no código de teste. Com o intuito de facilitar a detecção de *smells* e validar o nosso estudo, desenvolvemos uma ferramenta de detecção automática desses 16 *smells* através da técnica de análise estática de código e executamos um estudo exploratório com 65 projetos, sendo 61 de código-livre e 4 de

código-fechado. Como resultados, concluímos que as incidências de alguns destes *test smells* estão moderadamente correlacionadas entre si e com as métricas de qualidade em sentido positivo.

Adicionalmente, realizamos um estudo complementar envolvendo desenvolvedores e testadores com o intuito de investigar o impacto da experiência no reconhecimento desses *smells* em *JavaScript*. Este estudo visou entender não apenas a prevalência de *test smells* específicos na percepção desses profissionais, mas também as estratégias e ferramentas utilizadas por eles para mitigar tais anti-padrões. Este aspecto do estudo nos permitiu avaliar a conscientização e as práticas adotadas no contexto real de desenvolvimento e teste de *software*, enriquecendo a compreensão dos desafios enfrentados e das possíveis soluções aplicáveis.

Com este trabalho esperamos ter contribuído para a formalização inicial dos estudos de *test smells* na linguagem *JavaScript*, tendo em vista que embora venha sendo cada vez mais utilizada nos mais diversos contextos de desenvolvimento, ainda é muito pouco investigada quanto as suas especificidades e desdobramentos no âmbito de *test smells*.

Abstract

Software testing is an essential activity in system development, as it minimizes the possibility of manifesting problems and maximizes the chances that the production code meets its specifications. However, just as a poor design choice (bad smell) affects the production code, tests are also susceptible to the same type of problem. Thus, a test indicating possible design defects (test smell) can lead to challenges ranging from code readability, making maintenance difficult, to being prone to giving false positives or simply ceasing to function. Indeed, test smells are anti-patterns adopted in the test code implementation. These smells can vary or trigger the emergence of new types, depending on the language or technology used for test writing.

In parallel, the JavaScript language has developed over decades since its inception and a notable evolution since 2015 with the ECMAScript 6 (ES6) specification. JavaScript is a high-level language, which includes the following peculiarities: being multi-paradigm, interpreted, using dynamic typing, having prototype-based object orientation, and even first-class functions. Given this, we focused our study on smells in JavaScript, because it is a constantly evolving language and widely used to develop software on various platforms, such as: servers, desktop, command line, mobile systems, and the internet of things.

In summary, the aim of this study is to investigate the occurrence and severity of classic and specific test smells in the JavaScript language, and to check if these anti-patterns might be correlated with each other and with the quality metrics of the test code. To achieve this goal, we initially selected 15 classic smells from the literature and one inherent to the language, to assess their occurrence in the test code. To facilitate the detection of smells and validate our study, we developed an automated detection tool for these 16 smells using the static code analysis technique and conducted an exploratory study with 65 projects, 61 open-source and 4 proprietary. As results, we concluded that the incidence of some of these test smells is moderately correlated with each other and with quality metrics in a positive sense.

Additionally, we conducted a complementary study involving developers and testers with the aim of investigating the impact of experience on the recognition of these smells

in JavaScript. This study aimed to understand not only the prevalence of specific test smells in the perception of these professionals, but also the strategies and tools they used to mitigate such anti-patterns. This aspect of the study allowed us to assess the awareness and practices adopted in the real context of software development and testing, enriching the understanding of the challenges faced and the possible applicable solutions.

With this study, we hope to have contributed to the initial formalization of test smell studies in the JavaScript language, considering that although it has been increasingly used in various development contexts, it had not yet been academically explored regarding its specifics and ramifications in the realm of test smells.

Dedicatória

Dedico este trabalho aos pilares da minha vida: meus pais e irmãos, cujo amor incondicional e apoio inabalável foram meu farol nas noites mais escuras; minha esposa, companheira de vida, cuja paciência, compreensão e encorajamento me inspiraram a buscar sempre o meu melhor; e a minha família estendida, incluindo meus sogros e cunhados, que me acolheram como mais um dos seus e ofereceram suporte e carinho em cada etapa desta jornada.

Agradecimentos

Primeiramente, gostaria de expressar minha profunda gratidão aos meus orientadores, Prof.ª Dra. Patricia Machado e Prof. Dr. Wilkerson Andrade, cuja sabedoria, dedicação e orientação foram fundamentais para a realização deste trabalho. Seu comprometimento não apenas com a excelência acadêmica, mas também com o meu desenvolvimento pessoal e profissional, foi uma fonte de inspiração constante.

Aos meus colegas de doutorado, agradeço pelas discussões estimulantes, pela solidariedade nos desafios e pela amizade que transcendeu os muros da universidade. Cada um de vocês contribuiu para tornar esta jornada mais rica e gratificante.

Um agradecimento especial aos funcionários da Universidade Federal de Campina Grande, cuja dedicação, muitas vezes nos bastidores, garantiu um ambiente propício ao nosso crescimento acadêmico e pessoal. Sua assistência, paciência e bom humor não passaram despercebidos.

À minha família, meu porto seguro, agradeço por entenderem minhas ausências, celebrarem minhas vitórias e me oferecerem um amor que energiza e conforta. Minha esposa, em particular, merece um agradecimento especial por sua paciência, amor e incansável apoio, sendo uma fonte de paz e equilíbrio.

À Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) pela concessão da bolsa, que foi indispensável para o desenvolvimento da pesquisa.

Por fim, agradeço a todos que, de alguma forma, contribuíram para a minha formação e para a conclusão deste trabalho. Esta tese não é apenas um reflexo do meu esforço, mas também do apoio e carinho que recebi ao longo deste caminho.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Siglas e Acrônimos

Steel *teST smEll dEtection tooL*

AR *Assertion Roulette*

AST *Abstract Syntax Tree*

CC *Cyclomatic Complexity*

CLI *Command Line Interface*

CSV *Comma-separeted Values*

CT *Conditional Test*

DA *Duplicate Assert*

EaT *Eager Test*

EmT *Empty Test*

ExT *Exception Test*

GV *Global Variable*

HE *Halstead Effort*

HT *Halstead Time*

HTML *Hypertext Markup Language*

HTTP *Hypertext Transfer Protocol*

HVoc *Halstead Vocabulary*

IT *Ignored Test*

JSON *JavaScript Object Notation*

LSP *Language Server Protocol*

LT *Lazy Test*

MA *Maintainability*

MG *Mystery Guest*

MN *Magic Number*

RA *Redudant Assertion*

RO *Redudant Optimism*

RP *Redudant Print*

ST *Sleepy Test*

SUT *System Under Test*

UT *Unknown Test*

Lista de Código-fontes

Capítulo 1

Introdução

JavaScript é uma linguagem de programação de *scripts* criada em 1995, que tem como características o fato de ser interpretada, dinâmica, possuir uma tipagem fraca, além de permitir implementações multi-paradigma [2]. As melhorias adotadas a partir do ano de 2015, fizeram com que ela se tornasse a linguagem mais popular em repositórios de códigos [3] [4] [5] e utilizada não apenas no desenvolvimento de páginas da web, mas também em aplicações do lado do servidor, em programas de *desktop* [1], em automação de robótica e *internet* das coisas [2], em aplicações de linha de comando para terminal [3], em aplicativos para dispositivos móveis [4] [5], dentre outros.

Embora tenha uma larga popularidade e adoção por desenvolvedores e também pela indústria, a linguagem *JavaScript* traz sérios desafios como efeitos colaterais [6] de suas especificidades citadas anteriormente: ser interpretada, ser dinâmica e permitir conceitos de programação mais avançados como *prototypes*, *closures* e funções *first-class* [7]. Tais características permitem, ao mesmo tempo, recursos e flexibilidades na programação, assim como também permitem brechas para que o desenvolvedor com pouca experiência na linguagem cometa *smells* durante a escrita de código.

Os *smells* são anti-padrões [8] que dão indícios de problemas na legibilidade do código, além de dificultar a manutenção durante o ciclo de vida de um programa.

[1] `https://www.electronjs.org/`
[2] `http://johnny-five.io/`
[3] `https://oclif.io/`
[4] `https://reactnative.dev/`
[5] `https://nativescript.org/`

Enquanto a qualidade do código de um sistema pode ser afetada por *smells*, é relevante notar que os testes de *software*, utilizados para a verificação e validação desse código, também podem ser prejudicados por essa mesma questão. Os anti-padrões existentes em testes de *software* são habitualmente chamados de *test smells* e têm sido objeto de interesse em estudos da Academia [9] [10].

Além dos anti-padrões que sinalizam adversidade na saúde do código, o interesse pela qualidade do código-fonte fez com que surgissem trabalhos acadêmicos para quantificar essa qualidade por meio de métricas de código [11] [12]. Um exemplo de métrica mais simples é a quantidade de linhas de código, que pode indicar diretamente a complexidade de um programa. Ainda em relação à quantidade de linhas, a medida de linhas lógicas de código é uma evolução da métrica anterior. Outros parâmetros foram sugeridos, tais como a Complexidade Ciclomática [13] e os índices de *Halstead* [14], que serão discutidos com mais detalhes no Capítulo 2.

Sendo a incidência de *smells* um aspecto não desejável nos casos de teste, compreender o seu impacto no desenvolvimento de *software* e detectar os *test smells* são atividades de extrema importância e necessárias para a avaliação e melhoria da qualidade dos testes. É aqui que o nosso presente estudo se encaixa no contexto relatado e que exemplificaremos na Seção 1.1.

1.1 Exemplo Motivante

A listagem 1.1 é um caso de teste extraído do *Sails*, um *framework* para desenvolvimento de aplicações *web*. Neste exemplo, faremos uma avaliação minuciosa para detectar problemas de *design* no código de teste. Inicialmente, podemos observar que as linhas 6 e 7 possuem asserções que avaliam os valores das propriedades *status* e *body* do objeto *err*. No entanto, nenhuma das asserções possuem mensagens de erro para quando o teste falhe. Esta ausência de mensagens de erro como argumento quando houver múltiplas asserções em um mesmo caso de teste caracteriza um tipo de *test smell* chamado *Assertion Roulette*.

Ainda no exemplo do caso de teste em questão, podemos averiguar que a linha 4 levanta uma exceção por meio da declaração *throw*. Recomenda-se evitar tal prática em

códigos de teste, visto que adiciona complexidade supérflua. Caso o objetivo do teste seja avaliar se o código de produção levantará ou não uma exceção, o desenvolvedor deverá utilizar os tratamentos específicos de biblioteca de testes adotada. Esse tipo de anti-padrão é chamado de *Exception Handling*.

Por fim, é possível perceber que a asserção da linha 7 faz uma comparação da propriedade *err.status* com o literal numérico 500. Um desenvolvedor iniciante, ou novato na equipe de desenvolvimento, terá dificuldades em reconhecer o propósito da asserção. Este tipo de anti-padrão é chamado *Magic Number* e tem como consequência prejudicar a compreensão e manutenção do caso de teste.

```
1  it('should return the expected error when something throws', function
       (done) {
2    var ERROR = 'oh no I forgot my keys';
3    sails.get('/errors/1', function (req, res) {
4      throw ERROR;
5    });
6    sails.request('GET /errors/1', {}, function (err) {
7      assert.deepEqual(500, err.status);
8      assert.deepEqual(ERROR, err.body);
9      done();
10   });
11 });
```

Listagem 1.1: Exemplo de presença de *Smell* no *framework Sails*

Ilustramos neste exemplo a avaliação do *design* do código, por meio da identificação de três tipos de *smells* presentes no caso de teste. Esta atividade de detecção terá sua complexidade aumentada se expandirmos o catálogo de tipos de *test smells* detectados. Temos que considerar também que o exemplo ilustra apenas um caso de teste. A versão do projeto ao qual este exemplo pertence possui 18 suítes de testes com 135 casos de testes no total. Realizar a detecção manualmente é um trabalho que demanda tempo e esforço, principalmente em uma linguagem com as características que o *JavaScript* possui. Além disso, até o momento não identificamos trabalhos acadêmicos que explorem os desdobramentos da ocorrência dos *test smells* no contexto da linguagem *JavaScript*.

1.2 Escopo e Objetivos

O escopo do nosso trabalho compreende uma investigação de *smells* em códigos de teste unitários no domínio da linguagem *JavaScript*, utilizando as asserções nativas do *Node.js*, os *frameworks Mocha*[6] e *Jest*[7], e as bibliotecas de asserções *Chai*[8], *Sinon*[9] e *Nock*[10]. Vale ressaltar que apesar estarmos observando apenas testes de unidade, é possível que nosso estudo inclua testes de integração. Isso porque a distinção entre estes tipos de teste é raramente seguida na prática e há indícios de que esta distinção não é significativa [15] [16]. Adicionalmente, esta investigação explora: *i)* as relações entre os *test smells* detectados e *ii)* a relação dos *test smells* com as métricas de qualidade extraída dos casos de testes.

Estando com o nosso escopo definido, o objetivo geral deste trabalho é investigar a ocorrência e desdobramentos de *test smells* na linguagem *JavaScript*. Com esse objetivo em vista, elaboramos as seguintes questões de pesquisa:

- **Questão de Pesquisa 1**: Quais são os *test smells* mais frequentemente detectados em projetos *JavaScript*? Ao responder a esta questão, é possível obter informações relevantes para os desenvolvedores e testadores, contribuindo para a criação de estratégias eficazes de refatoração e, consequentemente, para a melhoria da qualidade do software.

- **Questão de Pesquisa 2**: Quais as associações significativas entre os *test smells* detectados? Com esta questão, buscamos indícios se a presença de um tipo específico de *test smell* pode ser um indicativo da probabilidade de ocorrência de outros. Essa informação pode ser utilizada para otimizar esforços de detecção e refatoração.

- **Questão de Pesquisa 3**: Quais as associações significativas entre os *test smells* detectados e métricas de qualidade de *software*? Ao responder a esta questão, procuramos evidências de que a incidência de um tipo específico de *test smell*

[6] `https://mochajs.org`
[7] `https://jestjs.io`
[8] `https://www.chaijs.com/`
[9] `https://sinonjs.org`
[10] `https://github.com/nock/nock`

pode estar associada a índices ruins de métricas de qualidade de *software*, o que nos permite uma compreensão mais ampla do impacto desses *test smells* em um projeto de *software*.

A partir das questões de pesquisas apresentadas acima, delineamos os seguintes objetivos com o intuito de respondê-las:

1. Propor um catálogo de *test smells* para a linguagem *JavaScript*;
2. Desenvolver uma ferramenta para detecção de *smells* e extração de métricas de qualidade de código dos testes;
3. Investigar quais dos *test smells* propostos possuem maior incidência em uma seleção de projetos de *softwares open-source*;
4. Estudar as relações entre os *test smells* que ocorrerem conjuntamente;
5. Estudar as relações entre os *test smells* e as métricas de *software*;

1.3 Metodologia

A metodologia deste trabalho de doutorado foi delineada para fornecer uma abordagem abrangente para a identificação e avaliação de *Test Smells* em códigos *JavaScript*. A pesquisa segue um projeto metodológico dividido em quatro etapas principais, ilustradas na Figura 1.1 e descritas detalhadamente a seguir.

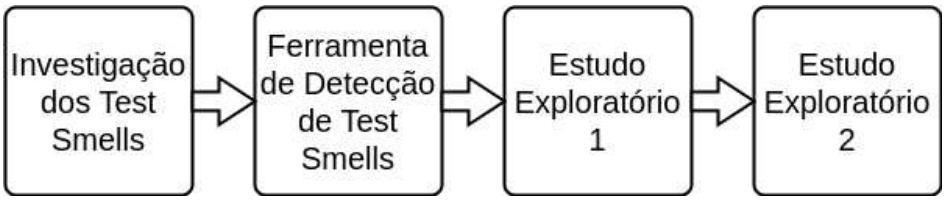


Figura 1.1: Metodologia adotada neste trabalho de doutorado

1. **Estudo dos *Test Smells* Clássicos e da Linguagem *JavaScript***: A primeira fase da pesquisa envolveu um estudo dos *Test Smells* clássicos, conforme definido na literatura existente [9, 10]. Paralelamente, realizou-se um estudo

da linguagem de programação *JavaScript*, focando em suas peculiaridades que podem afetar a qualidade dos testes. Este estágio foi crucial para estabelecer a fundamentação teórica que guiou o desenvolvimento da ferramenta de análise estática e os estudos exploratórios posteriores.

2. **Criação da Ferramenta de Análise Estática para Detecção de *Test Smells***: Nesta etapa, foi desenvolvida uma ferramenta de análise estática visando detectar *Test Smells* em códigos *JavaScript*. O desenvolvimento foi iterativo e baseado nos princípios e nas métricas identificados na fase anterior. A ferramenta foi validada por meio de testes unitários e inspeções de especialistas, garantindo sua eficácia e precisão.

3. **Estudo Exploratório com Projetos em Repositórios de Código Aberto e Fechado**: Posteriormente, realizou-se um estudo exploratório envolvendo uma amostra significativa de projetos *JavaScript*, tanto de repositórios de código aberto quanto fechado. O objetivo foi avaliar a ocorrência e correlações de *Test Smells* com métricas de qualidade de *software*, utilizando a ferramenta desenvolvida na fase anterior como instrumento de coleta de dados. As análises estatísticas foram conduzidas para identificar padrões e tendências.

4. **Estudo Exploratório com Desenvolvedores *JavaScript* para Avaliar a Percepção sobre a Ocorrência de *Test Smells***: Na etapa final, foi conduzido um estudo exploratório com desenvolvedores de *JavaScript* para entender suas percepções sobre a ocorrência e o impacto de *Test Smells* em seus códigos. Utilizou-se um questionário para coletar dados qualitativos que complementaram as descobertas quantitativas das etapas anteriores.

1.4 Relevância

Este trabalho visou contribuir com o campo de *Test Smells*, expandindo para o contexto da linguagem *JavaScript*, dado a sua crescente adoção no desenvolvimento de soluções para as mais diversas plataformas. Apesar dos diversos trabalhos existentes na Academia sobre *Test Smells*, a maioria contempla a linguagem *Java*, que possui

características e desafios bastante diferentes do *JavaScript*. Este trabalho também demonstra sua relevância no âmbito profissional da Indústria, uma vez que alerta para os efeitos que os *test smells* podem causar no processo de desenvolvimento de *software* na linguagem *JavaScript*.

1.5 Contribuições

Nosso objetivo foi alcançar as seguintes contribuições:

- A formalização inicial dos estudos de *test smells* na linguagem *JavaScript*, explorando as especificidades e desdobramentos;
- A criação de um catálogo de *test smells* para a linguagem *JavaScript*;
- A implementação de uma ferramenta para auxiliar na detecção dos *test smells*.

De modo geral, esperamos que ao responder as nossas questões de pesquisa, possamos proporcionar subsídios para a melhoria da qualidade das suítes de testes em um projeto de desenvolvimento de *software* na linguagem *JavaScript*.

1.6 Resultados

Esta pesquisa trouxe descobertas importantes sobre a incidência e as implicações de *Test Smells* em projetos que utilizam *JavaScript*. Os resultados apontaram para uma alta ocorrência de *Duplicate Assert* e *Conditional Test*, destacando a urgência de revisitar e otimizar tanto o *design* quanto a implementação das suítes de testes. Além disso, foi observada uma correlação significativa entre diferentes *Test Smells*, evidenciando a complexidade inerente às práticas de codificação e sugerindo que a resolução de um *Test Smell* pode inadvertidamente desencadear outros. Os resultados também indicaram que *Test Smells* exibem fortes correlações com métricas consagradas de qualidade de *software*, como *Maintainability* e *Cyclomatic Complexity*.

1.7 Organização do Documento

Os capítulos subsequentes deste documento estão estruturados da seguinte forma: o Capítulo 2 introduz os conceitos básicos necessários para a compreensão desta proposta de pesquisa. Já o Capítulo 3 apresenta um catálogo com os *test smells* do presente estudo. O Capítulo 4 expõe a ferramenta denominada Steel, desenvolvida e patenteada sob nossa autoria, para realizar a detecção de *test smells* na linguagem *JavaScript*. O Capítulo 5 traz o detalhamento do estudo exploratório com projetos de software e o Capítulo 6 introduz o estudo com desenvolvedores e testadores acerca da percepção sobre *test smells* em *JavaScript*. No Capítulo 7, são apresentados os trabalhos relacionados com o tema de *code smells* e *test smells*. Por fim, o Capítulo 8 apresenta a conclusão deste trabalho de doutorado.

Capítulo 2

Fundamentação Teórica

O presente capítulo visa apresentar o arcabouço teórico que fundamenta os principais conceitos aplicados neste trabalho. Sendo assim, serão abordados e discutidos os conceitos básicos relacionados a teste de *software* (Seção 2.1), a testes unitários (Seção 2.2), a *code smells* (Seção 2.3), a *test smells* (Seção 2.4) e a métricas de *software* (Seção 2.6).

2.1 Teste de *Software*

A cada dia que passa, surgem novos dispositivos e serviços – ou mesmo evoluções dos já existentes – que nos oferecem melhorias e facilidades nas mais diversas áreas do nosso cotidiano: comunicação, transportes, saúde, entretenimento, finanças, dentre outros. Sendo assim, é idealizado que os componentes de *software* que controlam esses dispositivos e/ou serviços operem do modo como se é esperado e sem surpresas desagradáveis para seus usuários.

É imprescindível que o *software* esteja conforme os requisitos aos quais foram implementados, visto que a falta de conformidade nas premissas de um sistema poderá implicar, dependendo do caso, em custos altíssimos para seus desenvolvedores [17]. Mesmo as boas práticas de programação não são suficientes para garantir a qualidade esperada e, dessa forma, o teste de *software* é o principal método que a indústria utiliza para avaliar o sistema em desenvolvimento antes de colocá-lo em produção [18].

McGregor and Sykes definem [19] teste de *software* como uma atividade que objetiva

identificar indícios de defeitos que tenham sido inseridos em alguma etapa durante o processo de desenvolvimento ou de manutenção de um *software*. Esses defeitos são consequência de erros, enganos, omissões ou interpretações equivocadas dos requisitos por parte dos desenvolvedores. No que diz respeito à terminologia empregada em teste de *software*, existe uma certa inconsistência na literatura, gerada pela confusão no emprego de alguns termos, dentre os quais podemos citar:

- **Erro**: é a ação cometida por engano pelo ser humano, em virtude de fatores como: prazos apertados, código complexo, complexidade na infraestrutura, mudanças de tecnologia e/ou múltiplas interações no sistema [20].
- **Defeito**: é o resultado do erro no código do sistema ou documento. Um defeito pode ou não dar origem a uma falha [20].
- **Falha**: é a manifestação executável do defeito, que impede o sistema de executar o que deveria ser feito, ou o contrário, fazer uma ação que não estava prevista [20].
- **Caso de teste**: é o conjunto de entradas (pré-condições e dados de entrada) e saídas esperadas (pós-condições e dados de saída) para um determinado comportamento de um programa [21].
- **Validação**: avalia se o *software* entregue está em concordância com o uso esperado e depende do conhecimento do domínio [18].
- **Verificação**: determina se os artefatos de *software* resultantes de uma fase do desenvolvimento atendem aos requisitos firmados para esta fase [18].

Em síntese, teste de *software* é a atividade de detectar falhas através da execução de um sistema. É relevante frisar que existem outras técnicas, diferentes e complementares, para melhoria da qualidade do *software*, tais como: análise estática, inspeções, revisões, depuração (*debugging*) e correção de erros. Os dois últimos processos são realizados após a detecção das falhas [1].

Quanto à classificação do teste de software, Utting and Legeard [1] demonstram os diversos tipos distribuídos em um sistema tridimensional de eixos ortogonais entre

si (adaptado de Tretmans [22]) conforme é ilustrado na Figura 2.1. O eixo vertical z apresenta a escala do sistema sob desenvolvimento, e partindo do ponto mais próximo da origem, encontramos *(i)* o teste de unidade (ou teste unitário), *(ii)* o teste de componente, *(iii)* teste de integração e, por fim, *(iv)* o teste de sistema.

Já o eixo x compreende as características que se deseja testar e engloba *(i)* o teste funcional, *(ii)* teste de robustez, *(iii)* teste de performance e *(iv)* teste de usabilidade. No eixo y encontramos o tipo de informação necessária para criar o teste, sendo eles: *(i)* o teste de caixa-preta, que significa que não se sabe detalhes da implementação – apenas com informações dos requisitos – e *(ii)* o teste de caixa-branca, que utiliza o código de implementação para guiar os testes.

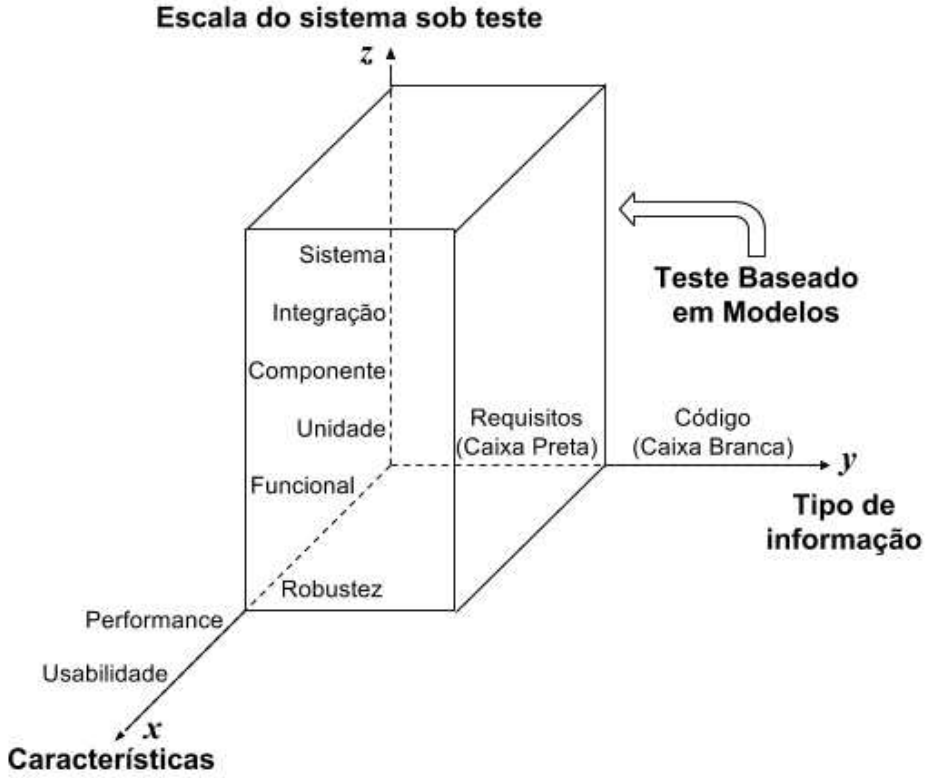


Figura 2.1: Classificação de Teste de Software (fonte: Utting and Legeard)

O tipo de teste mais comum é o teste funcional, também conhecido por teste comportamental. O objetivo do teste funcional é verificar se o programa está conforme as suas especificações, sem se deter aos detalhes de implementação. Por consequência, os testes funcionais podem ser adotados bem antes da disponibilização de código executável, e caso as especificações sejam escritas em linguagem formal, os testes funcionais

podem ter algum nível de automatização, economizando esforços e tempo, além de evitar erros inseridos em um processo de execução manual.

As práticas de desenvolvimento tradicionais, tais como Cascata ou Prototipação, adotam o processo de teste de *software* como uma das etapas finais em um projeto de desenvolvimento. No entanto, com a experiência construída através dos anos, tem se verificado uma maior eficiência quando os testes são aplicados ao longo de todo o processo de desenvolvimento do *software* [19]. Dessa forma, cada etapa do desenvolvimento possui um nível de abstração que deriva um respectivo tipo de teste. Este mapeamento está representado na Figura 2.1 por meio do diagrama *V-Model* ou modelo de verificação e validação.

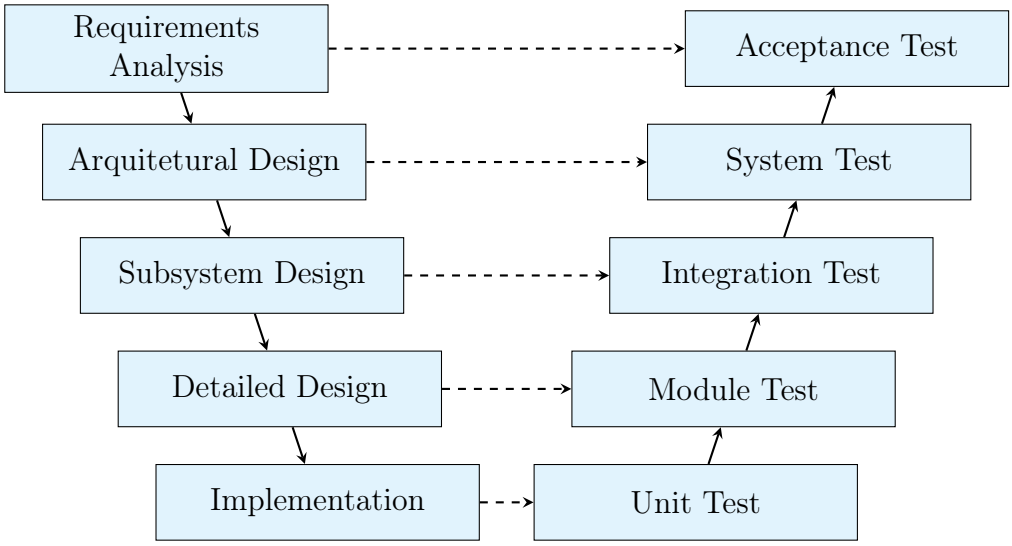


Figura 2.2: Diagrama *V-Model*

2.2 Testes Unitários

O teste unitário tem como objetivo testar a funcionalidade do menor bloco de declarações de código que pode ser chamado de outra parte do sistema. Este bloco de código funcional pode ser um objeto, uma função, um método ou um procedimento, conforme a linguagem empregada [18]. É importante salientar que os testes podem ser escritos após a codificação do bloco de código de produção, ou antes, consoante a prática de desenvolvimento orientado a testes.

Para escrever testes unitários é necessário compreender claramente a implementação do código a ser testado, fato este que caracteriza-os como testes do tipo caixa-branca. Sendo assim, as partes interessadas pela validação e verificação dos testes de unidade são, na maioria das vezes, os desenvolvedores. Em contrapartida, os testes de aceitação, os quais são do tipo caixa-preta, dispensam o conhecimento da implementação e são, preferencialmente, de interesse do cliente ou de algum especialista nas regras de negócio [23].

