

UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

Larissa de Cássia Nazaré Bicalho

**Identificação de Smells em Testes Fim-a-Fim Implementados Usando a
Ferramenta Cypress**

Belo Horizonte
2024

Larissa de Cássia Nazaré Bicalho

**Identificação de Smells em Testes Fim-a-Fim Implementados Usando a
Ferramenta Cypress**

Versão Final

Dissertação apresentada ao Programa de Pós-Graduação em
Ciência da Computação da Universidade Federal de Minas
Gerais, como requisito parcial à obtenção do título de Mestre
em Ciência da Computação.

Orientador: Marco Túlio de Oliveira Valente

Coorientador: João Eduardo Montadon de Araujo Filho

Belo Horizonte
2024

Bicalho, Larissa de Cássia Nazaré.

B583i Identificação de smells em testes fim-a-fim implementados usando a ferramenta Cypress [recurso eletrônico] / Larissa de Cássia Nazaré Bicalho – 2024.

1 recurso online (87 f. il, color.) : pdf.

Orientador: Marco Túlio de Oliveira Valente.

Coorientador: João Eduardo Montadon de Araújo Filho.

Dissertação (Mestrado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciências da Computação.

Referências: f. 72-76

1. Computação – Teses. 2. Engenharia de software– Teses. 3. software – Avaliação - Teses. 4. JavaScript (Linguagem de programação de computador) – Teses. I. Valente, Marco Túlio de Oliveira. II. Araújo Filho, , João Eduardo Montadon de. III. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Computação. IV. Título.

CDU 519.6*32(043)



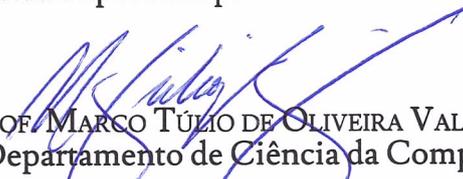
UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

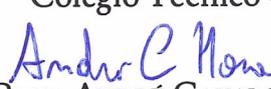
Identificação de Smells em Testes Fim-a-Fim Implementados Usando a Ferramenta Cypress

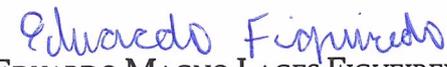
LARISSA DE CASSIA NAZARE BICALHO

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:


PROF. MARCO TÚLIO DE OLIVEIRA VALENTE - Orientador
Departamento de Ciência da Computação - UFMG


PROF. JOÃO EDUARDO MONTANDON DE ARAUJO FILHO - Coorientador
Colégio Técnico - UFMG


PROF. ANDRÉ CAVALCANTE HORA
Departamento de Ciência da Computação - UFMG


PROF. EDUARDO MAGNO LAGES FIGUEIREDO
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 5 de julho de 2024.

Dedico este trabalho a minha avó que sempre acreditou em meus sonhos, pois, “Um sonho que se sonha só é só um sonho que se sonha só. Mas sonho que se sonha junto é realidade.”- Raul Seixas

Agradecimentos

Esta dissertação é o resultado de dedicação, persistência e muitos aprendizados. Nada disso teria sido possível sem o apoio de diversas pessoas que acompanharam e contribuíram para esta jornada. Por isso, meu agradecimento especial:

Em primeiro lugar, agradeço a **Deus**, pois sem Ele nada é possível.

À minha avó, minha gratidão eterna. Ela é sempre a primeira pessoa a acreditar em mim e me apoiar em todas as etapas da minha vida.

À minha mãe, agradeço profundamente por todo o apoio ao longo desse caminho.

À minha tia, por não medir esforços para me ajudar.

Ao meu irmão Lucas, por sempre acreditar em mim e ser a minha força.

Ao meu tio Ladinho, que sempre me apoiou e me acompanhou nas idas à UFMG.

À minha amiga de ensino médio, Ana, que me apoiou e esteve ao meu lado em todos os momentos.

Aos meus amigos, Beatriz, André e Hudson, pela amizade e apoio constante.

À minha amiga Érika, e aos meus colegas de trabalho **Cleyton e Luana**, que inúmeras vezes leram este trabalho para me ajudar a melhorar a escrita.

