UNIVERSIDADE ESTADUAL DA PARAÍBA
CAMPUS I
CENTRO DE CIÊNCIAS E TECNOLOGIA
DEPARTAMENTO DE COMPUTAÇÃO
CURSO DE BACHARELADO EM COMPUTAÇÃO

JARDESSON ELLÍUDO LÁZARO DA COSTA

AVALIAÇÃO DE TEST SMELLS EM TESTES DE API NO CONTEXTO DE EVOLUÇÃO

CAMPINA GRANDE
2023

JARDESSON ELLÍUDO LÁZARO DA COSTA

AVALIAÇÃO DE TEST SMELLS EM TESTES DE API NO CONTEXTO DE EVOLUÇÃO

Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Computação do Departamento de Computação do Centro de Ciências e Tecnologia da Universidade Estadual da Paraíba, como requisito parcial à obtenção do título de bacharel em Computação.

Área de concentração: Testes de Software

Orientador: Dra. Sabrina de Figueiredo Souto

CAMPINA GRANDE
2023

C838a Costa, Jardesson Elliudo Lazaro da.

Avaliação de *test smells* em testes de API no contexto de evolução [manuscrito] / Jardesson Elliudo Lazaro da Costa. - 2023.

69 p. : il. colorido.

Digitado.

Trabalho de Conclusão de Curso (Graduação em Computação) - Universidade Estadual da Paraíba, Centro de Ciências e Tecnologia, 2023.

"Orientação : Profa. Dra. Sabrina de Figueiredo Souto, Coordenação do Curso de Computação - CCT. "

1. Desenvolvimento de software. 2. Teste de software. 3. Testes de API. I. Título

21. ed. CDD 005.3

Elaborada por Talita M. A. Tavares - CRB - CRB 15/971 BC/UEPB

JARDESSON ELLÍUDO LÁZARO DA COSTA

AVALIAÇÃO DE TEST SMELLS EM TESTES DE API NO CONTEXTO DE EVOLUÇÃO

Trabalho de Conclusão de Curso de Graduação em Ciência da Computação da Universidade Estadual da Paraíba, como requisito à obtenção do título de Bacharel em Ciência da Computação.

Aprovada em 03 de Julho de 2023.

Sabrina de F. Souto.

Profa. Dra. Sabrina de Figueiredo Souto (CCT/UEPB)
Orientador(a)

Luciana de Queiroz Leal Gomes

Profa. Me. Luciana de Queiroz Leal Gomes (CCT/UEPB)
Examinador(a)

José Glauber Braz de Oliveira

José Glauber Braz de Oliveira (UFCG)
Examinador(a)

A Deus, por me conceder forças e sabedoria durante todo o processo, a minha família, que sempre esteve presente, apoiando-me em todas as etapas desta jornada e a meus amigos pelo companheirismo e amizade, DEDICO.

RESUMO

A garantia da qualidade dos testes é uma prática essencial no desenvolvimento de software. No entanto, testes de má qualidade podem levar a uma série de problemas, como aumento de custos, atrasos na entrega e falhas no software em produção. Nesse contexto, surgem os test smells, que são sinais de problemas potenciais nos testes. O objetivo geral deste trabalho é identificar problemas de design e padrões comuns que podem afetar negativamente a eficiência e a eficácia dos testes de API através da análise de test smells. Para isso, foi feito um estudo empírico com dois projetos de software reais Regnutes e Priorize. será utilizada uma abordagem que envolve o uso de ferramentas automatizadas para auxiliar na detecção dos test smells.Observamos um aumento de test smells no projeto Priorize, enquanto no Regnutes houve uma diminuição. Isso sugere a importância de uma abordagem proativa na prevenção e correção desses problemas ao longo da evolução do software. Esses resultados reforçam a relevância de se realizar uma avaliação contínua dos testes de API, buscando identificar e corrigir os test smells para melhorar a qualidade dos testes e, consequentemente, do software. Além disso, a utilização de ferramentas automatizadas para auxiliar na detecção dos test smells pode ser uma estratégia eficaz para facilitar o processo de análise e refatoração dos códigos de teste.

Palavras-chave: Test Smells, Testes de API, Garantia de Qualidade, Evolução.

ABSTRACT

Quality assurance is an essential practice in software development. However, poor-quality tests can lead to various problems, such as increased costs, delivery delays, and software failures in production. In this context, test smells emerge as potential signs of issues in tests. The overall objective of this work is to identify common design problems and patterns that can negatively impact the efficiency and effectiveness of API tests through the analysis of test smells. To achieve this, an empirical study was conducted with two real software projects, Regnutes and Priorize. An approach involving the use of automated tools to assist in detecting test smells was employed. An increase in test smells was observed in the Priorize project, while a decrease was noticed in the Regnutes project. This suggests the importance of a proactive approach in preventing and correcting these problems throughout software evolution. These results reinforce the relevance of conducting continuous evaluation of API tests, aiming to identify and address test smells to enhance test quality and, consequently, software quality. Furthermore, the utilization of automated tools to assist in detecting test smells can be an effective strategy to facilitate the process of analysis and refactoring of test code.

Keywords: Test Smells, API testing, Quality Assurance, Evolution.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

A qualidade dos produtos e serviços é muito enfatizada na sociedade atual, e temos visto melhorias significativas em seu nível durante as últimas décadas. O desenvolvimento de software é um processo de lidar com muitos riscos. O objetivo da garantia da qualidade de software é reduzir muitos desses riscos, principalmente o risco de não produzir um produto de qualidade, especialmente para sistemas críticos, como áreas médicas ou biomédicas (ANJARD, 1992). “O software não é diferente de outros processos físicos onde as entradas são recebidas e as saídas são produzidas. Onde o software difere é na maneira como ele falha.” (TUTEJA, DUBEY, 2012). O princípio KISS (*keep it simple stupid*) significa “mantenha simples, estúpido”. A ideia é que o código fique mais simples possível para que seja mais fácil trabalhar nele mais tarde. Código complexo ou complicado leva mais tempo para ser projetado, escrito e testado (CARULLO, 2020). Problemas de design e evolução de código podem impactar significativamente a qualidade do software. Um design ruim pode tornar o código difícil de entender, testar e manter, enquanto uma evolução inadequada do código pode introduzir bugs e falhas. O uso excessivo de padrões de projeto complexos ou a falta de encapsulamento de dados podem tornar o código difícil de entender e mudar, enquanto o uso excessivo de código duplicado ou a falta de refatoração regular podem levar a problemas de manutenção e escalabilidade. Além disso, um código mal estruturado, com pouco ou nenhum comentário, pode tornar difícil para outros desenvolvedores entender e trabalhar com o código, o que pode atrasar o progresso do projeto e aumentar o risco de bugs (CARULLO, 2020). Portanto, é crucial garantir que o design e a evolução do código sejam cuidadosamente planejados e gerenciados para garantir a qualidade do software.

Um conceito relacionado é o de "code smells", para (FOWLER, 1999) são sinais de que algo está errado em um código. Esses sinais podem ser indicadores de problemas de design ou estruturais que afetam a qualidade do código e a facilidade de manutenção. A detecção e correção de code smells é uma parte importante do processo de manutenção e melhoria contínua de um software. Além dos code smells, os testes também são uma forma de assegurar a qualidade do software. Testes unitários, integração e testes de aceitação são algumas das maneiras de garantir que o software esteja funcionando conforme o esperado. Eles também podem ajudar a

detectar problemas de design e evolução do código. Testar é checar, através de uma execução controlada, se o comportamento corresponde ao especificado (RIOS, MOREIRA, 2013, p. 8,9).

"O teste é o processo de executar um programa com a intenção de encontrar erros" (MYERS; SANDLER; BADGETT, 2011). Os testes são usados amplamente pela indústria de software, e sua relevância é reforçada pelo fato de que muitas empresas e organizações possuem processos de desenvolvimento de software baseados em testes. Os testes são considerados como uma prática essencial para garantir a qualidade do software, pois eles permitem a detecção de problemas de forma precoce, evitando que eles sejam propagados para outras partes do código e tornando-se mais difíceis de serem corrigidos (CRESPO et al., 2004). Como também, os testes ajudam a garantir a confiabilidade e estabilidade do software, o que é crítico para minimizar os erros e diminuir custos (SILVA et al., 2016).

Os problemas comuns no desenvolvimento de testes incluem a escrita de testes ruins ou inadequados, a falta de cobertura de testes, a falta de manutenção de testes e a dificuldade em depurar falhas nos testes (MYERS; SANDLER; BADGETT, 2011). Os testes ruins ou inadequados podem não detectar corretamente falhas no código, o que pode levar a falhas no sistema em produção. A falta de cobertura de testes pode ser um problema se algumas partes do código não forem testadas, o que pode levar a falhas não detectadas. A falta de manutenção de testes pode causar problemas se os testes não estiverem atualizados para refletir mudanças no código (MYERS; SANDLER; BADGETT, 2011). Há a preocupação com a qualidade desses códigos, devido a isso surgiu a Engenharia de Código de Teste de Software (ECTS) (*Software Test-Code Engineering*), que se refere aos métodos, às técnicas e às ferramentas utilizadas para verificar e manter código de teste (YUSIFOĞLU et al., 2015).

Dentro desse contexto, os testes automatizados desempenham um papel fundamental. Alguns de seus benefícios incluem, execução eficiente de testes, é possível rodar os testes inúmeras vezes, a capacidede testar várias situações e condições de forma rápida, e também testar o código sempre que o mesmo é modificado (KRAUSE, 2000). É importante o uso de testes automatizados para

assegurar que os desenvolvedores executem testes com rapidez e precisão. Os testes automatizados podem ser executados com mais frequência e podem ser integrados ao fluxo de trabalho de construção do software, o que permite detectar problemas rapidamente. Outro benefício dos testes automatizados é que eles são mais confiáveis do que os testes manuais, como são executados pelo computador, eles são menos propensos a erros humanos e aceleram acelerar o processo de execução do teste (KUMAR, MISHRA, 2016). Além disso, permitem que os desenvolvedores testem cenários de teste que seriam difíceis de serem reproduzidos manualmente, como testar uma grande quantidade de dados, testar com diferentes configurações, ou testar cenários que envolvem vários dispositivos ou sistemas (KRAUSE, 2000). Em resumo, testes automatizados são importantes porque eles são rápidos, precisos, confiáveis e podem ser integrados ao fluxo de trabalho de construção do software, assegurando assim o aumento da qualidade do software. Os testes automatizados podem ser executados para gerar as validações, além de que o teste manual abre margens para erros, o que poderia ser evitado utilizando o processo automatizado (KUMAR, MISHRA, 2016).

Escrever testes automáticos é desafiador pois requer tempo e esforço para serem criados e mantidos (GRAHAM, FEWSTER, 2012). É necessário garantir que os testes estejam sincronizados com as mudanças no código de produção (MYERS; SANDLER; BADGETT, 2011), e é importante garantir que os testes estejam isolados, evitando dependências indesejadas com outros componentes do sistema. A criação de testes que possuem lógica complexa, ou que sejam dependentes de recursos externos, pode tornar os testes mais difíceis de manter durante a evolução do software de assegurar sua confiabilidade (GRAHAM, FEWSTER, 2012).

Na prática, nem sempre a ECTS é seguida, resultando na inserção de antipadrões em códigos de testes “test smells” (PALOMBA et al., 2018). “Como um tipo de antipadrão, os cheiros de teste são definidos como testes mal projetados e sua presença afeta negativamente a qualidade dos conjuntos de teste e do código de produção.” (GAROUSI et al., 2018). Eles podem ser comparados aos code smells, que são problemas de design ou más práticas que afetam a qualidade do código de produção (FOWLER, 1999). A pesquisa sobre test smells é relativamente nova, começaram por volta de 2001 quando a automação de teste começou a se tornar

popular (GAROUSI et al., 2018), mas tem crescido rapidamente nos últimos anos, com muitos trabalhos publicados em conferências e periódicos de alto impacto. Esses problemas podem tornar os testes menos confiáveis, mais difíceis de entender e manter, e podem até mesmo impedir que os desenvolvedores detectem falhas no código. É importante notar que, assim como os code smells, os test smells não são erros, mas são sinais de que algo pode estar errado e pode precisar ser investigado.

Test smells podem indicar violações de princípios que afetam negativamente a qualidade do design do código de teste. Eles são baseados em princípios de boas práticas de teste, e sua detecção pode ajudar a identificar problemas no código de teste, levando a uma melhoria na qualidade e manutenibilidade dos testes (SPADINI et al., 2018). “Tests smells são mais propensos a mudanças e defeitos, 'Indirect Test', 'Eager Test' e 'Assertion Roulette' são os mais significativos a propensão a mudanças e, o código de produção é mais propenso a defeitos quando testado por testes que contêm test smells” (SPADINI et al., 2018). Não é possível afirmar definitivamente se a evolução do código de teste é a responsável por introduzir test smells (GREILER et al., 2013). No entanto, mesmo que sejam removidos, novos test smells podem ser inseridos (KIM, 2019).

Como os testes são uma parte importante do desenvolvimento de software, é essencial que eles sejam escritos de forma clara, concisa e confiável, e a detecção de test smells pode ajudar a garantir isso, como também pode melhorar a confiança no software e reduzir o esforço de manutenção e evolução do código. Além disso, (ALJEDAANI et al., 2021, p. 1) mostrou que a detecção precoce de maus cheiros reduz os custos de manutenção, destacando a importância de tais ferramentas de detecção. Existem ferramentas que detectam especificamente test smells, enquanto outras detectam code smells, em várias linguagens de programação, e podem ser utilizadas tanto para códigos de teste quanto para códigos de produção. Diante disso, algumas ferramentas foram propostas para automatizar a detecção de test smells. Dentre essas ferramentas, destacam-se JNose Test (VIRGÍNIO et al., 2019), que detecta 21 tipos de test smells e a quantidade de vezes em que eles ocorrem e a ferramenta VITrUM (PECORELLI et al., 2020), que analisa classes de teste e utiliza cores diferentes conforme a presença de test smells, sendo possível filtrar os resultados por período e por tipo de test smells.

A detecção de test smells em códigos JavaScript é importante pois é uma das linguagens de programação mais populares e amplamente utilizadas na atualidade. Está na sétima posição de acordo com o TIOBE index de junho de 2023 e na terceira posição de acordo com o Github index (**Github** e **Tiobe**). Dada a sua crescente adoção no desenvolvimento de soluções para as mais diversas plataformas, o JavaScript tem sido muito empregado para desenvolvimento de aplicações web, aplicativos móveis e outras aplicações. Como muitos projetos usam JavaScript, é importante detectar problemas de design e qualidade no código de teste para garantir que os testes sejam eficazes e confiáveis.

STEEL (JORGE et al., 2021) é uma ferramenta de análise estática para linha de comando que possibilita a detecção automática de smells em códigos de testes na linguagem JavaScript. Ela pode ser usada para identificar problemas de design no código de teste, ajudando a equipe de desenvolvimento a melhorar a qualidade dos testes. Ela pode ser executada em um conjunto de arquivos de teste de unidade, analisando-os e gerando relatórios sobre os problemas encontrados. Isso pode ajudar a equipe a localizar e corrigir problemas de maneira mais eficiente, aumentando a confiabilidade dos testes e a qualidade do software. Além disso, a análise de test smells também pode ajudar a equipe a entender como os testes estão sendo escritos e a identificar boas práticas de teste para serem seguidas no futuro.