A evolução sustentável de um projeto de software depende fortemente da implementação contínua de testes unitários eficazes ao longo de todo o seu ciclo de vida [24]. Além de garantir a consistência no desenvolvimento do software, devido a características como a execução rápida e o isolamento de dependências, os testes unitários, quando bem elaborados, são uma proteção contra possíveis reduções na qualidade do projeto. No entanto, como destacado por Moonen et al., existe uma relação dualística intricada entre a evolução do software e o teste do software. Apesar de cruciais para garantir a evolução segura e contínua do sistema, a necessidade de coevolução constante dos testes em resposta a mudanças no código pode se tornar, paradoxalmente, um obstáculo para a própria evolução do software. Essa interdependência complexa entre teste e evolução enfatiza a necessidade de uma abordagem estratégica e bem informada no design e manutenção de suítes de testes durante o ciclo de vida do software.

Outra observação no estudo de Moonen et al. é a de que testes de software e compreensão do programa possuem uma relação simbiótica durante a evolução de software. Os testes provêm documentação executável do sistema, aumentando o entendimento da equipe. Por sua vez, o ato de escrever novos testes expande a compreensão sobre o software. Assim, a disponibilidade de uma ampla suíte de testes reforça e é apoiada pelo conhecimento do sistema, gerando um ciclo benéfico que favorece a evolução do software.

Dentre as vantagens proporcionadas pelos testes de unidade destacam-se as seguintes [26]:

- O teste unitário é empregado na etapa de implementação, possibilitando a identificação rápida e antecipada de problemas que muitas vezes não são identificados por testes de níveis mais altos.

- A execução rápida dos testes unitários aliada ao fato de dispensar o processo de construção e execução de um sistema torna mais rápidos e ágeis o diagnóstico e o conserto de falhas detectadas.

- Por focar em apenas blocos funcionais de código, e de preferência que estejam isolados de recursos externos, a execução dos testes é bem mais rápida do que a dos testes de níveis de abstração mais altos.

- O teste de unidade possibilita que o desenvolvedor entenda melhor a construção do seu código de produção e permite que ele crie testes específicos para seu estilo de programação.

- A execução do teste unitário informa se os requisitos para os quais foram programados ainda são válidos ou não, assumindo dessa forma um papel de documentação viva.

- A escrita do teste de unidade força o desenvolvedor a repensar seu código em outra perspectiva, o que pode favorecer uma melhoria no *design* do teste. Além disso, o fato de escrever o teste e código de produção dobra as chances de obter uma codificação sem problemas.

- Uma suíte de testes unitários bem construída permite ao desenvolvedor identificar a localização de algum possível código problemático em qualquer parte do sistema, o que não seria possível utilizando outros tipos de testes.

Diante do que foi exposto sobre os testes unitários, observa-se que sua finalidade é garantir que o bloco de código que está sob teste atenda completamente as especificações de seus requisitos. No entanto, os testes unitários também são códigos que estão suscetíveis a terem problemas que comprometerão a sua qualidade. Estes problemas serão explorados nas seções seguintes.

2.3 *Code Smells*

No processo de desenvolvimento de *software*, idealiza-se que a codificação dos requisitos só se inicie após a concepção de um *design* de aplicação impecável. A lógica subjacente a

essa abordagem visa evitar o desperdício de tempo e recursos financeiros na reescrita de código problemático que poderia comprometer a qualidade do produto final. Contudo, essa situação ideal raramente é alcançável devido a diversos desafios inerentes a projetos de *software*, como prazos apertados, limitações orçamentárias e barreiras técnicas.

A premissa de que um design bem concebido precede a codificação não elimina o risco de que a evolução contínua do sistema deprecie o código-fonte ao longo do tempo. Consequências diretas de um código de baixa qualidade incluem ilegibilidade [27], complexidade na manutenção e evolução, limitações na escalabilidade, problemas de desempenho e maior susceptibilidade a erros [28].

Frente aos desafios impostos por um *design* de código deficiente, a refatoração emerge como uma técnica essencial. Esta abordagem envolve a reestruturação do código-fonte para aprimorar sua estrutura interna sem alterar seu comportamento externo, favorecendo um desenvolvimento mais ágil e adaptável [29].

Embora a refatoração melhore significativamente o *design* do código, surge a questão de identificar o momento oportuno para iniciar esse processo. Nesse contexto, o termo *Code Smell*, cunhado por *Kent Beck*, desempenha um papel crucial ao indicar padrões de codificação que sugerem problemas de design. A identificação desses padrões, organizada em catálogos detalhados, orienta os desenvolvedores sobre quando e como refatorar o código [29] [30].

O interesse em identificar designs de código inadequados e explorar técnicas de aprimoramento não é recente. O estudo pioneiro de Opdyke and Johnson [31], publicado em 1990, inaugurou a pesquisa acadêmica sobre refatoração de código. Desde então, a literatura especializada expandiu consideravelmente, com destaque para o trabalho de Fowler, que compila um extenso catálogo de code smells e técnicas de refatoração em seu livro *Refactoring: Improving the Design of Existing Code* [29].

2.4 *Test Smells*

Test smells refere-se aos padrões específicos presentes nos testes automatizados, os quais, apesar de serem tecnicamente corretos, podem indicar possíveis deficiências ou áreas que necessitam de aperfeiçoamento. A nomenclatura é semelhante à de *code*

smells, como demonstrado no contexto do código de produção por Fowler [29]. Da mesma forma que os *code smells* são características do código que podem indicar possíveis inconsistências no *design* ou na implementação de um sistema, sem necessariamente indicar falhas diretas, os *test smells* são padrões que podem revelar problemas [10], tais como fragilidade, acoplamento excessivo e complexidade exagerada. Tais problemas não implicam a presença de falhas que prejudiquem sua execução, mas dificultam a legibilidade e a manutenção do teste, além de comprometerem a confiabilidade dos resultados [32].

Os autores van Deursen et al. [9] foram os primeiros a empregar o termo *test smell*, no qual delinearam um catálogo de 11 *test smells* com as respectivas descrições e operações de refatoração para remoção. Este catálogo inclui padrões como, por exemplo, duplicação de código de testes, asserções sem mensagens explicativas e o uso de recursos externos. Em seguida, estudos posteriores ampliaram e aperfeiçoaram esse compêndio [33, 23]. A revisão sistemática de Garousi and Küçük [10] cataloga um conjunto abrangente de 139 *test smells* clássicos oriundos de publicações científicas e também de fontes técnicas, não estritamente acadêmicas. O objeto de estudo do presente trabalho de doutorado se deteve, de forma mais específica, aos *test smells* adotados nos trabalhos [34] e [35], cujo enfoque foi o desenvolvimento na plataforma *android*, e aqui foi adaptado para o desenvolvimento na linguagem *JavaScript*.

A importância de identificar e tratar *test smells* no desenvolvimento de software é evidente, uma vez que os testes têm um papel fundamental no ciclo de vida e na garantia da qualidade do software. Sendo o teste uma atividade que visa identificar sinais de defeitos que possam ter sido introduzidos em algum momento do desenvolvimento ou manutenção de *software* [19], a presença de *test smells* no código de teste pode prejudicar a legibilidade e, consequentemente, a manutenibilidade não apenas das suítes de testes, mas também do código de produção.

A tabela 2.1 apresenta o corpo de *test smells* utilizado neste estudo. No Capítulo 3, são apresentadas as definições, consequências, regras de detecção e exemplos para cada *test smell*.

Tabela 2.1: *Test smells* adotados neste estudo

Smell	Característica
Assertion Roulette	Mais de uma asserção no caso de teste sem conter a mensagem de falha.
Conditional Test	Estruturas de controle ou repetição.
Duplicate Assert	Múltiplas chamadas de asserções iguais no mesmo caso de teste.
Eager Test	Múltiplas chamadas para métodos de produção no mesmo caso de teste.
Empty Test	Caso de teste vazio (sem código).
Exception Test	Estrutura para tratar ou levantar uma exceção.
Ignored Test	Algum caso de teste ignorado.
Global Variable	Uso de varáveis globais na suíte de teste.
Lazy Test	Múltiplas chamadas para um mesmo método de produção.
Magic Number	Asserções contendo literais numéricos como argumentos.
Mystery Guest	Instâncias de recursos externos como um dado de teste.
Redudant Assertion	Asserções onde o parâmetro atual e o esperado são iguais.
Redudant Print	Chamadas para métodos do tipo *console.log*.
Redudant Optimism	Instâncias de recursos externos para inferir estado.
Sleepy Test	Casos de testes com chamadas para *setTimeout()*.
Unknown Test	Casos de testes com código, mas sem asserções.

2.5 Ferramentas para Detecção de Test Smells

2.5.1 TSDetector

O trabalho de Peruma et al. apresenta o *tsDetect*, uma ferramenta de detecção de *test smells* em suítes de teste baseadas em *JUnit*. Os *test smells* representam desvios das práticas estabelecidas de teste e a detecção desses desvios é geralmente uma tarefa manual e propensa a erros. O *tsDetect* automatiza esse processo, identificando 19 tipos de *test smells* em arquivos de teste unitário *JUnit*. A ferramenta alcançou um alto desempenho, com uma média de 96,5% de *F-score* na detecção dos diferentes tipos de *test smells*. O projeto *tsDetect* é de código aberto e está disponível para a comunidade, com documentação detalhada, exemplos e um vídeo de demonstração. O trabalho destaca a importância da qualidade das suítes de teste e encoraja a contribuição da comunidade para a melhoria e expansão contínua do *tsDetect*.

2.5.2 JNose Test

O estudo realizado por Virgínio et al. concentra-se na identificação de *test smells* em códigos de teste, avaliando especificamente a precisão da ferramenta *JNose Test* em detectar esses problemas de *design* em projetos *Java*. O trabalho ressalta a importância de estratégias para avaliar a qualidade do código de teste, incluindo a análise de *test*

smells, e destaca a necessidade de suporte automatizado para identificar eficientemente esses problemas em uma escala ampla.

Para validar as regras de detecção implementadas no *JNose-Core*, os pesquisadores conduziram um estudo empírico comparativo, avaliando a precisão da ferramenta *JNose* em relação a outras ferramentas e à análise manual. Eles estabeleceram um conjunto de dados de referência contendo sessenta e cinco classes de teste analisadas por especialistas no assunto para realizar essa comparação. O estudo se concentrou na comparação entre o *JNose* e a ferramenta *tsDetect* em um nível de classe.

2.5.3 PyNose

No artigo *"PyNose: A Test Smell Detector For Python"*, [38] introduzem uma abordagem pioneira para a detecção de *test smells* em código de teste *Python*. Dada a predominância de estudos anteriores focados em linguagens estaticamente tipadas como *Java* e *Scala*, a pesquisa em questão aborda uma lacuna significativa no contexto de *Python*, uma linguagem dinamicamente tipada que ganhou imensa popularidade, especialmente nos domínios de ciência de dados e aprendizado de máquina. A equipe de pesquisa identificou inicialmente 17 *test smells* considerados agnósticos à linguagem ou com equivalências funcionais no *framework Unittest* do *Python*. Além disso, foi introduzido um *test smell* específico para *Python*, denominado *Suboptimal Assert*, identificado por meio da análise de padrões de mudança de código frequentemente associados à introdução ou correção de *test smells*. Para facilitar a detecção desses *smells*, os autores desenvolveram a ferramenta *PyNose*, implementada como um *plugin* para o *PyCharm*, um ambiente de desenvolvimento integrado amplamente utilizado por desenvolvedores *Python*.

Um estudo empírico abrangente, realizado com o auxílio do *PyNose* em 248 projetos *Python*, revelou a presença de *test smells* em 98% dos projetos analisados, com 84% dos conjuntos de testes examinados contendo pelo menos um *test smell*. Notavelmente, o *test smell* "Suboptimal Assert", proposto pelos autores, foi detectado em 70,6% dos projetos, evidenciando sua prevalência no ecossistema *Python*. Este estudo não apenas destaca a necessidade crítica de ferramentas automatizadas para a identificação e mitigação de *test smells* em projetos *Python*, mas também contribui significativamente

para a literatura existente em engenharia de *software*, fornecendo percepções valiosas sobre a manutenção de código de teste e a qualidade geral do *software* desenvolvido nesta linguagem em ascensão.

2.5.4 Pytest-Smells

No contexto da programação em *Python*, existe uma lacuna significativa em ferramentas que automatizam a detecção dos *test smells*, especialmente para os testes escritos com o *framework Pytest*. Bodea nos apresenta o *Pytest-Smell*, uma solução inovadora para esse problema, sendo uma biblioteca *Python* de fácil instalação e uso, destinada à detecção automatizada de *test smells* em códigos que utilizam *Pytest*. Este trabalho destaca a importância dos testes unitários no desenvolvimento ágil de *software* e os desafios associados à manutenção da qualidade desses testes.

A pesquisa realizada valida a ferramenta *Pytest-Smell* através de um estudo experimental em três bibliotecas de código livre *Python* amplamente utilizadas, demonstrando que aproximadamente 90% dos conjuntos de testes analisados continham *test smells*, com os *smells* mais comuns sendo *Assertion Roulette*, *Magic Number Test* e *Conditional Test Logic*. Esses resultados são consistentes com estudos anteriores em outras linguagens de programação, ressaltando a prevalência e a necessidade de abordar *test smells* no desenvolvimento de *software*. O artigo também estabelece um caminho para trabalhos futuros, incluindo a expansão da detecção para mais *test smells* e a exploração de correções automáticas para os *smells* detectados, sublinhando o potencial impacto positivo do *Pytest-Smell* na comunidade de desenvolvimento de software *Python*.

2.6 Métricas de *Software*

À medida que os sistemas ganham mais espaço e importância em todos os âmbitos de nossas vidas, é natural que se deseje fazer medições nos softwares para avaliar e caracterizar os mais diversos aspectos de suas naturezas [40]. O conceito de métrica na engenharia de *software* é aplicado em diversos contextos e isto ocasiona uma falta de consenso na validação de cada uma das propostas [41]. A imaturidade na definição de

métrica faz com que ora a métrica seja usada para indicar um valor medido, ora um procedimento de medição, noutro momento os resultados de medições ou modelos de relacionamentos entre múltiplas medidas, ou até mesmo a medição dos próprios objetos. Recentes estudos nos mostram haver uma preocupação com a elaboração e qualidade das métricas, empregando alguns conceitos relacionados à metrologia na engenharia de *software*.

Nas subseções seguintes, explicitamos algumas métricas clássicas para avaliarmos e tirarmos conclusões acerca dos nossos estudos.

2.6.1 Complexidade Ciclomática

A complexidade ciclomática, do inglês *Cyclomatic Complexity* (CC), é uma métrica que representa o grau de dificuldade na estrutura de um algoritmo, e quanto maior for o seu índice, mais difícil será a legibilidade, a manutenibilidade, a criação de testes e consequentemente a cobertura de código [13]. Para calcular este índice, leva-se em consideração os caminhos linearmente independentes que existem em um programa quando representado em um grafo de fluxo de controle. Estes caminhos independentes são computados a partir da quantidade de estruturas lógicas de decisões, como, por exemplo, estruturas do tipo *if-else* e laços *for* ou *while*. Sendo assim, considerando F o fluxo de um programa qualquer, então o valor da complexidade ciclomática é calculado pela expressão:

$$v(F) = E - N + 2P$$

onde

- E é o total de arestas contidos em F
- N é o total de vértices contidos em F
- P é o total de componentes conectados em F (geralmente 1 para a maioria dos programas)

Para ilustrar o cálculo da complexidade ciclomática, considere a Listagem 2.1 no qual possui um comando de repetição e um comando de decisão.

```
1  function product(a, b) {
2    let c = 0;
3    while (b !== 0) {
4      if ((b % 2) == 0) {
5        b = parseInt(b / 2);
6        a = 2 * a;
7      } else {
8        b = b - 1;
9        c = c + a;
10     }
11   }
12   return c;
13 }
```

Listagem 2.1: Exemplo de código para cálculo da Complexidade Ciclomática

A Figura 2.6.1 demonstra o grafo com os caminhos possíveis para o código-exemplo. As etiquetas contidas nos vértices se referem ao índice das linhas no código.

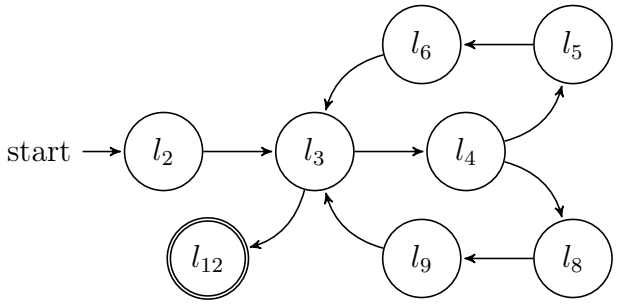


Figura 2.3: Grafo do código-exemplo para cálculo da complexidade ciclomática

A partir do grafo acima, temos os seguintes pontos de decisão:

- A condição *while (b !== 0)*.
- A condição *if ((b % 2) == 0)*.

Calculando E e N:

1. Começamos com um nó de entrada (linha 1).
2. Temos um nó para o *while* (linha 3).
3. Dentro do *while*, temos um nó para o *if* (linha 4).

4. Se o *if* for verdadeiro, temos um nó que representa as linhas 5 e 6.

5. Se o *if* for falso, temos um nó que representa as linhas 8 e 9.

6. Temos um nó de saída após a linha 9.

Isso nos dá um total de $N = 6$ nós.

Para as arestas:

1. Uma do nó de entrada para o *while*.

2. Uma do *while* para o *if*.

3. Uma do *if* para o bloco verdadeiro (linhas 5 e 6).

4. Uma do *if* para o bloco falso (linhas 8 e 9).

5. Uma do bloco verdadeiro (linhas 5 e 6) de volta para o *while*.

6. Uma do bloco falso (linhas 8 e 9) de volta para o *while*.

7. Uma do *while* para o nó de saída (quando o *while* for falso).

Isso nos dá um total de $E = 7$ arestas.

Agora, usando a fórmula $M = E - N + 2P$, onde $P = 1$ (já que temos um único componente conectado):

$$M = 7 - 6 + 2(1)$$

Dessa forma, $M = 3$.

2.6.2 Densidade Ciclomática

Diferentemente de métricas tradicionais que quantificam a complexidade geral de um programa, a Densidade Ciclomática realiza uma análise da complexidade ciclomática por linha de código-fonte. Essa abordagem permite executar comparações relativizadas da complexidade entre módulos e programas distintos, mesmo aqueles com grandes variações de tamanho. Dessa forma, a Densidade Ciclomática provê uma métrica de

complexidade modular invariante ao tamanho, sendo particularmente útil para avaliações contrastantes do nível de complexidade entre diversos componentes de um sistema de software. Tal utilidade advém de sua natureza normalizada, que abstrai o efeito da extensão do código na complexidade mensurada. Para isso, a densidade ciclomática recorre à métrica de linha de código *LOC*. A fórmula é a seguinte:

$$DC = \frac{CC}{LOC}$$

onde:

- CC é a Complexidade Ciclomática.
- LOC é o número de linhas de código.

2.6.3 Métricas de *Halstead*

As métricas de *Halstead*, comumente chamadas de ciência de *software*, empregam como parâmetros de suas funções os conjuntos de *tokens*, classificados como operandos e operadores [41]. Mais especificamente, as métricas de *Halstead* são um refinamento de métricas que tomam o software em relação ao seu número de linhas ou procedimentos [40]. Diversos trabalhos revisaram estas métricas de complexidade, trazendo evidências em diversos contextos, linguagens de programação e correlações com outras métricas de *software* [42, 43, 44].

Antes de detalhar as métricas *Halstead*, é necessário fazer a definição das medições de base alicerçados na contagem dos *tokens* operandos e operadores:

- $n1$: Número de operadores distintos.
- $n2$: Número de operandos distintos.
- $N1$: Número total de ocorrências dos operadores.
- $N2$: Número total de ocorrências dos operandos.
- $n1*$: Número de potenciais operadores.
- $n2*$: Número de potenciais operandos.

Com estas medições-base extraídas dos *tokens*, *Halstead* define as seguintes métricas:

1. **Tamanho do Programa - *Length* (N)**

   Definição: Representa o total de operadores e operandos em um programa:

   $$N = N_1 + N_2 \tag{2.1}$$

   Impacto:

   - Um tamanho de programa maior pode indicar um código mais extenso e, potencialmente, mais complexo;
   - Pode ser usado para comparar o tamanho relativo de diferentes programas ou módulos.

   Exemplo: Se um programa tem 50 operadores e 100 operandos, seu tamanho será 150.

   Interpretação Prática:

   - Um valor alto para N pode indicar um programa extenso;
   - Não é necessariamente um indicador de complexidade, mas de extensão.

2. **Tamanho do Vocabulário - *Vocabulary* (n)**

   Definição: Representa o número total de operadores únicos e de operandos únicos em um programa:

   $$n = n_1 + n_2 \tag{2.2}$$

   Impacto:

   - Um vocabulário maior pode indicar maior diversidade nas operações e dados manipulados.
   - Pode refletir uma maior complexidade em termos de variedade de operações.

Exemplo: Se um programa usa 20 operadores distintos e 30 operandos distintos, seu tamanho de vocabulário é 50.

Interpretação Prática:

- Um valor elevado para n sugere uma diversidade de operações e operandos, o que pode complicar a manutenção ou compreensão.

3. **Volume - *Volume* (V)**:

   Definição: Representa o tamanho em bits do espaço necessário para armazenar o programa:

$$V = N * log_2(n) \tag{2.3}$$

   Impacto:

   - O volume é uma métrica da densidade de informação no código.
   - Um volume alto pode ser um indicador de complexidade.

   Exemplo: Dado um tamanho de programa de 150 e um tamanho de vocabulário de 50, o volume é $150 \times \log_2(50)$.

   Interpretação Prática:

   - Um valor elevado de V sugere que o código pode ser denso em informações, tornando-o potencialmente mais desafiador de compreender e modificar.

4. **Dificuldade - *Program Difficulty* (D)**

   Definição: Indica o quão difícil é entender ou modificar o código:

$$D = \frac{n_1}{2} \times \frac{N_2}{n_2} \tag{2.4}$$

   Impacto:

   - A dificuldade é um indicador direto da complexidade cognitiva do programa.

- Um valor alto indica um programa potencialmente difícil de entender ou modificar.

Exemplo: Se um programa tem 20 operadores distintos e 100 operandos totais, com 30 operandos distintos, sua dificuldade é $\frac{20}{2} \times \frac{100}{30}$.

Interpretação Prática:

- Um valor elevado de D sugere que o código pode ser intrinsecamente complexo, exigindo mais esforço para compreensão e modificação.

5. **Esforço - *Effort* (E)**

   Definição: Estima o esforço cognitivo necessário para desenvolver ou compreender o programa:

$$E = D \times V \tag{2.5}$$

   Impacto:

   - O esforço combina a dificuldade e o volume para fornecer uma estimativa do esforço total necessário.
   - É uma métrica abrangente que pode ser usada para estimar o tempo ou recursos necessários para compreender, modificar ou desenvolver o código.

   Exemplo: Dado um volume de 500 e uma dificuldade de 3, o esforço é 1500.

   Interpretação Prática:

   - Um valor elevado de E indica que o programa pode exigir um investimento significativo de tempo e recursos.
   - Pode ser usado para estimar a carga de trabalho ou alocar recursos em projetos de software.

6. **Tempo de Entendimento - *Time* (T)**

   Definição: Estima o tempo necessário para um programador compreender o programa:

$$T = \frac{E}{S} \tag{2.6}$$

Onde S é uma constante que representa a taxa de processamento cognitivo do desenvolvedor, medido em segundos, e seu valor é 18.

Impacto:

- Esta métrica fornece uma estimativa do tempo (em segundos, se E for medido em operações elementares por segundo) necessário para compreender o programa.
- Pode ser usado para planejar alocações de tempo para revisões de código ou para entender um novo código.

Exemplo: Se o esforço (E) para um dado programa é de 3600, o tempo de entendimento estimado seria:

$$T = \frac{3600}{18} = 200 \text{ segundos}$$

Interpretação Prática:

- Um valor elevado de T sugere que o programa pode ser mais difícil de entender e que os programadores podem precisar de mais tempo para se familiarizar com ele.
- Esta métrica pode ser particularmente útil para equipes que estão avaliando a complexidade de um código antes de iniciar tarefas de manutenção ou revisão.

7. **Erros - *Bugs* (B)**

   Definição Estima o número de erros na aplicação:

$$B = V/3000 \tag{2.7}$$

Onde 3000 é uma constante que representa um fator de normalização para converter o volume em uma estimativa razoável do número de erros.

Impacto:

- Esta métrica fornece uma estimativa grosseira do número de bugs que um programa pode ter com base na sua complexidade.
- É importante notar que esta é uma estimativa empírica e pode não refletir o número real de bugs.

Exemplo: Se o volume (V) para um dado programa é de 6000, a estimativa de bugs seria:

$$B = \frac{6000}{3000} = 2$$

Isso sugere que o programa pode ter aproximadamente 2 bugs.

Interpretação Prática:

- Um valor elevado de B sugere que um programa pode ser propenso a ter mais defeitos devido à sua complexidade.
- Embora essa métrica forneça uma estimativa, é essencial validar com testes reais e análises de código para determinar o número real de bugs.
- Esta métrica pode ser útil para priorizar revisões de código e testes em programas com uma estimativa de bugs mais alta.

2.6.4 Manutenibilidade

A finalidade deste índice é fornecer uma visão geral da carga relativa de manutenção para diferentes seções do projeto, através da combinação de uma série de métricas diferentes [45]. As métricas envolvidas no índice de Manutenibilidade são:

- Volume de Halstead - HV
- Complexidade Ciclomática - CC
- Linhas de Código - LOC
- % de Comentários - PCOM

A fórmula original do índice de Manutenibilidade é:

$$MA = 171 - 5.2 * ln(HV) - 0.23 * CC - 16.2 * ln(LOC) + 50 * \sqrt{2.46 * pCOM}$$

O Índice de Manutenibilidade foi introduzido pela *Microsoft* em seu ambiente de desenvolvimento *Visual Studio*[1] em 2011. A fórmula matemática foi redefinida pelos engenheiros da companhia com o intuito de adequá-lo a uma escala que varia de 1 a 100. Com isso, a fórmula ficou da seguinte maneira:

$$MA = MAX(0, (171 - 5.2 * ln(HV) - 0.23 * (CC) - 16.2 * ln(LOC)) * 100/171)$$

Os limiares de alerta para o índice de manutenibilidade são:

- 0-9 = Vermelho
- 10-19 = Amarelo
- 20-100 = Verde

A fim de evitar ruídos nos alertas, a *Microsoft* adota como índice de manutenibilidade aceitável valor superior a 20.

2.7 Considerações Finais

Neste capítulo foi apresentado o arcabouço teórico-conceitual que alicerça nosso trabalho de doutorado. No próximo capítulo apresentaremos o catálogo de *smells* selecionados para nossa pesquisa.

[1] `https://learn.microsoft.com/en-us/visualstudio/code-quality/code-metrics-maintainability-index-range-and-meaning`

Capítulo 3

Catálogo de Test Smells

O propósito deste capítulo é expor o conjunto de *test smells* utilizado na presente pesquisa. Para a escolha dos *test smells*, consideramos diversos fatores para assegurar a pertinência e viabilidade do estudo. Primeiramente, a seleção foi embasada na relevância de estudos anteriores, garantindo que os *test smells* escolhidos refletissem preocupações e padrões previamente identificados na literatura científica, estabelecendo assim um fundamento sólido para a investigação. Além disso, levou-se em conta o contexto específico de desenvolvimento em *JavaScript*, uma linguagem de programação dinâmica e amplamente utilizada, cujas peculiaridades podem influenciar a manifestação e detecção de *test smells* de maneiras distintas em comparação a outros ambientes de desenvolvimento. Por fim, as limitações de tempo e escopo do projeto também foram critérios decisivos, orientando a seleção para um conjunto de *test smells* que pudesse ser investigado de forma profunda e significativa nas restrições do estudo.

A escolha pela adoção da análise estática como técnica de detecção de *smells* em nossa ferramenta, se deve ao fato de dispensarmos a necessidade de execução do código-fonte do *System Under Test* (SUT), e com isso evitarmos gastos de tempo e problemas com versionamento de dependências com pacotes de bibliotecas. Em vista disso, utilizamos o *parser* de código *JavaScript* para obtermos a árvore sintática do SUT e por conseguinte empregamos a abordagem baseada em regras, a exemplo dos trabalhos [46], [47], [48]. Dessa forma, pudemos abstrair as diferenças de sintaxe entre as bibliotecas e *frameworks* de testes que a ferramenta *teST smEll dEtection tooL* (Steel) compreende, e obter resultados mais promissores em comparação com uma abordagem

de busca de termos por palavras-chave.

A construção das regras para a detecção dos *test smells* engloba a utilização de funções para identificação dos vários tipos de nós da *Abstract Syntax Tree* (AST), como, por exemplo: o corpo de um caso de teste, uma declaração de asserção ou uma estrutura de repetição. Dessa forma, construímos o catálogo a seguir contendo a definição, a consequência, a regra de detecção e um código de exemplo para cada um dos *test smells* abordados.

3.1 *Assertion Roulette* (AR)

3.1.1 Definição

Este tipo de *smell* ocorre quando um caso de teste possui mais de uma asserção, e se, em pelo menos uma destas asserções não possuir argumento para uma mensagem de erro. Vale ressaltar que se existir apenas uma asserção, a ausência da mensagem de erro será compensada pelo nome do caso de teste em questão.

3.1.2 Consequência

O relatório de saída dos testes não informará uma mensagem explicativa se a asserção sem mensagem falhar, o que dificultará a identificação imediata do erro pelo desenvolvedor.

3.1.3 Detecção

A detecção deste *smell* é realizada por meio de funções que avaliam o tipo do nó durante a varredura da AST, segundo as condições contidas na Subseção 3.1.1 acima. Sendo assim, a detecção é implementado no Algoritmo 1.

3.1.4 Exemplo

O exemplo da Listagem 3.1 mostra que existem três asserções (linhas 8, 10 e 11) e que uma delas está sem o argumento para mensagem de erro (linha 11), confirmando a detecção do *smell Assertion Roulette*.