Por fim, agradeço ao meu orientador **Marco Túlio Valente** e ao meu coorientador **João Eduardo Montandon**, pela paciência, pelos ensinamentos e pela sabedoria compartilhada, que levarei comigo para sempre.

“Sonhar, nunca desistir. Ter fé, pois fácil não é nem vai ser”
(Guilherme Alves)

Resumo

Considerando que o sistema é uma das construções humanas mais complexas já realizadas, é natural que uma variedade de erros e inconsistências possam ocorrer. Para evitar que tais problemas cheguem aos usuários finais e causem prejuízos, são necessárias atividades de teste em projetos de desenvolvimento de software. Um dos métodos mais comuns é o teste fim-a-fim, que visa verificar o comportamento dos requisitos do sistema como um todo. Para implementar esse tipo de teste, os desenvolvedores contam com várias ferramentas, como Selenium, Cypress e Playwright, entre outras.

Apesar do aumento no uso dessas ferramentas, poucos estudos avaliam más práticas associadas ao seu uso. Para abordar esse assunto, esta pesquisa investigou as más práticas relacionadas com o uso do *framework* Cypress, um *framework* JavaScript para testes fim-a-fim. Inicialmente, foi realizado um estudo para catalogar os *smells* mais comuns em tais testes por meio de uma Revisão Sistemática da Literatura (SLR) e uma Revisão da Literatura Cinza (GLR), resultando na identificação de 14 *smells* específicos de testes fim-a-fim implementados com o Cypress.

Em seguida, avaliou-se métodos para identificar automaticamente esses *smells*. Para isso, recorreu-se aos Modelos de Linguagem de Grande Escala (LLMs), como o ChatGPT, que são utilizados para automatizar uma variedade de tarefas, incluindo aquelas pertinentes ao desenvolvimento de software. A capacidade do ChatGPT em identificar esses problemas foi avaliada por meio de um estudo de caso e um estudo com aplicações GitHub.

No estudo controlado, o ChatGPT conseguiu identificar com sucesso 12 dos 14 *smells* catalogados. Oito dos *smells* considerados no estudo foram detectados após a primeira solicitação (67%). O estudo de campo avaliou testes fim-a-fim implementados em três sistemas de código aberto: Pigallery2, Livewire e GlobaLeaks. Os resultados mostraram que o sistema Pigallery2 teve uma precisão de 0.31 e um *recall* de 0.62. Para o Livewire, os valores foram de 0.24 para precisão e 0.44 para *recall*. Por fim, o GlobaLeaks apresentou o pior desempenho, com uma precisão de 0.15 e um *recall* de 0.31. A principal causa para os baixos índices de precisão e *recall* obtidos nesse segundo estudo foi devido à ineficiência na detecção de certos *smells*, como o *Brittle Selectors*.

A pesquisa obteve resultados promissores ao integrar um estudo da SLR e GLR, com isso determinando um catálogo de *smells* para os testes desenvolvidos com o Cypress. Em relação a detecção dos *smells* pode-se concluir que o ChatGPT não é eficiente para detecção destes.

Palavras-chave: testes fim-a-fim; code smells; test smells; modelos de linguagem de grande escala; chatGPT; cypress; javaScript.

Abstract

Considering that software systems are among the most complex human constructions ever made, it is natural for a variety of errors and inconsistencies to occur. To prevent such issues from reaching end-users and causing harm, testing activities are necessary in software development projects. One of the most common methods is end-to-end testing, which aims to verify the behavior of system requirements as a whole. To implement this type of testing, developers rely on various tools such as Selenium, Cypress, and Playwright, among others.

Despite the increasing use of these tools, few studies evaluate the bad practices associated with their use. To address this issue, this research investigated the bad practices related to the use of the Cypress framework, a JavaScript framework for end-to-end testing. Initially, a study was conducted to catalog the most common smells in such tests through a Systematic Literature Review (SLR) and a Grey Literature Review (GLR), resulting in the identification of 14 specific smells in end-to-end tests implemented with Cypress.