Assim como os testes de unidade, os testes de API também podem sofrer de problemas de design e evolução de código, o que pode afetar negativamente a qualidade dos testes e torná-los mais difíceis de manter e evoluir. A detecção e correção de test smells nos testes de API é importante para garantir que os testes estejam sendo escritos de forma eficiente e que possam ser facilmente manuseados e evoluídos ao longo do tempo. Os testes de API são fundamentais para garantir que uma aplicação esteja funcionando corretamente e esteja preparada para lidar com as necessidades dos usuários. Eles verificam se a API está retornando as respostas esperadas, se ela está lidando com as solicitações de forma adequada e se ela está protegida contra erros e falhas (REDDY, 2011).

Considerando o contexto apresentado, a ferramenta Steel será utilizada para detectar test smells neste trabalho. A escolha da ferramenta STEEL para realizar a análise de test smells em JavaScript pode ser justificada por dois motivos:

- Funcionalidade específica: A ferramenta STEEL é projetada especificamente para analisar e detectar test smells em código de teste.
- Suporte para JavaScript: A linguagem JavaScript é amplamente utilizada no desenvolvimento web e de aplicativos. Portanto, é importante ter uma ferramenta que seja capaz de analisar test smells em código escrito em JavaScript.

Ela irá analisar o código dos testes de API, identificando violações de princípios que afetam negativamente a qualidade do design do código de teste. A ferramenta STEEL pode ajudar a encontrar esses problemas de design, facilitando a sua compreensão e, consequentemente, seu uso e evolução. STEEL foi desenvolvida devido a falta de detecção de test smells na linguagem JavaScript. (JORGE et al., 2021). 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.

1.1 Objetivo

O objetivo geral deste trabalho é identificar problemas de design e padrões comuns que podem afetar negativamente a eficiência e a eficácia dos testes de API através da análise de test smells investigando sua evolução, ao longo do desenvolvimento do software, observando possíveis mudanças e padrões. Para tanto, será utilizada uma abordagem que envolve o uso de ferramentas automatizadas para auxiliar na detecção dos test smells. O intuito é compreender melhor como os test smells se desenvolvem e como podem ser mitigados ao longo da evolução dos testes de API. A partir dessa avaliação, analisar os tests smells identificados, investigando suas possíveis causas e discutindo estratégias de melhoria e prevenção. Para isso, foi feito um estudo empírico com dois projetos de software reais (Regnutes e Priorize - mais detalhes na Seção 4.2 - **Projetos**).

Com base no objetivo geral, elencamos os seguintes objetivos específicos:

- Identificar os principais test smells detectados em ambos os projetos;
- Verificar a evolução dos test smells em relação a evolução do software e consequentemente dos testes de API;
- Discutir as melhores práticas para evitar a inserção de test smells em testes de API durante o desenvolvimento;
- Avaliar como o uso de ferramentas automatizadas para detectar test smells pode ajudar a melhorar a qualidade dos testes de API neste estudo.

1.2 Questões de pesquisa

Nesta seção, serão apresentadas as questões que nortearam a pesquisa e cujas respostas foram buscadas ao longo do trabalho. As questões de pesquisa fornecem uma estrutura para a investigação e ajudam a delimitar o escopo do estudo. Elas são formuladas com o objetivo de explorar e obter respostas sobre determinados aspectos do tema em questão.

- Quais os test smells mais frequentes, considerando os sujeitos analisados?

- É possível identificar uma evolução ou regressão dos test smells em relação ao avanço no desenvolvimento dos sujeitos analisados?

- É possível identificar práticas para evitar a inserção de test smells em testes de API durante a evolução?

- O uso de ferramentas automatizadas para detectar test smells pode ajudar a melhorar a qualidade dos testes de API?

1.3 Estrutura do documento

O trabalho segue a estrutura descrita pelos capítulos citados a seguir:

- Capítulo 2: fornece embasamento teórico para compreensão do problema contextualizado neste trabalho;

- Capítulo 3: apresenta a abordagem para identificar e avaliar os test smells;

- Capítulo 4: descreve as questões de pesquisa e a aplicação da abordagem aos objetos de estudo Regnutes e Priorize;

- Capítulo 5: apresenta as considerações finais assumidas após a problematização e a aplicação da abordagem proposta como solução, nos objetos de estudo Regnutes e Priorize, e suas contribuições para o meio de teste de software.

2 REFERENCIAL TEÓRICO

Este capítulo, fornece embasamento teórico para a compreensão de conceitos que são fundamentais para o entendimento do contexto a ser abordado na pesquisa realizada. Inicialmente serão apresentados os conceitos relacionados a Teste de Software e após isso, discutiremos sobre os tipos de test smells e por fim falaremos sobre a ferramenta utilizada.

2.1 Teste de Software

O software deve ser previsível e consistente, não apresentando surpresas aos usuários (MYERS; SANDLER; BADGETT, 2011). Para verificar que o software funciona de acordo com o esperado pelos usuários, é necessário testá-lo. “O teste de software é um processo, ou uma série de processos, projetado para garantir que o código de computador faça o que foi designado para fazer e, contrariamente, que não faça nada que não foi planejado” (MYERS; SANDLER; BADGETT, 2011).

Para melhor encontrar e organizar os erros, os testes de software são separados em tipos diferentes, os mais populares são os testes de caixa branca e caixa preta. No teste de caixa preta, o desenvolvedor responsável pelos testes não tem acesso ao código fonte, nem a qualquer outro detalhe específico de implementação, ele irá se basear nos requisitos e irá validar fluxos comuns aos usuários finais. No teste de caixa branca o profissional responsável tem acesso ao código fonte do sistema e pode observar mais atentamente determinadas etapas do código, como fluxo de dados e entre outros (REDDY, 2011). Os testes de API, estudados neste trabalho, são testes do tipo caixa preta.

2.1.1 Teste de sistema

O teste do sistema não é um processo de testar o funções do sistema ou programa completo, mas comparar o sistema ou programa com seus objetivos originais (MYERS; SANDLER; BADGETT, 2011). Envolve requisitos funcionais e não funcionais e se enquadra no tipo caixa preta. Nesse contexto, o ambiente de teste deve ser o mais semelhante possível ao ambiente de produção, a fim de maximizar a

identificação de falhas específicas de ambiente e para que os testes sejam executados em condições similares àquelas que o usuário final irá utilizar.

2.1.2 Teste de API

Uma API (*Application Programming Interface*) é uma interface de programação que possibilita a comunicação e a troca de dados entre dois sistemas de software diferentes. Uma API contém a lógica de negócios de uma aplicação, e as regras de como os usuários podem interagir com serviços, dados ou funções desta aplicação.

(ISHA; SHARMA e REVATHI, 2018) observam que o número de APIs está se expandindo exponencialmente a cada ano. Consequentemente, realizar testes de APIs tornou-se uma etapa fundamental no processo de desenvolvimento de software, pois APIs ineficazes ou com defeitos podem resultar em menor aquisição de produtos e, finalmente, perda de receita.

O teste de API é crucial e seu objetivo é verificar a funcionalidade, confiabilidade, desempenho e segurança das interfaces de programação. (REDDY, 2011). No teste de API, utiliza-se software para enviar chamadas para a API, obter a saída e validar a resposta do sistema. Os testes de API concentram-se principalmente na camada lógica de negócios da arquitetura de software. Existem várias formas de realizar testes de API, incluindo testes unitários, testes de integração e testes de aceitação (REDDY, 2011).

Testes unitários são usados para testar pequenas partes isoladas de código, geralmente uma função ou um método. Eles garantem que cada parte do código esteja funcionando corretamente (MYERS; SANDLER; BADGETT, 2011). Testes de integração verificam se as diferentes partes da API estão trabalhando corretamente em conjunto. Eles podem incluir testes para verificar se a API está se comunicando corretamente com outros sistemas, como bancos de dados ou outras APIs (REDDY, 2011). Testes de aceitação são usados para verificar se a API atende às necessidades do negócio. Eles podem incluir testes para verificar se a API está retornando os dados corretos e no formato correto (MYERS; SANDLER; BADGETT, 2011).

Além disso, existem algumas ferramentas e bibliotecas, como Postman, SoapUI, JUnit, Pytest, etc, que auxiliam nos testes de APIs, permitindo planejar, executar e analisar os resultados dos testes.

2.2 Test Smells

O termo “code smell” tornou-se amplamente adotado na indústria o suficiente para comandar sua própria taxonomia. Code smells são indícios de problemas de design em um código-fonte que podem indicar a necessidade de refatoração (FOWLER, 1999). Eles são sintomas que surgem no código que podem indicar problemas subjacentes no design do software. Alguns exemplos de code smells incluem:

- Duplicated code: quando há trechos de código repetidos em diferentes partes do software.
- Long method: quando um método é muito longo e realiza muitas tarefas diferentes.
- Large class: quando uma classe é muito grande e contém muitos métodos e atributos.
- Primitive obsession: quando há uma dependência excessiva de tipos primitivos em vez de objetos.

Ademais, podemos ter smells de teste de unidade. Afinal, o código de teste de unidade é simplesmente um tipo específico de código que você escreve. O código de teste de unidade pode fornecer uma visão única sobre a natureza do seu código de produção. Ele usa seu código de produção, por definição. Ao fazer isso, ele oferece uma janela para como é configurar, consumir e desmontar as construções que você cria em seu código.

Test Smells se refere a qualquer sintoma em um código de teste que pode indicar um problema futuro (GAROUSI, 2018). São definidos como más práticas de programação no código de teste que indicam possíveis problemas de design no código-fonte de teste (SPADINI et al., 2018). A principal diferença entre code smells e test smells é que os code smells estão relacionados ao design e à implementação

do software (PALOMBA et al, 2014), enquanto os test smells “são baseados em como os casos de teste são organizados, implementados e interagem uns com os outros. Devido a essa distinção, as operações de refatoração correspondentes para código de teste e código de produção diferem.” (BAVOTA et al, 2014). Os test smells são problemas específicos encontrados nos testes de software que podem afetar sua qualidade e eficácia. Ao contrário dos code smells, que se referem a problemas no código-fonte, os test smells surgem devido às características e necessidades distintas dos testes . Isso inclui considerações como a validação da funcionalidade, simulação de cenários de uso e interações com APIs externas (BAVOTA et al, 2014).

Nesta tabela, serão apresentados alguns dos principais tipos de test smells e suas características. Entre os test smells listados, temos aqueles relacionados à semântica e lógica dos testes, como o "Assertion Roulette" e o "Conditional Test". Também são abordados test smells relacionados ao código, como o "Duplicate Assert" e o "Magic Number". Além disso, são mencionados problemas nas etapas de teste, como o "Empty Test" e o "Unknown Test", e test smells relacionados a dependências e execução de testes, como o "Mystery Guest" e o "Ignored Test" (JORGE et al., 2021).

Estes smells foram selecionados dos quais se podem aplicar ao código de teste em JavaScript/TypeScript. E a seguir na **Tabela 1**, iremos detalhar cada um desses smells, organizados em cinco categorias, e explicar como eles podem afetar a qualidade dos testes de software.

Tabela 1 - Smells abordados nesse trabalho

Categoria	**Smell**	**Característica**
Teste Semântica/Lógica	Assertion Roulette	Mais de uma asserção no caso de teste sem mensagem de falha
	Conditional Test	Estruturas de controle ou repetição
	Eager Test	Múltiplas chamadas para métodos de produção no mesmo caso de teste
	Lazy Test	Múltiplas chamadas para o mesmo método de produção

Código relacionado	Duplicate Assert	Múltiplas chamadas de asserções iguais no mesmo caso de teste
	Magic Number	Asserções contendo literais numéricos como argumentos
	Redundant Print	Chamadas para métodos do tipo console.log
Problemas nas etapas de teste	Empty Test	Caso de teste vazio (sem código)
	Exception Handling	Estrutura para tratar ou levantar uma exceção
	Redundant Assertion	Asserções onde o parâmetro atual e o esperado são iguais
	Unknown Test	Casos de teste com código, mas sem asserções
Dependências	Mystery Guest	Instâncias de recursos externos como um dado de teste
	Resource Optimism	Instâncias de recursos externos para inferir estado
Execução de teste	Ignored Test	Algum caso de teste ignorado
	Sleepy Test	Casos de teste com chamadas para setTimeout()

Fonte: Jorge (2021)

2.2.1 Teste Semântica/Lógica

Testes de Semântica/Lógica são uma categoria de testes que se concentram em avaliar a corretude do comportamento lógico e semântico do código de teste. Esses testes visam garantir que o software esteja seguindo as regras e lógica de negócio corretamente, assegurando que os resultados produzidos estejam de acordo com as expectativas. Dentro dessa categoria, existem vários subtipos de test smells que podem surgir, como o "Assertion Roulette", "Conditional Test", "Eager Test", "Lazy Test" e outros. A seguir, será feita uma análise mais aprofundada desses subtipos e como eles podem afetar a qualidade dos testes de semântica/lógica.

2.2.1.1 Assertion Roulette (AR)

Assertion Roulette (AR) é um smell comum em testes unitários e ocorre quando um caso de teste possui múltiplas asserções e, em pelo menos uma delas, não há uma mensagem de erro clara ou específica para indicar qual assertiva falhou, tornando mais difícil identificar o problema (JORGE et al., 2021). O nome "Roulette" vem do fato de que a falha pode ocorrer em qualquer uma das asserções, e sem uma mensagem de erro clara, é como se o resultado fosse aleatório, como em uma roleta. A falta de clareza na mensagem de erro também pode dificultar a depuração do problema, especialmente se o caso de teste contiver muitas asserções. Isso geralmente acontece quando um desenvolvedor adiciona uma nova assertiva a um teste existente sem refatorar o código de teste. Isso pode levar a testes com muitas assertivas, tornando-os difíceis de ler, entender e manter.

Os testes com Assertion Roulette são geralmente longos e complexos, e tendem a ser menos confiáveis, pois é difícil determinar qual assertiva falhou se o teste falhar. Além disso, testes com muitas assertivas tendem a ser mais sensíveis a mudanças no código-fonte, o que significa que eles podem falhar por causa de mudanças não relacionadas ao código que está sendo testado.

Para evitar que ocorra o Assertion Roulette é recomendado seguir a técnica Three A's, onde o teste deve ser Atômico, Automatizado e Assertivo. Onde cada teste deve ser simples, ter um único objetivo e verificar uma única assertiva. Isso ajuda a tornar os testes mais legíveis, fáceis de entender e mais confiáveis. No exemplo de código apresentado **Figura 1**, temos um teste denominado 'testAssertionRoulette'. Nesse teste, são realizadas múltiplas asserções, ou seja, verificações de resultados esperados, sem o uso de mensagens de falha. Isso pode tornar a manutenção e depuração dos testes mais complexas, uma vez que se uma asserção falhar, não será possível identificar qual delas foi responsável pelo erro sem uma mensagem de falha.

Figura 1 - Exemplo de Assertion Roulette

```
test('testAssertionRoulette', () => {
  expect(1 + 1).toBe(2);
  expect(2 + 2).toBe(4);
  expect(3 + 3).toBe(6);
});
```

Fonte: Elaborado pelo autor (2023)

2.2.1.2 Conditional Test Logic (CT)

Conditional Test Logic é um test smell que se refere à presença de estruturas de controle de decisão ou repetição em casos de teste (JORGE et al., 2021). Isso pode levar a casos de teste excessivamente complexos, difíceis de manter e entender. A presença dessas estruturas pode dificultar a identificação de falhas específicas no código, já que a execução do teste pode seguir caminhos diferentes de acordo com a condição ou repetição definida.