Algorithm 1 Detectar Assertion Roulette

function DETECT(*AST*)
    results ← []
    Traverse *AST*, focusing on *CallExpression* nodes within test cases
    for each *CallExpression* representing a TestCase **do**
        Initialize an empty list for assertions
        Further traverse for *CallExpression* to find assertions
        for each *CallExpression* representing a Assertion **do**
            if assertion with message parameter is found **then**
                Add assertion node to the list
            end if
        end for
        if more than one assertion is found **then**
            Filter out assertions that lack a custom error message and add to *results*
        end if
    end for
    return *results* mapped to *Smell* objects with their locations
end function

```
describe('Test creation file', () => {
  it('create a text', () => {
    const path = "data.txt";
    const text = "1) some random text;\\n2) more random text.";
    createFileText(text);
    const contents = fs.readFileSync(path, "utf8").split("\\n");
    fs.access(path, fs.constants.R_OK, (err) => {
      assert.ok(err, "message");}!
    });
    assert.equal(contents[0], text.split("\\n")[1], "line 1 must be
        '1) some random text;'");
    assert.equal(contents[1], text.split("\\n")[1]);
  });
});
```

Listagem 3.1: Exemplo de *Assertion Roulette*

3.2 *Conditional Test* (CT)

3.2.1 Definição

Refere-se à presença de estruturas de controle de decisão ou repetição (*if, for, while, etc*) nos casos de testes.

3.2.2 Consequência

Este *smell* adiciona um nível de complexidade que não é desejável em casos de testes, os quais devem ser objetivos e garantir que todas as suas asserções sejam avaliadas. As condições impostas pelo *Conditional Test Logic* modificam o comportamento do caso de teste, criando cenários onde asserções podem não ser avaliadas, comprometendo sua qualidade. Além disso, o entendimento do caso de teste pelo desenvolvedor será dificultado.

3.2.3 Detecção

O Algoritmo 2 demonstra que a varredura da AST testa se os nós da árvore são do tipo decisão (*if*) ou repetição (*for, forIn, forOf, while, switch e forEach*).

Algorithm 2 Detectar Conditional Test

```
1:  function DETECT(AST)
2:      results ← []
3:      statements ← [If, For, ForIn, ForOf, While, Switch]
4:      for each node in AST do
5:          if type of node in statements then
6:              Add its start location to results
7:          else
8:              if type of node = CallExpression then
9:                  if expression matches a 'forEach' on an array then
10:                     Add its start location to results
11:                 end if
12:             end if
13:         end if
14:     end for
15:     return results
16: end function
```

```
1 it('check if a number is even', () => {
2   [2, 4, 6].forEach(item => {
3     assert.equal(modulusByTwo(item), 0, "it's an odd number")
4   });
5 });
```

Listagem 3.2: Exemplo de *Conditional Test*

3.2.4 Exemplo

A linha 2 da Listagem 3.2 possui uma instrução de repetição (*forEach*).

3.3 *Duplicate Assert* (DA)

Este *test smell* indica que o caso de teste possui asserções repetidas que avaliam a mesma condição de teste com os mesmos valores como argumento. Isso ocorre quando o desenvolvedor não percebe a repetição durante o ato de copiar e colar um bloco de código ou então utiliza asserções repetidas como forma de depuração de código.

3.3.1 Consequência

A duplicação de asserções faz com que haja desperdício de processamento e consequentemente se gaste mais tempo na execução do caso de teste.

3.3.2 Detecção

Os nós da AST são filtrados por cada caso de teste, e em seguida as asserções são reunidas em uma coleção no qual são detectados as asserções repetidas, conforme o Algoritmo 3.

3.3.3 Exemplo

A Listagem 3.3 apresenta duplicação de asserções nas linhas 3 e 4.

Algorithm 3 Detectar Duplicate Assert

```
1:  function DETECT(AST)
2:      results ← []
3:      for each CallExpression in AST do
4:          if node is a TestCase then
5:              callings ← Empty list to collect assertion calls
6:              for each CallExpression in TestCase do
7:                  if node is an Assertion call then
8:                      Collect expression and its start location
9:                      Add to callings
10:                 end if
11:             end for
12:             if callings has more than one element then
13:                 Identify duplicate assertion calls in callings
14:                 Add duplicates to results
15:             end if
16:         end if
17:     end for
18:     return results
19: end function
```

```
1 it('check line has content', () => {
2   ...
3   assert.equal(contents[0], text.split("\\n")[0], "line 1 must be '1)
        some random text;'");
4   assert.equal(contents[0], text.split("\\n")[0], "line 1 must be '1)
        some random text;'");
5   assert.equal(contents[0], text.split("\\n")[1], "line 2 must be '2)
        more random text;'");
6 });
```

Listagem 3.3: Exemplo de *Duplicate Assert*

3.4 *Eager Test* (EaT)

3.4.1 Definição

A constatação deste *smell* é caracterizada pela existência de múltiplas chamadas de diferentes métodos do SUT.

3.4.2 Consequência

O *Eager Test* adiciona mais dependência ao caso de teste, tornando a sua compreensão mais complexa, e dessa forma, dificultando sua manutenibilidade ou evolução.

3.4.3 Detecção

A detecção deste *smell* exige que seja informado o código-fonte a ser testado juntamente com a suíte de testes. Com isso, serão gerados duas AST: uma para o código de produção e outra para os testes. A regra de detecção varrerá a AST do código de produção em busca de métodos públicos para a geração de uma lista de métodos a serem buscados nos casos de testes. Assim, quando houver a invocação de mais de um método do código de produção em um caso de teste, o *smell* será apontado conforme o Algoritmo 4.

3.4.4 Exemplo

A Listagem 3.4 indica a existência do *Eager Test* nas linhas 19 e 20, onde há duas chamadas para métodos de produção.

3.5 *Empty Test* (EmT)

3.5.1 Definição

Este anti-padrão é caracterizado pela existência de um caso de teste no qual o seu corpo é um bloco vazio ou com conteúdo totalmente comentado.

Algorithm 4 Detectar Eager Test

```
1:  function DETECT(AST)
2:      results ← []
3:      imports ← ResolveImportPaths(AST)
4:      productionMethods ← []
5:      for each import in imports do
6:          filepath ← LoadJSFileOrPackage(import)
7:          if filepath ≠ 'Not Found!' then
8:              code ← ReadFile(filepath)
9:              prod ← ParseCodeToAST(code)
10:             Append production methods from prod to productionMethods
11:         end if
12:     end for
13:     Traverse AST to detect CallExpression within test cases
14:     for each CallExpression in test cases do
15:         if callee is identified and in productionMethods then
16:             Collect such call expressions
17:         end if
18:     end for
19:     if Multiple calls to the same production method detected then
20:         Add call locations as Smell to results
21:     end if
22:     return results
23: end function
```

```
1  // Production code (circle.js)
2  const PI = 3.14159265359;
3
4  export function area(radius) {
5    return (radius ** 2) * PI;
6  }
7
8  module.exports = function circumference(radius) {
9    return 2 * radius * PI;
10 }
11
12 // Test code
13 import { area, circumference } from './circle.js';
14
15 describe('Test geometry functions', () => {
16   it('check the area of a circle', () => {
17     const PI = 3.14159265359;
18     const r = 3;
19     assert.equal(area(r), r ** 2);
20     assert.equal(circumference(r), 2 * PI * r);
21   });
22 });
```

Listagem 3.4: Exemplo de *Eager Test*

3.5.2 Consequência

Mesmo sem um corpo, o caso de teste com este *smell* será chamado para execução. Além do desperdício de tempo e processamento, mesmo que mínimos, o *Empty Test* poderá dar a falsa sensação ao desenvolvedor de que os testes foram executados com sucesso.

3.5.3 Detecção

A detecção deste tipo de *smell* se dá pela busca de casos de testes no qual o seu corpo é um bloco vazio ou comentado, conforme o Algoritmo 5.

Algorithm 5 Detectar Empty Test

```
1:  function DETECT(AST)
2:      results ← []
3:      Traverse AST, focusing on CallExpression nodes
4:      for each CallExpression do
5:          if node represents a TestCase then
6:              Define emptyBody as a function to check if
7:                  a function or arrow expression has an empty body
8:              if any of node's arguments satisfy emptyBody then
9:                  Add a new Smell object to results using node's start location
10:             end if
11:         end if
12:     end for
13:     return results
14: end function
```

3.5.4 Exemplo

A Listagem 3.5 apresenta um caso de teste com o bloco de código do seu corpo comentado (linhas de 3 à 10).

```
describe('Test creation file', () => {
  it('create a text', () => {
    // const path = "data.txt";
    // const text = "1) some random text;\\n2) more random text.";
    // createFileText(text);
    // const contents = fs.readFileSync(path, "utf8").split("\\n");
    // fs.access(path, fs.constants.R_OK, (err) => {
    //   assert.ok(err);
    // });
    // assert.equal(contents[0], text.split("\\n")[0]);
  });
});
```

Listagem 3.5: Exemplo de *Empty Test*

3.6 *Exception Test* (ExT)

3.6.1 Definição

Este *smell* é caracterizado pela presença de um bloco de tratamento de exceção no corpo do caso de teste. Isto ocorre devido ao interesse do desenvolvedor em testar a ocorrência de exceção no código do método a ser testado. No entanto, esta situação deve ser resolvida por meio de asserções específicas da biblioteca de testes.

3.6.2 Consequência

Tratar uma exceção em um caso de teste com blocos do tipo *try* ou levantar exceção com declarações como *throw*, pode adicionar complexidade e dependência desnecessárias.

3.6.3 Detecção

O Algoritmo 6 busca por declarações do tipo *try* e *throw*.

3.6.4 Exemplo

A Listagem 3.6 mostra que o *smell* ocorre no bloco *try* na linha 11 e na instrução *throw* da linha 12.

Algorithm 6 Detectar Exception Test

```
1:  function DETECT(AST)
2:      results ← []
3:      Traverse AST, focusing on CallExpression nodes within test cases
4:      for each CallExpression representing a TestCase do
5:          Traverse inside the TestCase for:
6:              TryStatement and ThrowStatement
7:          for each found statement do
8:              Add a new Smell object to results, using statement's start location
9:          end for
10:     end for
11:     return results
12: end function
```

```
1  import fs from "fs";
2  import {createFileText} from "../src/production";
3
4  describe('Test creation file', () => {
5    it('first line size is 20', () => {
6      const path = "data.txt";
7      const text = "1) some random text;\\n2) more random text.";
8      createFileText(text);
9      const contents = fs.readFileSync(path, "utf8").split("\\n");
10     console.log(contents);
11     try {
12       throw "myException";
13     }
14     catch (e) {
15      logMyErrors(e);
16     }
17     fs.access(path, fs.constants.R_OK, (err) => {
18       assert.ok(err);
19     });
20     assert.equal(contents[0].length, 20);
21   });
22 });
```

Listagem 3.6: Exemplo de *Exception Handling*

```
1 x = 5; // atribui 5 à variável global x
2 function exemplo() {
3     console.log(x); // undefined, devido ao hoisting
4     var x = 10;
5     console.log(x); // 10
6 }
7 exemplo();
```

Listagem 3.7: Exemplo de hoisting

3.7 *Global Variable* (GV)

3.7.1 Definição

O *test smell Global Variable* está relacionado aos mecanismos que ocorrem durante a execução do código e o tratamento de escopo das variáveis pelo interpretador *JavaScript*. A linguagem JavaScript apresenta um comportamento chamado *hoisting*, que significa "elevação". Este comportamento da linguagem é responsável por mover as declarações para o topo do seu escopo antes da execução do código. A Listagem 3.7 ilustra o *hoisting*.

A saída do *console.log(x)* na linha 3 é "*undefined*", pois a declaração foi "içada"para o topo do escopo e alocado um espaço de memória com o conteúdo *undefined*. Caso não existisse o comportamento de *hoisting*, a saída seria a mensagem de erro "*Uncaught ReferenceError: x is not defined*".

O escopo das variáveis declaradas com a palavra reservada *var* é o contexto de execução atual. Se ao atribuir um valor a uma variável não declarada previamente, ela é criada como global no momento da execução da atribuição.

As variáveis globais permitem que sejam redeclaradas e atualizadas. Com o intuito de oferecer mais segurança na manipulação dos escopos das variáveis, a especificação *ECMAScript 2015* (*ES6*) trouxe novos recursos como as palavras reservadas *let* e *const*. Ambas possuem escopo por bloco e não por função. A declaração com *let* permite atualização do valor da variável, mas impede a redeclaração. Já a declaração com *const* impede a atualização de valor e a redeclaração. A Listagem 3.8 demonstra as características de *let* e *const*.

Com base no que foi apresentado, é importante evitar as variáveis globais e, preferencialmente, utilizar as declarações *let* e *const*. Outra recomendação é sempre declarar

```
let greeting = "hey hi";
greeting = "say hello instead";
let greeting = "say goodbye"; // error:  Identifier 'greeting' has
                              // already been declared
const greeting2 = "hey hi";
greeting2 = "say Hello instead"; // error:  Assignment to constant variable.
const greeting2 = "say Hello instead"; // error:  Identifier 'greeting2'
                                       // has already been declared
```

Listagem 3.8: Exemplo de declarações com *let* e *const*

as variáveis previamente à utilização, tornando o *hoisting* explícito.

3.7.2 Consequência

Variáveis globais podem ocasionar:

- Poluição do objeto global, que no contexto dos navegadores é o objeto ***window***;
- Risco de sobrescrever variáveis;
- Dificuldade na depuração do código;
- Dificuldade na manutenção do código;
- Desempenho pode ser prejudicado, uma vez que o acesso às variáveis globais pode ser mais demorado devido à maneira como os interpretadores administram o escopo;
- Problemas no escopo de função. Por exemplo: uma variável declarada em um bloco condicional ou de repetição, estará disponível para toda a função.

3.7.3 Detecção

A detecção deste *smell* é realizada por funções que verificam se existe alguma declaração de variável do tipo "*var*"no caso de teste ou na suíte, ignorando o *var* caso seja um carregamento de módulo via *require*, conforme o algoritmo 7.

Algorithm 7 Detectar Global Variable

```
1: function DETECT(AST)
2:     results ← []
3:     Traverse AST, focusing on VariableDeclaration nodes
4:     for each VariableDeclaration do
5:         if declaration kind is var and name is not this then
6:             foundRequire ← false
7:             Traverse within the declaration for CallExpression
8:             for each CallExpression do
9:                 if expression is a require call then
10:                    Set foundRequire to true
11:                end if
12:            end for
13:            if foundRequire is false then
14:                Add a new Smell object to results, using declaration's start location
15:            end if
16:        end if
17:    end for
18:    return results
19: end function
```

3.7.4 Exemplo

A listagem 3.9 apresenta a presença do *smell* nas linhas 8 e 9 e as linhas 1, 2 e 3 são ignoradas.

3.8 *Ignored Test* (IT)

3.8.1 Definição

Em geral, o *Ignored Test* recorre a alguma sintaxe específica para evitar a avaliação das asserções de um caso de teste ou toda a suíte de testes. Essa situação deveria ser utilizada apenas como um recurso momentâneo para desativar o teste e não permanentemente, caracterizando como um *test smell*.

3.8.2 Consequência

Este *smell* faz com que os testes afetados não sejam executados, acarretando tanto a não avaliação do código de produção, como também a possível obsolescência dos testes

```
var async = require('../lib');
var expect = require('chai').expect;
var assert = require('assert');

describe('applyEach', function () {
    it('applyEach', function (done) {
        var call_order = [];
        var one = function (val, cb) {
            expect(val).to.equal(5);
            setTimeout(function () {
                call_order.push('one');
                cb(null, 1);
            }, 10);
        };
...
```

Listagem 3.9: Exemplo de *Ignored Test*

ignorados.

3.8.3 Detecção

A detecção deste *smell* é realizada por funções que verificam se algum nó do caso deteste ou suíte na AST possui a propriedade *skip*, conforme o algoritmo 8.

Algorithm 8 Detectar Ignored Test

```
1: function DETECT(AST)
2:     results ← []
3:     Traverse AST, focusing on CallExpression nodes
4:     for each CallExpression do
5:         if expression is marked as ignored (describe, test case, or this) then
6:             Add a new Smell object to results, using callee property's start location
7:         end if
8:     end for
9:     return results
10: end function
```

3.8.4 Exemplo

A listagem 3.10 apresenta a presença do *smell* na linha 2.

```
describe('Test creation file', () => {
  it.skip('create a text', () => {
    const path = "data.txt";
    const text = "1) some random text;\\n2) more random text.";
```

```
5       createFileText(text);
6       const contents = fs.readFileSync(path, "utf8").split("\\n");
7       fs.access(path, fs.constants.R_OK, (err) => {
8         assert.ok(err);
9       });
10      assert.equal(contents[0], text.split("\\n")[0]);
11    });
12  });
```

Listagem 3.10: Exemplo de *Ignored Test*

3.9 *Lazy Test* (LT)

3.9.1 Definição

Este tipo de *smell* ocorre quando um mesmo método de produção é passado como argumento nas asserções em múltiplos casos de testes de uma mesma suite.

3.9.2 Consequências

O efeito da ocorrência destes *smell* é dificultar a manutenibilidade da suite de testes.

3.9.3 Detecção

Algorithm 9 Detectar Lazy Test

```
1: function DETECT(AST)
2:     findings ← []
3:     Resolve and load imports to identify production methods
4:     productionMethods ← ExtractProductionMethods(imports)
5:     Traverse AST for CallExpression in test cases
6:     for each CallExpression do
7:         Collect unique production method calls and add to findings
8:     end for
9:     Identify duplicates among collected method calls
10:    duplicates ← FilterDuplicates(findings)
11:    return Duplicates mapped to Smell objects
12: end function
```

3.9.4 Exemplo

A Listagem 3.11 apresenta a incidência do *smell Lazy Test* nas linhas 19 e 27.

```
1  // Production code (circle.js)
2  const PI = 3.14159265359;
3
4  export function area(radius) {
5    return (radius ** 2) * PI;
6  }
7
8  module.exports = function circumference(radius) {
9    return 2 * radius * PI;
10 }
11
12 // Test code
13 import { area, circumference } from './circle.js';
14
15 describe('Test creation file', () => {
16   it('create a text', () => {
17     const PI = 3.14159265359;
18     const r = 3;
19     assert.equal(area(r), r ** 2);
20   });
21 });
22
23 describe('Test updating file', () => {
24   it('update a text', () => {
25     const PI = 3.14159265359;
26     const r = 3;
27     assert.equal(area(r), r ** 2);
28     assert.equal(circumference(r), 2 * PI * r);
29     assert.equal(circumference(r), 2 * PI * r);
30   });
31 });
```

Listagem 3.11: Exemplo de *Lazy Test*

3.10 *Magic Number* (MN)

3.10.1 Definição

Este tipo de *smell* ocorre quando as asserções contêm literais numéricos como argumentos a serem avaliados.

3.10.2 Consequência

A presença do *Magic Number* dificulta a compreensão do código do teste, impossibilitando ao desenvolvedor ter uma noção clara da intenção do teste, além de comprometer a manutenabilidade.

3.10.3 Detecção

A ocorrência deste *smell* é sinalizada quando algum dos argumentos a serem avaliados nas asserções são do tipo literal numérico, conforme o Algoritmo 10.

Algorithm 10 Detectar Magic Number

```
1:  function DETECT(AST)
2:      results ← []
3:      Traverse AST to identify CallExpression within test cases
4:      for each CallExpression in a TestCase do
5:          if expression is an Assertion or specific to a testing framework then
6:              Filter arguments that are numeric literals
7:              Create a Smell object for each numeric argument
8:              Add these Smell objects to results
9:          end if
10:     end for
11:     return results
12: end function
```

3.10.4 Exemplo

A listagem 3.12 apresenta a presença do *Magic Number* nas linhas 10, 11, 12.

```
1 describe('Test creation file', () => {
2   it('first line size is 20', () => {
3     const path = "data.txt";
```

```
4       const text = "1) some random text;\\n2) more random text.";
5       createFileText(text);
6       const contents = fs.readFileSync(path, "utf8").split("\\n");
7       fs.access(path, fs.constants.R_OK, (err) => {
8         assert.ok(err);
9       });
10      assert.equal(contents[0].length, 20);
11      expect(object).to.have.lengthOf(3);
12      object.should.to.have.lengthOf(3);
13    });
14  });
```

Listagem 3.12: Exemplo de *Magic Number*

3.11 *Mystery Guest* (MG)

3.11.1 Definição

Este *smell* ocorre quando o caso de teste usa recursos externos, como, por exemplo, arquivos, banco de dados ou serviços *web*. No entanto, a prática correta seria o emprego de artefatos como *mocks* e *stubs* para simular o recurso desejado.

3.11.2 Consequência

A presença do *Mystery Guest* torna o teste não-determinístico, em virtude da dependência de um recurso que está fora de um ambiente controlado.

3.11.3 Detecção

Até o momento, a detecção deste smell contempla apenas o reconhecimento de manipulação de arquivos e o acesso a serviços *web*. Para isso é necessária a identificação de importação das bibliotecas *fs* e *http*, além de chamadas para os métodos dessas, conforme o Algoritmo 11.

Algorithm 11 Detectar Mystery Guest

```
1:  function DETECT(AST)
2:      results ← []
3:      Define a list of questionable methods across various modules
4:      if Mocks are present and specific mocking expressions are found then
5:          Return empty results
6:      end if
7:      Identify all required modules in the AST
8:      Traverse AST focusing on CallExpression within test cases
9:      for each CallExpression representing a TestCase do
10:         Further traverse for MemberExpression
11:         for each MemberExpression do
12:             Compare against the list of questionable methods
13:             if method is called without being mocked then
14:                 Add a new Smell object with the location of the call to results
15:             end if
16:         end for
17:     end for
18:     return results
19: end function
```

3.11.4 Exemplo

A Listagem 3.13 mostra a ocorrência do *smell* nas linhas 9 e 10.

```
1  import fs from "fs";
2  import {createFileText} from "../src/production";
3
4  describe('Test creation file', () => {
5    it('create a text', () => {
6      const path = "data.txt";
7      const text = "1) some random text;\\n2) more random text.";
8      createFileText(text);
9      const contents = fs.readFileSync(path, "utf8").split("\\n");
10     fs.access(path, fs.constants.R_OK, (err) => {
11       assert.ok(err);
12     });
13     assert.equal(contents[0], text.split("\\n")[1], "line 1 must be
           '1) some random text;'");
14   });
15 });
```

Listagem 3.13: Exemplo de *Mystery Guest*

3.12 *Redudant Assertion* (RA)

3.12.1 Definição

Este *test smell* é caracterizado pela existência de asserções cujo parâmetro atual e o esperado são iguais. Em geral, esta situação ocorre quando o desenvolvedor modifica o comportamento de asserção para que esta sempre passe com sucesso.

3.12.2 Consequência

A presença do *Redudant Assertion* indica que a asserção não tem objetivo claro. Com isso, causa o comprometimento da qualidade do caso de teste, transmitindo a falsa sensação de um teste finalizado com sucesso.

3.12.3 Detecção

O *Redudant Assertion* é identificado pela existência de asserções cujos argumentos para avaliação são iguais, conforme o Algoritmo 12.

Algorithm 12 Detectar Redundant Assertion

```
1: function DETECT(AST)
2:     results ← []
3:     Define helper functions to identify literal arguments and assertions
4:     Traverse AST to identify CallExpression relevant to assertions
5:     for each CallExpression do
6:         if expression is an assertion then
7:             if callee name starts with 'assert' or specific patterns then
8:                 if all arguments are literals then
9:                     Add a new Smell object with the location of the call to results
10:                end if
11:            else if callee name starts with 'expect' then
12:                Consider both current and chained call arguments
13:                if all considered arguments are literals then
14:                    Add a new Smell object with the location of the call to results
15:                end if
16:            end if
17:        end if
18:    end for
19:    return results
20: end function
```

```
1  import {createFileText} from "../src/production";
2
3  describe('Test creation file', () => {
4    it('first line size is 20', () => {
5      const path = "data.txt";
6      const text = "1) some random text;\\n2) more random text.";
7      createFileText(text);
8      assert.equal(true, true);
9      expect(true).to.be.equal(true);
10   });
11 });
```

Listagem 3.14: Exemplo de *Redudant Assertion*

3.12.4 Exemplo

A Listagem 3.14 mostra a presença do *smell* nas linhas 8 e 9:

3.13 *Redudant Print* (RP)

3.13.1 Definição

Este *smell* decorre de uma má prática de depuração. Nela, o desenvolvedor escreve chamadas que deveriam ser temporárias para métodos que imprimem valores na saída padrão da aplicação, mas que acabam ficando permanentes no código.

3.13.2 Consequência

O *Redudant Print* pode aumentar consideravelmente o tempo de execução do caso de teste afetado, impactando negativamente no desempenho geral das suítes de testes.

3.13.3 Detecção

A identificação deste *smell* se dá pela existência de métodos que imprimem na saída padrão, tais como *console.log, console.error, console.info, console.trace* e *console.warn* conforme o Algoritmo 13.

Algorithm 13 Detectar Redundant Print

```
1: function DETECT(AST)
2:     results ← []
3:     Traverse AST, focusing on CallExpression nodes within test cases
4:     for each CallExpression do
5:         Further traverse for MemberExpression
6:         if expression involves console usage then
7:             Add a new Smell object with the location of the usage
8:         end if
9:     end for
10:    return results
11: end function
```

3.13.4 Exemplo

A Listagem 3.15 apresenta a presença do *smell Redudant Print* na linha 10 no qual é feita uma chamada para *console.log*.

```
1  import fs from "fs";
2  import {createFileText} from "../src/production";
3
4  describe('Test creation file', () => {
5    it('first line size is 20', () => {
6      const path = "data.txt";
7      const text = "1) some random text;\\n2) more random text.";
8      createFileText(text);
9      const contents = fs.readFileSync(path, "utf8").split("\\n");
10     console.log(contents);
11     fs.access(path, fs.constants.R_OK, (err) => {
12       assert.ok(err);
13     });
14     assert.equal(contents[0].length, 20);
15   });
16 });
```

Listagem 3.15: Exemplo de *Redudant Print*

3.14 *Redudant Optimism* (RO)

3.14.1 Definição

Este *smell* indica que o teste pressupõe a existência de uma instância de arquivo, contudo sem efetuar a devida confirmação desse recurso externo ao ambiente do teste.

3.14.2 Consequência

O *Redudant Optimism* possui a mesma consequência do *smell Mystery Test* (SubSeção 3.11).

3.14.3 Detecção

A constatação deste *smell* ocorre por meio da presença da importação da biblioteca *"fs"* e nas chamadas de algum dos métodos *"open"*, *"readFile"* e *"writeFile"* 14.

Algorithm 14 Detectar Redudant Optimism

```
1: function DETECT(AST)
2:     results ← []
3:     Identify all 'require' statements for the 'fs' module
4:     accessCalls ← []                          ▷ To store calls to 'fs.access'
5:     fsMethodsCalls ← []                       ▷ To store calls to other 'fs' methods
6:     Traverse AST, focusing on CallExpression within test cases
7:     for each CallExpression do
8:         Check if node calls 'fs.access' or other 'fs' methods
9:         if 'fs.access' is called then
10:            Add node to accessCalls
11:        else
12:            Add node to fsMethodsCalls if other 'fs' method is called
13:        end if
14:    end for
15:    if no 'access' calls are made then
16:        Add all nodes from fsMethodsCalls to results as Smell
17:    end if
18:    return results
19: end function
```

```
1  import fs from "fs";
2  import {createFileText} from "../src/production";
3
4  describe('Test creation file', () => {
5    it('first line size is 20', () => {
6      const path = "data.txt";
7      const text = "1) some random text;\\n2) more random text.";
8      createFileText(text);
9      const contents = fs.readFile(path, "utf8").split("\\n");
10     assert.equal(contents[0].length, 20);
11   });
12 });
```

Listagem 3.16: Exemplo de *Redudant Optimism*

3.14.4 Exemplo

A Listagem 3.16 exibe a ocorrência do *smell Redudant Optimism* na linha 9.

3.15 *Sleepy Test* (ST)

3.15.1 Definição

Este *smell* ocorre quando um caso de teste possui chamadas de funções para interrupção temporária de *threads*, com o objetivo de simular eventos externos como chamadas assíncronas, atrasos e pausas no processamento.

3.15.2 Consequências

O emprego de funções de tempo real para interrupção temporária de *threads* pode tornar imprevisíveis os resultados dos testes afetados. Para remediar este problema, as bibliotecas disponibilizam mecanismos próprios para lidar com esta situação.

3.15.3 Detecção

A detecção deste *smell* ocorre com a identificação de chamadas para a função *setTimeout()*, conforme o algoritmo 15.

Algorithm 15 Detectar Sleep Test

```
1: function DETECT(AST)
2:     results ← []
3:     Traverse AST, looking for CallExpression within test cases
4:     for each CallExpression in a TestCase do
5:         Further traverse for nested CallExpression
6:         if expression is identified as a sleep function then
7:             Add a new Smell object with the location of the call
8:         end if
9:     end for
10:    return results
11: end function
```

```
1  function myFunc(arg) {
2    console.log(\`arg was => \${ arg} \`);
3  }
4
5  describe('Test creation file', () => {
6    it('first line size is 20', () => {
7      const path = "data.txt";
8      const text = "1) some random text;\\n2) more random text.";
9      createFileText(text);
10     setTimeout(myFunc, 1500, 'funky');
11   });
12 });
```

Listagem 3.17: Exemplo de *Sleepy Test*

3.15.4 Exemplo

A Listagem 3.17 demonstra a ocorrência do *Sleepy Test* na linha 10.

3.16 *Unknown Test* (UT)

3.16.1 Definição

Este anti-padrão é caracterizado pela inexistência de asserções no corpo do caso de teste, e pode ocorrer quando o desenvolvedor esquece de implementar a asserção ou exclui a asserção existente.

```
describe('Test creation file', () => {
  it('first line size is 20', () => {
    const path = "data.txt";
    const text = "1) some random text;\\n2) more random text.";
    createFileText(text);
  });
});
```

Listagem 3.18: Exemplo de *Unknown Test*

3.16.2 Detecção

A detecção deste *smell* ocorre quando ao fim da varredura dos nós do corpo do caso de teste, não é detectada nenhuma asserção 16.

Algorithm 16 Detectar Unknown Test

```
1:  function DETECT(AST)
2:      results ← []
3:      Define helper function hasBody to check for non-empty function bodies
4:      Traverse AST, focusing on CallExpression nodes within test cases
5:      for each CallExpression representing a TestCase do
6:          if test case has a non-empty body then
7:              Initialize an empty list of assertions
8:              Traverse for assertions using CallExpression and MemberExpression
9:              if no assertions are found then
10:                 Add a new Smell object with the location of the test case
11:             end if
12:         end if
13:     end for
14:     return results
15: end function
```

3.16.3 Exemplo

A Listagem 3.18 demonstra que o caso de teste não possui chamadas para asserções.

3.17 Considerações Finais

Neste capítulo apresentamos um catálogo de 15 *test smells* clássicos e um específico da linguagem *JavaScript*. Para cada *test smell* foi informado a definição, consequência, forma de detecção e um exemplo. No próximo capítulo apresentaremos a ferramenta de detecção de *test smells* desenvolvida neste trabalho de doutorado.