Subsequently, methods for automatically identifying these smells were evaluated. Large Language Models (LLMs), such as ChatGPT, which are used to automate a variety of tasks, including those relevant to software development, were utilized. The ability of ChatGPT to identify these problems was assessed through a case study and a study with GitHub applications.

In the controlled study, ChatGPT successfully identified 12 of the 14 cataloged smells. Eight of the smells considered in the study were detected after the first request (67%). The field study evaluated end-to-end tests implemented in three open-source systems: Pigallery2, Livewire, and GlobaLeaks. The results showed that the Pigallery2 system had a precision of 0.31 and a recall of 0.62. For Livewire, the values were 0.24 for precision and 0.44 for recall. Finally, GlobaLeaks had the worst performance, with a precision of 0.15 and a recall of 0.31. The main cause for the low precision and recall rates obtained in this second study was due to inefficiency in detecting certain smells, such as Brittle Selectors.

The research yielded promising results by integrating an SLR and GLR study, thus determining a catalog of smells for tests developed with Cypress. Regarding the detection of smells, it can be concluded that ChatGPT is not efficient in detecting them.

Keywords: end-to-end tests; code smells; test smells; llm's; chatGPT; cypress; javaS-cript.

Lista de Figuras

2.1	Pirâmide de Teste, retirado do livro Engenharia de Software Moderna.	24
3.1	Protocolo da Revisão da Literatura Científica	34
3.2	Processo Literatura Cinza	40
4.1	Metodologia do Estudo do Mantis	51
4.2	Metodologia do Estudo das Aplicações GitHub	58
4.3	Taxas de Precisão de cada smell para o sistema Pigallery	62
4.4	Taxas de Recall de cada smell para o sistema Pigallery	62
4.5	Taxas de Precisão de cada smell para o sistema Livewire	63
4.6	Taxas de Recall de cada smell para o sistema Livewire	63
4.7	Taxas de Precisão de cada smell para o sistema GlobaLeaks	63
4.8	Taxas de Recall de cada smell para o sistema GlobaLeaks	64

Lista de Tabelas

3.1	Quantidade de artigos encontrados em cada biblioteca	35
3.2	Quantidade de artigos excluídos em cada biblioteca digital. A terceira coluna apresenta a quantidade de artigos excluídos por motivo	35
3.3	Artigos selecionados da RSL	36
3.4	Smells identificados em cada artigo	36
3.5	Critérios utilizados para excluir artigos durante a filtragem. A segunda coluna apresenta a quantidade de artigos excluídos por motivo	41
4.1	Smells injetados em cada teste	52
4.2	Prompt enviado ao ChatGPT	52
4.3	Lista de <i>smells</i> solicitados ao ChatGPT. O símbolo de marcador (●) indica qual rodada o <i>smell</i> foi identificado pelo ChatGPT.	53
4.4	Respostas do ChatGPT na tentativa de identificar o smell Missing Global BaseUrl	57
4.5	Respostas do ChatGPT na tentativa de identificar o smell Use Relative URLs Locator Instead Of Absolute Ones	57
4.6	Tabela de Sistemas Selecionados do GitHub	59
4.7	Quantidade de Smells por Sistema	59
4.8	Prompt enviado ao ChatGPT durante o Segundo Experimento	60
4.9	Avaliação Geral da Precisão e Recall dos Sistemas	61
4.10	Quantidade do smell Brittle Selector no Oráculo vs Quantidade desse smell detectada pelo ChatGPT	65
4.11	Resultado do ChatGPT	65
A.1	Smells Excluídos RSL	78
A.2	Smells Excluídos RLC	79