Para resolver este problema, é recomendado que os testes sejam divididos em vários testes menores e mais focados, cada um verificando um cenário específico. Isso ajuda a tornar os testes mais legíveis e mais confiáveis. É importante evitar o uso excessivo dessas estruturas em casos de teste, buscando simplificar o código e torná-lo mais fácil de entender e manter.

No exemplo de código apresentado **Figura 2**, podemos observar que é utilizado um fluxo condicional para verificar a lógica do código em diferentes cenários. O teste contém uma estrutura de controle, um "if", que direciona a execução do teste com base em uma condição específica. Isso pode tornar o teste difícil de entender, manter e modificar, pois, cada ramificação condicional requer uma verificação separada e pode levar a uma maior duplicação de código nos casos de teste.

Figura 2 - Exemplo de Conditional Test Logic

```
✋  Conditional Test Logic test smell

%s/home/jardesson/Documentos/Nutes/regNUTES/Container/system-test/regnutes-deployment/audit-log
/unit/mappers/audit.log.entity.mapper.spec.ts%s:33:16
  31 |                     assert.propertyVal(result.request, 'body', auditLog.request.body)
  32 |                 }
> 33 |                 if (auditLog.user) {
     |                 ^ 'if' statement detected
  34 |                     assert.propertyVal(result.user, 'id', auditLog.user.id)
  35 |                     assert.propertyVal(result.user, 'type', auditLog.user.type)
  36 |                 }
```

Fonte: Saída da ferramenta STEEL (2023)

2.2.1.3 Eager Test (EaT)

Eager Test (EaT) é um smell onde o teste inicializa ou configura um objeto ou sistema completo antes de realizar as verificações (assertivas) (JORGE et al., 2021). Isso geralmente acontece quando os desenvolvedores criam testes que dependem de muitos objetos ou componentes diferentes para funcionar, tornando-os muito complexos e difíceis de entender e manter.

Os testes com o smell Eager Test tendem a ser longos e complexos, e também tendem a ser menos confiáveis, pois é difícil identificar qual parte do sistema está causando uma falha se o teste falhar. Esses testes também são propensos a mudanças não relacionadas ao código que está sendo testado, fazendo com que o teste possa falhar por causa disso.

Para solucionar o problema de Eager Test, é recomendado que os testes sejam escritos de forma isolada, iniciando apenas os objetos e dependências necessárias para a verificação específica. Isso ajuda a tornar os testes mais legíveis, fáceis de entender e mais confiáveis. Ademais, é recomendado usar técnicas como injeção de dependência, mock objects para remover a dependência de objetos e sistemas externos e evitar que testes complexos e demorados sejam executados.

No exemplo de código apresentado **Figura 3**, temos um conjunto de testes relacionados ao componente chamado 'MyComponent'. Nesses testes, é criada uma instância do componente e são chamados os métodos 'init()' e 'doSomething()' antes de cada teste ser executado. Podemos notar que há múltiplas chamadas para métodos de produção no mesmo caso de teste, como no caso em que são chamados os métodos 'init()' e 'doSomething()' antes de cada teste. Essa abordagem pode levar a uma duplicação de código desnecessária nos testes e torná-los mais complexos.

Figura 3 - Exemplo de Eager test

```
describe('MyComponent', () => {
  let myComponent;

  beforeEach(() => {
    myComponent = new MyComponent();
    myComponent.init();
    myComponent.doSomething();
  });

  it('should do something', () => {
    expect(myComponent.something).toBe(true);
  });

  it('should do something else', () => {
    expect(myComponent.somethingElse).toBe(false);
  });
});
```

Fonte: Elaborado pelo autor (2023)

2.2.1.4 Lazy Test (LT)

Lazy Test é um tipo de Test Smell que ocorre quando um caso de teste usa métodos de produção de forma inadequada, geralmente chamando múltiplas vezes o mesmo método em diferentes asserções em um mesmo caso de teste (JORGE et al., 2021). Isso pode levar a testes lentos e difíceis de manter, pois torna mais difícil identificar qual assertiva causou uma falha no teste. Além disso, pode haver a necessidade de configurar adequadamente o estado do objeto ou sistema antes de realizar as verificações, a fim de garantir que os testes sejam mais confiáveis e precisos.

Os testes com Lazy Test tendem a ser menos confiáveis, pois não garantem que o sistema está em um estado válido antes da verificação ser feita. O teste pode passar mesmo que o sistema não esteja funcionando corretamente. Isso ocorre porque o teste não configura adequadamente o objeto ou sistema antes de realizar as verificações (assertivas), podendo levar a falsos positivos ou falsos negativos. Portanto, é importante configurar o objeto ou sistema adequadamente antes de realizar as verificações para garantir a confiabilidade dos testes.

Para corrigir esses problemas, é importante que o teste seja reescrito de forma a configurar adequadamente o objeto ou sistema antes de realizar as verificações. Em

vez de chamar diretamente métodos de produção, é importante criar instâncias dos objetos necessários e configurá-los de forma adequada, de acordo com os cenários de teste. Dessa forma, é possível garantir que os testes sejam mais confiáveis e que as falhas sejam detectadas de forma mais precisa.

No exemplo de código apresentado **Figura 4**, temos um conjunto de testes relacionados ao método 'myMethod' de uma classe chamada 'MyService'. No primeiro teste, é realizada uma chamada ao método 'myMethod' com o parâmetro 'param1', e o resultado é armazenado na variável 'result'. Em seguida, é feita uma asserção para verificar se o resultado é igual ao objeto esperado, com as propriedades 'prop1' e 'prop2' e seus respectivos valores.

No segundo teste, novamente é realizada uma chamada ao método 'myMethod' com o parâmetro 'param1', e o resultado é armazenado na variável 'result'. Em seguida, são feitas asserções individuais para verificar se as propriedades 'prop1' e 'prop2' do resultado são iguais aos valores esperados.

Esse exemplo ilustra o teste smell denominado 'Lazy Test' (Teste Preguiçoso). Esse smell ocorre quando há múltiplas chamadas para o mesmo método de produção no mesmo caso de teste, como no caso em que 'myMethod' é chamado duas vezes com o mesmo parâmetro. Essa abordagem pode indicar uma falta de modularidade nos testes e um possível desperdício de recursos.

Figura 4 - Exemplo de Lazy Test

```
describe('Exemplo de Teste com Lazy Test', () => {
  const sut = new MyService();

  it('Deveria retornar o objeto esperado na primeira chamada', () => {
    const result = sut.myMethod('param1');
    expect(result).toEqual({ prop1: 'value1', prop2: 'value2' });
  });

  it('Deveria retornar o objeto esperado na segunda chamada', () => {
    const result = sut.myMethod('param1');
    expect(result.prop1).toBe('value1');
    expect(result.prop2).toBe('value2');
  });
});
```

Fonte: Elaborado pelo autor (2023)

2.2.2 Código relacionado

Test smells relacionados ao código são aqueles que se referem a problemas encontrados no código de teste, que podem afetar sua legibilidade, manutenibilidade e eficiência. Esses test smells estão relacionados a questões específicas do código de teste, como duplicação de asserções, uso de números mágicos (valores literais numéricos) como argumentos em asserções e chamadas redundantes para métodos de impressão (como console.log). A seguir, será feita uma análise mais aprofundada desses subtipos e como eles podem afetar a qualidade dos testes garantindo assim uma base sólida para a construção de software confiável e de alta qualidade.

2.2.2.1 Duplicate Assert (DA)

Duplicate Assert (DA) é um smell comum em testes unitários onde há mais de uma assertiva verificando a mesma coisa (JORGE et al., 2021). Isso geralmente acontece quando desenvolvedores adicionam mais de uma assertiva a um teste existente sem verificar se essa assertiva já existe. Isso pode levar a testes com várias assertivas verificando a mesma coisa, tornando-os difíceis de ler, entender e manter. Além disso, esse tipo de teste pode tornar o processo de debug mais complicado devido a necessidade de verificar cada uma das assertivas.

Para evitar o problema de Duplicate Assert, é recomendado remover todas as assertivas redundantes, mantendo apenas a assertiva mais importante. Isso ajuda a tornar os testes mais legíveis, fáceis de entender e mais confiáveis. Também é importante evitar adicionar assertivas novas a um teste existente sem verificar se já há uma assertiva existente verificando a mesma coisa, dessa forma, evitando criar assertivas duplicadas.

Nesse exemplo da **Figura 5**, podemos observar que há uma duplicação de asserção na verificação do valor da propriedade 'event_name'. Essa duplicação é redundante e desnecessária, pois o mesmo valor está sendo testado duas vezes consecutivas.

Figura 5 - Exemplo de Duplicate Assert

```
✋  Duplicate Assert test smell

%s/home/jardesson/Documentos/Nutes/regNUTES/Container/system-test/regnutes-deployment/accou
it/events/email.reset.password.event.spec.ts%s:74:16
  72 |                 const result: any = emailResetPasswordEvent.toJSON()
  73 |
> 74 |                 assert.propertyVal(result, 'event_name', 'EmailResetPasswordEvent')
     |                ^ duplicate assert
  75 |                 assert.propertyVal(result, 'event_name', 'EmailResetPasswordEvent')
  76 |                 assert.propertyVal(result, 'type', EventType.EMAIL)
  77 |                 assert.propertyVal(result, 'timestamp', timestamp)
```

Fonte: Saída da ferramenta STEEL (2023)

2.2.2.2 Magic Number Test (MN)

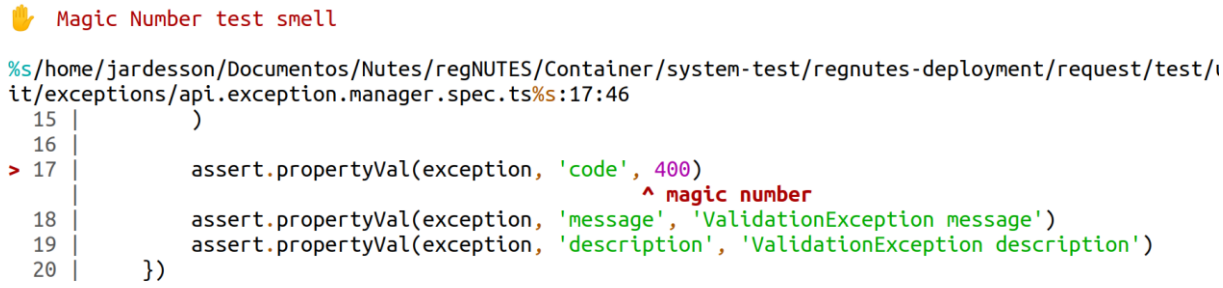
Magic Number Test (MN) é um smell comum em testes unitários onde os números utilizados como parâmetros para as assertivas são difíceis de entender ou são valores "mágicos" sem contexto ou explicação (JORGE et al., 2021). Esses números podem ser constantes, valores de retorno ou valores de comparação. Isso geralmente acontece quando os desenvolvedores não dão a atenção necessária aos valores utilizados nos testes, ou quando os testes são escritos de forma genérica sem se preocupar com a compreensão de quem os lerá.

Os testes com Magic Number tendem a ser difíceis de ler e entender, pois os valores usados não têm contexto e podem ser confusos. Além de tudo, esses valores são difíceis de manter, pois qualquer alteração pode afetar vários testes.

Para resolver o problema de Magic Number Test, é importante dar nomes significativos para esses números, ou utilizar constantes ou enumerações, buscando manter uma coesão entre o valor e o contexto em que o teste é escrito.

Nesse exemplo da **Figura 6**, o teste verifica se uma exceção lançada contém um código de erro específico, representado pelo número 400. O uso de um número literal sem explicação ou contexto adequado é considerado um "Magic Number", pois dificulta a compreensão do propósito do teste.

Figura 6 - Exemplo de Magic Number



Fonte: Saída da ferramenta STEEL (2023)

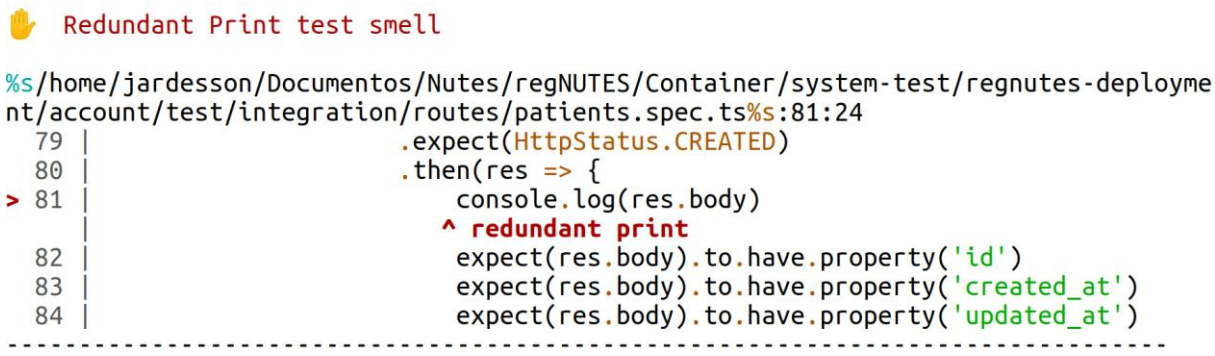
2.2.2.3 Redundant Print (RP)

Redundant Print (RP) é um smell comum em testes unitários onde são adicionados prints desnecessários para debugar um teste falhando (JORGE et al., 2021). Isso geralmente acontece quando os desenvolvedores não conseguem identificar a causa da falha e adicionam prints para obter mais informações sobre o estado do sistema. Isso pode deixar os testes mais verbosos e difíceis de ler, e também pode tornar o processo de depuração mais complicado. Tais testes podem ser mais propensos a modificações, o que significa que eles podem falhar por causa de mudanças não relacionadas ao código que está sendo testado.

Para solucionar o problema de Redundant Print, é recomendado remover todos os prints desnecessários e utilizar ferramentas de depuração mais adequadas, como breakpoints, watchpoints, etc. É de tamanha relevância evitar adicionar prints novos a um teste existente sem antes verificar se já existe uma forma de obter a informação desejada sem precisar usar prints desnecessários.

No exemplo de código apresentado na **Figura 7**, após realizar uma requisição e obter a resposta, o teste imprime o corpo da resposta utilizando console.log. Esse uso de console.log é considerado um "Redundant Print" porque não tem um propósito claro ou informativo no contexto do teste.

Figura 7 - Exemplo de Redundant Print



Fonte: Saída da ferramenta STEEL (2023)

2.2.3 Problemas nas etapas de teste

Os "Problemas nas etapas de teste" se referem a test smells relacionados a deficiências nas etapas de execução e verificação dos testes. Esses problemas podem impactar a qualidade dos testes e comprometer a detecção de erros e falhas no software. Alguns sub-tipos comuns nessa categoria incluem: "Empty Test" (caso de teste vazio, sem código de teste), "Exception Handling" (falhas na manipulação de exceções), entre outros. A seguir, iremos aprofundar cada um desses sub-tipos, discutindo suas características e impactos na qualidade dos testes.