Capítulo 4

STEEL

O presente capítulo apresenta a ferramenta de análise estática para detecção de *test smells* desenvolvida neste trabalho. O repositório da ferramenta se encontra no link: `https://github.com/daltonjorge/steel`.

4.1 Ferramenta STEEL

Nesta seção apresentaremos a ferramenta STEEL. Inicialmente realizaremos uma definição do que se trata a ferramenta e quais os seus objetivos, em seguida detalharemos sua arquitetura e tecnologias empregadas, e por fim demonstraremos sua utilização.

4.1.1 Definição

STEEL é uma ferramenta de análise estática para linha de comando que possibilita a detecção automática de *smells* em códigos de testes na linguagem *JavaScript*. Além de detectar os *test smells*, a ferramenta STEEL também reporta métricas de qualidade de código para os testes. Dentre as métricas de qualidade estão: o número de linhas físicas e lógicas do código-fonte, os índices de complexidade ciclomática simples e condensada, a manutenibilidade, e as propriedades de *Halstead* (*Bugs*, *Difficulty*, *Effort*, *Length*, *Time*, *Vocabulary*, *Volume*).

4.1.2 Objetivos

Alguns *test smells*, tais como os do tipo *Magic number*, são facilmente detectados a olho nu pelo desenvolvedor. No entanto, outros *smells* mais complexos são difíceis de identificar sem o auxílio de uma ferramenta adequada. Em adição à complexidade do *smell*, o tamanho de um projeto também torna os *smells* menos visíveis. Um dos objetivos da ferramenta Steel é permitir a descoberta automática de *smells* nos testes, apontando o tipo do *smell* e o respectivo local de ocorrência no código-fonte.

4.1.3 Arquitetura

O conjunto arquitetural da ferramenta Steel é composto pelos seguintes módulos: o *parser* da linguagem JavaScript contendo os analisadores léxico e sintático, os detectores de *test smells* baseados em regras, o analisador da qualidade de código, e o gerador de relatórios nos formatos *JavaScript Object Notation* (JSON), *Hypertext Markup Language* (HTML) e *Command Line Interface* (CLI). A Figura 4.1.3 ilustra a referida arquitetura Steel.

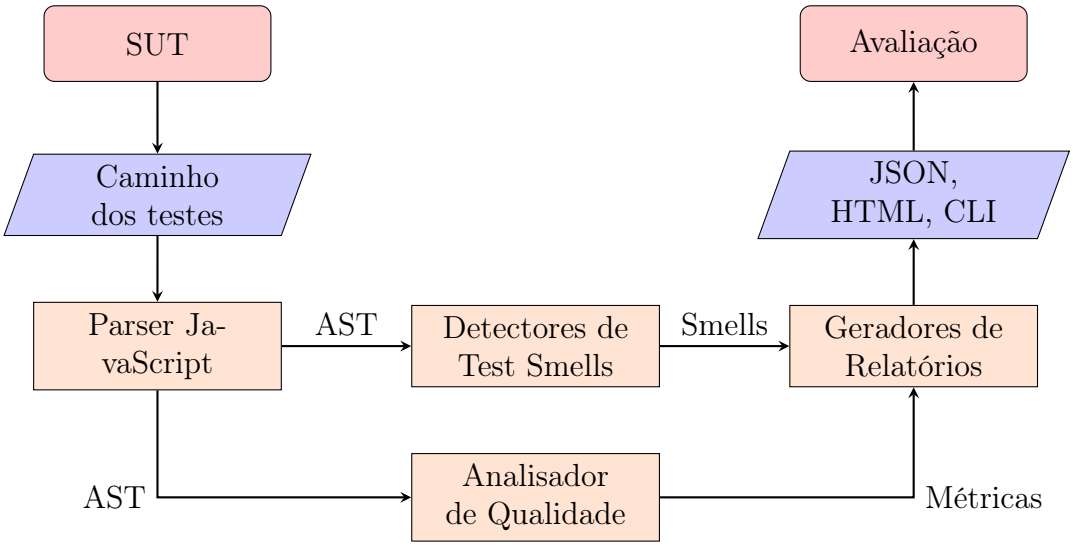


Figura 4.1: Arquitetura da ferramenta

Fluxo

O fluxo desta arquitetura se inicia a partir do repositório do SUT, em que o usuário da ferramenta deve informar os nomes dos arquivos de código-fonte dos testes a serem

analisados pela ferramenta. Para isso, deve-se utilizar o padrão *gobbling*, o qual é uma forma de especificação de nomes de arquivos por meio de caracteres coringas e casamento de padrões. Embora esse conceito seja comum em sistemas *Unix-like*, ele também é aplicado em muitos outros contextos, incluindo programação e ferramentas de busca. Os caracteres coringas mais comuns são:

- Asterisco (`*`): Corresponde a qualquer sequência de caracteres (incluindo nenhuma);
- Interrogação (`?`): Corresponde a exatamente um caractere, qualquer que seja;
- Colchetes (`[]`): Permitem especificar um conjunto de caracteres. Por exemplo, [a-z] corresponde a qualquer letra minúscula.

Por exemplo, se quisermos elencar todos os testes unitários do repositório do projeto *React*, a construção pode ser definida por `**/__tests__/*-test.internal.js`. Este casamento de padrão define que serão encontrados todos os arquivos que terminam com terminação `-test.internal.js` e que se localizam em qualquer subdiretório de nome `__tests__` no repositório do projeto. O uso do padrão *gobbling* é uma evolução da ferramenta para facilitar sua utilização, principalmente em projetos com muitas suítes de testes. As versões iniciais de STEEL obrigavam o usuário da ferramenta fornecer um arquivo *Comma-separeted Values* (CSV) contendo duas colunas, sendo a primeira coluna o caminho e nome do arquivo de teste, e a segunda coluna o caminho e nome do arquivo de produção sob teste.

Em seguida, a primeira fase interna da ferramenta é a transformação do código-fonte do teste em estrutura de árvore sintática, do inglês AST. A segunda fase compreende dois caminhos, sendo que o primeiro tem o objetivo de detectar os *smells* na AST por meio de analisadores estáticos específicos para cada tipo de *smell*. Ainda nesta etapa, mais especificamente nos *smells Eager Test* (EaT) e *Lazy Test* (LT), será feito a transformação do respectivo código-fonte de produção sob teste em árvore sintática. No segundo caminho, a AST é avaliada pelo módulo de análise de qualidade, onde são extraídas as métricas (número de linhas de código, complexidade ciclomática, manutenibilidade e índices de *Halstead*).

Por fim, os resultados das análises de detecção de *smells* e extração das métricas de qualidade são utilizados na fase de geração de relatórios. Os *smells* detectados e as métricas de complexidade do código são exportados nos formatos JSON e HTML. Além disso, os resultados também são impressos na interface de linha de comando, com o intuito de utilização em processo de integração contínua.

Tecnologias adotadas

A ferramenta STEEL foi desenvolvida em *Typescript*[1] e escolhemos esta linguagem pelas vantagens que suas características apresentam sobre o *JavaScript*. *Typescript* é uma extensão da linguagem *JavaScript* e seu principal benefício é a sua inerente tipagem forte. Essa característica aliada ao *Language Server Protocol* (LSP)[2] implementado pelo servidor da linguagem e ao editor utilizado (*Visual Studio Code*) permitiu usufruir dos recursos de diagnósticos e de autocompletar, os quais viabilizaram a mitigação dos erros de tipos em tempo de execução e a codificação mais rápida e segura.

Para a transformação dos códigos-fontes em árvores sintáticas, utilizamos o módulo *@babel/parser*[3] do compilador *Babel*[4]. Este compilador segue as especificações do padrão *EcmaScript-262*[5] e permite a retrocompatibilidade (versão a partir do ES2015) do código analisado com versões antigas de navegadores e ambientes de execução *JavaScript*, como o *Node.js*[6]. A árvore sintática gerada pelo *@babel/parser* emprega a especificação *ESTree*[7], que possibilita a identificação dos tipos dos nós da árvore e sua localização no código-fonte.

O analisador de qualidade de código emprega a biblioteca *typhonjs-escomplex-module*[8] para a extração das métricas de complexidade de código.

Para facilitar a criação de novas regras para detectores de *test smells*, desenvolvemos um mecanismo de carga dinâmica de *plugin*. Para isso, o novo tipo de *test smell* deve estender uma classe abstrata *Rule* e implementar o método abstrato *detect* em um

[1] `https://www.typescriptlang.org`
[2] `https://langserver.org`
[3] `https://babeljs.io/docs/en/babel-parser`
[4] `https://babeljs.io`
[5] `https://www.ecma-international.org/publications/standards/Ecma-262.htm`
[6] `https://nodejs.org/en/`
[7] `https://github.com/estree/estree`
[8] `https://github.com/typhonjs-node-escomplex/typhonjs-escomplex-module`

arquivo *Typescript*. A ferramenta STEEL fará a carga dinâmica dos arquivos *plugins* que estão na pasta de nome *plugins*. A Figura 4.2 ilustra a sequência de chamadas durante a carga do detector *EagerTest*.

4.1.4 Exemplo de uso

Como foi citado anteriormente, STEEL é uma ferramenta de linha de comando e para sua utilização é necessário definir primeiramente a máscara *gobbling* para a detecção dos arquivos de testes, como ilustrado nas Figuras 4.3 e 4.4

A Figura 4.5 ilustra o relatório gerado em formato HTML para projeto *Day.js*. Nas caixas coloridas estão as métricas gerais para todos os arquivos de testes (totais de arquivos de testes, de métodos de teste, de arquivos de testes infectados e de *smells* detectados.

4.1.5 Escopo

Devido ao vasto ecossistema de bibliotecas e *frameworks* da linguagem *JavaScript* tivemos que limitar o alcance da ferramenta STEEL. Priorizamos o suporte da ferramenta para asserções nativas do *Node.js* e aos *frameworks Mocha*[9] e *Jest*[10], e as bibliotecas de asserções *Chai*[11], *Sinon*[12] e *Nock*[13]. A escolha destas tecnologias foram tomadas com base na popularidade e flexibilidade de uso, como o *Mocha/Chai*, e também na facilidade de uso e forte suporte da indústria, como o *Jest* desenvolvida pelo *Facebook*. O *Jest* possui um ferramental completo e funcional sem necessidade prévia de configuração. Acrescentamos também a biblioteca de dublês de testes *Sinon* e o *Nock*, específico para *mocking* de servidor *Hypertext Transfer Protocol* (HTTP).

[9] `https://mochajs.org`
[10] `https://jestjs.io`
[11] `https://www.chaijs.com/`
[12] `https://sinonjs.org`
[13] `https://github.com/nock/nock`

4.2 Validação da Ferramenta

Para realizar a validação da ferramenta STEEL , foi utilizado um *dataset* compreendendo várias instâncias de incidência de *smells*, identificados em suítes de teste específicas. Cada instância no *dataset* foi classificada segundo a sua verdadeira natureza (*smell* detectado corretamente ou não, avaliado manualmente por um especialista) e a predição feita pela ferramenta STEEL (verdadeiro ou falso). Com base nestas classificações, foi possível calcular duas métricas fundamentais para avaliar o desempenho da ferramenta: precisão (*precision*) e *recall*.

A precisão, calculada como a razão entre o número de verdadeiros positivos (TP) e a soma dos verdadeiros positivos e falsos positivos ($TP + FP$), foi de 87.23%. Esta métrica indica que, das instâncias classificadas pela ferramenta como *test smells*, aproximadamente 87% foram classificações corretas. Em contrapartida, o *recall*, que mede a proporção de verdadeiros positivos detectados pela ferramenta em relação ao total de instâncias positivas reais ($TP/(TP + FN)$), atingiu 85.42%. Este valor sugere que a ferramenta conseguiu identificar cerca de 85% dos *test smells* presentes no *dataset*.

É importante destacar que os resultados aqui discutidos refletem uma versão antiga da ferramenta, o qual já foi atualizado com base nestes resultados preliminares. Novas avaliações de precisão e *recall*, que esperamos possam demonstrar melhorias significativas, serão disponibilizadas no repositório da ferramenta para consulta. Ademais, ressalta-se que os resultados obtidos são relativos à detecção de *smells* em um único projeto de software. Reconhecemos a importância de uma análise mais abrangente e, portanto, mais projetos serão avaliados futuramente para se obter uma visão mais robusta e generalizável sobre a precisão e *recall* da ferramenta STEEL .

4.3 Considerações Finais

Neste capítulo apresentamos a ferramenta STEEL , desenvolvida em nossa pesquisa para auxiliar na detecção automática de *test smells*, conforme as regras de detecção estipuladas no catálogo do Capítulo 3. No próximo capítulo apresentaremos o estudo exploratório sobre a incidência de *test smells* na linguagem *JavaScript* e sua relação com as métricas de qualidade em um conjunto de projetos de *software*.

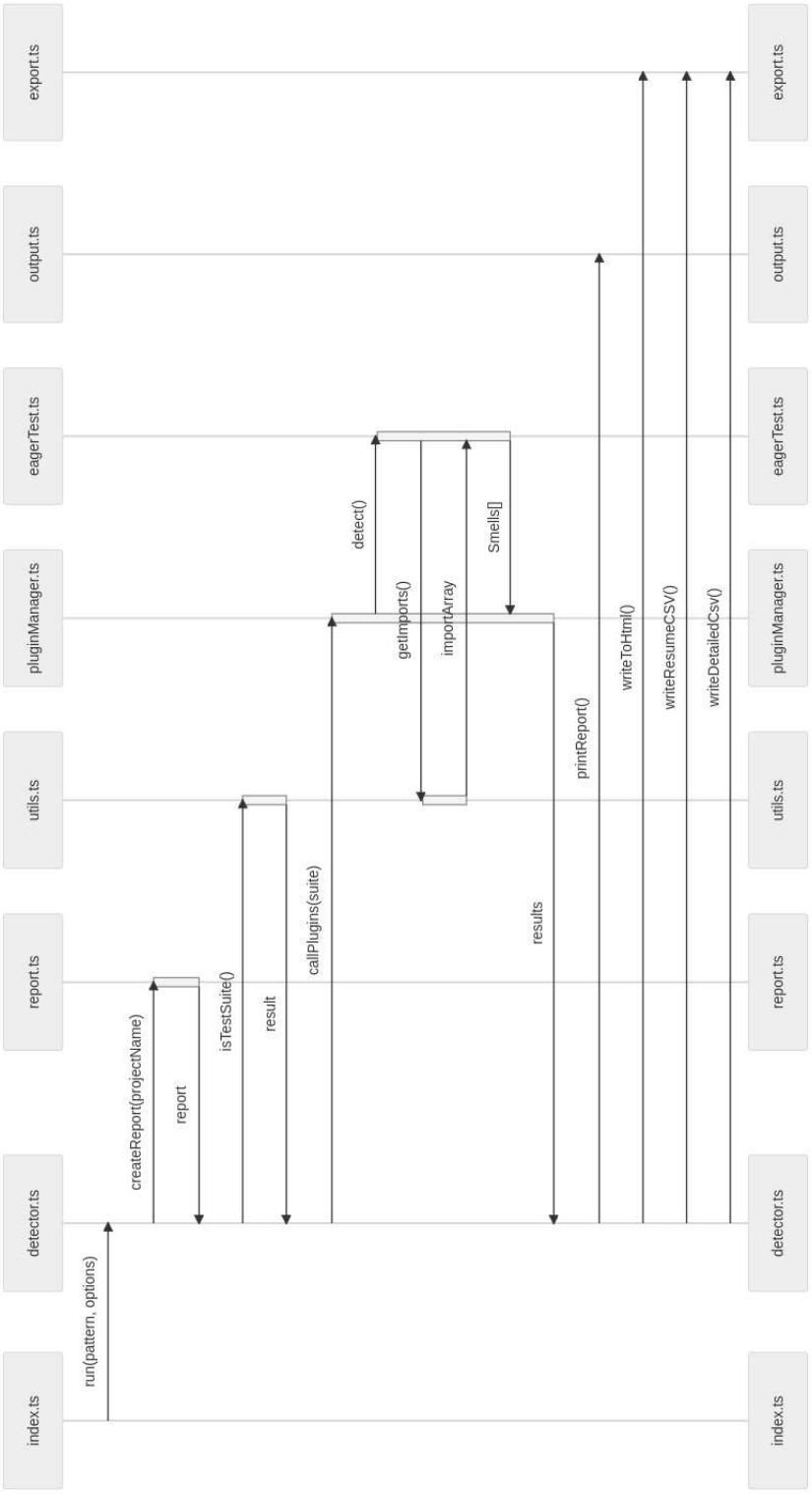


Figura 4.2: Diagrama de Sequência para a detecção do *Eager Test*

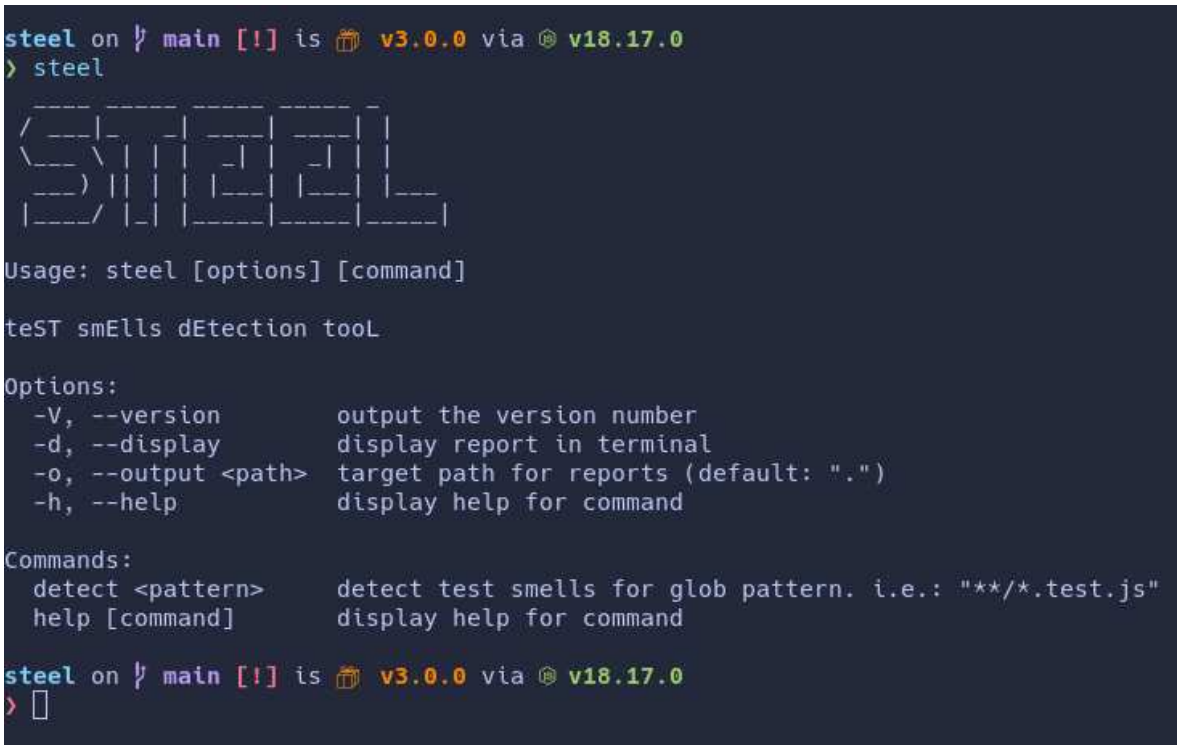


Figura 4.3: Exemplo das opções e comandos disponíveis em STEEL

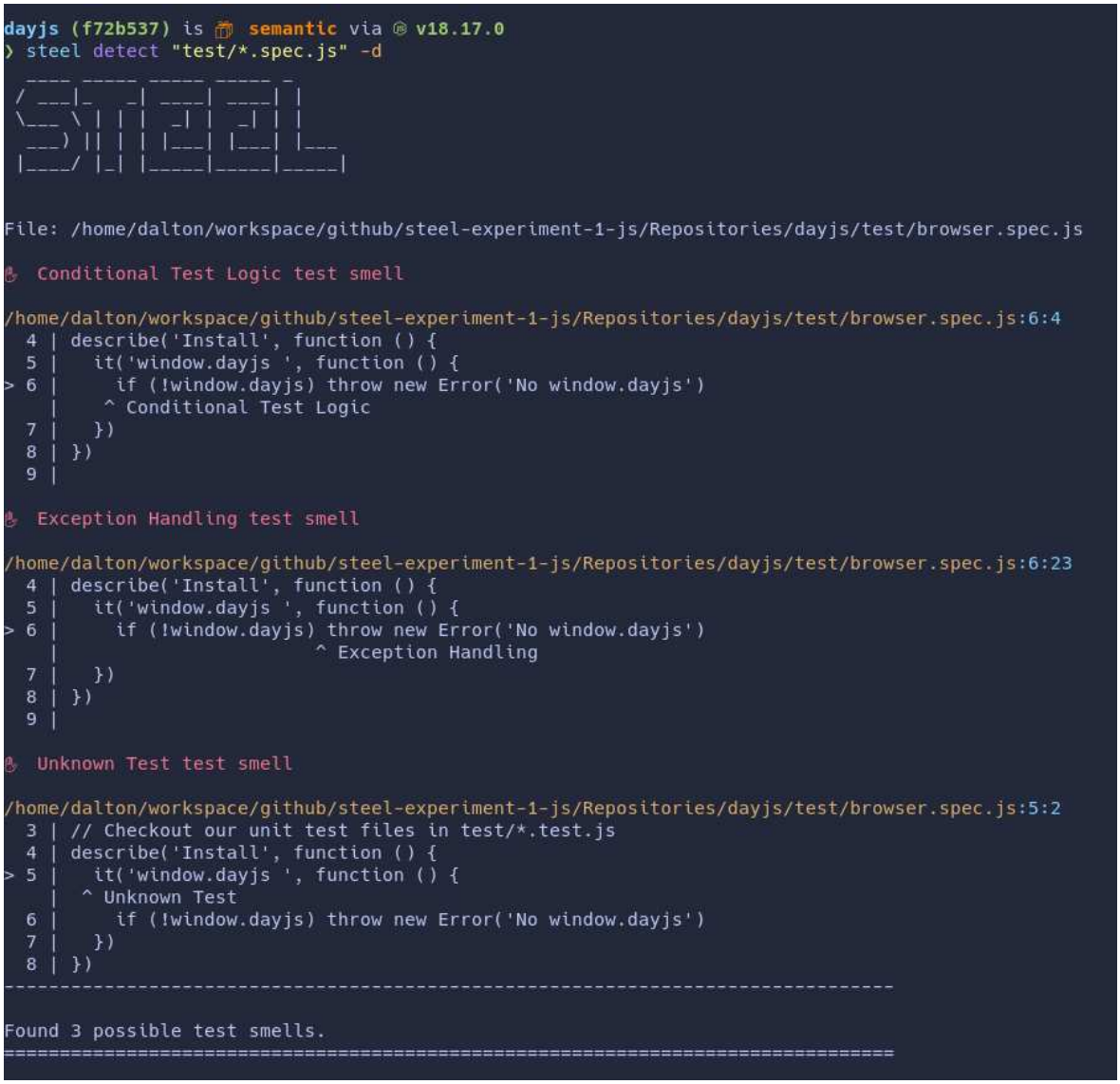


Figura 4.4: Exemplo de saída em linha de comando

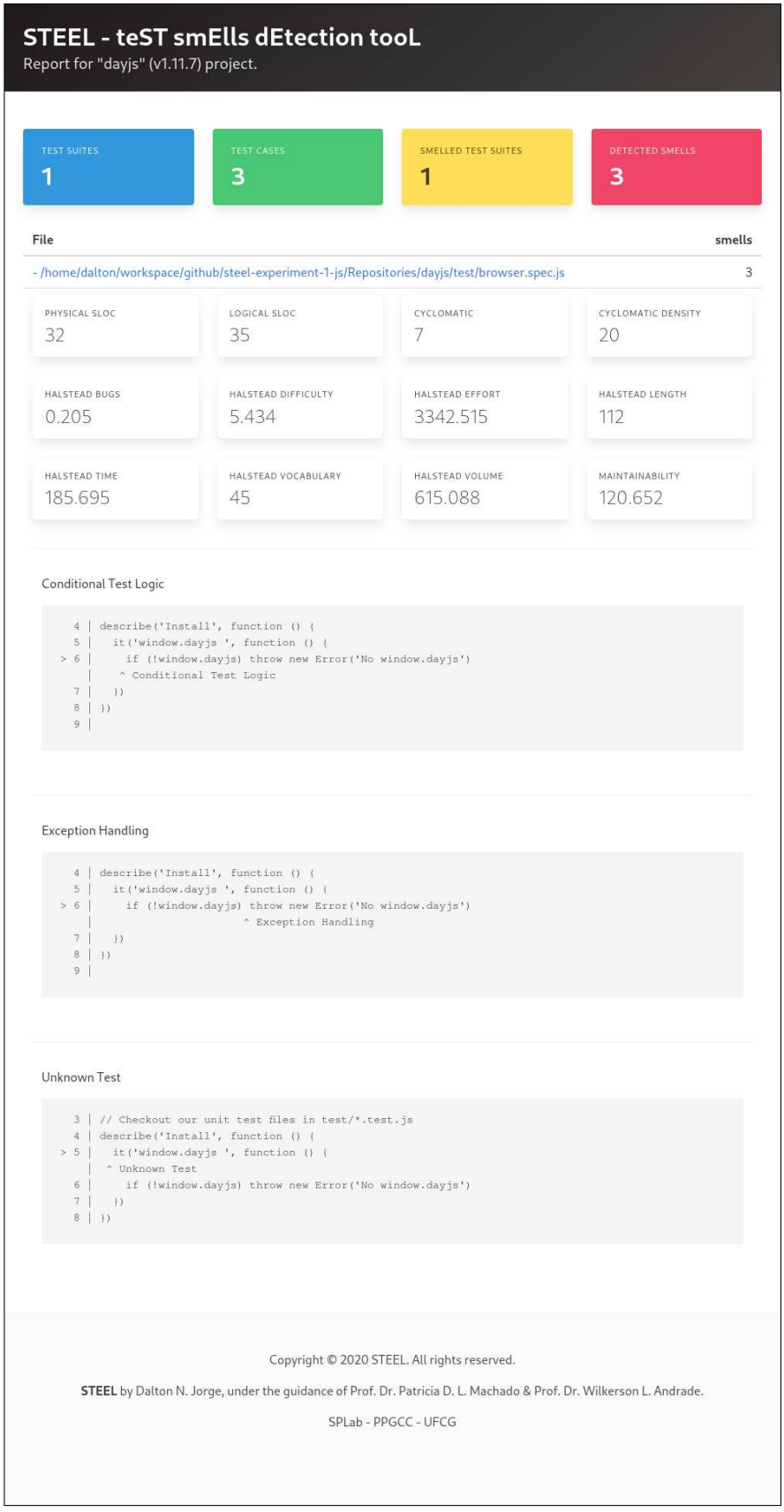


Figura 4.5: Exemplo de saída em HTML

Capítulo 5

Investigando *Test Smells* em código de teste *JavaScript*

Este capítulo apresenta uma ampliação do estudo exploratório publicado por Jorge et al. sobre a incidência de *Test Smells* na linguagem de programação *JavaScript*. Foram analisadas as inter-relações identificadas entre os *smells* e a conexão desses com as métricas de qualidade de *software* mais relevantes. Por meio da utilização de uma ferramenta especialmente concebida para esse propósito, conduziu-se uma análise estática detalhada em um conjunto de 65 projetos de código-fonte em *JavaScript*, abrangendo tanto o domínio público quanto o privado. Além disso, foram coletadas métricas de qualidade de *software* relacionadas à complexidade e manutenibilidade. Os resultados mostraram que a presença de *Test Smells* está correlacionada com métricas de qualidade de *software*, indicando que a detecção e correção desses padrões de código defeituosos podem contribuir para melhorar a qualidade dos testes e, consequentemente, a qualidade do *software* em geral.

Estruturamos o capítulo da seguinte forma: a Seção 5.1 introduz o estudo empírico que realizamos para investigar a ocorrência de *test smells* e sua associação com métricas de qualidade de código. A Seção 5.2 expõe o delineamento do estudo, onde apresentamos o escopo na Subseção 5.2.1, o objetivo na Subseção 5.2.2 e as questões de pesquisa na Subseção 5.2.3. A Seção 5.3 apresenta a metodologia do estudo, onde detalhamos a amostra na Subseção 5.3.1, os instrumentos de coleta na Subseção 5.3.2, os procedimentos na Subseção 5.3.3, a qualidade dos dados na Subseção 5.3.4 e a análise

dos dados na Subseção 5.3.5. A Seção 5.4 apresenta os resultados obtidos no estudo, respondendo às questões de pesquisa (Subseções 5.4.2, 5.4.3 e 5.4.4) e apresentando as ameaças à validade (Seção 5.4.5). Por fim, a Seção 5.5 discute as implicações dos resultados.

5.1 Contextualização

A prática de teste de *software* desempenha um papel crucial na garantia da qualidade do *software*. No entanto, os casos de teste podem conter problemas de *design* e implementação que afetam negativamente sua legibilidade, manutenibilidade e eficácia dos resultados. Esses problemas são comumente conhecidos como *Test Smells*. Os *Test Smells* são indicadores de possíveis deficiências nos casos de teste, semelhantes aos *Code Smells* encontrados no código-fonte do *software*. Identificar e avaliar os resultados são passos importantes para a tomada de decisão: caso os indícios apontem possíveis problemas nas suítes de testes, a remoção desses *Test Smells* é o passo seguinte em direção a uma melhoria na qualidade dos testes.

Embora muitos estudos tenham abordado a detecção e a correção de *Test Smells* em diferentes linguagens de programação, ainda há uma lacuna no conhecimento sobre a incidência desses problemas na linguagem *JavaScript* e a sua relação com métricas de qualidade de *software*, bem como ferramentas específicas para detecção de *test smells* [50]. JavaScript é uma das linguagens de programação mais amplamente utilizadas na indústria de desenvolvimento de *software*. Por isso, entender como os *Test Smells* afetam a qualidade dos testes em projetos *JavaScript* pode fornecer percepções valiosas para melhorar as práticas de teste e a qualidade do *software* em geral.

5.2 Escopo, Objetivo Geral e Questões de Pesquisa

Nesta seção, abordaremos o escopo do estudo, o objetivo geral que orienta a pesquisa e as questões de pesquisa a serem respondidas ao longo do estudo.