Lista de Códigos

1	Teste de Login com smell Unnecessary Waiting	19
2	Login Page	27
3	Home Page	27
4	Exemplo de Teste Fim-a-Fim.	28
5	Using Cypress Calls In Assignments example	45
6	Trecho de código contendo o smell Missing Global BaseUrl	45
7	Trecho de código contendo o smell Dirty inputs	46
8	Trecho de código contendo o smell Cleaning up state with <code>after()</code> or <code>afterEach()</code> exemplo	47
9	Trecho de código contendo o smell Direct Access to List Elements	47
10	Trecho de código contendo como evitar o smell Avoid Direct Access to List Elements	48
11	Exemplo de teste contendo o smell Unnecessary Waiting	54
12	Exemplo de teste contendo o <i>smell Starting Web Servers</i>	55
13	Exemplo de teste contendo o <i>smell Missing Global BaseUrl</i>	56
14	Code snippet containing the Using Cypress Calls In Assignments smell	84
15	Code snippet containing the Missing Global BaseUrl smell	85
16	Code snippet containing the Dirty inputs smell	85
17	Code snippet containing the Cleaning up state with <code>after()</code> or <code>afterEach()</code> smell	86
18	Code snippet containing the Direct Access to List Elements smell	87
19	Code snippet containing how avoid Direct Access to List Elements smell	87

Sumário

1	Introdução	16
1.1	Motivação	16
1.2	Trabalho Proposto	20
1.3	Contribuições	21
1.4	Estrutura da Dissertação	21
2	Conceitos Básicos	23
2.1	Qualidade de Software	23
2.2	Testes Fim-a-Fim	25
2.3	Cypress	25
2.4	Outras Ferramentas	29
2.5	Trabalhos Relacionados	30
2.5.1	Code Smells	30
2.5.2	Test Smells	31
2.6	Considerações Finais	32
3	Revisões Sistemáticas da Literatura	33
3.1	Revisão da Literatura Científica	33
3.2	Catálogo de Smells RSL	37
3.2.1	Unnecessary Waiting	37
3.2.2	Brittle Selectors	38
3.2.3	Adopt Page Object Pattern	38
3.2.4	Use Relative URLs Locator Instead Of Absolute Ones	38
3.2.5	Unnecessary Navigation	39
3.3	Revisão da Literatura Cinza	40
3.4	Catálogo de Smells RLC	42
3.4.1	Unnecessary Waiting	42
3.4.2	Using Force: True When Interacting With Elements	42
3.4.3	Visiting External Sites	43
3.4.4	Starting Web Servers	43
3.4.5	Brittle Selectors	43
3.4.6	Unnecessary Visibility Checks For Previously Actionable Elements	44
3.4.7	Adopt Page Object Pattern	44

3.4.8	Using Cypress Calls In Assignments	44
3.4.9	Missing Global BaseUrl	45
3.4.10	Dirty Inputs	46
3.4.11	Cleaning up state with after() or afterEach()	46
3.4.12	Direct Access to List Elements	47
3.5	Ameaças à Validade	48
3.6	Considerações Finais:	49
4	Detecção de Smells	50
4.1	Introdução	50
4.2	Estudo Controlado - Mantis App	51
4.2.1	Metodologia	51
4.2.2	Resultados	53
4.2.3	Discussão	54
4.2.3.1	Descrição dos Smells	54
4.2.3.2	Dificuldade na Identificação de Missing Global URL e Use Relative URLs	56
4.3	Estudo com Aplicações GitHub	58
4.3.1	Metodologia	58
4.3.2	Resultados	61
4.3.2.1	Resultado Pigallery	62
4.3.2.2	Resultado Livewire	62
4.3.2.3	Resultado GlobaLeaks	63
4.3.2.4	Discussão	64
4.4	Ameaças à Validade	66
4.5	Considerações Finais	67
5	Conclusão	68
5.1	Contribuições	69
5.1.1	Catálogo de Smells	69
5.1.2	Detecção de Smells	69
5.1.2.1	Estudo Controlado - Mantis	70
5.1.2.2	Estudo com Sistemas GitHub	70
5.1.3	Trabalhos Futuros	70
	Referências	72
	Apêndice A Tabelas de Exclusão de Smells Candidatos ao Catálogo	77
A.1	Tabela de Exclusão de Smells da RSL	77
A.2	Tabela de Exclusão de Smells da RLC	78