2.2.3.1 Empty Test (Emt)

Empty Test é um smell comum em testes unitários onde o corpo do teste está vazio ou não contém nenhuma assertiva ou verificação para validar o comportamento esperado do código (JORGE et al., 2021). Esses métodos são possivelmente criados para fins de depuração e, em seguida, esquecidos ou contêm código comentado.

Os testes vazios são inúteis, pois eles não verificam se o código está comportando-se como esperado, portanto, não fornecem nenhum valor para garantir a qualidade do código e nem para detectar regressões no futuro. Para corrigir o problema de Empty Test é recomendado escrever testes unitários com assertivas ou verificações que validem o comportamento esperado do código.

No exemplo da **Figura 8** podemos observar um caso de teste vazio, onde o método testAdd() não contém nenhum código de teste. Isso significa que não há nenhuma verificação ou asserção sendo realizada para testar o comportamento esperado do método add() da classe Calculator. Um caso de teste vazio como esse não contribui para a garantia da qualidade do software, pois não realiza nenhum tipo de verificação ou validação.

Figura 8 - Exemplo de Empty test

```
public class CalculatorTest {

    @Test
    public void testAdd() {
        // Empty test body
    }
}
```

Fonte: Elaborado pelo autor (2023)

2.2.3.2 Exception Handling (ExT)

Exception Handling (ExT) é um smell comum em testes unitários onde o código de teste contém lógica para lidar com exceções ou erros (JORGE et al., 2021). Isso geralmente acontece quando os desenvolvedores não estão tratando exceções de forma adequada no código de produção e, como resultado, precisam adicionar lógica de tratamento de exceção nos testes.

Os testes com Exception Handling tendem a ser menos confiáveis, pois eles podem conter lógicas complexas e confusas para lidar com exceções, onde o sucesso ou falha, do teste, depende de o método de produção/teste lançar uma exceção que é capturada por instruções catch, em vez de usar as declarações de exceção específicas do framework.

Para evitar isso, é recomendado escrever testes que validem se as exceções são lançadas corretamente e de forma esperada. Como também evitar capturar exceções de forma genérica ou supressa-las sem tratamento adequado. Usar valores de retorno ou estruturas de dados apropriadas para indicar erros ou condições anormais, é algo bastante positivo.

Na **Figura 9**, temos um caso de teste que lida com o tratamento de exceções. O teste está verificando o comportamento quando o objeto ID é válido. O código está envolvido em um bloco try-catch para capturar qualquer exceção que possa ser lançada durante a validação do objeto ID. A falta de um tratamento adequado de exceções nos testes pode levar a resultados inconsistentes ou a falhas não detectadas no software.

Figura 9 - Exemplo de Exception Handling

```
✋  Exception Handling test smell

%s/home/jardesson/Documentos/Nutes/regNUTES/Container/system-test/regnutes-deployment/req
it/validators/object.id.validator.spec.ts%s:9:12
   7 |     context('when the object id is valid', () => {
   8 |         it('should return undefined when the validation was successful', () => {
>  9 |             try {
     |            ^ exception handling [Try Statement]
  10 |                 const result = ObjectIdValidator.validate(`${new ObjectID()}`)
  11 |                 assert.isUndefined(result)
  12 |             } catch (err: any) {
```

Fonte: Saída da ferramenta STEEL (2023)

2.2.3.3 Redundant Assertion (RA)

Redundant Assertion (RA) é um smell comum em testes unitários onde as assertivas são desnecessárias ou não contribuem para a validação do comportamento esperado do código (JORGE et al., 2021). Isso pode ocorrer quando os desenvolvedores adicionam assertivas adicionais sem refatorar o código de teste, ou quando as assertivas são escritas sem considerar completamente o código que está sendo testado.

Para resolver o problema de Redundant Assertion, é recomendado remover as assertivas redundantes, mantendo apenas as assertivas que contribuem para a validação do comportamento esperado do código. É crucial evitar adicionar assertivas novas a um teste existente sem verificar se já existe uma assertiva existente verificando a mesma coisa, dessa forma, evitando criar assertivas redundantes.

No exemplo da **Figura 10**, temos um caso de teste que verifica se a função calculateTotal() retorna um número válido. O teste inclui três asserções redundantes. Embora essas asserções possam parecer úteis individualmente, elas são

redundantes, pois a segunda e a terceira asserções já são implicadas pela primeira asserção. Portanto, a terceira asserção é desnecessária.

Figura 10 - Exemplo de Redundant Assertion

```
test('calculateTotal() function should return a number', () => {
  const items = [1, 2, 3];
  const result = calculateTotal(items);

  expect(typeof result).toBe('number');
  expect(result).not.toBeNaN();
  expect(result).toBeGreaterThan(0);
});
```

Fonte: Elaborado pelo autor (2023)

2.2.3.4 Unknown Test (UT)

Unknown Test (UT) é um smell comum em testes unitários onde o propósito ou o comportamento esperado do código testado não é claro ou não está documentado (JORGE et al., 2021). Isso geralmente acontece quando os desenvolvedores não dão a atenção necessária à documentação dos testes, ou quando os testes são escritos sem se preocupar com a legibilidade e manutenção.

Os testes com Unknown Test são difíceis de ler, entender e manter, pois, eles não fornecem nenhuma informação útil sobre o comportamento esperado do código testado. O que significa que eles podem falhar ou passar incorretamente devido à falta de compreensão do comportamento esperado.

Para solucionar o problema de Unknown Test é recomendado escrever testes que incluam comentários ou documentação explicando o propósito e o comportamento esperado do código testado. Os novos desenvolvedores do projeto acharão difícil entender o propósito de tais métodos de teste (ainda mais se o nome do método de teste não for suficientemente descritivo).

No exemplo da **Figura 11**, temos um caso de teste chamado calculateTotal() que verifica se a função calculateTotal() está corretamente calculando o preço total de uma lista de itens. Porém, não há nenhuma asserção sendo feita para verificar se o resultado do cálculo está correto. Isso significa que o teste é considerado um

"unknown test" porque não está validando explicitamente o comportamento esperado da função.

Figura 11 - Exemplo de Unknown Test

```
describe('calculateTotal()', () => {
  it('should calculate the total price of items', () => {
    const items = [10, 20, 30];
    const total = calculateTotal(items);
  });
});
```

Fonte: Elaborado pelo autor (2023)

2.2.4 Dependências

Os testes relacionados a dependências são aqueles que envolvem o uso de recursos externos, como bancos de dados, APIs, serviços ou arquivos. Esses testes são importantes para verificar a integração correta entre o sistema em teste e as dependências externas. Eles ajudam a garantir que o software funcione corretamente em um ambiente real, levando em consideração todas as interações com essas dependências. Existem diferentes subtipos de testes de dependências, como "Mystery Guest" e "Resource Optimism", que se concentram em diferentes aspectos das dependências e seus impactos nos testes. A seguir, iremos explorar cada um desses subtipos com mais detalhes.

2.2.4.1 Mystery Guest (MG)

Mystery Guest (MG) é um smell comum em testes unitários onde existe uma dependência de objetos ou sistemas externos que não são adequadamente configurados ou inicializados, tornando os testes difíceis de entender e reproduzir (JORGE et al., 2021). Isso geralmente acontece quando os desenvolvedores não dão a atenção necessária à configuração e inicialização de objetos e dependências nos testes, ou quando os testes são escritos sem se preocupar com a independência do código testado.

Para corrigir o problema de Mystery Guest é recomendado escrever testes que devem usar objetos fictícios no lugar de recursos externos. Um exemplo disso seria

utilizar as técnicas já citadas, como injeção de dependência e mock objects para remover a dependência de objetos e sistemas externos e evitar testes complexos e demorados.

No exemplo da **Figura 12**, temos uma função getUsers() que faz uma requisição a uma API externa (https://api.example.com/users) para obter dados de usuários. O teste verifica se a função retorna todos os usuários e espera que o comprimento do array users seja 10.

O teste depende da disponibilidade e resposta da API externa. Ele assume que a API sempre retornará 10 usuários, o que pode nem sempre ser o caso. O teste não está isolado e depende do comportamento de um sistema externo, tornando-o menos previsível e potencialmente causando falhas ou inconsistências nos testes.

Figura 12 - Exemplo de Mystery Guest

```
function getUsers() {
  return fetch('https://api.example.com/users')
    .then(response => response.json())
    .then(data => {
      return data;
    });
}

test('should return all users', () => {
  const users = getUsers();

  expect(users.length).toBe(10);
});
```

Fonte: Elaborado pelo autor (2023)

2.2.4.2 Resource Optimism (RO)

Resource optimism é um smell comum em testes de software em que os testes assumem que recursos externos, como bancos de dados, APIs ou sistemas de arquivos, estarão sempre disponíveis e funcionando corretamente (JORGE et al., 2021). Isso pode levar a testes que passam na maioria das vezes, mas falham em situações reais de uso quando esses recursos estão indisponíveis ou apresentam falhas. Isso geralmente acontece quando os testes são escritos sem se preocupar com a independência do código testado.

Para evitar esse problema, é importante que os testes simulem cenários em que esses recursos estejam indisponíveis ou apresentem falhas, de forma a validar a capacidade do sistema de lidar com essas situações de forma adequada. Isso pode ser feito através do uso de mocks, stubs ou fakes que simulam o comportamento desses recursos externos em diferentes situações.

Na **Figura 13**, temos um teste que verifica se o serviço de usuário (userService) retorna um usuário com base em um determinado ID. O teste utiliza a função getUserById() do serviço para obter o usuário com o ID especificado e, em seguida, verifica algumas propriedades desse usuário, como id, name, email e status.

No entanto, esse teste pode estar sofrendo de "Resource Optimism" (otimismo de recursos). Esse otimismo de recursos pode levar a problemas caso o recurso externo não esteja disponível, a conexão falhe ou os dados esperados sejam diferentes. Isso pode resultar em falhas falsas nos testes ou na dependência excessiva de um ambiente específico para a execução dos testes.

Figura 13 - Exemplo de Resource Optimism

```
const userService = require('../services/userService');

describe('User Service', () => {
  it('should return a user by id', () => {
    const userId = 123;

    const user = userService.getUserById(userId);

    expect(user.id).toBe(userId);
    expect(user.name).toBeDefined();
    expect(user.email).toBeDefined();
    expect(user.status).toBe('active');
  });
});
```

Fonte: Elaborado pelo autor (2023)

2.2.5 Execução de teste

A categoria de "Execução de teste" refere-se aos testes que estão relacionados à execução propriamente dita dos testes, abordando aspectos como a seleção e ordem dos testes, tratamento de exceções e comportamentos específicos durante a

execução. Dentro da categoria de Execução de teste, existem subtipos que abordam diferentes aspectos desse processo. Alguns exemplos de subtipos comuns são: Ignored Test e Sleepy Test.

Esses subtipos de Execução de teste exigem uma abordagem cuidadosa para garantir que os testes sejam executados de maneira correta e confiável. A seguir, aprofundaremos cada um desses subtipos e forneceremos exemplos específicos para ilustrar melhor seu uso e impacto nos testes.

2.2.5.1 Ignored Test (IT)

Ignored Test é um smell comum em testes unitários onde um teste é marcado para ser ignorado ou desabilitado, geralmente com um comentário ou pela função "skip" de algum framework de teste (JORGE et al., 2021). Isso pode ocorrer por várias razões, como falha temporária no ambiente de teste, um bug conhecido no código ou uma funcionalidade que ainda não foi implementada. No entanto, manter testes ignorados no código-fonte pode levar a um acúmulo de testes obsoletos, aumentar o tempo de execução dos testes e reduzir a confiabilidade do conjunto de testes.

Os testes ignorados não fornecem nenhum valor para garantir a qualidade do código e detectar regressões no futuro, pois eles não são executados. É recomendável remover os testes desnecessários e desativados, e configurar corretamente os testes para garantir sua execução. E ainda, não escrever testes sem uma finalidade clara ou sem planejamento para incluir esses testes na execução automatizada e manter os testes atualizados.

No exemplo da **Figura 14**, temos um caso de teste que foi intencionalmente ignorado usando a função test.skip(). Ao utilizar test.skip(), indicamos que esse teste não deve ser executado no momento. Ao marcar um teste como ignorado, estamos comunicando que há uma razão específica para não executá-lo. Ignorar um teste pode ser útil durante o desenvolvimento, permitindo que os desenvolvedores se concentrem em outras partes do código sem interrupções. No entanto, é importante garantir que os testes ignorados sejam revisados e retomados quando apropriado, para evitar a omissão de problemas ou regressões no software.

Figura 14 - Exemplo de Ignored Test

```
test.skip('Exemplo de teste ignorado', () => {
  expect(2 + 2).toBe(5);
  // Esse teste não será executado
});

test('Outro exemplo de teste', () => {
  expect(2 + 2).toBe(4);
  // Esse teste será executado normalmente
});
```

Fonte: Elaborado pelo autor (2023)

2.2.5.2 Sleepy Test (ST)

Sleepy Test ocorre quando um teste precisa esperar por um determinado período de tempo ou até que uma condição específica seja atendida antes de continuar a execução do teste (JORGE et al., 2021). Essa espera pode ser realizada com o uso de comandos como setTimeout ou setInterval, por exemplo. Esse tipo de teste pode ser problemático, pois é difícil determinar se o teste falhou devido a um problema no código ou simplesmente porque a espera foi muito curta ou longa demais. Isso geralmente acontece quando os desenvolvedores não dão a atenção necessária às necessidades de sincronização dos testes, tal qual, pode levar a comportamentos indesejáveis, como condições de corrida.

São geralmente mais lentos e possíveis de falhar devido a mudanças no sistema ou problemas de desempenho, pois o tempo de processamento de uma tarefa pode diferir em diferentes dispositivos. Para resolver o problema de Sleepy Test é recomendado evitar o uso de 'sleeps' nos testes unitários e utilizar mecanismos de sincronização mais adequados como callbacks, eventos, monitores, etc. É importante também evitar depender de tempo de espera fixo, usando por exemplo a função 'Thread.sleep()' e buscar alguma forma de verificar a condição de finalização de uma tarefa. Isso ajuda a garantir que os testes estejam preparados para testar a funcionalidade específica e sejam mais confiáveis.

No exemplo da **Figura 15**, temos um caso de teste que utiliza a função setTimeout() para introduzir um atraso de 1000 milissegundos (1 segundo) antes de atribuir o valor true a uma variável.]

Figura 15 - Exemplo de Sleepy Test

```
test('test de exemplo', () => {
  let variavel = false;

  setTimeout(() => {
    variavel = true;
  }, 1000);

  setTimeout(() => {
    expect(variavel).toBe(true);
  }, 2000);
});
```

Fonte: Elaborado pelo autor (2023)

Os Sleepy Tests podem ser úteis para verificar o comportamento de código assíncrono ou interações com APIs externas que requerem um certo tempo de resposta. No entanto, é importante garantir que o tempo de espera seja razoável e necessário, e que as asserções corretas sejam feitas para validar o comportamento esperado após o atraso. Além disso, é recomendado utilizar ferramentas e bibliotecas adequadas para lidar com código assíncrono nos testes, como o uso de funções assíncronas ou a utilização de frameworks de teste específicos para lidar com essas situações de forma mais eficiente.