5.2.1 Escopo

O escopo deste estudo concentra-se na análise da incidência de *test smells* na linguagem de programação *JavaScript* e sua relação com métricas de qualidade de *software*. Serão considerados projetos JavaScript de diferentes domínios de aplicação, com o intuito de obter uma visão ampla e representativa dos desafios enfrentados na detecção e mitigação dos *Test Smells*. Utilizaremos as asserções nativas do *Node.js*, os *frameworks Mocha* e *Jest*, a biblioteca de asserções *Chai*, além das bibliotecas de dublês para testes *Sinon* e *Nock*.

É importante destacar que, embora estejamos focando apenas em testes unitários, nos projetos privados os testes de API foram inclusos. Isso ocorre porque, na prática, a distinção entre esses tipos de teste é raramente seguida e há indícios de que essa distinção não é significativa. Além disso, nossa pesquisa investigará: i) as relações entre os problemas de qualidade identificados nos testes e ii) a relação desses problemas de qualidade com as métricas extraídas dos casos de teste.

5.2.2 Objetivo Geral

O objetivo geral deste estudo é investigar a incidência e a relação entre os *test smells* e desses com as métricas de qualidade de *software* em projetos *JavaScript*. Busca-se compreender como a presença de *test smells* pode afetar a qualidade do código e identificar padrões e tendências que influenciem a eficácia das práticas de teste de *software* nesse contexto.

5.2.3 Questões de Pesquisa

Com base no objetivo geral, as seguintes questões de pesquisa foram formuladas:

QP1 Quais são os *test smells* mais frequentemente detectados em projetos *JavaScript*?
Com esta pergunta, cogitamos saber quais são os *test smells* mais frequentes na amostra selecionada, contabilizando suas ocorrências. Além disso, esta pergunta serve como um indicador de que o Steel cumpre seu papel como ferramenta de detecção de *test smells*.

QP2 Quais as associações significativas entre os *test smells* que foram detectados? Nesta questão, nosso objetivo é descobrir se os *test smells* fornecidos ocorrerão provavelmente juntos.

QP3 Quais as associações significativas entre os *test smells* detectados e métricas de qualidade de *software*? Esta última questão de pesquisa visa descobrir se a presença de certos *test smells* está relacionada a indicadores clássicos de *design* ruim no código de teste.

Para responder ao QP1, verificamos os resultados coletados para revelar quais *test smells* aparecem com mais frequência com suítes de testes de todos os projetos. Uma vez identificada e classificada a ocorrência dos *smells*, para responder o QP2, analisamos a relação entre os *test smells* usando o coeficiente de correlação de *Spearman*. Da mesma forma, para responder ao QP3, avaliamos a interdependência de *test smells* e métricas de qualidade de código usando o coeficiente de correlação de classificação de *Spearman*.

Essas questões de pesquisa orientaram a coleta de dados, a análise estatística e a interpretação dos resultados, permitindo compreender a relação entre *Test Smells* e qualidade de *software* em projetos *JavaScript*.

5.3 Metodologia

Nesta seção, apresentaremos a metodologia adotada neste estudo exploratório para investigar a relação entre os *Test Smells* e as métricas de qualidade de *software* em projetos *JavaScript*. Descreveremos detalhadamente os procedimentos utilizados para coleta de dados, a análise estatística realizada e as ferramentas específicas empregadas. A metodologia adotada viabiliza uma compreensão aprofundada e embasada sobre o impacto dos *Test Smells* na qualidade do código em projetos *JavaScript*, fornecendo percepções relevantes para a área de teste de *software* e contribuindo para o avanço do conhecimento nesse campo.

5.3.1 Amostra

Para a realização do presente estudo, optou-se pela utilização de uma amostragem não-probabilística, o que significa que a seleção dos elementos da amostra não foi baseada em uma distribuição randômica, resultando em diferentes chances de seleção para cada indivíduo. Nesse contexto, para a seleção dos projetos de código aberto, empregou-se a técnica de amostragem por julgamento, que se justifica pelo interesse específico em obter uma amostra que atendesse rigorosamente aos critérios estabelecidos para a inclusão. Dessa forma, a busca por tais projetos de código aberto foi realizada por meio da plataforma GitHub[1], na qual aplicamos um conjunto de critérios estabelecidos previamente para garantir a pertinência e adequação dos projetos selecionados para o escopo de nossa investigação. Os critérios foram:

- Linguagem: JavaScript;
- Popularidade: selecionamos projetos que possuíam um número significativo de estrelas (> 5000), indicando interesse da comunidade.
- Atividade contínua: não estar arquivado e possuir pelo menos um *commit* nos últimos 12 meses, indicando que o projeto está ativamente mantido.

A busca inicial resultou em um conjunto aproximado de 1211 projetos de *software* ordenados por popularidade (número de estrelas). Em seguida, procedeu-se à análise manual minuciosa desses repositórios, a fim de verificar a correta utilização de bibliotecas de teste e a existência de suítes de testes automatizados. Para tal avaliação, adotaram-se os seguintes critérios estabelecidos previamente:

- Possuir o arquivo "package.json", comum em projetos NodeJS e é responsável pela declaração das bibliotecas utilizadas;
- Adotar bibliotecas de testes mais populares e amplamente utilizadas na comunidade de desenvolvimento de software (como Jest, Mocha/Chai, Sinon e módulo Assert do NodeJS);
- Possuir suítes com testes unitários.

[1] `https://github.com`

Após uma análise minuciosa e abrangente dos repositórios, decidimos selecionar 61 projetos que atenderam plenamente aos critérios estabelecidos para compor a amostra final, rejeitando assim os outros repositórios que não atenderam a esses requisitos. A Tabela 5.1 apresenta a amostra contendo os projetos de código aberto.

Adicionalmente aos projetos de código aberto, incorporamos à presente pesquisa 4 projetos privados resultantes de uma colaboração entre a indústria e a Academia, na qual o pesquisador deste estudo se encontra envolvido. Tais projetos, nomeados "*eco-control*", "*mup*", "*soft-control*", e "*tcomserver*, destinam-se ao contexto de soluções para terminais de pagamentos, tendo sido desenvolvidos em linguagem *JavaScript*, empregam o arcabouço de testes *Jest* e apresentam suítes de testes automatizados. Da mesma maneira que os projetos de código aberto, a seleção dos projetos privados foi realizada por meio de uma abordagem não-probabilística, adotando-se a técnica de amostragem por conveniência a partir do ecossistema de projetos da empresa.

É imprescindível ressaltar que a incorporação desses projetos privados acrescenta ao escopo do estudo uma amostra diversificada de aplicações em *JavaScript*, atingindo uma variedade de domínios e níveis de complexidade, contribuindo significativamente para a ampliação da abrangência e representatividade dos dados analisados.

5.3.2 Instrumento de Coleta de Dados

Para a coleta de dados, desenvolvemos a ferramenta de análise estática chamada STEEL , que adota regras específicas para identificar possíveis problemas de design e implementação nos casos de teste. Essa ferramenta personalizada permitiu a detecção de *test smells* em projetos *JavaScript*, o que não seria possível com ferramentas genéricas de análise estática.

Durante a varredura realizada pela ferramenta STEEL , os *test smells* identificados foram registrados e contabilizados para análise posterior. Adicionalmente, os resultados obtidos foram validados por meio de revisão manual de uma amostra aleatória de testes, garantindo a precisão das detecções realizadas pela ferramenta. Além da identificação de *test smells*, a ferramenta STEEL também foi utilizada para coletar métricas de qualidade de software para a linguagem *JavaScript* durante a varredura dos casos de teste. Um módulo especializado foi adicionado à ferramenta para realizar essa coleta

automatizada.

Para avaliar a complexidade do código presente nos casos de teste, calculamos a complexidade ciclomática para cada um deles, fornecendo uma medida quantitativa da complexidade do código. Além disso, adotamos as métricas de *Halstead*, que visam avaliar quantitativamente a complexidade do código por meio dos operadores e operandos utilizados na linguagem *JavaScript*.

A manutenibilidade foi calculada a partir da métrica de volume de *Halstead* e da complexidade ciclomática. Essas métricas foram escolhidas com base na validação pela literatura especializada como indicadores de qualidade de código. Além disso, a escolha considerou a possibilidade de coletá-las de forma automatizada, facilitando o processo de análise.

A utilização da ferramenta STEEL e das métricas selecionadas permitiu uma abordagem abrangente na coleta de dados, possibilitando a identificação de *test smells* e a obtenção de métricas objetivas de qualidade de software para análise estatística e comparação entre os projetos estudados.

5.3.3 Procedimentos de Coleta de Dados

O processo de coleta de dados foi automatizado por meio de um programa desenvolvido em *JavaScript*, com objetivo de simplificar as etapas de clonagem dos projetos selecionados e execução da ferramenta de análise estática em cada um dos repositórios clonados.

O programa foi projetado para permitir a clonagem dos projetos a partir de uma versão específica de cada repositório. Essa abordagem garantiu que os dados coletados fossem consistentes e baseados em uma versão controlada dos projetos, evitando possíveis alterações que poderiam ocorrer em versões mais recentes.

Após a clonagem dos projetos, o programa executou a ferramenta de análise estática, no caso a ferramenta STEEL mencionada anteriormente, em cada um dos repositórios clonados. A ferramenta foi configurada para realizar a varredura e coletar informações relevantes sobre *test smells* e métricas de qualidade de software.

A automação dessas etapas permitiu uma coleta eficiente e precisa dos dados necessários para o estudo. Além disso, a utilização de um programa desenvolvido especi-

ficamente para esse fim em JavaScript garantiu a compatibilidade com a linguagem de programação dos projetos analisados, facilitando a análise e obtenção dos resultados desejados.

5.3.4 Qualidade dos Dados

A qualidade dos dados utilizados neste estudo foi avaliada com base em critérios de completude, consistência, precisão e a presença de valores ausentes ou *outliers*.

Para garantir a completude dos dados, foram adotados procedimentos rigorosos durante a coleta. O protocolo estabelecido incluiu a clonagem de cada versão específica dos repositórios selecionados consistentemente. Em relação à detecção dos *test smells*, os resultados apontam a localização (linha, coluna) de cada incidência no arquivo de teste e seu respectivo trecho de código para avaliação. Além disso, utilizamos uma biblioteca confiável para coletar todas as métricas de qualidade de software. Assim garantimos que todas as informações relevantes de cada projeto fossem capturadas.

No que diz respeito à consistência dos dados, foram adotados procedimentos padronizados para a coleta e análise. Isso incluiu a utilização de um programa desenvolvido em *JavaScript* para automatizar as etapas de clonagem e varredura dos projetos, garantindo a uniformidade e a comparabilidade dos dados obtidos.

Quanto à precisão dos dados, foram realizadas revisões manuais para validar a detecção dos *Test Smells* e minimizar possíveis falsos positivos ou falsos negativos. Essa revisão permitiu verificar a consistência dos dados coletados e assegurar que eles refletissem com precisão a presença e a natureza dos *Test Smells* nos projetos analisados.

Em relação à presença de valores ausentes, a ferramenta de análise estática faz o tratamento necessário para preenchê-los apropriadamente (valores zero). Os *outliers* foram considerados na análise com o intuito de capturar quaisquer nuances, dado a diversidade dos projetos.

5.3.5 Análise de Dados

Os dados coletados, tanto em relação à presença de *test smells* quanto às métricas de qualidade de software, foram submetidos a uma análise estatística abrangente. Foram

calculadas medidas descritivas, como média, mediana, desvio padrão e distribuições, a fim de compreender a natureza e a dispersão dos dados.

A análise estatística envolveu a avaliação das incidências de *test smells* e das métricas de qualidade de *software*. Para as incidências de *test smells*, foram calculadas as medidas descritivas mencionadas, permitindo uma visão geral da ocorrência desses problemas nos projetos estudados. Da mesma forma, as métricas de qualidade também foram analisadas estatisticamente, proporcionando uma compreensão detalhada da complexidade do código e de outros aspectos relevantes.

Além disso, foram conduzidas análises de correlação entre a presença de *test smells* e as métricas de qualidade de *software*. Dada a natureza não normal dos resultados, optou-se por utilizar o teste de correlação de *Spearman* como técnica estatística adequada. Essa abordagem permitiu identificar associações e relações entre os *test smells* e as métricas de qualidade, fornecendo informações valiosas sobre como esses fatores podem influenciar a qualidade do código.

Essa análise estatística e de correlações foi realizada considerando todo o conjunto de projetos estudados. O objetivo principal foi identificar padrões e tendências gerais que pudessem fornecer uma visão ampla sobre a presença de *test smells* e as métricas de qualidade de *software* nos projetos analisados. Essa compreensão contribuiu para a obtenção de conclusões significativas e embasadas sobre a relação entre esses aspectos.

5.4 Resultados

Esta seção apresenta os resultados obtidos pela análise dos dados obtidos no estudo exploratório. Começamos explorando o conjunto de dados por meio de uma análise descritiva. Em seguida, examinamos a distribuição dos *test smells* para identificar aqueles que são mais frequentemente detectados em projetos, o que nos dá uma indicação da prevalência de certos desafios durante a codificação. A seguir, exploramos as associações entre diferentes *test smells* para determinar se a presença de um pode sugerir a presença de outros, revelando assim interdependências potenciais que podem surgir durante o desenvolvimento. Por fim, discorremos as correlações entre *test smells* e as métricas de qualidade para determinar se a presença de certos *test smells* está

intrinsecamente relacionada a indicadores clássicos de qualidade no código de teste.

5.4.1 Análise Descritiva dos Dados

A tabela 5.2 apresenta as médias, desvios-padrão, quartis, valores máximos e mínimos dos totais de: 1) suítes de testes, 2) casos de testes, 3) suítes de testes que ocorreram incidência de smells, e 4) *test smells*. Os valores do desvio padrão foram superiores às médias, o que indica que os coeficientes de variação foram superiores a 100%, corroborando a grande dispersão dos dados entre os projetos da amostra. Os valores mínimo e máximo de *test smells* em um projeto foram respectivamente 3 (Day.js) e 12627 (Mongoose) ocorrências. O número de casos de testes variaram de 3 (Day.js) a 3898 (Math.js), o que significa que a amostra abrange desde projetos com poucas unidades até milhares de casos de testes.

A tabela 5.3 apresenta a estatística descritiva dos dados de *Test Smells*. O padrão de uma alta dispersão dos dados se repete aqui, com os valores do desvio padrão superando as médias. Em um *ranking* dos 3 maiores números de incidências de *test smells*, tem-se em primeiro o *Assertion Roulette* com 6576 ocorrências, em segundo o *Global Variable* que apresenta 5614 casos e em terceiro, o *Duplicate Assert* com 3510 casos. As colunas de assimetria (*skew*) e curtose (*kurtosis*) demonstram que a distribuição dos *test smells* é assimétrica com cauda longa à direita e que a maior concentração de dados é de valores próximos a zero. Isso é coerente com os valores de *p-value* encontrados na coluna *shapiro*, o que, por conseguinte, rejeita a hipótese de uma distribuição normal dos dados (*p-value* < 0.05).

5.4.2 Distribuição dos *Test Smells*

Ao avaliar as práticas de teste em projetos JavaScript, é essencial entender quais *test smells* são mais comuns e como eles se manifestam nos projetos. Com isso em mente, a Tabela 5.4 nos ajuda a responder à questão de pesquisa QP1: "Quais são os *test smells* mais frequentemente detectados em projetos *JavaScript*?". Esta pergunta não só visa identificar os *test smells* mais recorrentes em nossa amostra selecionada, mas também avaliar a eficácia da ferramenta de análise estática STEEL .

Test Smells Mais Frequentes:

- *Conditional Test*: Ocorre em 92.31% dos projetos, com um total de 4.960 ocorrências.
- *Duplicate Assert*: Presente em 76.92% dos projetos, totalizando 11.298 ocorrências.
- *Unknown Test*: Detectado em 72.31% dos projetos, com 4.785 ocorrências.
- *Exception Test*: Manifesta-se em 67.69% dos projetos, com 1.809 ocorrências.
- *Magic Number*: Encontrado em 61.54% dos projetos, acumulando 10.015 ocorrências.

Test Smells Menos Frequentes:

- *Empty Test*: Aparece em apenas 6.15% dos projetos, com 14 ocorrências.
- *Redudant Optimism*: Presente em 9.23% dos projetos, totalizando 68 ocorrências.

A observação direta entre a frequência de um *test smell* em projetos e seu total de ocorrências sugere que alguns *test smells*, embora frequentes, podem não ser tão prevalentes em termos de ocorrências totais. Por exemplo, *Conditional Test* é o *test smell* mais comum em termos de frequência, mas *Global Variable*, que ocorre em 29.23% dos projetos, tem o maior número de ocorrências, com 15.188. A prevalência de *smells* do tipo *Global Variable* pode ser explicada pelo fato dos projetos nos quais foram detectados, serem projetos criados anteriormente ao lançamento da especificação *ECMAScript 2015* que trouxe as declarações *let* e *const* como soluções melhores para tratar o escopo de variáveis. Também pode-se perceber que mesmo com os novos recursos para controle de escopo de variáveis, os testes continuaram com a forma antiga de declaração de variáveis.

A análise da distribuição dos *test smells* fornece percepções valiosas sobre os desafios comuns enfrentados em testes de projetos *JavaScript*. Além disso, a capacidade da ferramenta STEEL de identificar uma variedade de *test smells* em uma grande proporção de projetos confirma sua eficácia como ferramenta de análise estática.

5.4.3 Associação entre *Test Smells*

Nesta seção, abordamos a questão de pesquisa: "Quais as associações significativas entre os *test smells* que foram detectados?". Nosso objetivo é entender se a presença de um *test smell* específico em um projeto está potencialmente ligada à presença de outros *test smells*, indicando que certos *test smells* podem ocorrer em conjunto.

Para investigar essa questão, utilizamos a análise de correlação com o coeficiente de *Spearman* (ρ), identificando associações relevantes entre os *test smells*. A Figura 5.1 apresenta os principais resultados:

Eager Test e *Lazy Test*: Com uma correlação de 0.830, esta é a associação mais forte observada. Isso sugere que os testes que tentam fazer muito (*eager*) também podem conter partes que não exercitam testes o suficiente (*lazy*).

Conditional Test tem correlações significativas com vários outros *test smells*:

- Com *Unknown Test*: 0.639
- Com *Exception Test*: 0.616
- Com *Redudant Assertion*: 0.566
- Com *Assertion Roulette*: 0.466
- Com *Duplicate Assert*: 0.465

Estas correlações indicam que a presença de lógica condicional em testes está frequentemente associada a vários outros *test smells*, desde tratamento inadequado de exceções até afirmações redundantes.

Exception Test e *Unknown Test*: Com uma correlação de 0.504, sugere-se que os testes com um tratamento de exceção inadequado também podem conter partes do teste que não são claras ou reconhecíveis.

Assertion Roulette e *Global Variable*: Uma correlação de 0.486 indica que testes com várias afirmações sem mensagens descritivas também podem depender de variáveis globais.

Ignored Test e *Unknown Test*: Com uma correlação de 0.480, isso sugere que os testes que são frequentemente ignorados ou desativados também podem conter trechos de código não reconhecíveis ou não documentados.

Duplicate Assert e *Magic Number*: Uma correlação de 0.459 indica que testes com afirmações repetidas podem também conter números mágicos, tornando-os menos claros e possivelmente mais difíceis de manter.

A análise revela associações claras entre vários *test smells*, o que indica que a presença de um pode aumentar a probabilidade de que outro esteja presente. Estas associações oferecem percepções sobre os desafios inter-relacionados que os desenvolvedores enfrentam ao escrever testes e destacam áreas para foco e refatoração.

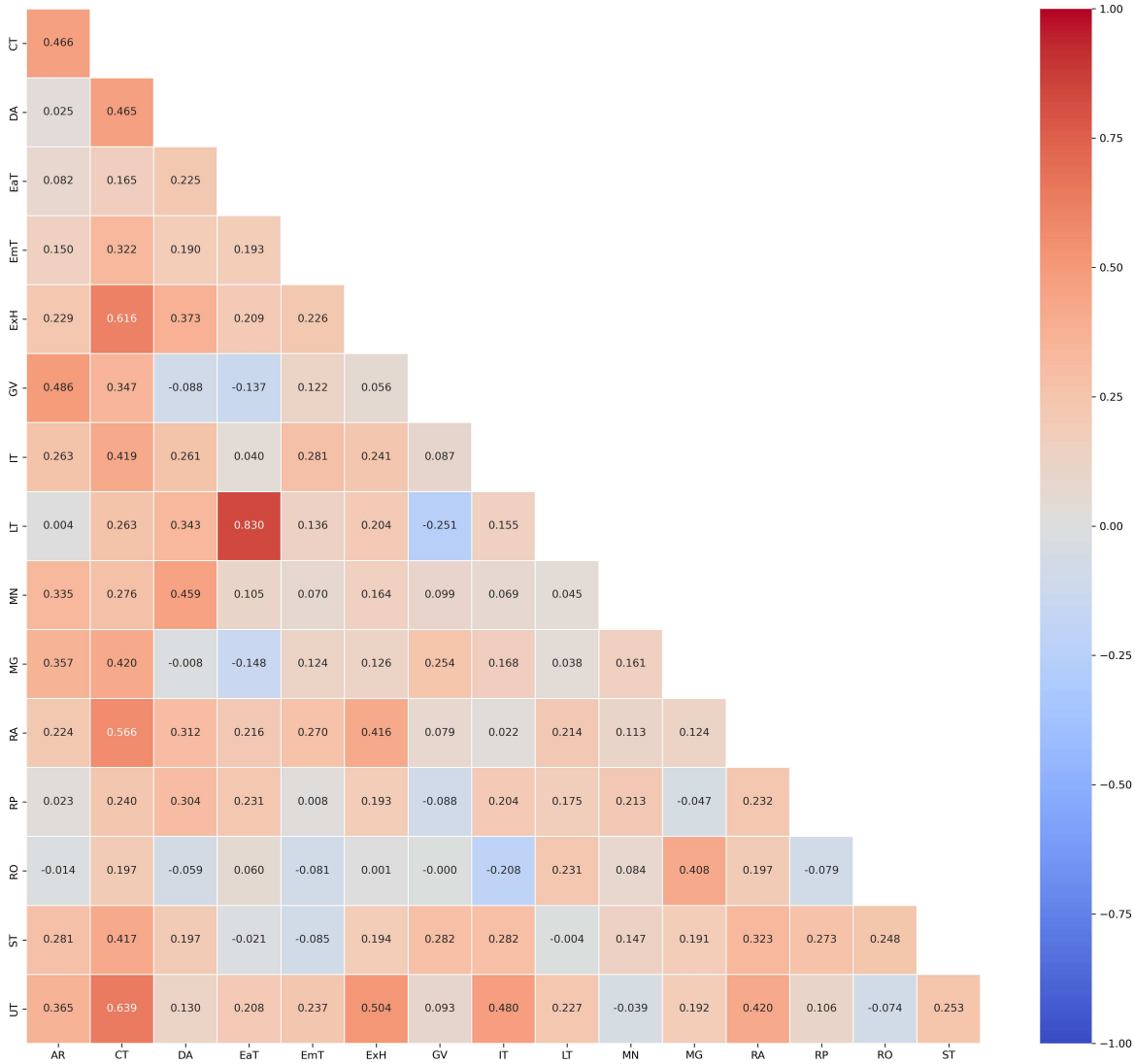


Figura 5.1: Correlação de Spearman entre Tipos de Test Smells

5.4.4 Associação entre *Test Smells* e Métricas de Qualidade de Código

Nesta seção, abordamos a questão de pesquisa QP3: "Quais as associações significativas entre os *test smells* detectados e métricas de qualidade de software?"O objetivo desta investigação é determinar se a presença de certos *test smells* está intrinsecamente relacionada a indicadores clássicos de design ruim no código de teste.

Através do coeficiente de correlação de Spearman, identificamos associações significativas entre diversos *test smells* e métricas de qualidade de software. Em particular, o *test smell Conditional Test* exibiu fortes correlações positivas com todas as métricas de qualidade, tendo os coeficientes entre 0.694(*Maintainability*) e 0.813 (*Cyclomatic Complexity*). Estas correlações são estatisticamente significativas com valores-p bem abaixo do nível de significância de 0,05.

Além disso, o *test smell Duplicate Assert* também mostrou correlações significativas com métricas como *Halstead Effort* e *Halstead Time*, com coeficientes de 0.670 para ambas.

Em resumo, a resposta para a questão de pesquisa é "há indícios de associação significativa entre os *test smells* detectados e métricas de qualidade de *software*, mesmo sem podermos afirmar a causalidade". Em particular, os *test smells Conditional Test* e *Duplicate Assert* estão fortemente associados a índices ruins das métricas clássicas de qualidade de *software*.

5.4.5 Ameaças

Embora nossos resultados sejam reveladores, é essencial reconhecer as limitações deste estudo. Questões pertinentes à validade externa referem-se à capacidade de generalização dos resultados obtidos no estudo em questão. Foram selecionados 65 projetos desenvolvidos em JavaScript e, devido às limitações de tempo inerentes ao estudo, usamos um método de amostragem não-probabilística, o que pode prejudicar os resultados do estudo, uma vez que a amostra pode não ser representativa da população-alvo. A seleção foi realizada criteriosamente visando contemplar projetos de diferentes contextos e em constante evolução, que utilizam as bibliotecas de teste em JavaScript de maior

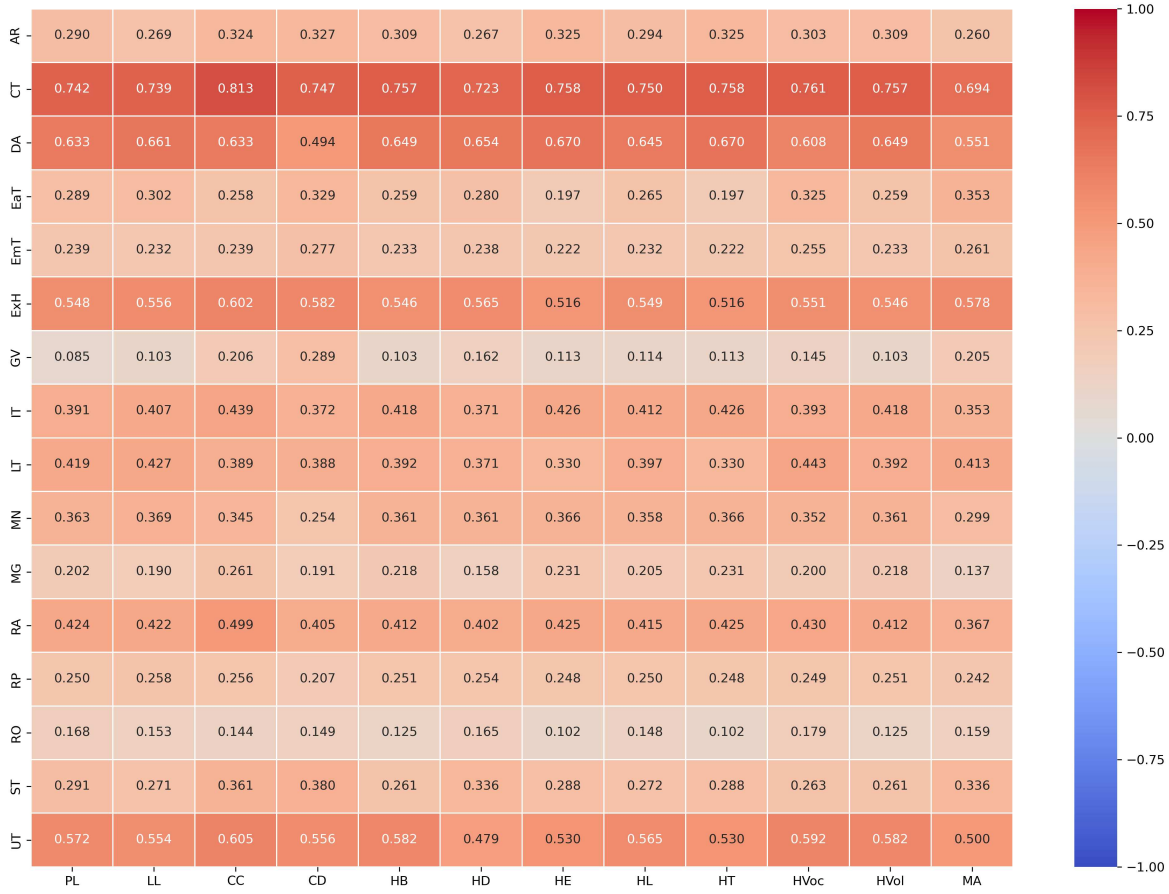


Figura 5.2: Correlação de Spearman entre Test Smells e Métricas de Qualidade

relevância no cenário atual. Analogamente, a ferramenta de detecção de *test smells* desenvolvido detecta apenas 16 tipos de *smells* em projetos que utilizam bibliotecas de teste específicas (módulo de asserções integrado do *Node.js*, *Mocha/Chai*, *Jest*, *Sinon* e *Nock*). No entanto, é importante salientar que esta ferramenta tem potencial para expandir por meio de um mecanismo de *plugins*, podendo ser adaptada para identificar novos *test smells* e integrar-se a outras bibliotecas de assertividade.

As ameaças à validade da conclusão estão relacionadas às limitações ou fraquezas nas conclusões advindas da análise de um estudo. A metodologia adotada neste estudo recorreu ao coeficiente de correlação de *Spearman* para analisar a relação entre os *test smells* e as métricas de qualidade. É importante salientar que o poder estatístico das análises realizadas $(1 - \beta)$ tem potencial de otimização se for ampliado o conjunto da amostra.

Por fim, em relação à validade interna, esta refere-se a variáveis não controladas

ou instrumentação e ferramental inadequados ou não-confiáveis que podem influenciar os resultados. A análise dos resultados gerados pela ferramenta de detecção é realizada manualmente, sendo, portanto, suscetível a erros humanos. Este potencial fonte de erro foi mitigado durante a fase de desenvolvimento do instrumento, por meio da aplicação de testes unitários destinados a avaliar a detecção de cada categoria de *test smell*. É importante salientar que estes testes concentram-se exclusivamente nos métodos de asserção dos *frameworks* de teste em estudo. Contudo, a possibilidade de incidências falso-positivas não pode ser descartada, especialmente em cenários que envolvem asserções não convencionais.

5.5 Discussão

A análise dos *test smells* em projetos *JavaScript* oferece uma perspectiva detalhada sobre os padrões de codificação e as áreas problemáticas comuns na comunidade de desenvolvimento. As questões de pesquisa abordadas neste estudo lançam luz sobre aspectos fundamentais da qualidade do código e das práticas de teste. Esta seção discute as implicações, nuances e possíveis direções futuras com base nos resultados obtidos.