Apêndice B Catálogo de Smells	80
B.1 Catálogo de Smells RSL em Inglês	80
B.1.1 Unnecessary Waiting	80
B.1.2 Brittle Selectors	80
B.1.3 Adopt Page Object Pattern	81
B.1.4 Use Relative URLs Locator Instead Of Absolute Ones	81
B.1.5 Unnecessary Navigation	81
B.1.6 Catálogo de Smells RLC em Inglês	82
B.1.6.1 Unnecessary Waiting	82
B.1.6.2 Using Force: True When Interacting With Elements	82
B.1.6.3 Visiting External Sites	82
B.1.6.4 Starting Web Servers	83
B.1.6.5 Brittle Selectors	83
B.1.6.6 Unnecessary Visibility Checks For Previously Actionable Elements	83
B.1.6.7 Adopt Page Object Pattern	84
B.1.6.8 Using Cypress Calls In Assignments	84
B.1.6.9 Missing Global baseUrl	85
B.1.6.10 Dirty Inputs	85
B.1.6.11 Cleaning up state with after() or afterEach()	86
B.1.6.12 Direct Access to List Elements	87

Capítulo 1

Introdução

Neste capítulo, é introduzido o problema que motiva a pesquisa relatada nesta dissertação (Seção 1.1). Em seguida, é destacado o trabalho proposto (Seção 1.2). Logo depois, é evidenciado as principais contribuições que foram obtidas em torno deste trabalho (Seção 1.3). Por fim, tem-se o esboço do restante desta dissertação (Seção 1.4).

1.1 Motivação

A constante problemática na área da computação reside no fato de que os sistemas de *software* já desenvolvidos frequentemente apresentam falhas ou não atendem às expectativas dos *stakeholders*, devido a defeitos identificados [23] [49]. Com o objetivo de mitigar tais questões e assegurar que os produtos entregues estejam alinhados com as suas finalidades previamente especificadas, além de evitar potenciais prejuízos de valor incalculável, torna-se imperativo integrar testes de qualidade no processo de desenvolvimento de software [46]. Um dos tipos de testes que podem ser empregados são os testes fim-a-fim.

Os testes fim-a-fim representam uma abordagem fundamental na verificação da funcionalidade de aplicativos Web, como destacado por Ricca e Stocco [35]. Esses testes são fundamentados na concepção de cenários de teste, que consistem em uma série de etapas ou ações realizadas em um aplicativo web. Um exemplo que ilustra esse conceito é o fluxo de *SignUp*, no qual o teste navega por todas as etapas do processo de cadastro, inserindo os dados necessários e concluindo o registro. O objetivo desse teste é verificar se um novo usuário é efetivamente criado e se todas as etapas do cenário são executadas conforme o esperado, refletindo assim o propósito essencial do teste fim-a-fim.

A automação da execução de cada caso de teste é viável por meio da implementação de *scripts* de teste, que podem adotar diferentes abordagens, como localização programável (*programmable location*) [26] ou baseada em DOM [27]. A escolha da abordagem apropriada é influenciada por diversos fatores, incluindo a tecnologia utilizada no

desenvolvimento do projeto e suas ferramentas associadas.

Os testes fim-a-fim podem ser comparados ao ciclo de vida do desenvolvimento de software demandando boas práticas para garantir sua eficácia e permitir sua adaptação às novas funcionalidades e tecnologias emergentes [22]. A relevância desses testes decorre da ampla utilização de aplicativos web e da demanda crescente por alta qualidade de código. São considerados cruciais para garantir os níveis de qualidade esperados pelos usuários, contribuindo significativamente para economia de tempo em testes e facilitando o lançamento de aplicativos web com menor incidência de defeitos [22].

Uma das principais vantagens dos testes fim-a-fim reside na capacidade de executar um conjunto completo de testes de forma rápida e autônoma após modificações no aplicativo web [4]. Para implementar esses testes, uma variedade de *frameworks* está disponível, sendo o Selenium a ferramenta mais amplamente utilizada atualmente. O Cypress¹ está emergindo como uma alternativa promissora, ganhando destaque no mercado devido às suas características avançadas e facilidade de uso, e será utilizado como foco neste estudo. Apesar da existência de outras ferramentas similares, como Playwright, optou-se por estudar o Cypress devido a familiaridade da autora com a ferramenta, e ser utilizada na empresa onde ela trabalha atualmente. Além disso, uma análise comparativa dos downloads registrados no *Node Package Manager* (NPM) nos últimos dois anos mostrou que o Cypress popular. Essa popularidade reforça a escolha do Cypress para este estudo, considerando seu crescimento significativo e adoção no mercado.