2.3 Ferramentas para detecção de test smells

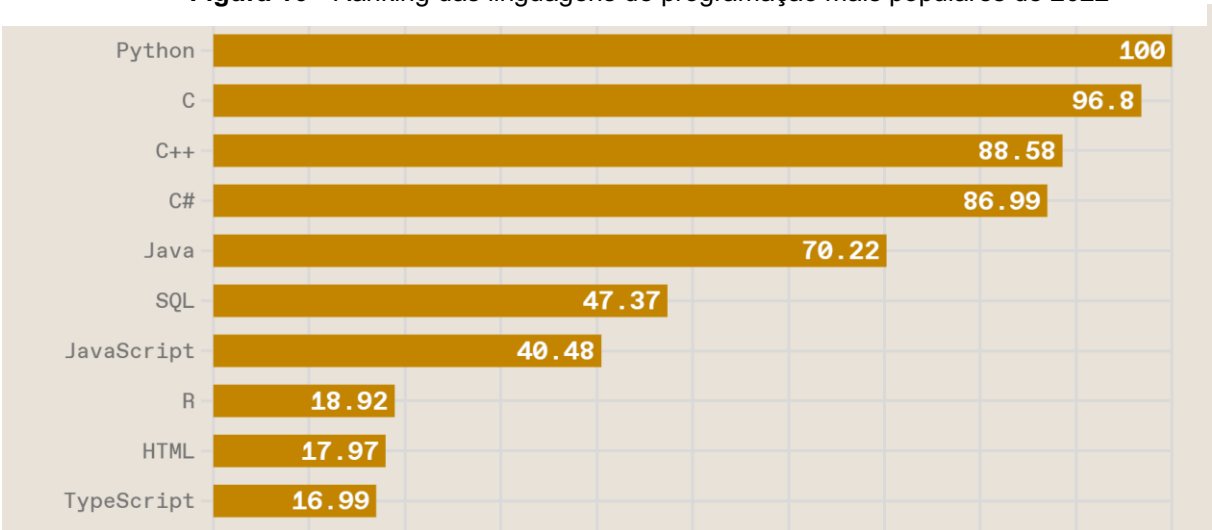
A detecção de Test Smells é uma tarefa importante para garantir a qualidade dos testes unitários e, consequentemente, a qualidade do software como um todo. Detectar esses smells é uma atividade que pode ser realizada tanto manualmente quanto de forma automatizada. No entanto, a detecção manual pode ser uma tarefa demorada e sujeita a erros, especialmente em projetos maiores (LACERDA et al, 2020, p. 16-17). Por essa razão, a detecção automática de Test Smells vem sendo cada vez mais estudada e utilizada na prática. Diversas ferramentas vêm sendo desenvolvidas para detectar automaticamente os Test Smells, incluindo tanto ferramentas genéricas que detectam vários tipos de smells, quanto ferramentas específicas para um determinado tipo de Test Smell. Essas ferramentas são capazes de identificar potenciais problemas em um conjunto de testes e apresentá-los aos desenvolvedores para que possam corrigi-los. Algumas dessas ferramentas incluem

o TRex, TestLint, TeReDetect entre outros (GAROUSI et al., 2018). Cada uma dessas ferramentas possui suas próprias abordagens e técnicas para detectar Test Smells, e a escolha da ferramenta mais adequada dependerá das necessidades e características do projeto em questão.

"Embora a maioria dos test smells se concentre nos sistemas Java tradicionais, os pesquisadores também estudaram o impacto desses smells em outras linguagens de programação" (ALJEDAANI et al., 2021, p. 1). Isso significa que há uma necessidade de mais ferramentas que possam detectar test smells em outras linguagens para garantir a qualidade e eficácia dos testes em um ambiente mais diversificado. "Embora a maioria das ferramentas detecte test smells ocorrendo em suítes de teste Java, há uma falta de suporte para outras linguagens populares, como JavaScript e Python" (ALJEDAANI et al., 2021, p. 7).

É essencial que mais pesquisas e desenvolvimentos sejam direcionados para a detecção de test smells em outras linguagens além do Java. Como também que evoluam para suportar essas linguagens populares e garantir que a qualidade dos testes seja mantida em todo o espectro de tecnologias utilizadas atualmente no desenvolvimento. A detecção de test smells em códigos JavaScript é importante pois é uma das linguagens de programação mais populares e amplamente utilizadas na atualidade. Como podemos observar na **Figura 16**, o ranking anual de linguagens de programação do ano de 2022 desenvolvido pela IEEE, e podemos observar também em (**Github** e **Tiobe**).

Figura 16 - Ranking das linguagens de programação mais populares de 2022



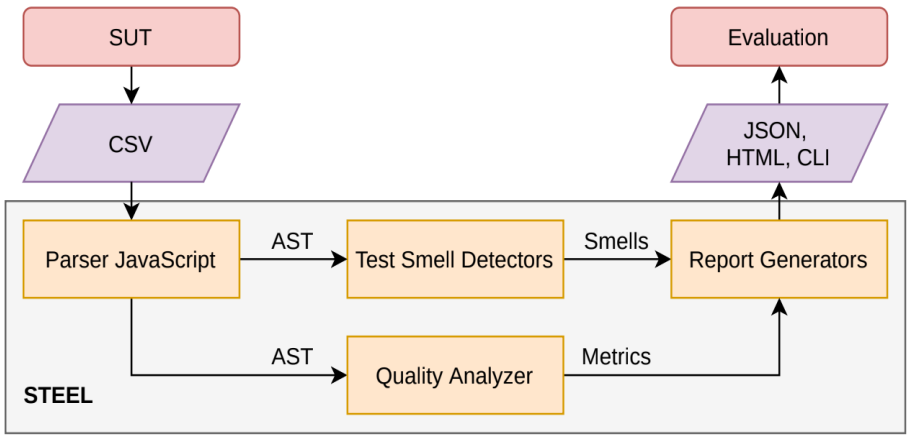
Fonte: IEEE Spectrum (2022)

2.3.1 STEEL

Para a realização deste estudo, era preciso de uma ferramenta que identificasse automaticamente os test smells na linguagem Javascript. Portanto, foi utilizada a ferramenta STEEL (teST smEll dEtection tooL), que é uma ferramenta de análise estática para a linha de comando que permite a detecção automática de smells em códigos de teste escritos na linguagem JavaScript, que além de detectar os test smells, também apresenta métricas de qualidade de teste. As métricas são: o número de linhas físicas do código do teste, tanto como o número de linhas lógicas, manutenibilidade, os índices de Complexidade Ciclomática simples e condensada, e propriedades Halstead (Bugs, Dificuldade, Esforço, Duração, Tempo, Vocabulário e Volume). Isso dá a capacidade para que os desenvolvedores possam identificar problemas com seus códigos de teste e melhorar a qualidade dos testes.

O conjunto arquitetônico da ferramenta STEEL **Figura 17** consiste nos seguintes módulos: o analisador da linguagem JavaScript contendo os analisadores léxicos e sintáticos, os detectores test smells baseados em regras, o analisador da qualidade do código e o gerador de relatórios nos formatos JSON, HTML e CLI.

Figura 17 - Arquitetura da ferramenta

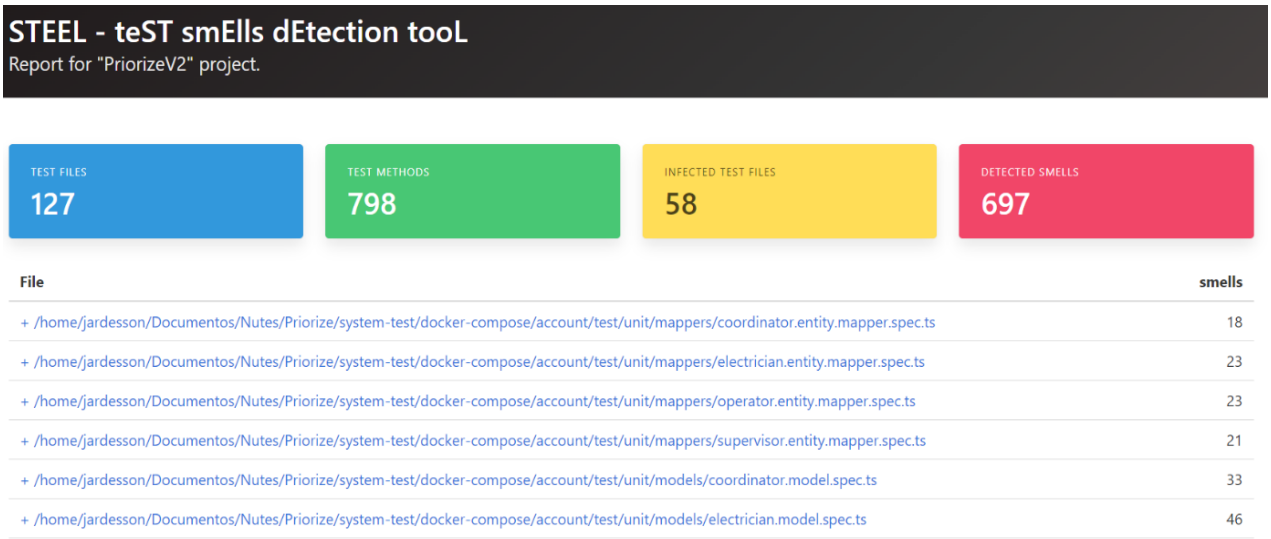


Fonte: JORGE, MACHADO, ANDRADE (2021)

No uso da ferramenta é necessário apenas um arquivo no formato CSV, que deve ser criado pelo usuário, contendo em cada linha da primeira coluna, todos os caminhos dos arquivos de teste a serem analisados, e, opcionalmente, os caminhos de seus respectivos arquivos de código-fonte de produção. Assim, a ferramenta será executada passando o arquivo CSV como parâmetro via linha de comando, ler cada caminho especificado, e analisar cada arquivo de teste presente nesse arquivo. A partir daí, começará a transformação dos códigos de testes em uma árvore sintática abstrata, onde os analisadores estáticos específicos para cada tipo de smell irão detectar os test smells e o módulo de análise de qualidade irá extrair as métricas.

A ferramenta possui três fases: a primeira é a transformação do código-fonte em estruturas de árvore sintática; a segunda consiste na detecção de test smells por meio de analisadores estáticos específicos e na extração de métricas de qualidade; e a terceira é a geração de relatórios a partir dos resultados das análises e da extração de métricas. Os resultados são exportados em formatos JSON e HTML e também podem ser impressos na interface de linha de comando para uso em integração contínua.

Figura 18 - Exemplo de saída em HTML



Fonte: Arquivo gerado pela ferramenta STEEL (2022)

A ferramenta apresenta os resultados das análises na fase de geração de relatórios, que incluem o número de arquivos de teste, o número de métodos de teste, o número de arquivos de testes infectados e a quantidade de test smells detectados. Além disso, os resultados incluem detalhamento de todos os arquivos de teste analisados, mostrando a quantidade de test smells presentes em cada um deles e também quais test smells foram detectados, tudo isso em um arquivo no formato HTML para facilitar a visualização e análise dos dados **Figura 18**. Dessa forma, é possível obter informações valiosas sobre a qualidade dos testes em um projeto e identificar áreas que precisam de melhorias.

3 ABORDAGEM

Nesta seção, apresentaremos a abordagem geral que será adotada para tratar os problemas relacionados aos test smells no contexto deste projeto de pesquisa. Discutiremos os conceitos e métodos que podem ser aplicados para identificar, analisar e compreender os test smells encontrados nos testes de API. Nosso objetivo é obter insights valiosos sobre esses test smells, buscando compreender suas características e possíveis causas, a fim de contribuir para a melhoria da qualidade dos testes de API.

Como mencionado anteriormente, para identificar os test smells nos códigos de teste em JavaScript, utilizamos a ferramenta de análise estática chamada STEEL (JORGE et al., 2021). Essa ferramenta foi desenvolvida especificamente para esse propósito e é capaz de detectar automaticamente os test smells, fornecendo métricas de qualidade de teste (ver Seção *2.3.1* - ***STEEL***). Essa abordagem baseada em ferramentas automatizadas facilita a detecção e análise dos test smells, permitindo que os desenvolvedores tomem medidas corretivas efetivas para aprimorar a qualidade dos testes.

3.1 Visão geral

Essa abordagem consiste em um processo que recebe como entrada os testes de software que serão analisados e visa identificar e analisar os test smells presentes nesses testes. O processo é composto por etapas distintas, incluindo a detecção automatizada dos test smells e a avaliação dos resultados obtidos. A saída desse processo é um conjunto de informações sobre os test smells encontrados nos testes, fornecendo uma visão clara dos problemas potenciais. Com base nesses resultados, os desenvolvedores podem tomar medidas apropriadas para corrigir os test smells, seja por meio de ajustes no código do teste ou na lógica subjacente. O objetivo final é melhorar a qualidade dos testes e garantir a eficácia dos testes de software. Ao longo deste capítulo, detalharemos cada uma dessas etapas, fornecendo diretrizes e boas práticas para realizar uma abordagem eficaz na detecção e análise dos test smells.

Podemos visualizar na **Figura 19** que o processo envolve três etapas. Primeiro, é gerado um arquivo de entrada contendo os caminhos para os arquivos de teste escritos em JavaScript. Em seguida, esse arquivo é utilizado como entrada para a ferramenta STEEL, que realiza a análise estática dos códigos, identificando os test smells. Após a execução da ferramenta, é realizada a análise dos resultados obtidos no relatório gerado, que contém informações sobre os test smells encontrados e métricas de qualidade de teste. Depois da identificação dos test smells, foi realizada uma análise detalhada dos resultados para identificar padrões recorrentes, como os tipos mais comuns de test smells, a distribuição dos test smells ao longo das diferentes versões do projeto e as possíveis causas para a sua introdução. É importante verificar se existem relações entre a evolução do código, a introdução de novas funcionalidades e o aumento ou diminuição dos test smells. Essa avaliação permite aos desenvolvedores compreender a qualidade dos testes e identificar oportunidades de melhoria.

Figura 19 - Processo da abordagem utilizada

Fonte: Elaborado pelo autor (2023)

3.1.1 Geração do arquivo de entrada

Inicialmente é indispensável a criação de um arquivo no formato CSV, que deve ser criado pelo usuário, contendo em cada linha da primeira coluna, todos os caminhos dos arquivos de teste a serem analisados, e, opcionalmente, os caminhos de seus respectivos arquivos de código-fonte de produção. Sendo assim, temos que é escrever todos os caminhos dos arquivos de teste que desejamos analisar em uma tabela. O arquivo pode ser preenchido manualmente ou pode ser gerado automaticamente, utilizando scripts que percorrem uma estrutura de diretórios, por exemplo, utilizando scripts que percorrem uma estrutura de diretórios. Utilizamos da segunda maneira para melhor execução em um menor tempo gasto. Foi desenvolvido um script onde era percorrido todos os diretórios dos arquivos de testes a serem analisados, para cada arquivo, seu caminho era listado e salvo em uma lista, garantindo assim de maneira eficiente, que todos os arquivos de teste sejam analisados. É importante frisar que esse script deve ser desenvolvido de forma que ele seja capaz de lidar com diretórios vazios, arquivos que não são testes, e arquivos que não existem mais. Isso garante que a lista de caminhos dos arquivos de teste contenha apenas arquivos válidos.

Uma vez que a lista de caminhos dos arquivos de teste está completa, devemos passar esta lista de caminhos extraídos pelo script para o arquivo CSV, que precisam ser organizados obrigatoriamente na primeira coluna. Isso permite que você execute a análise de uma só vez, em vez de ter que executar a ferramenta manualmente para cada arquivo de teste.