5.5.1 Frequência de Test Smells nos Projetos

Ao abordar a primeira questão de pesquisa sobre a frequência de *test smells*, observamos que certos *test smells*, como *Conditional Test*, *Global Variable* e *Duplicate Assert*, são particularmente frequentes nos projetos analisados. Esta frequência indica tendências em práticas de codificação que, apesar de amplamente adotadas, podem não ser consideradas as mais eficientes em um contexto de desempenho dos testes. No entanto, a identificação destes *test smells* em uma proporção tão significativa de projetos também valida a ferramenta STEEL . O fato de STEEL ter identificado uma variedade tão ampla de *test smells*, desde os mais comuns até os menos frequentes, indica sua robustez e capacidade de fornecer entendimento detalhado sobre a qualidade do código de teste.

5.5.2 Associações entre Test Smells

A segunda questão de pesquisa explora a associação entre diferentes *test smells*. As correlações significativas observadas entre certos *test smells* fornecem percepções sobre possíveis interdependências na codificação. Por exemplo, a forte associação entre *Eager Test* e *Lazy Test* pode sugerir que ao tentar resolver um *test smell*, os desenvolvedores inadvertidamente introduzem outro. Este entendimento é crucial, já que destaca a complexidade das práticas de codificação e a necessidade de abordagens holísticas de refatoração.

5.5.3 Associação entre Test Smells e Métricas de Qualidade do Código

A correlação entre o *test smell Conditional Test* e métricas como *Cyclomatic Complexity* e *Halstead Vocabulary* sugere uma relação profunda entre a estrutura dos testes e a complexidade inerente do código. Esta observação está consoante as literaturas anteriores que sustentam que a complexidade do código pode se refletir não apenas nas métricas tradicionais, mas também nas características e padrões dos testes associados.

A prevalência de *test smells* como *Global Variable* e *Duplicate Assert* tem implicações diretas para a prática de desenvolvimento de *software*. A presença destes *test smells* pode ser interpretada como um sinal de alerta para equipes de desenvolvimento, indicando áreas de código que podem se beneficiar de refatoração ou revisão. Por exemplo, variáveis globais em testes podem levar a fragilidade do código e estados imprevisíveis, enquanto asserções duplicadas podem indicar redundância ou falta de modularidade.

Nossa análise reforça a ideia de que *test smells* são não apenas indicativos de práticas de teste inadequadas, mas também de *design* de código potencialmente ruim. O fato de que *test smells* específicos estão fortemente correlacionados com métricas que indicam complexidade ou baixa manutenibilidade sugere que a qualidade do *design* do código de teste está intrinsecamente ligada à qualidade do código de produção.

Tabela 5.1: Projetos de código aberto

Projeto	Descrição
apidoc	Gera documentação de API a partir de comentários no código fonte
asyncc	Biblioteca para lidar com operações assíncronas em JavaScript
axios	Cliente HTTP baseado em promises para Node.js e browser
babel	Compilador que transpila código JavaScript moderno para versões antigas do JS
bull	Biblioteca para gerenciar filas de trabalho e jobs assíncronos
clipboard.js	Biblioteca para copiar e colar texto na área de transferência
collect.js	Biblioteca de funções utilitárias para arrays e objetos
commander.js	Biblioteca para criar linhas de comando interativas
dayjs	Biblioteca para parser, validar, manipular e formatar datas
docsify	Gerador de sites de documentação a partir de arquivos Markdown
flux	Arquitetura de aplicações web para implementar data flow unidirecional
Ghost	Plataforma de blogs open source
grapesjs	Construtor visual de templates para sites e apps
habitica	App gamificado de produtividade e tarefas
hexo	Framework para criar blogs estáticos rapidamente
highlight.js	Biblioteca de syntax highlighting para diversas linguagens
hubot	Framework para criar bots de chat simples para vários chat ops
Inquirer.js	Biblioteca para criar interfaces de linha de comando interativas
jsdom	Implementação em JS do que um browser faz, para testes
json-server	Cria uma fake REST API a partir de um arquivo JSON
KaTeX	Gerador rápido de fórmulas LaTeX para web
keeweb	Aplicativo web para gerenciar senhas
knex	Query builder para SQL em Node.js
koa	Framework web minimalista para Node.js
lerna	Gerenciador de monorepos para projetos JavaScript
luxon	Biblioteca para trabalhar com datas e horas
mathjs	Biblioteca extensa de matemática para JavaScript
mongoose	Modelagem de objetos do MongoDB para Node.js
mustache.js	Implementação da linguagem de templates Mustache
next.js	Framework React para renderização no lado do servidor
nock	Simulador de HTTP requests para testes em Node.js
nodemailer	Módulo para enviar emails em Node.js
nodemon	Monitora alterações e reinicia app Node.js automaticamente
nuxt.js	Framework Vue.js para renderização no lado do servidor e mais
p5.js	Biblioteca de JavaScript para criação de arte e design gerativo
parcel	Empacotador de assets e bundler de aplicações web
passport	Autenticação middleware para Node.js
pm2	Gerenciador de processos e cluster para Node.js
pug	Template engine de alta performance para Node.js
ramda	Biblioteca de programação funcional para JavaScript
react	Biblioteca para criar interfaces de usuário
riot	Biblioteca React-like pequena e rápida
sails	Framework MVC para aplicações Node.js
sequelize	ORM para Node.js que suporta PostgreSQL, MySQL e mais
serverless	Framework para desenvolver e deployar apps serverless
sharp	Biblioteca para processamento de imagens em Node.js
shepherd	GUIa e orienta usuários através de tours interativos
shields	Gera badges para projetos open source
strapi	Framework Node.js headless CMS para APIs rápidas
svgo	Otimizador de SVG via Node.js
ungit	Interface gráfica em browser para repositórios Git
validator.js	Biblioteca de validação de strings
vue-cli	Interface de linha de comando para inicializar projetos Vue.js
vue-router	Roteamento oficial para aplicativos Vue.js
vuex	Gerenciador de estado centralizado para Vue.js
webpack	Empacotador de módulos e assets para aplicações web
winston	Logger para Node.js
ws	Biblioteca para websockets
zotero	Gerenciador de referências open source

Tabela 5.2: Estatística Descritiva das Totalizações de Incidências

	Média	Desvio	Mín	25%	50%	75%	Máx
Suítes de testes	47.48	68.41	1	8	20	59	335
Suítes com smells	31.28	46.84	1	5	14	40	259
Casos de testes	483.09	711.85	3	73	201	592	3,898
Test Smells	975.02	2,208.90	3	78	261	758	12,627

Tabela 5.3: Estatísticas Descritiva, Distribuições e Normalidades dos Test Smells

	média	std_dev	min	25%	50%	75%	max	skew	kurtosis	shapiro
assertionRoulette	154.97	835.54	0	0	0	12	6,576	7.39	56.85	3.04919E-17
conditionalTestLogic	76.31	158.38	0	4	19	62	889	3.49	13.51	2.76553E-13
duplicateAssert	173.82	536.07	0	2	39	117	3,510	5.30	29.29	8.87796E-16
eagerTest	27.06	131.63	0	0	0	2	1,040	7.38	57.07	5.00658E-17
emptyTest	0.22	1.17	0	0	0	0	9	7.01	52.37	3.40667E-17
exceptionHandling	27.83	66.89	0	0	3	21	463	4.76	28.25	3.32072E-14
globalVariable	233.66	957.55	0	0	0	4	5,614	5.26	27.49	1.73519E-16
ignoredTest	3.26	12.92	0	0	0	1	95	6.10	41.28	2.40722E-16
lazyTest	28.55	72.82	0	0	0	18	361	3.52	12.92	3.29821E-14
magicNumberRule	154.08	395.38	0	0	8	39	1,828	3.13	9.28	2.41976E-14
mysteryGuest	4.42	10.62	0	0	0	3	57	3.29	11.62	9.75776E-14
redundantAssertion	4.95	22.33	0	0	0	1	175	7.18	54.71	7.93991E-17
redundantPrint	2.80	10.20	0	0	0	2	77	6.39	45.30	3.92898E-16
resourceOptimism	1.05	5.56	0	0	0	0	43	7.05	52.74	3.79313E-17
sleepyTest	8.43	27.39	0	0	0	5	208	6.38	45.50	7.74921E-16
unknownTest	73.62	161.11	0	0	12	62	1,009	3.90	18.37	1.66417E-13

Tabela 5.4: Totais de Incidências por Tipo de Test Smells

Test Smell	Total	Frequência
Conditional Test	4960	60
Duplicate Assert	11298	50
Unknown Test	4785	47
Exception Test	1809	44
Magic Number	10015	40
Sleepy Test	548	28
Lazy Test	1856	25
Assertion Roulette	10073	22
Mystery Guest	287	22
Ignored Test	212	20
Redudant Print	182	20
Global Variable	15188	19
Eager Test	1759	17
Redudant Assertion	322	17
Redudant Optimism	68	6
Empty Test	14	4

Capítulo 6

Estudo qualitativo com desenvolvedores e testadores da linguagem *JavaScript*

Neste capítulo apresentamos um estudo que investiga três aspectos cruciais relacionados aos *test smells* em projetos com a linguagem *JavaScript*: o impacto da experiência no reconhecimento desses *smells*, a prevalência de *test smells* específicos, e as estratégias e ferramentas utilizadas para sua mitigação. A primeira análise revela uma correlação fraca entre a experiência em desenvolvimento de software e a capacidade de identificar *test smells*, sugerindo que habilidades específicas em testes são necessárias além da experiência de desenvolvimento. A segunda parte do estudo destaca a frequente ocorrência de *test smells* como *"Magic Number Test"*, *"Exception Handling"* e *"Conditional Test Logic"*, apontando para a necessidade de práticas de teste mais rigorosas e bem estruturadas. Por fim, a pesquisa sobre estratégias e ferramentas de mitigação revela uma combinação de desafios e abordagens, incluindo a importância da padronização dos testes, treinamento em boas práticas e o uso de ferramentas de análise estática. Juntas, estas análises fornecem uma visão sobre as práticas atuais e as áreas que necessitam de melhorias nos testes de *software* em projetos *JavaScript*, destacando a importância de uma abordagem integrada que envolva educação, ferramentas adequadas e uma forte cultura de qualidade de teste.

6.1 Introdução

Neste estudo, examinamos três dimensões cruciais relativas à qualidade dos testes em projetos de *software* desenvolvidos com *JavaScript*, com ênfase especial na incidência e mitigação de *test smells*. Estes, definidos como padrões problemáticos em testes de software, têm o potencial de afetar adversamente a eficácia e a manutenibilidade das suítes de teste, comprometendo a qualidade geral do *software*.

A primeira questão de pesquisa aborda como a experiência dos desenvolvedores afeta o reconhecimento de *test smells*. Embora esta questão tenha sido amplamente estudada em contextos de outras linguagens, como *Java* — destacando-se o trabalho de Junior et al., que investiga a introdução inadvertida de *test smells* por profissionais apesar da adoção de práticas padronizadas —, seu impacto no contexto do *JavaScript* permanece relativamente inexplorado. Entender a relação entre a experiência em desenvolvimento e testes de *software* e a habilidade de identificar *test smells* é fundamental para aprimorar tanto as práticas de teste quanto a formação dos desenvolvedores.

Avançando, investigamos a prevalência de tipos específicos de *test smells* em projetos *JavaScript*, visando identificar os mais comuns e problemáticos. Esse exame fornece *insights* cruciais sobre áreas que demandam atenção redobrada nas práticas de teste. Paralelamente, o estudo de Soares et al. explora a percepção de desenvolvedores em projetos *Java* de código aberto quanto à existência de *test smells* e suas decisões de refatoração.

A terceira e última questão de pesquisa se concentra nas estratégias e ferramentas adotadas pelos desenvolvedores para mitigar *test smells*. Essa análise é essencial para elevar a qualidade e eficácia dos testes. O estudo de Damasceno et al. complementa essa discussão, avaliando as percepções e desafios enfrentados por testadores em relação à presença de *test smells* em *Java* e as estratégias de refatoração empregadas.

A combinação dessas três áreas de investigação oferece uma visão abrangente dos desafios enfrentados pelos desenvolvedores de *software* em relação aos *test smells*, destacando tanto as áreas problemáticas quanto as potenciais soluções para melhorar as práticas de teste em projetos *JavaScript*. Este estudo contribui para o campo de testes de *software* ao fornecer uma análise detalhada desses aspectos críticos, visando auxi-

liar desenvolvedores, gerentes de projeto e educadores na implementação de práticas de teste mais eficientes e eficazes.

6.2 Escopo, Objetivo Geral e Questões de Pesquisa

6.2.1 Escopo

O escopo deste estudo centra-se na avaliação do impacto da experiência dos desenvolvedores na identificação de *test smells*, à identificação dos tipos mais prevalentes desses padrões em projetos *JavaScript* e às abordagens utilizadas para mitigá-los. Limitamos a análise das experiências de 11 desenvolvedores, coletadas por meio de um formulário que abrange formação acadêmica, experiências profissionais, identificação de test smells, e opiniões sobre estratégias, dificuldades e ferramentas para mitigação de *test smells*. A metodologia envolve análises estatísticas descritivas e inferenciais para examinar correlações entre a experiência dos desenvolvedores e a presença de *test smells*, restrita a projetos *JavaScript*.

6.2.2 Objetivo Geral

O objetivo deste estudo é avaliar a incidência e as estratégias de mitigação de *test smells* em projetos de *software JavaScript*, analisando o impacto da experiência dos desenvolvedores na identificação desses padrões problemáticos. Busca-se obter subsídios para aperfeiçoar as práticas de teste e elevar a qualidade do *software*, contribuindo para a eficiência e eficácia dos desenvolvedores em ambientes de teste.

6.2.3 Questões de Pesquisa

QP1 Como a experiência de desenvolvedores influencia a capacidade de reconhecimento de *test smells* em ambientes de desenvolvimento de *software*?

QP2 Qual é a prevalência de *test smells* específicos em projetos de software desenvolvidos em *JavaScript*?

QP3 Quais estratégias e ferramentas são eficazes na mitigação de *test smells* em processos de desenvolvimento de *software*?

6.3 Metodologia

O Diagrama 6.1 apresenta a metodologia adotada neste estudo. Inicialmente foram elaboradas as questões de pesquisa referente ao objetivo geral do estudo e a construção do questionário com as perguntas relativas às questões de pesquisa. As perguntas foram criadas no formato de múltipla escolha, caixas de seleção e escalas *Likert*, visando agilizar as respostas dos participantes e simplificar a análise por parte do pesquisador. O recrutamento dos participantes ocorreu parte em ambiente acadêmico, mais especificamente em projetos de cooperação entre a instituição acadêmica e a iniciativa privada; e parte com profissionais que atuam no mercado de desenvolvimento de *software*. Para avaliar o questionário antes da execução do estudo, aplicamos um teste piloto onde pudemos sanar problemas e alinhar as perguntas segundo as questões de pesquisa.

Ao todo, foram coletados dados de 11 desenvolvedores, por meio do questionário com perguntas específicas relacionadas: *i)* as suas experiências em desenvolvimento e testes de *software*, *ii)* a prevalência de *test smells* específicos em projetos *JavaScript* e *iii)* as estratégias e ferramentas para mitigação de *test smells*. Esses dados foram posteriormente submetidos a análises estatísticas descritivas e inferenciais com o auxílio da linguagem *Python*. Os resultados das análises auxiliaram a responder às questões de pesquisa e a posterior discussão.

6.4 Resultados

O conjunto de dados com as respostas e as respectivas análises apresentadas nesta seção se encontram no link `https://colab.research.google.com/drive/1aaz2qQ21BI2gMFH6omfD9bpuWzGohfml?usp=sharing`.

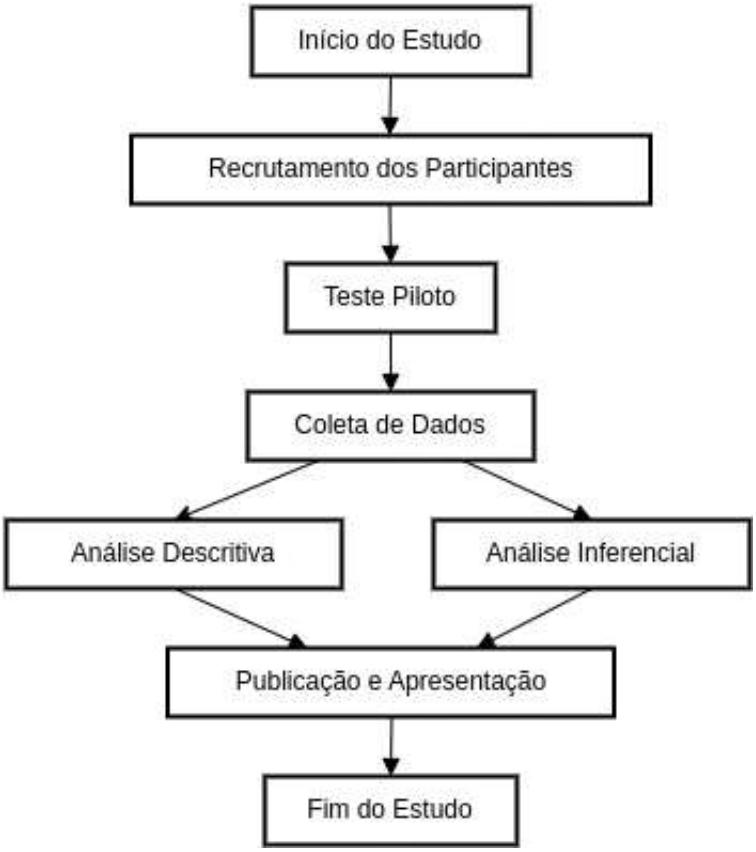


Figura 6.1: Diagrama da metodologia do estudo

6.4.1 Questão de pesquisa 1

A análise dos resultados referentes ao impacto da experiência no reconhecimento de *test smells* revelou aspectos importantes sobre a relação entre a experiência em desenvolvimento e testes de *software* e a habilidade de identificar *test smells*. Os dados mostraram que, embora a maioria dos participantes tivesse uma experiência significativa em desenvolvimento de *software*, a experiência em testes de *software* era mais variada, indicando diferentes níveis de familiaridade com práticas de testes.

Curiosamente, a análise estatística descritiva e inferencial, incluindo teste de correlação de *Spearman*, revelou uma correlação positiva, mas fraca, entre a experiência em desenvolvimento e testes de *software*. Essa constatação sugere que a experiência em desenvolvimento de *software* por si só pode não ser suficiente para garantir uma habilidade robusta em identificar *test smells*.

O resultado implica que habilidades específicas em testes são necessárias para reconhecer efetivamente os *test smells*, independentemente da experiência geral em desen-

volvimento de *software*, e aponta para a necessidade de uma investigação mais aprofundada sobre outros fatores que podem influenciar a capacidade dos desenvolvedores de identificar *test smells*.

A Tabela 6.1 apresenta a estatística descritiva da experiência dos participantes em termos de desenvolvimento e testes de *software*, oferecendo um panorama quantitativo sobre o impacto da experiência no reconhecimento de *test smells*.

Tabela 6.1: Estatística Descritiva das Experiências Profissionais

Estatística	Experiência em Desenvolvimento	Experiência em Testes
Média	4.18	2.73
Desvio	1.08	1.56
Mín	2.00	0.00
25%	3.50	2.00
50%	5.00	3.00
75%	5.00	3.50
Máx	5.00	5.00

Teste de Normalidade (*Shapiro-Wilk*)

1. **Experiência em Desenvolvimento de Software**:
   - Estatística: 0.78
   - p-valor: 0.0048
   - Interpretação: Como o p-valor é menor que 0.05, rejeitamos a hipótese nula de normalidade. Portanto, os dados não seguem uma distribuição normal.
2. **Experiência em Testes de Software**:
   - Estatística: 0.95
   - p-valor: 0.63
   - Interpretação: O p-valor é maior que 0.05, o que não nos permite rejeitar a hipótese nula. Assim, podemos considerar que os dados seguem uma distribuição aproximadamente normal.

Teste de Correlação de Spearman

- Correlação: 0.25
- p-valor: 0.45
- Interpretação: A correlação é positiva, mas fraca. O p-valor alto sugere que a correlação não é estatisticamente significativa.

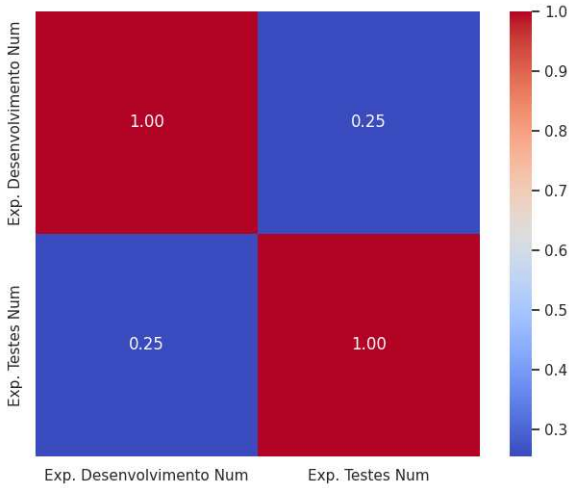


Figura 6.2: Correlação entre Experiência em Desenvolvimento e Testes de Software (Spearman)

6.4.2 Questão de pesquisa 2

Na investigação sobre a prevalência de *test smells* específicos em projetos JavaScript, a análise das respostas dos desenvolvedores revelou uma série de percepções sobre as tendências comuns na área de testes de *software*. Os dados coletados mostraram que certos *test smells* são particularmente prevalentes em projetos *JavaScript*. O *"Magic Number Test"*, caracterizado pelo uso de números sem explicação clara ou contexto nos testes, emergiu como o mais comum, seguido de perto por *"Exception Handling"*, que indica um tratamento inadequado de exceções nos testes. Além disso, *"Conditional Test Logic"*, que envolve o uso de lógica condicional complexa nos testes, foi identificado como um problema frequente. Esses resultados apontam para uma tendência de complexidade e falta de clareza nos testes, desafiando a manutenção e a eficácia desses testes ao longo do tempo.

Outros *test smells* frequentemente identificados incluíram *"Assertion Roulette"*, onde várias asserções são feitas sem mensagens claras, e *"Mystery Test"*, caracterizado pela dependência de recursos externos sem clara indicação ou documentação. A identificação desses *test smells* sugere uma série de desafios enfrentados pelos desenvolvedores, como a dificuldade em identificar a causa de falhas nos testes e a inconsistência dos testes em diferentes ambientes. Além disso, a presença de *"Resource Optimism"* e *"Eager Test"* reflete preocupações com a interação do teste com o ambiente externo e a abordagem de múltiplos métodos em um único teste, respectivamente. Estes padrões indicam áreas onde as práticas de teste podem ser aprimoradas, destacando a necessidade de estratégias mais eficazes e ferramentas para lidar com esses problemas comuns em testes *JavaScript*.

A Figura 6.3 mostra o gráfico de barras com os dez *test smells* mais frequentemente mencionados pelos desenvolvedores, com suas respectivas frequências. Este gráfico fornece uma visualização clara de quais *test smells* são mais comuns, ajudando a identificar as áreas que podem requerer maior atenção na melhoria das práticas de teste em projetos *JavaScript*.

Figura 6.3: Totais da prevalência de *test smells*

6.4.3 Questão de pesquisa 3

A pesquisa sobre estratégias e ferramentas para a mitigação de *test smells* em projetos *JavaScript* revelou uma série de práticas e desafios enfrentados pelos desenvolvedores.

A Figura 6.4 apresenta um gráfico de barras das estratégias e dificuldades apontadas pelos desenvolvedores com suas respectivas frequências. A estratégia mais mencionada foi a padronização da escrita e organização dos testes, destacando a importância de manter um padrão uniforme na abordagem dos testes. Isso inclui a adoção de convenções de nomenclatura consistentes, a estruturação lógica dos testes e a manutenção da consistência em toda a equipe. Paralelamente, o treinamento da equipe em boas práticas de testes surgiu como uma necessidade crucial, indicando que a capacitação contínua é vital para a prevenção e correção de *test smells*.

O uso de ferramentas de análise estática foi igualmente enfatizado como um método eficaz para identificar *test smells* automaticamente, permitindo uma correção mais rápida e precisa. Entretanto, um desafio significativo mencionado foi a inexistência ou escassez de documentação adequada, tanto dos testes quanto do sistema em si, dificultando a compreensão e manutenção dos testes. A automação também foi reconhecida como essencial para identificar rapidamente falhas nos testes, especialmente as que são intermitentes ou difíceis de reproduzir.

Além disso, a complexidade inerente de alguns sistemas foi citada como um fator que contribui para a ocorrência de *test smells*, ressaltando a necessidade de técnicas de teste mais avançadas. A falta de conhecimento sobre *test smells* e boas práticas de teste foi outro obstáculo identificado, sugerindo que a conscientização e educação são aspectos críticos para a melhoria da qualidade dos testes. Limitações de tempo e recursos também foram mencionadas como fatores que podem levar ao surgimento de *test smells*, apontando para a necessidade de uma melhor gestão de projetos e alocação de recursos.

Revisões de código focadas especificamente na qualidade dos testes foram sugeridas como uma prática útil para identificar e corrigir *test smells* precocemente no ciclo de desenvolvimento. No entanto, a resistência cultural à adoção de práticas rigorosas de teste foi um desafio citado, indicando a necessidade de uma mudança de mentalidade e uma maior valorização dos testes na cultura organizacional. A qualidade dos próprios testes, incluindo a clareza e a precisão, foi reconhecida como um fator crucial, com testes mal escritos contribuindo significativamente para o surgimento de *test smells*.

Por fim, a ausência de ferramentas adequadas para testar e identificar *test smells*

foi um desafio identificado, destacando a necessidade de investimento em ferramentas eficazes. A importância de monitorar continuamente a qualidade dos testes através da coleta de métricas foi reconhecida como uma prática que pode ajudar a identificar problemas proativamente.

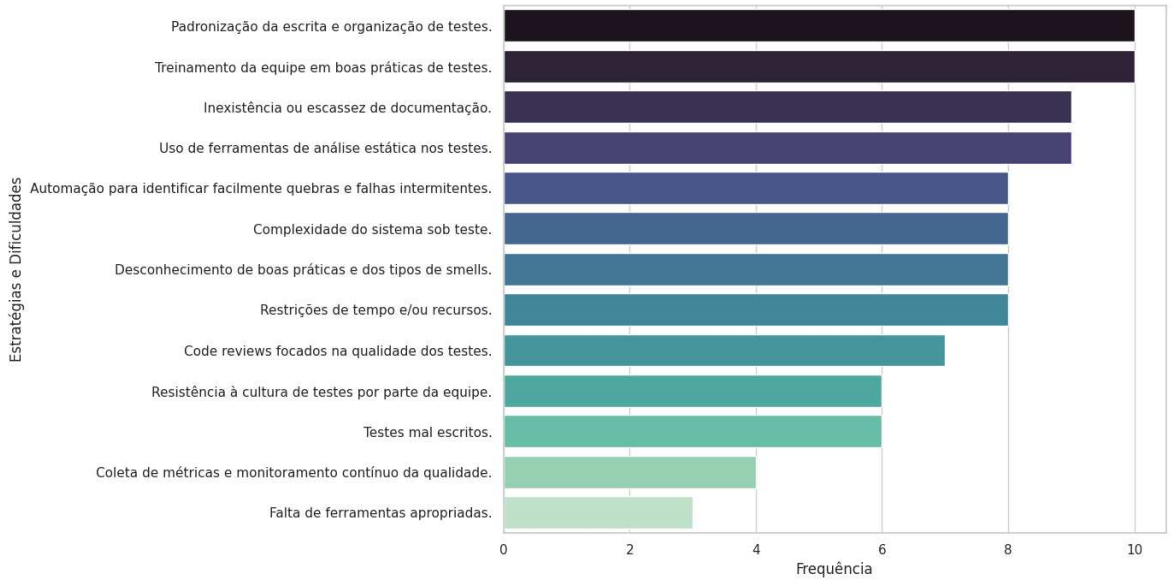


Figura 6.4: Frequência nas Estratégias e Dificuldades na Mitigação de *test smells*

Esses resultados fornecem uma visão abrangente das abordagens atuais e dos desafios na mitigação de *test smells* em projetos *JavaScript*, enfatizando a necessidade de uma combinação de treinamento, ferramentas apropriadas, práticas de desenvolvimento focadas na qualidade e uma mudança cultural em direção à valorização dos testes de *software*.

6.4.4 Ameaças à Validade

- **Validade Interna:** Pode ter sido afetada pela forma como o questionário foi construído e pela interpretação das perguntas pelos participantes. Tivemos o cuidado na elaboração do questionário, avaliando a coerência e coesão das perguntas com os objetivos das questões de pesquisa. Além disso, o pequeno tamanho da amostra limitou a capacidade de detectar diferenças significativas ou tendências sutis nos dados.

- **Validade Externa:** A generalização dos resultados para a população mais ampla de desenvolvedores de *software* pode ser limitada devido ao recrutamento específico de participantes e ao tamanho da amostra. A diversidade de contextos de desenvolvimento e experiências dos participantes, embora benéfica, pode não ser suficientemente representativa.

- **Validade de Construto:** A utilização de escalas *Likert* e outras formas de resposta fechada pode limitar a capacidade de capturar a complexidade e a nuance das experiências e percepções dos participantes em relação aos *test smells* e suas estratégias de mitigação.

- **Validade de Conclusão:** As análises estatísticas foram planejadas e executadas para evitar conclusões errôneas. No entanto, o uso de uma amostra pequena aumenta o risco de erros do tipo I e II, podendo levar a conclusões incorretas sobre as relações entre variáveis.

6.5 Discussão

6.5.1 Questão de pesquisa 1

Discutir como a experiência afeta o reconhecimento de problemas em testes de *software* em *JavaScript* ajuda a entender como identificar problemas de qualidade em testes de *software*. A correlação fraca entre a experiência em desenvolvimento de *software* e a habilidade em identificar test smells sugere que a competência em testes de *software* é uma habilidade distinta, não necessariamente garantida pela experiência geral em desenvolvimento. Isso indica que, embora a experiência em desenvolvimento possa proporcionar uma base sólida, são necessárias habilidades e conhecimentos específicos em testes para identificar e tratar eficazmente os *test smells*. Além disso, a variação na experiência dos desenvolvedores em testes de software ressalta a diversidade de abordagens e práticas em diferentes contextos de trabalho. Esta descoberta sublinha a importância de uma formação específica em qualidade de teste e práticas de teste eficazes, bem como a necessidade de ferramentas e métodos que apoiem os desenvolvedores na identificação de *test smells*. A pesquisa aponta para a necessidade de mais estudos

que explorem como diferentes contextos de trabalho, culturas organizacionais e métodos de treinamento podem influenciar a habilidade dos desenvolvedores em reconhecer e mitigar *test smells*, visando aprimorar as práticas de teste e, consequentemente, a qualidade do *software* desenvolvido.