A linguagem utilizada por essa ferramenta é o JavaScript,² a qual tem experimentado um crescimento significativo ao longo dos anos e consolidou-se como a linguagem de programação padrão para a Web. Por exemplo, segundo a pesquisa mais recente do StackOverflow,³ JavaScript tem sido a linguagem de programação mais popular do mundo pelo décimo primeiro ano consecutivo. Além disso, HTML e CSS têm ocupado consistentemente o segundo lugar nos últimos seis anos. Essa popularidade reflete a importância cada vez maior dos sistemas modernos baseados na Web.

Ao explorar o *framework* Cypress neste trabalho é importante compreender a linguagem JavaScript. Esta é uma linguagem de *script* multiplataforma, orientada a objetos, empregada para tornar as páginas da web interativas, permitindo recursos como animações complexas, botões clicáveis, menus *pop-up*, entre outros [1].

No contexto do *framework* Cypress, os testes são exclusivamente implementados em JavaScript [12]. Com o crescimento das aplicações Web desenvolvidas em JavaScript, a curva de aprendizagem torna-se mais acessível e não é necessário fazer o download de inúmeras bibliotecas e recursos separados para configurar o conjunto de testes. Assim, o Cypress se apresenta como uma alternativa ao popular Selenium⁴ oferecendo diversas

¹<https://docs.cypress.io/guides/overview/why-cypress>

²https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Introductionwhat_is_javascript

³<https://survey.stackoverflow.co/2023/>

⁴selenium <https://www.selenium.dev/>

funcionalidades para o desenvolvimento de testes fim-a-fim [37].

Algumas das características notáveis do Cypress incluem a automação de esperas, eliminando a necessidade de inserir comandos de espera (como *wait* ou *sleep*) antes de interagir com elementos, captura de tela e gravação de vídeos automáticos em caso de falhas, e a capacidade de executar testes em diferentes navegadores da família Firefox e Chrome (incluindo Edge e Electron), tanto localmente quanto em pipelines de Integração Contínua.

O diferencial principal do Cypress reside na sua interação direta com os navegadores. Ao contrário do Selenium, que exige *drivers* específicos para cada navegador, o Cypress não necessita de um *driver* separado. Ele se comunica diretamente com o navegador, enviando comandos por meio do DOM para executar eventos, como cliques. Dado que é um *framework* de testes fim-a-fim, o Cypress concentra-se na verificação de cenários de uso, proporcionando uma abordagem eficaz para garantir a qualidade e a integridade dos aplicativos Web.

Por outro lado, é crucial que os testes fim-a-fim mantenham um padrão de qualidade elevado, devido a uma variedade de fatores [4]:

- **Garantia da Qualidade do Produto Final:** Os testes fim-a-fim são projetados para simular o comportamento do usuário final, abrangendo todos os componentes do sistema. Caso os testes não sejam confiáveis e consistentes, a experiência do produto final será comprometida.
- **Identificação de Problemas não Detectáveis em Outras Camadas de Teste:** Os testes fim-a-fim têm a capacidade de identificar problemas que podem passar despercebidos nos testes de unidade ou de integração. Isso inclui problemas de integração entre componentes, falhas de comunicação entre sistemas, questões de usabilidade, entre outros.
- **Redução de Custos:** Detectar e corrigir problemas em estágios avançados do desenvolvimento ou após o lançamento do sistema pode ser extremamente dispendioso. Investir na criação de testes fim-a-fim de alta qualidade permite a identificação precoce de problemas durante o ciclo de desenvolvimento, o que ajuda a reduzir os custos de manutenção do software, incluindo correções e atualizações.
- **Compatibilidade e Conformidade:** É fundamental garantir a compatibilidade do software com uma variedade de navegadores, dispositivos e ambientes de execução. Os testes fim-a-fim desempenham um papel crucial na verificação dessa compatibilidade, garantindo que o sistema funcione corretamente em todas as plataformas suportadas.