3.1.2 Execução da ferramenta STEEL

Para a execução da ferramenta STEEL, devemos fornecer apenas um comando através do terminal, onde o usuário passa os parâmetros necessários para que ela possa analisar os arquivos de teste desejados. Um exemplo de comando para a ferramenta STEEL **Figura 20**, pode ser dado pelo nome da ferramenta seguido pelo caminho do arquivo CSV que contém a lista dos caminhos dos arquivos de testes.

Figura 20 - Exemplo do comando

Fonte: Elaborado pelo autor (2023)

3.1.3 Análise dos resultados

A análise dos resultados da ferramenta STEEL envolve examinar o relatório gerado pela ferramenta após a análise estática dos códigos de teste. O relatório pode conter informações sobre os test smells identificados, como o tipo de test smell, a localização no código, e possíveis sugestões de correção. Além disso, o relatório pode fornecer métricas de qualidade de teste, como a quantidade total de test smells encontrados, a distribuição por tipo de test smell, entre outros dados relevantes.

É importante ressaltar que a avaliação dos resultados não se limita apenas à identificação dos test smells, mas também envolve uma compreensão mais ampla da qualidade dos testes, levando em consideração o contexto do projeto, as necessidades do cliente e as restrições do ambiente de desenvolvimento.

3.2 Exemplo Ilustrativo

Nesta seção, apresentaremos um exemplo ilustrativo para demonstrar a aplicação da abordagem de análise de test smells em testes de API. Este exemplo ajudará a entender como o arquivo de entrada é gerado, como a execução da ferramenta é realizada e como os resultados são analisados.

3.2.1 Arquivo de Entrada

Para começar, vamos considerar um cenário em que temos um conjunto de testes de API escritos em JavaScript para uma aplicação de gerenciamento de tarefas. Esses testes têm o objetivo de verificar se as funcionalidades da API estão funcionando corretamente. Primeiramente, é necessário gerar o arquivo de entrada que será utilizado pela ferramenta (no nosso caso, a ferramenta STEEL). A **Figura 21** abaixo ilustra um trecho do arquivo de entrada gerado a partir dos testes de API.

Figura 21 - Exemplo de arquivo de entrada

1	/home/jardesson/Documentos/src/tests/apiTests.js
2	
3	
4	
5	

Fonte: Elaborado pelo autor (2023)

Neste exemplo, o arquivo de entrada contém apenas um arquivo com os testes de API escritos em JavaScript, incluindo as asserções e as chamadas para a API. É importante que o arquivo de entrada esteja bem estruturado e seguindo as terminologia correta para que a ferramenta possa analisá-lo corretamente. Mas como é um arquivo de teste de API? Na **Figura 22** vemos um exemplo.

Figura 22 - Exemplo de teste de API

```
1  const axios = require('axios');
2
3  describe('Task API', () => {
4    let taskId;
5
6    beforeAll(async () => {
7      // Configuração inicial, cria uma tarefa para ser usada nos testes
8      const response = await axios.post('https://api.example.com/tasks', {
9        title: 'Test Task',
10       description: 'This is a test task',
11       status: 'todo'
12     });
13     taskId = response.data.id;
14   });
15
16   afterAll(async () => {
17     // Limpeza após os testes, remove a tarefa criada
18     await axios.delete(`https://api.example.com/tasks/${taskId}`);
19   });
20
21   it('should create a new task', async () => {
22     const response = await axios.post('https://api.example.com/tasks', {
23       title: 'New Task',
24       description: 'This is a new task',
25       status: 'todo'
26     });
27
28     expect(response.status).toBe(201);
29     expect(response.data.title).toBe('New Task');
30     expect(response.data.description).toBe('This is a new task');
31     expect(response.data.status).toBe('todo');
```

Fonte: Elaborado pelo autor (2023)

No arquivo de teste, serão definidos casos de teste que abrangem diferentes cenários e funcionalidades da API. Esses testes geralmente envolvem a preparação dos dados de entrada, a execução das chamadas para a API e a verificação das respostas recebidas. As asserções são usadas para verificar se as respostas da API estão de acordo com o esperado, garantindo que a API esteja se comportando corretamente.

3.2.2 Execução da Ferramenta STEEL

Com o arquivo de entrada gerado, podemos prosseguir para a execução da ferramenta STEEL para identificar possíveis test smells presentes nos testes de API. A **Figura 23** abaixo mostra a execução da ferramenta STEEL:

Figura 23 - Execução do STEEL

```
jardesson@PC:~/Downloads$ steel /home/jardesson/Documentos/src/tests/Apitests.csv
  ____ _____ _____ _____ _
 / ___|_   _| ____| ____| |
 \___ \ | | |  _| |  _| | |
  ___) || | | |___| |___| |___
 |____/ |_| |_____|_____|_____|


File: /home/jardesson/Documentos/src/tests/apiTests.js

✋  Duplicate Assert test smell

%s/home/jardesson/Documentos/src/tests/apiTests.js      %s:38:4
  36 |
  37 |     expect(response.status).toBe(200);
> 38 |     expect(response.data.id).toBe(taskId);
     |    ^ duplicate assert
  39 |     expect(response.data.title).toBe('Test Task');
  40 |     expect(response.data.description).toBe('This is a test task');
  41 |     expect(response.data.status).toBe('todo');

%s/home/jardesson/Documentos/src/tests/apiTests.js      %s:39:4
  37 |     expect(response.status).toBe(200);
  38 |     expect(response.data.id).toBe(taskId);
> 39 |     expect(response.data.title).toBe('Test Task');
     |    ^ duplicate assert
  40 |     expect(response.data.description).toBe('This is a test task');
  41 |     expect(response.data.status).toBe('todo');
  42 |
```

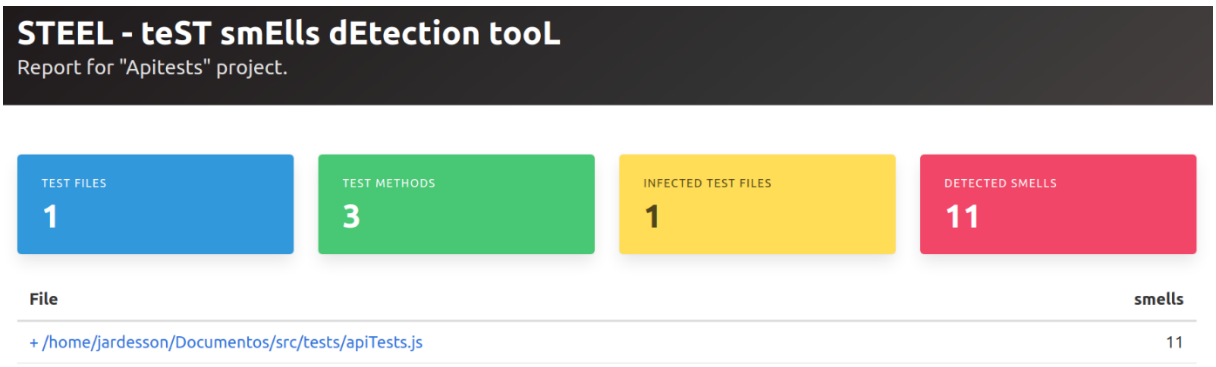
Fonte: Saída da ferramenta STEEL (2023)

Durante a execução, a ferramenta STEEL analisa o código de teste, identifica padrões e estruturas que indicam a presença de test smells, e gera um relatório com os resultados da análise.

3.2.3 Análise dos Resultados

Após a execução da ferramenta STEEL, é necessário analisar os resultados obtidos. O relatório gerado pela ferramenta fornece informações valiosas sobre os test smells encontrados, as métricas de qualidade de teste e outras análises relevantes. A **Figura 24** abaixo ilustra um trecho do relatório de resultados:

Figura 24 - Relatório



Fonte: Saída da ferramenta STEEL (2023)

Neste exemplo, o relatório destaca a presença de um test smell relacionado a asserts duplicados nos testes de API. A ferramenta STEEL identificou que há asserções duplicadas em um dos testes, o que pode indicar uma oportunidade de otimização e refatoração do código de teste. Ao analisar o relatório, e identificar os test smells encontrados, podemos entender suas causas e impactos, e tomar medidas para corrigir e melhorar os testes de API.

3.2.4 Considerações finais do exemplo

Observamos como a abordagem de análise de test smells em testes de API no contexto de evolução pode ser aplicada. O processo envolve a geração do arquivo de entrada, a execução da ferramenta STEEL e a análise dos resultados obtidos. Essa abordagem proporciona insights valiosos para aprimorar a qualidade dos testes, identificando padrões de test smells e oferecendo métricas de qualidade de teste.

4 AVALIAÇÃO

Nesta seção, voltaremos as questões de pesquisas investigadas neste trabalho, iremos apresentar os projetos utilizados nesta pesquisa e por fim respondemos as questões de pesquisa.

4.1 Questões de Pesquisa

Voltaremos nossa atenção para as questões de pesquisa abordadas ao longo deste trabalho, relembrando o objetivo principal que buscamos alcançar. O objetivo geral deste trabalho é identificar problemas de design e padrões comuns que podem afetar negativamente a eficiência e a eficácia dos testes de API através da análise de test smells investigando sua evolução, ao longo do desenvolvimento do software, observando possíveis mudanças e padrões.

Cada uma das questões de pesquisa abordadas neste trabalho foi cuidadosamente elaborada com o intuito de contribuir para o alcance desse objetivo. Essas questões foram selecionadas com base na relevância e na importância de compreender e abordar os test smells em testes de API, considerando o contexto de evolução de um sistema.

A seguir, estão elencadas as questões respondidas por este trabalho.

4.1.1 Quais os test smells mais frequentes, considerando os sujeitos analisados?

4.1.2 É possível identificar uma evolução ou regressão dos test smells em relação ao avanço no desenvolvimento dos sujeitos analisados?

4.1.3 É possível identificar práticas para evitar a inserção de test smells em testes de API durante a evolução?

4.1.4 O uso de ferramentas automatizadas para detectar test smells pode ajudar a melhorar a qualidade dos testes de API?

4.2 Projetos

Neste trabalho foi feita uma avaliação dos test smells encontrados ao longo da construção de dois projetos de software, Priorize e Regnutes, no qual teve o autor deste trabalho como parte da equipe de desenvolvimento de testes. Também foi realizada uma verificação da incidência desses smells no decorrer da evolução de tais projetos, e com o progresso de incrementos, fazer uma comparação se houve aumento ou diminuição em relação aos smells detectados, consequentemente levando a uma melhora da escrita dos códigos.

A **Tabela 2** a seguir apresenta dados gerais das diferentes versões dos projetos que foram utilizados neste trabalho. Esses dados incluem a quantidade de arquivos de testes examinados, a quantidade de linhas de código de teste avaliados e o número de métodos de testes de API aplicados em cada versão. Essas informações são relevantes para compreender a complexidade e a evolução do programa ao longo do tempo, além de permitir análises comparativas entre as versões. Através desses dados, podemos ter uma visão mais ampla do escopo e do tamanho dos testes executados e utilizá-los como base para a avaliação de test smells e melhorias na qualidade dos testes.

Tabela 2 - Dados dos projetos

Programa	Arquivos de teste	Métodos de teste	Linhas de código
Regnutes v1	325	6486	104523
Regnutes v2	361	5759	82284
Priorize v1	113	725	6211
Priorize v2	127	798	6939

Fonte: Elaborado pelo autor (2023)

Esses dados gerais nos fornecem uma visão quantitativa do tamanho do código, da quantidade de arquivos de teste, da quantidade de casos de teste de API e da extensão das linhas de código que foram avaliados em diferentes versões da dos dois projetos utilizados. Essas informações são importantes para entender a amplitude dos testes realizados, bem como para comparar e analisar possíveis variações ao longo das versões.

4.2.1 Regnutes

A partir de algumas demandas do setor de saúde da Paraíba, surgiu uma parceria entre a Secretaria de Saúde da Paraíba e o NUTES. A principal problemática demandada pela secretaria é uma solução para informatizar uma Central de Regulação de Leitos Hospitalares na Secretaria de Saúde do Estado da Paraíba.

O projeto Regnutes foi idealizado com intuito de resolver tal problemática com objetivo principal de facilitar a regulação para leitos hospitalares. A Central de Regulação recebe a solicitação de uma vaga a partir de uma Unidade Solicitante, que não possui leitos de terapia ou não dispõe de vaga no momento. A equipe médica da Central classifica o risco, através de informações sobre as condições clínicas, exames complementares e diagnóstico médico, e procura, na rede Estadual, pelo serviço que atenda às necessidades do paciente. A partir dessas informatizações será possível, além de ser célere nos serviços de saúde ofertados à população, dar mais transparência no item ocupação de leitos hospitalares.

4.2.1 Priorize

A partir de algumas demandas da empresa do setor energético Light-RJ, surgiu uma parceria entre a empresa e as universidades UEPB e UFCG, a partir da qual foi constatada a necessidade de realização de um projeto de pesquisa e desenvolvimento para atender tais demandas. A principal problemática demandada pela empresa é em relação a quais chamadas devem ser priorizadas em detrimento a outras, de modo a maximizar o atendimento de chamadas procedentes.

O projeto Priorize foi idealizado com intuito de resolver tal problemática com objetivo principal de evitar os deslocamentos desnecessários de equipes de manutenção para atender chamadas improcedentes, melhorando assim o atendimento ao cliente de chamadas procedentes. Utilizando análises do histórico de chamadas anteriores, dados climáticos, dados do sistema elétrico, dentre outros fatores, aliados às mais recentes técnicas de machine learning, o sistema Priorize é capaz de organizar as chamadas de forma a priorizar as chamadas com maior probabilidade de serem procedentes em detrimento as de menor probabilidade.

4.3 Resultados

4.3.1 Test smells mais comuns encontrados nesse estudo

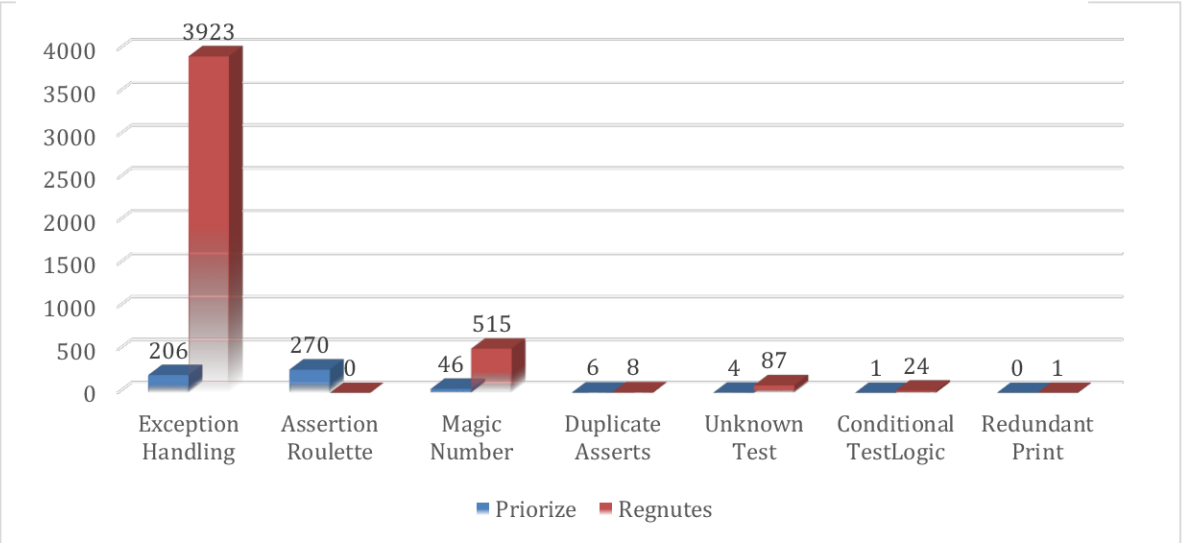
Avaliamos a quantidade de cada tipo de test smell que foi identificado na execução da ferramenta. Identificar os test smells mais comuns em testes de API pode ajudar a equipe de desenvolvimento a melhorar a qualidade dos testes direcionando os esforços em eliminar padrões problemáticos adotando práticas mais eficientes. Para cada test smell detectado pela ferramenta STEEL foi coletada a quantidade de ocorrências. A **Tabela 3** e a **Figura 25** apresentam a quantidade de cada test smell que foi encontrado em cada projeto, como também a quantidade de arquivos de testes analisados.