6.5.2 Questão de pesquisa 2

Nesta subseção, discutimos como os resultados obtidos na análise da prevalência de *test smells* específicos em projetos *JavaScript* podem evidenciar diversas implicações importantes para a prática de testes de *software*. A frequência elevada de *test smells* como *"Magic Number Test"* e *"Exception Handling"* sugere que os desenvolvedores muitas vezes podem se deparar com dificuldades ao tentar tornar seus testes tanto compreensíveis quanto robustos. O uso de números mágicos nos testes, por exemplo, pode indicar uma tendência a priorizar a conveniência ou a velocidade de escrita dos testes em detrimento da clareza e manutenção a longo prazo. Da mesma forma, o tratamento inadequado de exceções nos testes pode comprometer a confiabilidade dos mesmos, levando a uma falsa sensação de segurança quanto à robustez do *software*. Estes padrões sugerem a necessidade de uma maior ênfase na formação e na adoção de melhores práticas de codificação, visando aprimorar a qualidade geral dos testes.

Além disso, a identificação frequente de *test smells* como *"Conditional Test Logic"* e *"Assertion Roulette"* aponta para desafios na estruturação eficaz de testes e na comunicação clara dos seus resultados. A complexidade desnecessária nos testes, decorrente de lógica condicional excessiva, pode obscurecer o propósito do teste e dificultar a identificação de falhas específicas. A presença de múltiplas asserções sem mensagens claras também dificulta a depuração eficiente quando os testes falham. A ocorrência desses padrões sugere uma área de melhoria significativa no *design* e na execução de testes, enfatizando a importância de uma abordagem mais sistemática e bem documentada na escrita de testes. O reconhecimento desses *test smells* fornece uma oportunidade valiosa para equipes de desenvolvimento revisarem e melhorarem suas práticas de teste, para aumentar a eficácia, a eficiência e a confiabilidade de seus processos de teste em projetos *JavaScript*.

6.5.3 Questão de pesquisa 3

Os resultados obtidos neste estudo relacionados às estratégias e ferramentas para a mitigação de *test smells* em projetos *JavaScript* revelaram um cenário complexo e multifacetado no campo de testes de *software*. Os desenvolvedores destacaram uma variedade de práticas e desafios, indicando uma consciência crescente da importância de abordagens estratégicas para melhorar a qualidade dos testes.

A ênfase na padronização da escrita e organização dos testes reflete uma necessidade reconhecida de clareza e consistência nos processos de teste. Esta padronização é crucial para evitar ambiguidades e erros comuns, facilitando a manutenção e a compreensão dos testes ao longo do tempo. O treinamento da equipe em boas práticas emerge como um elemento fundamental, sugerindo que a educação contínua e o desenvolvimento de habilidades são essenciais para capacitar desenvolvedores a identificar e corrigir *test smells* de maneira eficaz.

Além disso, o uso de ferramentas de análise estática é frequentemente citado, sublinhando a importância de recursos automatizados no processo de detecção e correção de problemas nos testes. Essas ferramentas podem dar informações importantes que seriam difíceis de obter manualmente, aumentando a eficiência e a eficácia do processo de teste.

Por outro lado, os desafios identificados, como a falta de documentação adequada e o desconhecimento sobre tipos específicos de *test smells* e práticas eficazes, apontam para lacunas críticas nas abordagens atuais. A inexistência ou a escassez de documentação clara é particularmente problemática, uma vez que prejudica a capacidade dos desenvolvedores de entender e melhorar os testes existentes. Igualmente, o desconhecimento sobre boas práticas e os tipos de *test smells* sugere uma área de melhoria potencial para os programas de treinamento e desenvolvimento profissional.

Outro aspecto notável é a menção às restrições de tempo e recursos, um desafio comum no desenvolvimento de *software* que pode levar à adoção de atalhos ou práticas menos rigorosas nos testes. Isso reforça a necessidade de uma gestão eficaz de projetos que priorize a qualidade dos testes, mesmo em face de prazos apertados e recursos limitados.

Finalmente, a pesquisa sublinha a importância de uma cultura organizacional que

valorize a qualidade dos testes. A resistência à adoção de uma cultura de testes robusta e a prevalência de testes mal escritos são desafios que vão além das habilidades técnicas, envolvendo aspectos de gestão de equipe e cultura organizacional.

Em resumo, esta discussão destaca a necessidade de uma abordagem holística para a mitigação de test smells em projetos *JavaScript*. As soluções passam não apenas pelo desenvolvimento de habilidades técnicas e o uso de ferramentas apropriadas, mas também pela criação de uma cultura organizacional que valorize e priorize a qualidade dos testes. Este entendimento é crucial para guiar esforços futuros na melhoria de práticas de teste em ambientes de desenvolvimento de *software*.

6.6 Considerações Finais

Neste capítulo conduzimos um estudo qualitativo com foco em três aspectos fundamentais dos *test smells* em projetos de *software* que utilizam a linguagem *JavaScript*: a influência da experiência dos desenvolvedores no reconhecimento de test smells, a prevalência de tipos específicos de test smells, e as estratégias e ferramentas adotadas para sua mitigação. No próximo capítulo, exploraremos os trabalhos relacionados, onde examinaremos como as descobertas deste estudo se alinham com a literatura existente, ampliando nossa compreensão sobre test smells e estratégias de mitigação em projetos *JavaScript*.

Capítulo 7

Trabalhos Relacionados

Após a fundamentação teórica apresentada no Capítulo 2, apresentamos neste capítulo uma síntese de trabalhos significativos em que estão relacionados ao nosso campo de pesquisa e apontaremos onde nosso estudo contribui para o estado da arte. As Seções 7.1, 7.2, 7.3, 7.4 e 7.5 listam, respectivamente, os trabalhos de Tufano et al., Bavota et al., Spadini et al., Garousi and Küçük e Junior et al. na área de *test smells*. Já as Seções 7.6, 7.7, 7.8, 7.9 e 7.10 descrevem os trabalhos de Saboury et al., Johannes et al., Damasceno et al., Zozas et al. e Malavolta et al. que focam em *code smells*, débito técnico e código morto na linguagem *JavaScript*.

7.1 Tufano et al

Muitos trabalhos na literatura buscam compreender os mais diversos aspectos da existência de *test smells* em projetos de *software*, visando sanar dúvidas que vão desde o ciclo de vida dos *test smells*, até o impacto causado no processo de desenvolvimento. Diante disso, Tufano et al. [54] nos apresentam um estudo empírico que contemplou duas etapas, sendo a primeira etapa para avaliar as percepções dos desenvolvedores perante os *test smells* e a segunda etapa examina de forma mais aprofundada como os desenvolvedores lidam com o código que possui problemas de *design* decorrente de *smells*.

Nesta primeira fase do estudo, foi realizado um questionário com 19 participantes com o intuito de avaliar o entendimento destes a respeito da presença de *smells* em

testes de *software*. Este questionário apresentou 95 *test smells* de 5 tipos diferentes, nos quais em cada um destes o participante teve que responder: *i)* se havia algum problema de *design*; *ii)* explicar os prováveis problemas de *design*; *iii)* porque o mau design foi introduzido; *iv)* se existe a necessidade de refatoração do código; e *v)* como seria a refatoração. A análise dos resultados revelou que os *test smells* são ignorados como ameaças pela maioria dos participantes, e apenas 2% destes conseguiram identificar corretamente a presença de um *test smell*. Ademais, quase a totalidade dos participantes responderam que a refatoração não ajudaria a melhorar o *design* do teste e, consequentemente, não souberam sugerir técnicas para refatorar o código.

A segunda etapa se aprofundou no entendimento do ciclo de vida dos *test smells* e para isso foram analisados 152 projetos *open-source*. Em vista disso, os autores observaram que a escolha de péssimos designs ocorreram logo na criação dos testes e não durante a evolução destes. Outro ponto observado foi que os testes evoluem raramente e a remoção dos *test smells* se deram brevemente, exigindo poucas modificações por parte dos desenvolvedores.

No que se refere a quantidade de *smells* removidos, o percentual foi baixo e isto reflete o impacto negativo dos *smells* na legibilidade e manutenibilidade do código. Mediante este fato, os autores reforçam a importância de ferramentas automáticas que auxiliem o desenvolvedor na identificação e remoção dos *test smells*. Uma vez que a retirada do *smell* é dificultada, a sua capacidade de sobrevivência é alta, pois o estudo apontou que 80% dos *smells* ainda sobrevivem após 1000 dias de introdução. Se por um lado este índice de sobrevivência clama por ferramentas de detecção automáticas de *smells*, por outro lado, sugere uma discussão sobre os custos envolvidos na manutenção destes casos de testes. Por fim, esta etapa do estudo mostrou uma forte correlação entre alguns *code smells* e *test smells*, o que demonstra que códigos de produção mal construídos e complexos corroboram para a presença de *smells* em seus respectivos casos de testes.

Em nosso trabalho de doutorado, encontramos resultados semelhantes aos de Tufano et al. em termos da prevalência de *test smells* e sua correlação fraca com a experiência de desenvolvimento. Além disso, também identificamos práticas atuais e desafios na mitigação de *test smells* em projetos *JavaScript*, enfatizando a importância de educa-

ção, ferramentas adequadas e uma cultura de qualidade de teste forte.

7.2 Bavota et al

Neste trabalho empírico de Bavota et al. [55], o objetivo foi responder uma lacuna existente até então sobre qual é a real dimensão da presença de *test smells* em projetos de *software* e as consequências na assimilação do código pelo desenvolvedor. Com isso em mente, os autores revisitam 2 estudos que indicam que tanto projetos *open-source* como industriais estão sujeitos a incidência de *smells*, fazendo com que a compreensão e manutenção dos testes sejam comprometidas.

No primeiro estudo, os autores analisaram 27 projetos, sendo 25 projetos *open-source* e 2 projetos industriais, dos quais foram extraídos os seguintes dados: número de linhas do projeto, número de classes no código de produção, número de classes *JUnit*[1], número de linhas *JUnit*, idade do projeto, idade dos testes e tamanho da equipe. Após a análise, os resultados revelaram uma alta distribuição de *test smells* entre os casos de testes, independente de serem *open-source* ou não.

Continuando as conclusões relativas a qual tipo de projeto é mais suscetível ao aparecimento de *smells* nos testes, os números foram semelhantes. Porém, foi percebido um percentual um pouco maior nos projetos industriais, que pode ser explicado pela constante pressão no tempo de entrega dos artefatos. É relevante frisar que o número reduzido de projetos industriais neste estudo – em virtude da dificuldade de estarem disponibilizados para uma pesquisa – fez com que existisse a possibilidade de ameaças à validade externa relacionadas à generalização dos dados.

Por fim, o segundo estudo envolveu um experimento controlado juntamente com mais 3 replicações, e teve a participação de 61 indivíduos nos mais diversos níveis de experiência em desenvolvimento de *software*. As conclusões mostraram que os acertos nas tarefas dos participantes sofreram consequências com o *test smells* e que o tempo não influenciou as tarefas, embora o código refatorado permitiu economia de tempo na execução de tarefas. Além disso, os *test smells* tiveram impacto diferente dependendo do nível de experiência do participante e com uma vantagem para desenvolvedores da

[1] `https://junit.org/junit5`

indústria.

O estudo de Bavota et al. e o nosso compartilham o objetivo comum de avaliar o impacto dos *test smells* na qualidade do *software*, adotando metodologias empíricas para a detecção automática desses *smells* em conjuntos de testes e examinando sua correlação com métricas de qualidade de código. No entanto, divergem no escopo e contexto: enquanto o estudo de Bavota et al. incide sobre códigos de teste em *Java*, especificamente suítes de teste *JUnit*, o nosso trabalho aborda o espaço menos explorado dos códigos de teste em *JavaScript*, demonstrando desafios específicos dessa linguagem e de seus *frameworks* de teste.

7.3 Spadini et al

Os autores deste trabalho trazem um extenso estudo com dados obtidos a partir de 221 *releases* oriundos de 10 projetos de *software open-source*, com o propósito de buscar um maior entendimento acerca das consequências da presença de *test smells* para a base de testes, assim como no código de produção [56]. Estes projetos participantes foram selecionados de forma aleatória após uma busca no *Github*[2]. Para garantir a representatividade dos dados da amostra, foi empregado no filtro da busca o critério de possuir mais de 1000 casos de testes em todos os *releases*.

Mais especificamente, os autores investigaram se existe tendência de mudanças e propensão a erros nos casos de testes e código de produção, em virtude da presença de *test smells*. Para isso, Spadini et al. recorrem a métricas indicadoras de predisposição a mudanças e também de defeitos numa base de dados contendo mais de um milhão de casos de testes.

As conclusões obtidas pela análise dos resultados confirmaram as suspeitas de que a presença de *test smells* aumente a chance dos testes estarem suscetíveis a defeitos e também a mudanças. Além disso, o estudo revelou os tipos de *smells* que são prováveis de ocasionar mudanças no código. Por último, os resultados demonstraram que a existência de *smells* ampliam as possibilidades do código de produção estar vulnerável a ocorrência de defeitos.

[2] `https://github.com`

7.4 Garousi e Küçük

A relevância em compreender os mecanismos dos *test smells* é demonstrada pelos inúmeros trabalhos da literatura, sejam eles provenientes da academia ou da indústria. Diante deste farto conjunto de conhecimentos espalhados, Garousi and Küçük [10] apresentaram uma revisão da literatura visando reunir o estado da arte a respeito de *test smells*, servindo assim como um índice para facilitar novos trabalhos de pesquisadores ou interessados na área.

Este trabalho amplia sua relevância ao adotar a técnica de *Multivocal Literature Review* [61], o qual engloba artefatos da literatura científica com outros provenientes de fontes de menor formalismo científico. Tais fontes de menor rigor são conhecidos como literatura cinza (do inglês *gray literature*), que compreendem postagens de *blogs*, *white papers* da indústria, vídeos de apresentação ou qualquer outro formato que veicule um determinado tópico ou conhecimento contemporâneo.

Em suma, os autores mapearam os estudos em diversas classificações: *i)* pela perspectiva da origem de contribuição, *ii)* pelo tipo de pesquisa, *iii)* pelas questões de pesquisa, *iv)* por tipos de *smells*, *v)* por consequências da presença de *smells* e *vi)* por abordagens, *frameworks* e ferramentas para lidar com os *smells*.

7.5 Junior et al

Seguindo a mesma linha dos trabalhos de Tufano et al. e Bavota et al., os autores Junior et al. realizaram um estudo [51] para entender como as práticas diárias de desenvolvedores profissionais impactam no surgimento de *test smells* em projetos de *software*. Diante do fato de que as atividades de testes representam um custo altíssimo para o desenvolvimento de *software* [62] e que a presença de *smells* podem refletir em riscos e ampliar ainda mais os gastos [28], os pesquisadores buscaram respostas para inferir sobre a origem de anti-padrões no código de testes.

Sendo assim, os autores aplicaram um estudo de caso-controle [63], no qual uma amostra com 60 participantes responderam um questionário contendo perguntas sobre o perfil profissional e as perspectivas acerca da criação e execução dos casos de testes. Para evitar interferências no estudo, o questionário evitou o uso do termo *test smell*

ou palavras de sentido semelhante. Os resultados obtidos mostraram que as práticas adotadas de design podem levar ao aparecimento de *test smells*, mesmo sem o desenvolvedor se dar conta. Também foi observado que os resultados foram inconclusivos sobre a influência que experiência profissional exerce na inserção de *smells* durante a criação dos casos de testes. Além disso, os resultados demonstraram que as práticas mais comuns que podem originar os *test smells* foram o uso de dados genéricos de configuração e o uso de comandos de decisão e repetição.

Por fim, os resultados ainda evidenciaram que a maioria dos casos de testes foram criados por desenvolvedores com mais experiência profissional, enquanto os desenvolvedores menos experientes foram responsáveis pela maior parte das execuções. Em relação aos motivos que impactam no surgimento de *smells*, os desenvolvedores citaram os padrões de codificação da empresa, as restrições de tempo e os esforços para alcançar melhor cobertura e desempenho no código. Ambos os estudos de Junior et al. e o nosso destacam a necessidade de aumentar a conscientização sobre a qualidade do código de teste entre os desenvolvedores e a importância de ferramentas automatizadas para detecção e refatoração de *test smells*.

7.6 Saboury et al

Os pesquisadores Saboury et al. [57] realizaram uma investigação quantitativa e qualitativa para compreender o impacto dos *smells* de código na propensão a falhas das aplicações desenvolvidas na linguagem *JavaScript*. Mais precisamente, o estudo detectou 12 tipos de *smells* em 537 *releases* de 5 aplicações *open-source* em *JavaScript* (*express*, *grunt*, *bower*, *less.js*, e *request*) e realizou uma análise de sobrevida para o aparecimento de falhas entre os arquivos com e sem *smells*. Em um segundo momento, o estudo também avaliou qualitativamente as respostas de 1484 desenvolvedores para um questionário visando entender suas percepções acerca dos *smells* detectados.

Os resultados obtidos no estudo quantitativo demonstraram indícios de que a presença de *smells* no código proporcionam uma alta propensão a falhas. Em contrapartida, os arquivos livres de *smell* possuíram 65% a menos de chance de ocorrer falhas quando comparados aos arquivos com a presença de *code smells*. Deste modo, a

ocorrência de *smells* tem impacto real no custo de manutenção e evolução das aplicações e é um indicador revelante para gerentes estimarem os custos de mão de obra no orçamento de projetos.

Já os resultados do estudo qualitativo revelou que os desenvolvedores priorizam a resolução de *code smells* conforme o grau de ameaça proporcionado. Dentre os *smells* detectados no estudo quantitativo, os participantes indicaram os que possuem maior impacto nocivo na manutenibilidade e confiabilidade das aplicações. São eles: *i)* o *"Nested Callbacks"*, sendo o *Callback* um recurso utilizado para lidar com tarefas assícronas e de forma não-bloqueante, mas o seu aninhamento torna o código complexo (*Callback Hell*); *ii)* o *"Variable Re-assign"*, que ocorre devido a tipagem fraca que a linguagem possui; e *iii)* o *"Long Parameter List"*, o qual é um indicador de que a função tem mais responsabilidades do que deveria ou que os parâmetros deveriam estar contidos em uma estrutura mais organizada.

Os trabalhos de Saboury et al. e o nosso compartilham um foco comum na análise de projetos *JavaScript* para identificar problemas de qualidade, utilizando métodos empíricos e ferramentas de análise para coletar dados. Ambos visam melhorar a qualidade e a manutenção do *software*, embora em diferentes aspectos: Saboury et al. concentram-se em *code smells* em aplicações *server-side*, destacando a importância de abordar problemas de manutenibilidade e complexidade, enquanto focamos em *test smells*, investigando como esses *smells* afetam a qualidade do código de teste e desenvolvemos a ferramenta STEEL para auxiliar na detecção desses *smells*. As diferenças entre os estudos refletem a diversidade de desafios na qualidade do *software* em *JavaScript*, abordando tanto o código de produção quanto o código de teste para promover práticas de desenvolvimento melhoradas.

7.7 Johannes et al.

O trabalho de Johannes et al. [58] é uma ampliação do estudo da Seção 7.6 no qual são analisados 12 tipos de *smells* em 1807 versões de 15 aplicações *open-source* desenvolvidas com *JavaScript*. Além do objetivo de medir o tempo de aparecimento de falhas nos códigos infectados e não-infectados via análise de sobrevivência, os pesquisadores

também examinaram a incidência e retirada dos *smells* usando modelos de sobrevida.

Os resultados obtidos neste trabalho apontaram que arquivos sem indícios de *smells* tiveram 33% menos chances de ter problemas do que os arquivos com *smells* identificados. Foi observado também que os tipos de *smells* que ofereceram maiores riscos foram *"Variable Re-assign"*, *"Assignment in Conditional statements"* e *"Complex Code"*. Além disso, foi percebido que os *smells*, e especialmente o *"Variable Re-assign"*, são introduzidos no processo de criação dos arquivos de código e igualmente permanecem no código por um longo tempo até a remoção.

7.8 Damasceno et al.

O estudo realizado por Damasceno et al. [53] investiga a refatoração de *test smells* sob a perspectiva dos desenvolvedores e seus impactos nas qualidades internas do código de teste, como tamanho, coesão, acoplamento e complexidade. Analisando 100 exemplos de cinco tipos de *test smells* em quatro sistemas de código aberto desenvolvidos em Java e as percepções de vinte desenvolvedores, o estudo revela que a refatoração desses *smells* específicos de teste pode melhorar significativamente os atributos de qualidade interna, exceto pelo tamanho do código, que tende a aumentar. Entre os achados, destaca-se que os *smells Eager Test* e *Duplicate Assert* são considerados os mais críticos pelos desenvolvedores, enquanto *Assertion Roulette* e *Magic Number Test* são vistos como menos prejudiciais. Uma das principais dificuldades encontradas durante a refatoração é a compreensão do código-fonte, sugerindo que práticas de codificação claras podem facilitar o processo de refatoração de *test smells*.

Comparando com o nosso estudo do Capítulo 6, ambos abordam a importância da qualidade dos testes em projetos de software e a influência dos *test smells* nessa qualidade. No entanto, enquanto o estudo de Damasceno et al. foca em aspectos práticos da refatoração de *test smells* e sua percepção pelos desenvolvedores, o nosso estudo enfoca na incidência e mitigação de *test smells* em projetos *JavaScript*, destacando a experiência dos desenvolvedores na identificação desses *smells* e as estratégias utilizadas para sua mitigação. Além disso, nosso estudo propõe investigar a prevalência de tipos específicos de *test smells* e como a experiência de desenvolvimento afeta a capacidade

de identificá-los, complementando a abordagem de Damasceno et al. ao considerar a percepção dos desenvolvedores sobre a gravidade dos test smells e as dificuldades encontradas durante a refatoração.

7.9 Zozas et al.

O estudo de Zozas et al. apresenta a questão da dívida técnica do código em aplicações *JavaScript*, abordando como a flexibilidade e as características da linguagem podem tanto acelerar o desenvolvimento quanto introduzir riscos significativos relacionados à dívida técnica. Esse estudo parece enfatizar a previsão e a gestão da dívida técnica, um conceito que engloba não apenas os problemas de qualidade do código identificados como *smells*, mas também outras questões que podem afetar a sustentabilidade e a evolução do software a longo prazo.

Enquanto o nosso estudo do Capítulo 5 se concentra na detecção e no impacto dos *test smells* em *JavaScript*, o segundo expande a discussão para a dívida técnica do código, sugerindo uma abordagem mais ampla para entender e gerenciar os desafios de qualidade no desenvolvimento de aplicações *JavaScript*.

7.10 Malavolta et al.

O estudo de Malavolta et al. apresenta o *Lacuna*, uma abordagem para detecção e eliminação de código morto em *JavaScript* em aplicações *web*. Essa abordagem é significativa, por combinar análises estáticas e dinâmicas para otimizar aplicações *web* pela remoção de código não utilizado, melhorando potencialmente os tempos de carregamento e reduzindo o uso de recursos. A metodologia é extensível, permitindo a aplicação do *Lacuna* em qualquer base de código *JavaScript* sem restrições quanto ao estilo de codificação ou construtos de *JavaScript*. Uma avaliação empírica foi conduzida em 30 aplicações *web* móveis para avaliar o impacto do código morto em métricas de desempenho como consumo de energia, uso de rede e uso de recursos. Os resultados sugerem que a remoção de código morto em *JavaScript* afeta positivamente os tempos de carregamento das aplicações *web* móveis e diminui significativamente os bytes

transferidos pela rede.

Comparando com o nosso estudo do Capítulo 5, que foca em investigar *test smells* no código de teste *JavaScript* e sua correlação com métricas de qualidade de *software*, observamos um foco distinto, porém complementar, entre os dois. Enquanto Malavolta et al. se concentram na identificação e remoção de código morto em *JavaScript* para melhorar o desempenho de aplicações *web*, o nosso estudo se concentra na qualidade do próprio código de teste, examinando como certos padrões no código de teste (*smells*) se correlacionam com a qualidade geral do *software*.

7.11 Considerações Finais

Apresentamos neste capítulo alguns trabalhos que julgamos pertinentes e que estão relacionados a *test smells* no contexto da linguagem *Java*. Também expomos alguns trabalhos no âmbito da linguagem *JavaScript*, com foco em *code smells*, dívida técnica e código morto. Estes trabalhos nos trouxeram o alicerce fundamental sobre *smells* e serviram de motivação, guiando a nossa pesquisa. Diante disso, nosso trabalho contribui no sentido de trazer entendimento para as lacunas existentes, sendo especificamente os *test smells* na linguagem *JavaScript*. No próximo capítulo apresentaremos a conclusão do nosso trabalho de doutorado.

Capítulo 8

Conclusão

Este capítulo apresenta um resumo dos principais resultados deste trabalho e apresenta algumas sugestões para futuras pesquisas. Primeiramente, as conclusões são delineadas na Seção 8.1 e os possíveis trabalhos futuros são apresentados na Seção 8.2.

8.1 Conclusões

O principal objetivo deste trabalho de doutorado foi investigar a incidência e a relação entre os *test smells* e desses com as métricas de qualidade de software em projetos *JavaScript*, evidenciando a relevância do teste de *software* no desenvolvimento de sistemas e os desafios intrínsecos a esses *test smells*.

Considerando as questões de pesquisa definidas anteriormente e visando responder o objetivo do nosso trabalho, os seguintes resultados foram alcançados:

QP1 Quais são os *test smells* mais frequentemente detectados em projetos *JavaScript*?

No contexto da engenharia de *software*, é inegável que testes em *JavaScript* estão, assim como em outras linguagens, suscetíveis a *test smells*. Entretanto, dada a natureza intrínseca de *JavaScript* como uma linguagem dinâmica, multi-paradigma e com particularidades como o hoisting, pode-se afirmar que existem *test smells* que são específicos ou mais prevalentes nesse ambiente. Além disso, evidências apontam para uma frequência elevada de *smells* como *Duplicate Assert* e *Conditional Test*. Esta ocorrência não apenas indica possíveis problemas de design

no código de produção, o que, consequentemente, aumenta a complexidade na elaboração dos testes, como também evidencia a necessidade iminente de revisão e aprimoramento no design e implementação das suítes de testes.

Os resultados desse estudo mostram a importância de se investigar *test smells* no âmbito do *JavaScript*, e também reforçam a necessidade de se conduzir pesquisas adicionais para o desenvolvimento de técnicas avançadas e ferramentas dedicadas à análise e detecção desses *smells*. Esta preocupação torna-se ainda mais relevante quando consideramos a utilização do *JavaScript* no cenário atual de desenvolvimento de software. A linguagem se estende por uma ampla gama de aplicações, abrangendo desde sistemas *web* e *desktop* até dispositivos móveis e soluções para a *Internet* das Coisas, reiterando sua importância e a necessidade de garantir práticas de teste robustas e eficientes.

QP2 Quais as associações significativas entre os *test smells* que foram detectados?

As evidências coletadas evidenciaram correlações significativas entre certos *test smells*, sugerindo a possibilidade de interdependências na prática de codificação. Notavelmente, a forte correlação entre *Eager Test* e *Lazy Test*, emerge como a mais robusta, indicando que testes que são excessivamente abrangentes (*eager*) muitas vezes contêm segmentos que não são suficientemente rigorosos em sua avaliação (*lazy*). Além disso, o *Conditional Test* apresentou correlações substanciais com uma variedade de outros *test smells*, sugerindo sua prevalência e potencial interação com outros problemas no código.

Essas descobertas revelam a complexidade das práticas de codificação, salientando que a mitigação de um *test smell* pode, inesperadamente, introduzir ou estar associada a outro. Isso demonstra a complexidade da codificação e da refatoração, bem como a urgência de abordagens abrangentes e bem informadas na prática de desenvolvimento. No entanto, é preciso reconhecer que, apesar das associações identificadas, uma compreensão completa das relações e causas entre os *test smells* requer investigações mais aprofundadas. Esta tese é um ponto de partida, enfatizando a necessidade de estudos futuros para compreender completamente as nuances dessas associações e orientar práticas de refatoração mais

eficientes.

QP3 Quais as associações significativas entre os *test smells* detectados e métricas de qualidade de *software*?

Os *test smells* emergem como indicadores críticos de potenciais deficiências no design e implementação das suítes de testes. Essas deficiências têm um impacto direto na qualidade, especialmente na legibilidade e manutenibilidade dos testes. Consequentemente, é lógico inferir uma correlação destes *smells* com métricas estabelecidas de qualidade de *software*. Nos nossos resultados, o *test smell Conditional Test* demonstrou correlações positivas fortes com todas as métricas de qualidade examinadas, particularmente exibindo coeficientes de 0.694 com a *Maintainability* e 0.813 com a *Cyclomatic Complexity*.

Além disso, o *Duplicate Assert*, outro *test smell*, também apresentou associações significativas, especialmente com métricas como *Halstead Effort* e *Halstead Time*, ambas com coeficientes de 0.670. A inter-relação observada entre os *test smells* e as métricas de qualidade enfatiza a relevância destes *smells* como indicativos de problemas existentes nos testes, servindo como um guia valioso para estratégias direcionadas de refatoração. Contudo, é importante salientar a necessidade de mais pesquisas envolvendo uma variedade maior de projetos para aperfeiçoar a nossa compreensão sobre as associações entre *test smells* e métricas de qualidade de *software*.

8.2 Trabalhos Futuros

A pesquisa desenvolvida forneceu percepções valiosas sobre *test smells* em códigos de testes *JavaScript*. No entanto, há diversos aspectos que podem ser aprofundados em estudos futuros:

- Explorar métodos experimentais ou estudos longitudinais que possam estabelecer uma relação causal entre *test smells* e métricas de qualidade. Além disso, a integração de análises qualitativas, mediante entrevistas ou estudos de caso, poderia fornecer informações sobre as experiências práticas dos desenvolvedores em

relação aos *test smells*. Adicionalmente, a avaliação do impacto dos *test smells* em diferentes contextos de projeto e domínios de aplicação pode enriquecer nossa compreensão sobre sua influência na qualidade do software.

- Dada a utilização da técnica de análise estática na ferramenta desenvolvida neste estudo, sugere-se que sejam desenvolvidos estudos futuros que analisem métodos de detecção mais precisos para diminuir a ocorrência de falsos positivos. A integração de técnicas de análise dinâmica, juntamente com aprendizado de máquina, pode oferecer uma maior precisão na identificação de *smells* nos testes. Além disso, a validação dessas abordagens em diferentes contextos de projetos pode proporcionar ferramentas mais robustas e confiáveis para as equipes de desenvolvimento.