Tabela 3 - Estatísticas da quantidade de Test Smells

Test Smell	Priorize	Regnutes
Exception Handling	206	3923
Assertion Roulette	270	0
Magic Number	46	515
Duplicate Asserts	6	8
Unknown Test	4	87
Conditional TestLogic	1	24
Redundant Print	0	1

Fonte: Elaborado pelo autor (2023)

Figura 25 - Gráfico da Tabela 3



Fonte: Elaborado pelo autor (2023)

Com base nessas informações, pode-se perceber que o maior número de ocorrências de test smells para o projeto Priorize foram respectivamente: Assertion Roulette com 270 ocorrências seguido do Exception Handling com 206 ocorrências. Esses resultados podem ser atribuídos a várias razões. O Assertion Roulette ocorre quando há múltiplas asserções dentro de um único teste, o que pode indicar uma falta de foco e clareza na definição dos casos de teste. Já o Exception Handling se refere à forma como as exceções são tratadas nos testes. Nesse caso, a alta ocorrência pode indicar que os desenvolvedores estão escrevendo testes que não estão adequadamente preparados para tratar exceções.

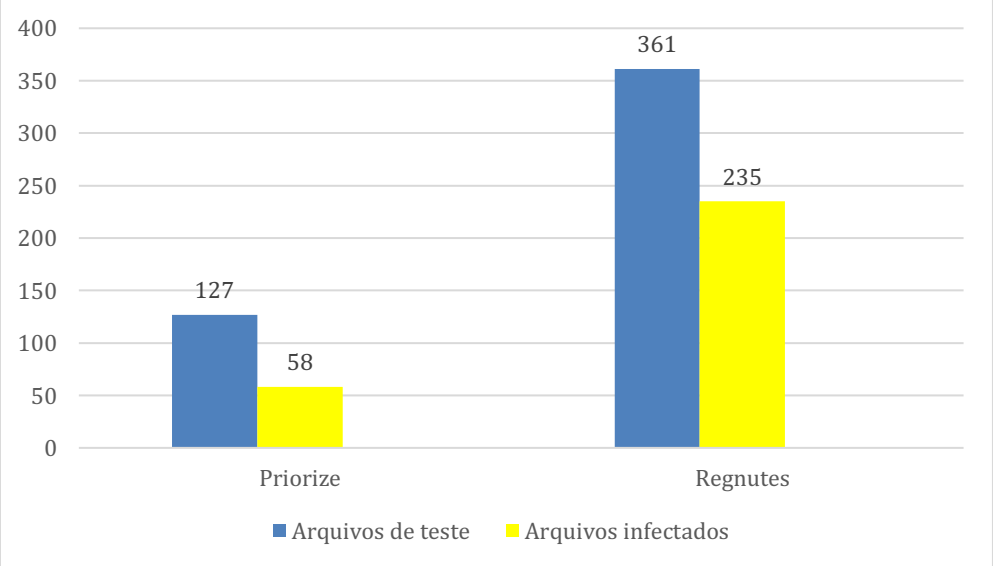
Esses resultados sugerem que, na prática, muitos desenvolvedores podem não estar aplicando boas práticas na definição e estruturação de seus testes. Isso pode ser resultado de falta de conhecimento sobre as melhores práticas de teste ou de falta de tempo e recursos para revisar e refatorar os testes existentes. Para mitigar esses test smells, é importante que os desenvolvedores recebam treinamento adequado sobre boas práticas de teste, como definir casos de teste mais focados e concisos e como lidar corretamente com exceções nos testes. Além disso, é importante realizar revisões regulares dos testes existentes para identificar e corrigir esses problemas.

Enquanto isso, no projeto Regnutes, foram detectados os test smells Exception Handling com 3923 ocorrências, acompanhado pelo test smell Magic Number com 515 ocorrências. Esses resultados podem ser explicados por algumas razões específicas relacionadas a esse projeto em particular. O test smell Exception Handling ocorre quando os testes não lidam corretamente com exceções. No projeto Regnutes, é possível que haja uma maior propensão a erros ou exceções durante a execução dos testes. Isso pode ser devido à complexidade das funcionalidades implementadas ou à presença de dependências externas que podem levar a falhas durante os testes. Como resultado, os desenvolvedores podem estar escrevendo testes que não são adequadamente preparados para capturar e lidar com exceções, resultando em um grande número de ocorrências desse test smell. Já o test smell Magic Number ocorre quando valores numéricos são usados diretamente no código, sem uma explicação clara do seu significado. No Regnutes, a presença de muitas ocorrências desse test smell pode indicar que os desenvolvedores estão usando valores numéricos de forma arbitrária em seus testes, sem uma contextualização adequada.

É importante considerar o contexto específico do projeto Regnutes e suas características. Dependendo da natureza do sistema sendo desenvolvido, certos test smells podem ser mais comuns ou relevantes. Por exemplo, se o projeto envolve operações matemáticas complexas ou manipulação intensiva de dados, é esperado que o test smell Magic Number seja mais prevalente. Esses resultados destacam a importância de analisar os test smells em um contexto específico, levando em consideração as particularidades do projeto e das funcionalidades implementadas.

A figura **Figura 26** mostra o número de arquivos analisados nos dois projetos de software e o número de arquivos de teste que foram identificados como infectados por algum tipo de test smell. A partir desses números, é possível calcular a taxa de infecção de cada projeto. No projeto Priorize, a taxa de infecção foi de aproximadamente 45%. No projeto Regnutes, obteve uma taxa de 65,10% de arquivos infectados. Com essa informação, é possível tomar ações específicas para corrigir ou evitar esses test smells, visando reduzir a taxa de infecção nos projetos. Medidas como revisões de código mais rigorosas, refatoração dos testes afetados, podem ajudar a reduzir a ocorrências de test smell e melhorar a qualidade dos testes.

Figura 26 - Comparação entre os dois projetos



Fonte: Elaborado pelo autor (2023)

4.3.2 Variação dos test smells em comparação a evolução do desenvolvimento do software

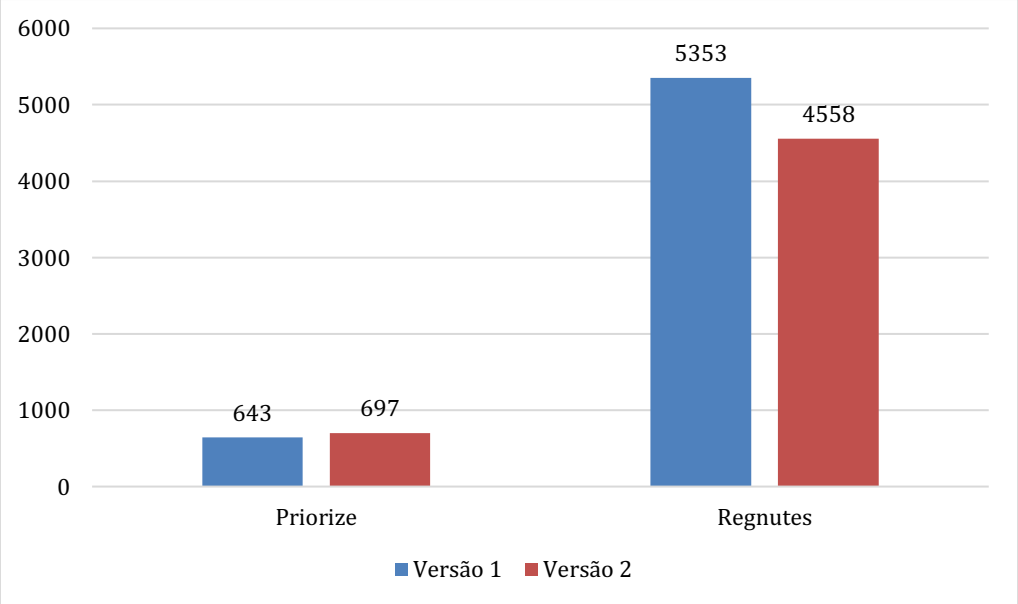
Para avaliar a variação dos test smells em comparação com a evolução do desenvolvimento do software, analisamos duas versões de cada projeto ao longo do tempo. Isso nos permitiu identificar mudanças nos testes, bem como avaliar se a introdução de novas funcionalidades ou correções introduziram novos test smells.

Tabela 4 - Comparação entre versões dos projetos

	Priorize	**Regnutes**
Versão 1	643	5353
Versão 2	697	4558

Fonte: Elaborado pelo autor (2023)

Figura 27 - Gráfico da Tabela 4



Fonte: Elaborado pelo autor (2023)

A **Tabela 4** e a **Figura 27** apresentam a quantidade de ocorrências dos test smells em duas versões diferentes de cada projeto de software utilizado. Podemos notar que houve um aumento de 8,39% no número de test smells entre as duas versões do projeto Priorize, o que pode indicar a presença de possíveis problemas de qualidade nos testes implementados nessa evolução. Por outro lado, observamos uma diminuição de 14,87% no número de test smells entre as duas versões do projeto

Regnutes. Essa redução significativa sugere que, a partir das novas implementações e refatorações realizadas nos testes do projeto Regnutes, houve uma melhora na qualidade dos testes e uma redução nos test smells identificados. Isso demonstra a importância de realizar revisões e refatorações nos testes à medida que o projeto evolui, a fim de aprimorar a robustez e a confiabilidade dos testes e garantir um melhor desempenho do software como um todo.

Enquanto isso, no projeto Priorize, é importante ressaltar que não foram realizadas refatorações significativas nos testes entre as versões analisadas. Em vez disso, houve a introdução de novas funcionalidades e, consequentemente, a criação de novos testes. Essa adição de código e funcionalidades sem uma devida revisão e refatoração pode levar a um aumento no número de test smells identificados. É importante notar que o surgimento de novas funcionalidades e alterações no software também podem introduzir novos test smells. Esses resultados reforçam a importância de aplicar práticas de manutenção contínua nos testes, mesmo durante o desenvolvimento de novas funcionalidades, a fim de evitar a acumulação de test smells e garantir a qualidade dos testes ao longo do tempo.

Observamos que as mudanças no código podem ter um impacto direto nos testes, resultando na ocorrência de test smells. No entanto, é importante ressaltar que essa relação causal entre a evolução do código e a presença de test smells é apenas uma hipótese e requer uma investigação mais aprofundada para ser confirmada. Existem diversas outras hipóteses que podem explicar os resultados observados, como a falta de conhecimento sobre boas práticas de teste, a ausência de revisões de código ou a pressão para entregar novas funcionalidades em curtos prazos. Portanto, como trabalho futuro, recomenda-se a realização de estudos mais abrangentes e aprofundados para compreender melhor os fatores que contribuem para a expansão dos test smells e propor estratégias eficazes para mitigá-los.

4.3.3 Práticas para evitar a introdução de test smells em testes de API durante a evolução

A partir dos dados analisados, pudemos identificar práticas que podem ajudar a evitar a introdução de test smells em testes de API durante a evolução. Ao

compreender os tipos mais comuns de test smells introduzidos ao longo do tempo, pode ser um primeiro passo importante para compreender a qualidade dos testes e identificar possíveis áreas de melhoria em cada projeto. Isso permite uma abordagem mais personalizada para aprimorar os testes, abordando os problemas específicos encontrados em cada contexto de desenvolvimento. Com base nos dados fornecidos neste estudo, foi possível identificar os tipos de Test Smells mais frequentes nos sistemas Regnutes e Priorize.

No sistema Regnutes, os Test Smells mais comuns encontrados durante a evolução foram o Exception Handling, com 3923 ocorrências, e o Magic Number, com 515 ocorrências. No caso do sistema Priorize, os Test Smells mais comuns identificados durante a evolução foram o Duplicated Assert, com 270 ocorrências, seguido pelo Exception Handling, com 206 ocorrências. Essas informações podem orientar os desenvolvedores e profissionais de qualidade na aplicação de técnicas de refatoração e melhores práticas de teste, visando aprimorar a qualidade dos testes.

A partir disso, pode-se adotar práticas para evitar a introdução desses Test Smells, como:

- Estabelecer diretrizes para o desenvolvimento de testes de API;
- Envolver desenvolvedores de teste em discussões sobre práticas de desenvolvimento de testes;
- Utilizar ferramentas automatizadas para detectar Test Smells em testes de API existentes;
- Identificar padrões de smells por projeto/equipe através de ferramentas de detecção de test smells.
- Refatorar testes de API existentes para remover smells detectados;

A partir da revisão da literatura, pode-se constatar que a presença de Test Smells pode impactar negativamente a qualidade dos testes (VAN ROMPAEY et al, 2007) (BAVOTA et al, 2014), resultando em testes menos efetivos na detecção de erros e na garantia da correta funcionalidade da API. As práticas sugeridas, como estabelecer diretrizes, realizar revisões de código, refatorar testes existentes e utilizar

ferramentas automatizadas, são amplamente recomendadas na comunidade de desenvolvimento de software. Essas práticas têm como objetivo melhorar a legibilidade, a manutenibilidade e a eficácia dos testes, garantindo a confiabilidade do software. A literatura acadêmica e as boas práticas no desenvolvimento de software fornecem uma base sólida para essa conclusão.

Portanto, embora não haja um estudo específico neste trabalho com desenvolvedores, a importância de adotar essas práticas é fundamentada em evidências da literatura e na compreensão das melhores práticas no desenvolvimento de software e de testes. Essas abordagens proativas podem ajudar a prevenir a introdução de Test Smells nos testes de API, contribuindo para a melhoria da qualidade dos testes e, consequentemente, do software.

4.3.4 Melhorar a qualidade dos testes de API com o uso de ferramentas automatizadas para detectar test smells

Comparamos a qualidade dos testes de API antes e depois da aplicação dessa ferramenta para podermos avaliar a efetividade da ferramenta de detecção, a identificação de test smells relevantes, quais os impactos na qualidade dos testes e também se é eficiente o esforço investido na detecção dos test smells.

Durante a pesquisa, foi constatado que a ferramenta STEEL, utilizada neste estudo, é capaz de identificar automaticamente os test smells presentes nos códigos de teste e fornecer métricas de qualidade. Ao utilizar a ferramenta STEEL, foi possível identificar os test smells presentes nos testes de API, mostrando áreas de melhoria e fornecendo informações úteis para a refatoração dos códigos de teste. Essa abordagem permitiu aos desenvolvedores compreender quais os problemas presentes nos testes, possibilitando a correção e evitando que os mesmos erros ocorram novamente.

No entanto, é importante ressaltar que a ferramenta STEEL atualmente se concentra na detecção de test smells, e não possui a capacidade de corrigi-los automaticamente refatorando os códigos de teste. Embora a detecção seja um

primeiro passo importante, a refatoração manual ainda é necessária para corrigir os test smells identificados.

Considerando essa limitação, seria desejável o desenvolvimento de ferramentas automatizadas capazes de não apenas detectar, mas também corrigir os test smells automaticamente, refatorando os códigos de teste. Isso poderia aumentar significativamente a eficiência e a qualidade dos testes de API.

Portanto, embora o uso de ferramentas automatizadas para detectar test smells seja benéfico para a melhoria da qualidade dos testes de API, ainda existem oportunidades de pesquisa e desenvolvimento para criar ferramentas mais avançadas que possam também corrigir automaticamente os test smells, proporcionando uma abordagem mais completa e eficiente para a garantia de qualidade nos testes de API.

4.4 Ameaças à validade

O presente estudo sobre a avaliação de test smells em testes de API no contexto de evolução apresenta algumas limitações que podem afetar a validade dos resultados obtidos. É importante destacar que existem diferentes tipos de ameaças à validade, sendo duas delas particularmente relevantes para este trabalho: a validade de conclusão e a validade interna.

Neste estudo, os dados foram coletados a partir de uma amostra específica de projetos de software e suas versões. Portanto, os resultados e conclusões alcançados podem não ser diretamente aplicáveis a outros projetos ou ambientes de desenvolvimento. É importante considerar que os fatores que podem influenciar a ocorrência e evolução dos test smells são diversos e podem variar de acordo com cada projeto. Alguns desses fatores podem incluir a complexidade do sistema, a experiência da equipe de desenvolvimento, a qualidade dos requisitos, a dinâmica do processo de desenvolvimento, entre outros.

No caso desta pesquisa, embora tenhamos observado relações entre a evolução do código, introdução de novas funcionalidades e o aumento de test smells em alguns casos, é importante destacar que não podemos afirmar uma relação direta

de causa e efeito. Outros fatores, como a falta de conhecimento sobre boas práticas de teste ou pressões externas, podem ter contribuído para os resultados observados.

Na literatura, estudos discutem os fatores que podem contribuir para a ocorrência de test smells. (GAROUSI, 2018) relatou que um determinado smell pode causar outros smells, por exemplo, o test smell eager test (um único teste verificando muita funcionalidade) pode levar ao test smell Assertion Roulette (difícil dizer qual das várias asserções dentro do mesmo método de teste causou uma falha de teste). Ele destacou também que o test smell Eager Test ocorre quando um teste tenta verificar vários métodos do objeto a ser testado. O test smell General Fixture é causado principalmente quando a configuração do fixture (também conhecido como contexto de teste) é muito geral e tem uma funcionalidade ampla (GAROUSI, 2018). Portanto, é necessário considerar as especificidades do contexto do projeto em questão ao analisar os fatores que podem influenciar a ocorrência e evolução dos test smells.

Além das ameaças mencionadas, é importante reconhecer que outros tipos de ameaças à validade também podem estar presentes neste estudo. Neste caso, a amostra utilizada pode não representar adequadamente todos os projetos de software e ambientes de desenvolvimento existentes. Portanto, é fundamental ter consciência das limitações e ameaças à validade deste estudo, reconhecendo a necessidade de futuras pesquisas que abordem essas questões de forma mais abrangente e aprofundada, a fim de fortalecer os resultados e contribuir para o avanço do conhecimento na área de testes de API e qualidade de software.

5 CONSIDERAÇÕES FINAIS

A partir da avaliação dos projetos Priorize e Regnutes, nos quais o autor participou do desenvolvimento dos testes, foi possível identificar os test smells mais comuns encontrados ao longo das diferentes versões dos projetos. A identificação, análise e correção de test smells desempenham um papel fundamental na melhoria da qualidade dos testes de API. Por meio da detecção precoce e correção desses test smells, é possível aumentar a confiabilidade e a cobertura dos testes, reduzindo o número de bugs e falhas encontrados no sistema. Além disso, a análise dos test smells desempenha um papel fundamental na evolução dos testes de API, permitindo que os desenvolvedores identifiquem os problemas recorrentes e trabalhem para corrigi-los, como também, adotar medidas preventivas para evitar erros similares em testes futuros. Permitindo assim, aprender com os erros passados e promover um ciclo de melhoria contínua, garantindo que os testes de API sejam cada vez mais robustos e confiáveis.

Neste estudo, cujo o objetivo principal era identificar os test smells mais comuns e realizar uma análise desse test smells investigando sua evolução, ao longo do desenvolvimento dos projetos, foi possível identificar os test smells mais frequentes nos projetos Priorize e Regnutes, que foram, Assertion Roulette e Exception Handling, respectivamente. Através da análise desses test smells, foi possível compreender os padrões e problemas de design recorrentes que afetam a qualidade dos testes. Além disso, a investigação da evolução dos test smells ao longo do tempo permitiu observar o impacto das mudanças nos projetos. No projeto Priorize, foi observada uma evolução no número de test smells, indicando a introdução de novos smells ao longo das versões do software. Por outro lado, no projeto Regnutes, foi identificado um regresso na quantidade de test smells, sugerindo melhorias realizadas ao longo da evolução. A utilização de ferramentas automatizadas, como a ferramenta STEEL, desempenhou um papel fundamental nesse processo, auxiliando na identificação dos test smells mais recorrentes nos projetos analisados. O que permitiu uma compreensão mais aprofundada dos desafios enfrentados na manutenção e melhoria dos testes de API. A análise e identificação precoce de test smells possibilitam a adoção de medidas corretivas e preventivas de forma mais ágil, contribuindo para a qualidade e confiabilidade dos testes de API e, por consequência, dos sistemas

desenvolvidos. Essa abordagem proporciona a oportunidade de antecipar potenciais problemas e mitigar riscos, resultando em uma melhoria significativa na eficiência e eficácia dos testes de API. Com os resultados obtidos, foi possível atingir o objetivo principal deste trabalho.

No entanto, é importante ressaltar que o uso de ferramentas automatizadas é uma abordagem complementar e não substitui a necessidade de envolver os desenvolvedores de teste em discussões sobre práticas de desenvolvimento de testes. Além disso, as conclusões obtidas neste estudo são baseadas na avaliação dos projetos Priorize e Regnutes, e podem variar em diferentes contextos e projetos. Considerando o desafio de lidar com grandes conjuntos de testes, uma possível pesquisa futura seria o desenvolvimento de ferramentas que não apenas detectem, mas também corrijam automaticamente os test smells em testes de API. Essa abordagem automatizada poderia otimizar ainda mais o processo de manutenção e melhoria contínua dos testes de API, contribuindo para a qualidade e confiabilidade dos sistemas desenvolvidos.

Em resumo, a avaliação de test smells em testes de API no contexto da evolução é um tema relevante e necessário para garantir a qualidade dos testes. A utilização de ferramentas automatizadas para identificar esses test smells é uma abordagem eficaz, mas é importante complementá-la com a participação ativa dos desenvolvedores de teste e a definição de diretrizes e boas práticas para o desenvolvimento de testes de API. Essa combinação de esforços é essencial para aprimorar continuamente a qualidade dos testes de API e garantir a confiabilidade dos sistemas. Nossos resultados destacam a importância da comunidade desenvolver métodos e ferramentas capazes de (i) detectar instâncias de test smell e (ii) refatorá-las automaticamente.

REFERÊNCIAS

ALJEDAANI, W. et al. **Test Smell Detection Tools:** A Systematic Mapping Study, 2021. https://doi.org/10.1145/3463274.3463335

ANJARD, R. P. **Software quality assurance considerations**. Microelectronics Reliability, 32(3), 307–312. 1992. https://doi.org/10.1016/0026-2714(92)90058-S

BAVOTA G., QUSEF A., OLIVETO R., DE LUCIA A., BINKLEY D. **An empirical analysis of the distribution of unit test smells and their impact on software maintenance.** In: **28th IEEE International Conference on Software Maintenance (ICSM)**. IEEE, Trento, Italy, 56–65, 2012. https://doi.org/10.1109/ICSM.2012.6405253

BAVOTA, G., QUSEF, A., OLIVETO, R., DE LUCIA, A., & BINKLEY, D. **Are test smells really harmful? An empirical study**. Empirical Software Engineering, 20(4), 1052–1094. 2014. https://doi.org/10.1007/s10664-014-9313-0

CARULLO, G. **Implementing Effective Code Reviews**: How to Build and Maintain Clean Code, 2020. https://doi.org/10.1007/978-1-4842-6162-0

CASS, S. Top Programming Languages 2022. **Ieee Spectrum**, 2022. Disponível em: https://spectrum.ieee.org/top-programming-languages-2022#toggle-gdpr. Acesso em: jul. 2023

CRESPO, A. N., SILVA, O. J., BORGES, C. A., SALVIANO, C. F., ARGOLLO, M., JINO, M. **Uma Metodologia para Teste de Software no Contexto da Melhoria de Processo**. In: **Simpósio Brasileiro de Qualidade de Software**, p. 271-285. 2004. https://doi.org/10.5753/sbqs.2004.16194

FOWLER, M. **Refactoring**: Improving the Design of Existing Code, Addison-Wesley Professional, ISBN: 9780134757681 1999.

GAROUSI, VAHID; KÜÇÜK, BARIŞ. **Smells in software test code**: A survey of knowledge. in Industry and academia. **Journal of Systems and Software** 138, 2018, 52 – 81. https://doi.org/10.1016/j.jss.2017.12.013.

GAROUSI, V., KUCUK, B., & FELDERER, M. **What We Know About Smells in Software Test Code**. **IEEE Software**, 1–1. 2018 https://doi.org/10.1109/MS.2018.2875843

GRAHAM, D., FEWSTER, M. **Experiences of test automation**: Case studies of software test automation. Addison-Wesley Professional, 2012. https://dl.acm.org/doi/10.5555/2132831

GREILER, M.; VAN DEURSEN, A.; STOREY, M. A. **Automated Detection of Test Fixture Strategies and Smells**. In: **2013 IEEE Sixth International Conference on Software Testing (ICST)**. p. 322-331. 2013. https://doi.org/10.1109/ICST.2013.45

ISHA; SHARMA, A.; REVATHI, M. **Automated api testing**. In: **2018 3rd International Conference on Inventive Computation Technologies (ICICT)**. Coimbatore, India, 2018, pp. 788-791, https://doi.org/10.1109/ICICT43934.2018.9034254.

JORGE, D.; MACHADO, P.; ANDRADE, W. **Investigating Test Smells in JavaScript Test Code**. In: **Brazilian Symposium on Systematic and Automated Software Testing**, 2021. https://doi.org/10.1145/3482909.3482915.

KIM, D. J. **An Empirical Study on the Evolution of Test Smell**, 2019. Disponível em: https://djaekim.github.io/djae.io/img/EvolutionOfTestSmell.pdf

KRAUSE, P. J. **Software Test Automation**: Effective Use of Test Execution Tools. By Mark Fewster and Dorothy Graham. Published by Addison-Wesley, Harlow, Essex, U.K., 1999. ISBN: 0-201-33140-3, 574 pages, 2000. https://doi.org/10.1002/1099-1689(200006)10:2%3C139::AID-STVR201%3E3.0.CO;2-K

KUMAR, D.; MISHRA, K. **The impacts of test automation on software's cost, quality and time to market**. Procedia Computer Science, Elsevier, v. 79, 2016. https://doi.org/10.1016/j.procs.2016.03.003

LACERDA, G., PETRILLO, F., PIMENTA, M., GUÉHÉNEUC., Y. G. **Code smells and refactoring**: A tertiary systematic review of challenges and observations. Journal of Systems and Software 167, 110610, 2020. https://doi.org/10.1016/j.jss.2020.110610

MYERS, G. J.; SANDLER, C.; BADGETT, T. **The Art of Software Testing**. 3rd. ed. [S.l.]: Wiley Publishing. ISBN 1118031962. 2011. https://dl.acm.org/doi/book/10.5555/2161638

PALOMBA, F.; ZAIDMAN, A.; DE LUCIA, A. **Automatic Test Smell Detection Using Information Retrieval Techniques**. In: **2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. p. 311-322. 2018. https://doi.org/10.1109/ICSME.2018.00040

PALOMBA, F. et al. **Do They Really Smell Bad?** A Study on Developers' Perception of Bad Code Smells**.** In: **2014 IEEE International Conference on Software Maintenance and Evolution (ICSME).** 2014. https://doi.org/10.1109/ICSME.2014.32

PECORELLI, F.; DO LILLO, G.; PALOMBA, F.; DE LUCIA, A. **VITRuM-A Plug-in for the Visualization of Test-Related Metrics**.In: **International Conference on Advanced Visual Interfaces**, 1-3. 2020 https://doi.org/10.1145/3399715.3399954

REDDY, M. **Testing. API Design for C++**, Morgan Kaufmann. p. 291–327, 2011 https://doi.org/10.1016/C2010-0-65832-9

RIOS, E.; MOREIRA, T. R. F. **Teste de Software**. 3ª Edição revisada e atualizada. p 8-9. P Rio de Janeiro: Alta Books, 2013. https://www.academia.edu/37989821/Teste_de_Software

SILVA, R. O.; MACHADO, G. B. G.; VIANA, G. B.; DOS SANTOS SILVA, J. S. O **Processo de Teste de Software**. In: **Tecnologias em Projeção**. v. 7, 2016.

SPADINI, D.; PALOMBA, F.; ZAIDMAN, A.; BRUNTINK, M.; BACCHELLI, A. **On the Relation of Test Smells to Software Code Quality**. In: **2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. p. 1-12. 2018. https://doi.org/10.1109/ICSME.2018.00010

TUTEJA, M.; DUBEY, G. **A Research Study on importance of Testing and Quality Assurance in Software Development Life Cycle (SDLC) Models**. In: International Journal of Soft Computing and Engineering (IJSCE) ISSN: 2231-2307, Volume-2, Issue-3, July 2012. Disponível em: https://www.ijsce.org/wp-content/uploads/papers/v2i3/C0761062312.pdf

VAN ROMPAEY, B., DU BOIS, B., DEMEYER, S., & RIEGER, M. **On The Detection of Test Smells**: A Metrics-Based Approach for General Fixture and Eager Test. In: **IEEE Transactions on Software Engineering**, 33(12), p. 802. 2007. https://doi.org/10.1109/tse.2007.70745

VIRGÍNIO, T.; SANTANA, R.; MARTINS, L. A.; SOARES, L. R.; COSTA, H.; MACHADO, I. **On the Influence of Test Smells on Test Coverage**. In: **SBES**. p. 467-471. 2019. https://doi.org/10.1145/3350768.3350775

YUSIFOĞLU, V. G.; AMANNEJAD, Y.; CAN, A. B. **Software Test-Code Engineering**: A Systematic Mapping. In: **Information and Software Technology**. p. 123-147. 2015. https://doi.org/10.1016/j.infsof.2014.06.009

[n.d.]. Popularity of Programming Language index. **Github,** Disponível em: http://pypl.github.io/PYPL. Acesso em: jul. 2023.

[n.d.]. Popularity of programming languages index. **Tiobe**, Disponível em: https://www.tiobe.com/tiobeindex/. Acesso em: jul. 2023