- Expansão do catálogo de *test smells*, incorporando não apenas os *smells* clássicos que não foram abordados neste estudo, mas também identificando e caracterizando novos *smells* intrínsecos às particularidades do JavaScript. Dada a natureza dinâmica e versátil do JavaScript, é provável que existam *smells* específicos que mereçam atenção. Uma análise sistemática dessas peculiaridades pode enriquecer significativamente o corpo de conhecimento sobre *test smells* na linguagem.

- Para aprofundar nossa compreensão sobre a influência dos *test smells* em *JavaScript*, seria valioso investigar se esses padrões também se manifestam em projetos desenvolvidos na linguagem *Typescript*. Além disso, uma análise comparativa entre os *test smells* em ambos os idiomas poderia fornecer um entendimento sobre a eficácia das tipagens estáticas na prevenção desses problemas. Por fim, seria interessante avaliar se as ferramentas de detecção de *test smells* para *JavaScript* são igualmente eficazes para *Typescript*.

Em resumo, esta tese forneceu uma base sólida para a compreensão e abordagem dos *test smells* em códigos de testes *JavaScript*. As possibilidades para futuras pesquisas nesta área são vastas, prometendo avanços significativos na qualidade e eficácia dos testes de *software* em *JavaScript*.

Referências Bibliográficas

[1] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[2] D. Flanagan and W. S. Like, *JavaScript: The Definitive Guide, 7th*. Sebastopol: O'Reilly Media, 2020.

[3] S. Overflow, "2019 developer survey results," https://insights.stackoverflow.com/survey/2019, acessado em: 08-10-2020.

[4] GitHut, "A small place to discover languages in github," http://githut.info, acessado em: 08-10-2020.

[5] RedMonk, "The redmonk programming language rankings: January 2020," https://redmonk.com/sogrady/2020/02/28/language-rankings-1-20/, acessado em: 08-10-2020.

[6] A. M. Fard and A. Mesbah, "JSNOSE: Detecting javascript code smells," *IEEE 13th International Working Conference on Source Code Analysis and Manipulation, SCAM 2013*, pp. 116–125, 2013.

[7] Mozilla, "Object prototypes," https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes, acessado em: 08-10-2020.

[8] A. Koenig, "Patterns and antipatterns," *The patterns handbook: techniques, strategies, and applications*, vol. 13, p. 383, 1998.

[9] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok, "Refactoring Test Code," in *Proceedings 2nd International Conference on Extreme Programming and*

Flexible Processes in Software Engineering (XP2001), M. Marchesi and G. Succi, Eds., 2001.

[10] V. Garousi and B. Küçük, "Smells in software test code: A survey of knowledge in industry and academia," *Journal of Systems and Software*, vol. 138, pp. 52–81, apr 2018. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0164121217303060

[11] M. Alfadel, A. Kobilica, and J. Hassine, "Evaluation of halstead and cyclomatic complexity metrics in measuring defect density," *2017 9th IEEE-GCC Conference and Exhibition, GCCCE 2017*, 2018.

[12] J. L. Anderson, "Using software tools and metrics to produce better quality test software," *AUTOTESTCON (Proceedings)*, pp. 293–297, 2004.

[13] T. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, dec 1976. [Online]. Available: http://ieeexplore.ieee.org/document/1702388/

[14] M. H. Halstead, "Toward a theoretical basis for estimating programming effort," *Proceedings of the 1975 Annual Conference, ACM 1975*, pp. 222–224, 1975.

[15] F. Trautsch and J. Grabowski, "Are There Any Unit Tests? An Empirical Study on Unit Testing in Open Source Python Projects," in *Proceedings - 10th IEEE International Conference on Software Testing, Verification and Validation, ICST 2017*. Institute of Electrical and Electronics Engineers Inc., may 2017, pp. 207–218.

[16] F. Trautsch, S. Herbold, and J. Grabowski, "Are unit and integration test definitions still valid for modern Java projects? An empirical study on open-source projects," *Journal of Systems and Software*, vol. 159, jan 2020.

[17] C. Baier and J.-P. Katoen, *Principles Of Model Checking*. MIT Press, 2008.

[18] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge University Press, dec 2016.

[19] J. D. McGregor and D. A. Sykes, *A practical guide to testing object-oriented software*. Addison-Wesley Professional, 2001.

[20] F. ISTQB, "Foundation level syllabus version 2011," *International Software Testing Qualifications Board*, 2011.

[21] P. Jorgensen and P. C., *Software testing : a craftsman's approach*. Auerbach Publications, 2008.

[22] J. Tretmans, "Model-based testing: Property checking for real," in *International Workshop for Construction and Analysis of Safe Secure, and Interoperable Smart Devices*, 2004.

[23] G. Meszaros, *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[24] V. Khorikov, *Unit Testing Principles, Practices, and Patterns*. Manning Publications, 2020. [Online]. Available: https://books.google.com.br/books?id=CbvZyAEACAAJ

[25] L. Moonen, A. Van Deursen, A. Zaidman, and M. Bruntink, "On the Interplay Between Software Testing and Evolution and its Effect on Program Comprehension," in *Software Evolution*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 173–202. [Online]. Available: http://link.springer.com/10.1007/978-3-540-76440-3_8

[26] B. Laboon, *A Friendly Introduction to Software Testing*. CreateSpace Independent Publishing Platform, 2016.

[27] M. Abbes, F. Khomh, Y. G. Guéhéneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, Blob and Spaghetti Code, on program comprehension," *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR*, pp. 181–190, 2011.

[28] F. Khomh, M. D. Penta, Y. G. Guéhéneuc, and G. Antoniol, "An exploratory study of the impact of antipatterns on class change- and fault-proneness," *Empirical Software Engineering*, vol. 17, no. 3, pp. 243–275, 2012.

[29] M. Fowler, *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 2018.

[30] W. J. Brown, R. C. Malveau, T. J. Mowbray, and J. Wiley, "AntiPatterns: Refactoring Software , Architectures, and Projects in Crisis," *Crisis*, 1998.

[31] W. F. Opdyke and R. E. Johnson, "Refactoring: an aid in designing application frameworks and evolving object-oriented systems," *Proc. SOOPPA '90 : Symposium on Object-Oriented Programming Emphasizing Practical Applications*, 1990.

[32] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "An empirical analysis of the distribution of unit test smells and their impact on software maintenance," *IEEE International Conference on Software Maintenance, ICSM*, pp. 56–65, 2012.

[33] S. Reichhart, T. Gîrba, and S. Ducasse, "Rule-based Assessment of Test Quality." *The Journal of Object Technology*, vol. 6, no. 9, p. 231, 2007. [Online]. Available: http://www.jot.fm/contents/issue_2007_10/paper12.html

[34] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "On the Distribution of Test Smells in Open Source Android Applications: An Exploratory Study," *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, no. November, pp. 193–202, 2019. [Online]. Available: http://dl.acm.org/citation.cfm?id=3370272.3370293

[35] A. S. A. Peruma, "What the Smell ? An Empirical Investigation on the Distribution and Severity of Test Smells in Open Source Android Applications What the Smell ? An Empirical Investigation on the Distribution and Severity of Test Smells in Open Source Android Applications," Ph.D. dissertation, Rochester Institute of Technology, 2018.

[36] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba, "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.*

Virtual Event USA: ACM, Nov. 2020, pp. 1650–1654. [Online]. Available: https://dl.acm.org/doi/10.1145/3368089.3417921

[37] T. Virgínio, L. Martins, R. Santana, A. Cruz, L. Rocha, H. Costa, and I. Machado, "On the test smells detection: an empirical study on the JNose Test accuracy," *Journal of Software Engineering Research and Development*, vol. 9, Sep. 2021. [Online]. Available: https://sol.sbc.org.br/journals/index.php/jserd/article/view/1893

[38] T. Wang, Y. Golubev, O. Smirnov, J. Li, T. Bryksin, and I. Ahmed, "PyNose: A Test Smell Detector For Python," 2021. [Online]. Available: https://arxiv.org/abs/2108.04639v1

[39] A. Bodea, "Pytest-smell: A smell detection tool for python unit tests," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 793–796. [Online]. Available: https://doi.org/10.1145/3533767.3543290

[40] H. Li and W. Cheung, "An Empirical Study of Software Metrics," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 6, pp. 697–708, jun 1987. [Online]. Available: http://ieeexplore.ieee.org/document/1702275/

[41] A. Abran, *Software Metrics and Software Metrology*. Hoboken, NJ, USA: John Wiley & Sons, Inc., may 2010.

[42] A. Fitzsimmons and T. Love, "A Review and Evaluation of Software Science," *ACM Computing Surveys*, vol. 10, no. 1, pp. 3–18, Mar. 1978. [Online]. Available: https://dl.acm.org/doi/10.1145/356715.356717

[43] S. A. Abdulkareem and A. J. Abboud, "Evaluating Python, C++, JavaScript and Java Programming Languages Based on Software Complexity Calculator (Halstead Metrics)," *IOP Conference Series: Materials Science and Engineering*, vol. 1076, no. 1, p. 012046, Feb. 2021. [Online]. Available: https://iopscience.iop.org/article/10.1088/1757-899X/1076/1/012046

[44] R. Tavares Coimbra, A. M. P. De Resende, and R. Terra, "A correlation analysis between halstead complexity measures and other software measures," *Proceedings - 2018 44th Latin American Computing Conference, CLEI 2018*, pp. 31–39, 2018.

[45] D. Coleman, D. Ash, B. Lowther, and P. Oman, "Using metrics to evaluate software system maintainability," *Computer*, vol. 27, no. 8, pp. 44–49, Aug. 1994, conference Name: Computer.

[46] G. Rosu, M. Di Penta, T. N. Nguyen, ACM Sigsoft, ACM SIGAI, IEEE Computer Society, and Institute of Electrical and Electronics Engineers, "Detecting Fragile Comments," in *ASE'17 : proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering : October 30 - November 3, 2017, Urbana-Champaign, IL, USA*, 2017.

[47] L. Tan, D. Yuan, G. Krishna, and Y. Zhou, "/* iComment: Bugs or Bad Comments? */," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles - SOSP '07*. New York, New York, USA: ACM Press, 2007, p. 145. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1294261.1294276

[48] A. Rahman, C. Parnin, and L. Williams, "The Seven Sins: Security Smells in Infrastructure as Code Scripts," in *Proceedings - International Conference on Software Engineering*, vol. 2019-May. IEEE Computer Society, may 2019, pp. 164–175.

[49] D. Jorge, P. Machado, and W. Andrade, "Investigating Test Smells in JavaScript Test Code," in *Brazilian Symposium on Systematic and Automated Software Testing*. Joinville Brazil: ACM, Sep. 2021, pp. 36–45. [Online]. Available: https://dl.acm.org/doi/10.1145/3482909.3482915

[50] W. Aljedaani, A. Peruma, A. Aljohani, M. Alotaibi, M. W. Mkaouer, A. Ouni, C. D. Newman, A. Ghallab, and S. Ludi, "Test Smell Detection Tools: A Systematic Mapping Study," pp. 170–180, 2021.

[51] N. S. Junior, L. Rocha, L. A. Martins, and I. Machado, "A survey on test practitioners' awareness of test smells," 2020. [Online]. Available: http://arxiv.org/abs/2003.05613

[52] E. Soares, M. Ribeiro, G. Amaral, R. Gheyi, L. Fernandes, A. Garcia, B. Fonseca, and A. Santos, "Refactoring Test Smells: A Perspective from Open-Source Developers," in *Proceedings of the 5th Brazilian Symposium on Systematic and Automated Software Testing*. Natal Brazil: ACM, Oct. 2020, pp. 50–59. [Online]. Available: https://dl.acm.org/doi/10.1145/3425174.3425212

[53] H. Damasceno, C. Bezerra, E. Coutinho, and I. Machado, "Analyzing Test Smells Refactoring from a Developers Perspective," in *Proceedings of the XXI Brazilian Symposium on Software Quality*. Curitiba Brazil: ACM, Nov. 2022, pp. 1–10. [Online]. Available: https://dl.acm.org/doi/10.1145/3571473.3571487

[54] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "An empirical investigation into the nature of test smells," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*. New York, New York, USA: ACM Press, 2016, pp. 4–15. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2970276.2970340

[55] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "Are test smells really harmful? An empirical study," *Empirical Software Engineering*, vol. 20, no. 4, pp. 1052–1094, aug 2015. [Online]. Available: http://link.springer.com/10.1007/s10664-014-9313-0

[56] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli, "On the Relation of Test Smells to Software Code Quality," *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 1–12, 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8529832/

[57] A. Saboury, P. Musavi, F. Khomh, and G. Antoniol, "An empirical study of code smells in JavaScript projects," *SANER 2017 - 24th IEEE International Conference on Software Analysis, Evolution, and Reengineering*, pp. 294–305, 2017.

[58] D. Johannes, F. Khomh, and G. Antoniol, "A large-scale empirical study of code smells in JavaScript projects," *Software Quality Journal*, vol. 27, no. 3, pp. 1271–1314, sep 2019. [Online]. Available: http://link.springer.com/10.1007/s11219-019-09442-9

[59] I. Zozas, S. Bibi, and A. Ampatzoglou, "Forecasting the Principal of Code Technical Debt in JavaScript Applications," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 2498–2512, Apr. 2023. [Online]. Available: https://ieeexplore.ieee.org/document/9954175/

[60] I. Malavolta, K. Nirghin, G. L. Scoccia, S. Romano, S. Lombardi, G. Scanniello, and P. Lago, "JavaScript Dead Code Identification, Elimination, and Empirical Assessment," *IEEE Transactions on Software Engineering*, vol. 49, no. 7, pp. 3692–3714, Jul. 2023. [Online]. Available: https://ieeexplore.ieee.org/document/10108937/

[61] R. T. Ogawa and B. Malen, "Towards Rigor in Reviews of Multivocal Literatures: Applying the Exploratory Case Study Method," *Review of Educational Research*, vol. 61, no. 3, pp. 265–286, 1991.

[62] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013.

[63] B. a. Kitchenham and S. L. Pfleeger, "Principles of Survey Research Part 2 : Designing a Survey," *Software Engineering Notes*, vol. 27, no. 1, pp. 18–20, 2002. [Online]. Available: http://portal.acm.org/citation.cfm?doid=566493.566495

Apêndice A

Test Smells na Linguagem *JavaScript*

Estamos conduzindo um estudo exploratório sobre *Test Smells* na linguagem *JavaScript* e gostaríamos de convidá-lo(a) a participar. Seu conhecimento e experiência são fundamentais para entendermos melhor este fenômeno e suas implicações na qualidade do código e na eficácia dos testes.

Esta pesquisa é conduzida por Dalton Nicodemos Jorge, doutorando do Programa de Pós-Graduação em Ciência da Computação pela Universidade Federal de Campina Grande (UFCG) e pesquisador do Laboratório de Práticas de Software (SPLab). Todas as respostas serão tratadas com confidencialidade e os resultados serão utilizados exclusivamente para fins científicos.

O questionário avalia trechos de código *JavaScript* e coleta opiniões em formatos de múltipla escolha e seleção, com opção para respostas abertas em 'Outros'. Assim, o questionário está dividido em 3 etapas:

1. Formação acadêmica e profissional (4 perguntas);
2. Identificação de Test Smells (8 perguntas);
3. Questões de opinião do participante (10 perguntas);

Etapa 1 - Formação acadêmica e profissional

1. Qual seu nível acadêmico?
    - ○ Graduação em Andamento
    - ○ Graduação
    - ○ Pós-graduação (lato sensu) em Andamento
    - ○ Pós-graduação (lato sensu)
    - ○ Mestrado em Andamento
    - ○ Mestrado
    - ○ Doutorado em Andamento
    - ○ Doutorado
    - ○ Outro:
2. Como você descreve sua atual vinculação profissional?

- ○ Funcionário(a) de empresa no setor público
- ○ Funcionário(a) de empresa no setor privado
- ○ Profissional bolsista atuando em Convênio Empresa - Universidade
- ○ Estudante bolsista atuando em Convênio Empresa - Universidade
- ○ Desenvolvedor(a) Autônomo(a)/Freelancer
- ○ Outro:

3. Quanto tempo de experiência você possui com desenvolvimento de *software*?

- ○ Sem experiência
- ○ 1 - 6 meses
- ○ 6 meses - 1 ano
- ○ 1 ano - 2 anos
- ○ 2 anos a 3 anos
- ○ Mais de 3 anos

4. Quanto tempo de experiência você possui com testes de *software*?

- ○ Sem experiência
- ○ 1 - 6 meses
- ○ 6 meses - 1 ano
- ○ 1 ano - 2 anos
- ○ 2 anos a 3 anos
- ○ Mais de 3 anos

Etapa 2 - Identificação de Test Smells

Os casos de testes apresentados neste formulário foram extraídos dos seguintes projetos *open-source* disponíveis no *Github*:

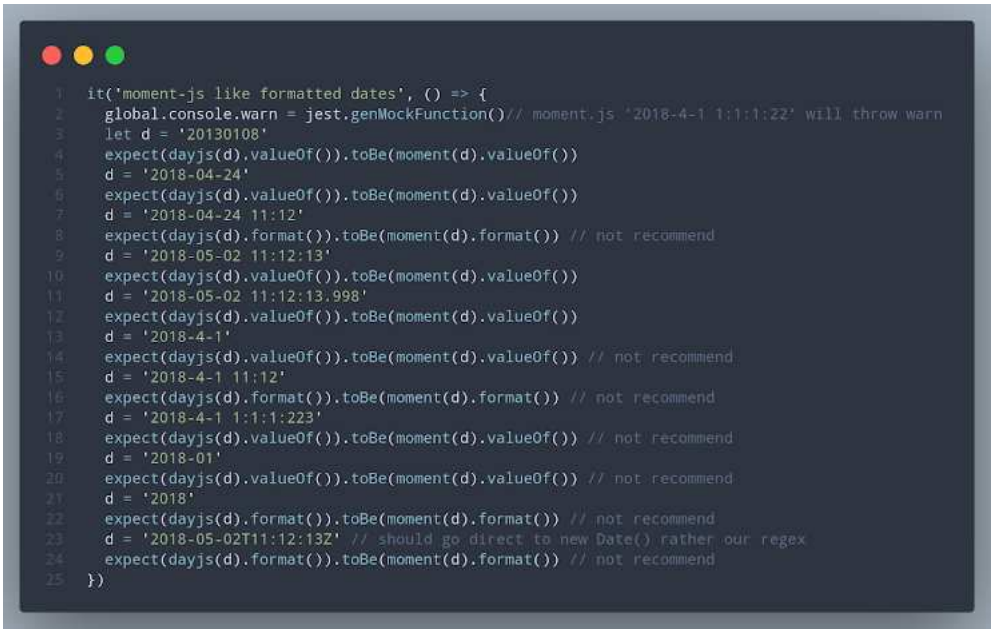
- `https://github.com/iamkun/dayjs`
- `https://github.com/balderdashy/sails`
- `https://github.com/nuxt/nuxt.js`
- `https://github.com/facebook/react`

Para as questões 5 à 14 adote as seguintes alternativas: * Marque todas que se aplicam.

- □ *Assertion Roulette* - Muitas assertions sem mensagem clara.
- □ *Conditional Test Logic* - Lógica condicional e/ou repetição nos testes.
- □ *Duplicate Asserts* - Mesma assertion repetida no teste.
- □ *Eager Test* - Teste que exercita múltiplos métodos do sistema sob teste.
- □ *Empty Test* - Teste sem assertions ou verificações.
- □ *Exception Handling* - Exceções não tratadas adequadamente.
- □ *Ignored Test* - Partes do teste são ignoradas.

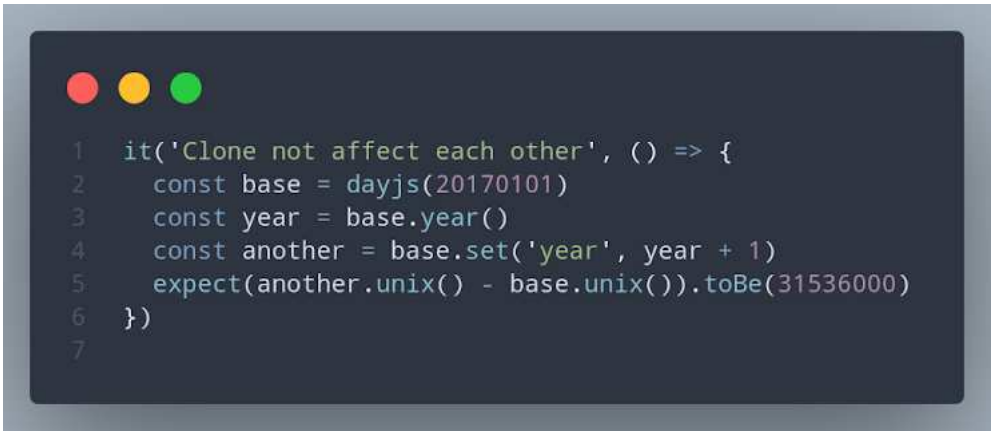
- ☐ *Lazy Test* - Múltiplos testes exercitam mesmo método do sistema sob teste.
- ☐ *Magic Number Test* - Uso de números mágicos no teste.
- ☐ *Mystery Test* - Teste depende de recursos externos.
- ☐ *Redundant Assertion* - Asserções que sempre retornam mesmo resultado.
- ☐ *Redundant Print* - Impressões de debug desnecessárias.
- ☐ *Resource Optimism* - Teste assume recurso externo disponível sem verificar.
- ☐ *Sleepy Test* - Teste inclui pausa explícita durante execução.
- ☐ *Unknown Test* - Teste não contém nenhuma asserção.
- ☐ Nenhum *smell*.
- ☐ Outro:

5. Avalie o caso de teste abaixo (Day.js - parse.test.js) e responda qual ou quais *smells* você detectou:



```
it('moment-js like formatted dates', () => {
  global.console.warn = jest.genMockFunction()// moment.js '2018-4-1 1:1:1:22' will throw warn
  let d = '20130108'
  expect(dayjs(d).valueOf()).toBe(moment(d).valueOf())
  d = '2018-04-24'
  expect(dayjs(d).valueOf()).toBe(moment(d).valueOf())
  d = '2018-04-24 11:12'
  expect(dayjs(d).format()).toBe(moment(d).format()) // not recommend
  d = '2018-05-02 11:12:13'
  expect(dayjs(d).valueOf()).toBe(moment(d).valueOf())
  d = '2018-05-02 11:12:13.998'
  expect(dayjs(d).valueOf()).toBe(moment(d).valueOf())
  d = '2018-4-1'
  expect(dayjs(d).valueOf()).toBe(moment(d).valueOf()) // not recommend
  d = '2018-4-1 11:12'
  expect(dayjs(d).format()).toBe(moment(d).format()) // not recommend
  d = '2018-4-1 1:1:1:223'
  expect(dayjs(d).valueOf()).toBe(moment(d).valueOf()) // not recommend
  d = '2018-01'
  expect(dayjs(d).valueOf()).toBe(moment(d).valueOf()) // not recommend
  d = '2018'
  expect(dayjs(d).format()).toBe(moment(d).format()) // not recommend
  d = '2018-05-02T11:12:13Z' // should go direct to new Date() rather our regex
  expect(dayjs(d).format()).toBe(moment(d).format()) // not recommend
})
```

6. Avalie o caso de teste abaixo (Day.js - parse.test.js) e responda qual ou quais *smells* você detectou:



```
it('Clone not affect each other', () => {
  const base = dayjs(20170101)
  const year = base.year()
  const another = base.set('year', year + 1)
  expect(another.unix() - base.unix()).toBe(31536000)
})
```

7. Avalie o caso de teste abaixo (Day.js - browser.spec.js) e responda qual ou quais *smells* você detectou:

```
describe('Install', function () {
  it('window.dayjs ', function () {
    if (!window.dayjs) throw new Error('No window.dayjs')
  })
})
```

8. Avalie o caso de teste abaixo (Sails.js - app.getRouteFor.test.js) e responda qual ou quais *smells* você detectou:

```
it('should throw usage error (i.e. `e.code===\'E_USAGE\'`) if target to search for not specified or is invalid', function (){
    try {
      app.getRouteFor();
      assert(false, 'Should have thrown an error');
    }
    catch (e) {
      if (e.code !== 'E_USAGE') { assert(false, 'Should have thrown an error w/ code === "E_USAGE"'); }
    }

    try {
      app.getRouteFor(3235);
      assert(false, 'Should have thrown an error');
    }
    catch (e) {
      if (e.code !== 'E_USAGE') { assert(false, 'Should have thrown an error w/ code === "E_USAGE"'); }
    }

    try {
      app.getRouteFor({ x: 32, y: 49 });
      assert(false, 'Should have thrown an error');
    }
    catch (e) {
      if (e.code !== 'E_USAGE') { assert(false, 'Should have thrown an error w/ code === "E_USAGE"'); }
    }

    try {
      app.getRouteFor(function(){});
      assert(false, 'Should have thrown an error');
    }
    catch (e) {
      if (e.code !== 'E_USAGE') { assert(false, 'Should have thrown an error w/ code === "E_USAGE"'); }
    }

  });
```

9. Avalie o caso de teste abaixo (Sails.js - app.initializeHooks.test.js) e responda qual ou quais *smells* você detectou:

```
it('should bind the routes in the correct order', function(done) {
  sails.request({
    method: 'get',
    url: '/foo'
  }, function (err, res, body) {
    if (err) return done(err);
    assert.equal(res.statusCode, 200);
    assert.equal(body, 'abc');
    return done();
  });
})
```

10. Avalie o caso de teste abaixo (Nuxt.js - custom-dirs.test.js) e responda qual ou quais *smells* você detectou:

```
test('custom assets directory', async () => {
  const readFile = promisify(fs.readFile)

  const extractedIndexCss = resolve(__dirname, '..', 'fixtures/custom-dirs/.nuxt/dist/client/app.css')
  const content = await readFile(extractedIndexCss, 'utf-8')

  expect(content).toContain('.global-css-selector{color:red}')
})
```

11. Avalie o caso de teste abaixo (Nuxt.js - dev.test.js) e responda qual ou quais *smells* você detectou:

```
test.skip('catches watchRestart error', async () => {
  const Nuxt = mockNuxt()
  const Builder = mockBuilder()

  await NuxtCommand.from(dev).run()
  jest.clearAllMocks()

  const builder = new Builder()
  builder.nuxt = new Nuxt()
  Builder.prototype.watchRestart = jest.fn().mockImplementationOnce(() => Promise.reject(new Error('watchRestart Error')))
  await Nuxt.fileRestartHook(builder)

  expect(consola.error).toHaveBeenCalledWith(new Error('watchRestart Error'))
  expect(Builder.prototype.watchRestart).toHaveBeenCalledTimes(2)
})
```

12. Avalie o caso de teste abaixo (React - ReactIncrementalTriangle-test.internal.js) e responda qual ou quais *smells* você detectou:

```
it('generative tests', () => {
  const limit = 100;
  for (let i = 0; i < limit; i++) {
    const actions = randomActionsPerRoot();
    try {
      simulateMultipleRoots(...actions);
    } catch (e) {
      console.error(
        `Triangle fuzz tester error! Copy and paste the following line into the test suite:
${formatActionsPerRoot(actions)}
        `,
      );
      throw e;
    }
  }
});
```

Etapa 3 - Questões de opinião do participante

13. Diante dos casos de teste avaliados, quais os *smells* que você julga os mais problemáticos para a qualidade dos testes?

14. Algum smell poderia permanecer nos casos de testes sem comprometer a qualidade? Qual(is)?

15. A existência dos *smells* pode trazer consequências negativas ao projeto. Sobre esta afirmação, você:

    - 1. Discordo totalmente
    - 2. Discordo parcialmente

- ○ 3. Não concordo e nem discordo
- ○ 4. Concordo parcialmente
- ○ 5. Concorda totalmente

16. Atenção: ignore esta questão caso sua resposta anterior tenha sido "(1) Discordo totalmente". Quais são os principais efeitos negativos de *test smells* em projetos de *software*?

- □ Aumentam custos de manutenção dos testes.
- □ Geram maior fragilidade e quebra frequente de testes.
- □ Tornam os testes pouco confiáveis e mascaram bugs.
- □ Dificultam refatoração e evolução do código.
- □ Aumentam dívida técnica ao longo do tempo.
- □ Pioram experiência do desenvolvedor com a base de testes.
- □ Aumentam tempo de *debug* e reduzem produtividade.
- □ Degradam cobertura e valor dos testes automatizados.
- □ Causam falhas intermitentes e resultados não determinísticos.
- □ Dificultam a compreensão do teste.
- □ Nenhum efeito negativo.
- □ Outro:

17. Quão fácil ou difícil foi a detecção dos *smells*?

- ○ 1. Muito difícil
- ○ 2. Difícil
- ○ 3. Nem fácil e nem difícil
- ○ 4. Fácil
- ○ 5. Muito fácil

18. Desenvolvedores precisam de treinamento em *test smells*. Sobre esta afirmação, você:

- ○ 1. Discordo totalmente
- ○ 2. Discordo parcialmente
- ○ 3. Não concordo e nem discordo
- ○ 4. Concordo parcialmente
- ○ 5. Concorda totalmente

19. Ferramentas de análise estática podem ser úteis na identificação de *test smells*. Sobre esta afirmação, você:

- ○ 1. Discordo totalmente
- ○ 2. Discordo parcialmente
- ○ 3. Não concordo e nem discordo
- ○ 4. Concordo parcialmente

- ○ 5. Concorda totalmente

20. Quais são as principais dificuldades que você enfrenta para identificar e corrigir *test smells* em projetos de *software*?

    - □ Desconhecimento de boas práticas e dos tipos de smells.
    - □ Complexidade do sistema sob teste.
    - □ Falta de ferramentas apropriadas.
    - □ Testes mal escritos.
    - □ Resistência à cultura de testes por parte da equipe.
    - □ Restrições de tempo e/ou recursos.
    - □ Inexistência ou escassez de documentação.
    - □ Não enfrento dificuldades.
    - □ Outro:

21. Quais são as principais estratégias que você julga servirem para evitar ou corrigir *test smells* em projetos de *software*?

    - □ Uso de ferramentas de análise estática nos testes.
    - □ Automação para identificar facilmente quebras e falhas intermitentes.
    - □ Padronização da escrita e organização de testes.
    - □ *Code reviews* focados na qualidade dos testes.
    - □ Treinamento da equipe em boas práticas de testes.
    - □ Coleta de métricas e monitoramento contínuo da qualidade.
    - □ Outro:

22. Gostaria de acrescentar algo (experiências, relatos, críticas, sugestões, etc.)?