Um dos problemas enfrentados na garantia da qualidade de um sistema são *smells* ou “más práticas no código-fonte” que podem levar a complicações futuras ou reduzir a qualidade

do sistema [7][17][25][44]. Eles são chamados de *smells* (cheiros) porque, assim como um cheiro ruim, indicam a presença de algo potencialmente problemático que pode precisar de atenção [9]. Para exemplificar esse conceito, é apresentado o teste presente na Listagem 1.

```
1 //File : loginTest.cy.js
2
3 describe('TEST', () => {
4   const expectedText = {
5     textMenu: 'administrator'
6   };
7
8   it('Login Test', { tags: '@smoke' }, () => {
9     const username = Cypress.config('username');
10    const password = Cypress.config('password');
11    cy.visit('/login_page.php');
12    cy.get('[id="username"]').clear().type(username);
13    cy.wait(100000);
14    cy.get('input[type="submit"]').click();
15    cy.get('[id="password"]').clear().type(password);
16    cy.get('input[type="submit"]').click();
17    cy.get('.user-info').invoke('text').should('include', expectedText.textMenu);
18  });
19 })
```

Listagem 1: Teste de Login com smell Unnecessary Waiting

Este teste realiza uma operação de login na página do Mantis—um sistema que gerencia falhas de outros softwares—e verifica se o procedimento foi concluído com êxito. No entanto, um aspecto problemático identificado neste teste é a presença do *smell* conhecido como *Unnecessary Waiting*. Esse problema surge quando a espera é excessiva e desnecessária, resultando em uma execução mais lenta dos testes.

Neste estudo, foi optado pelo uso do *framework* Cypress para garantir a qualidade do código dos testes desenvolvidos. Durante a pesquisa, foram identificados dois potenciais problemas que merecem atenção e para os quais este trabalho de mestrado propõe soluções.

- **Problema #1: Ainda não existem catálogos de *smells* para Cypress**
Este problema surge da ausência de catálogos de referência para detectar *smells* específicos no contexto do Cypress. O Cypress foi escolhido por ser uma ferramenta amplamente utilizada para testes de aplicações web, por empresas como PayPal e Intel, bem como por mais de 1M projetos de código aberto hospedados no GitHub.⁵
- **Problema #2: Ainda não existem estudos que usam ferramentas recentes de IA para detecção de *smells***

⁵<https://www.cypress.io/about-us>

Dada a ausência de um catálogo estabelecido de *smells*, não foram encontrados estudos que avaliam como as ferramentas existentes podem auxiliar na detecção de *smells*. Neste trabalho, foi estudado como Modelos de Linguagem já existentes —especificamente o ChatGPT—contribuem para a identificação dos *smells* identificados.

1.2 Trabalho Proposto

Esta dissertação de mestrado foi elaborada com o propósito de abordar os problemas identificados na seção anterior. Com essa finalidade, foram utilizadas duas linhas de trabalho principais:

Para encaminhar uma solução do problema 1, foi conduzida uma Revisão Sistemática da Literatura (RSL), um método de pesquisa estruturado que visa identificar, analisar e sintetizar artigos relacionados a uma questão de pesquisa específica [50]. Esse processo envolve a busca por estudos relevantes em bases de dados acadêmicas, a seleção de artigos de acordo com critérios predefinidos, a extração e análise dos dados desses estudos, e a síntese das respostas à questão de pesquisa em questão. No caso deste estudo, o objetivo foi investigar se há registros de *smells* em testes fim-a-fim nas fontes acadêmicas.

Para complementar esse estudo e encontrar resultados mais abrangentes, foi necessário realizar uma Revisão da Literatura Cinza. Esse tipo de literatura engloba materiais de pesquisa como *white-papers*, magazines, postagens em blogs *on-line*, sites de perguntas e respostas, resultados de pesquisas e informações técnicas [28]. A literatura cinza muitas vezes contém informações valiosas que não estão disponíveis em fontes convencionais e pode ser crucial para preencher lacunas de conhecimento em áreas de pesquisa específicas. Neste caso, foi utilizada para encontrar *smells* que poderiam complementar o catálogo identificado inicialmente na RSL.

Com base nessas investigações, foi proposto um catálogo de *smells* voltado para o desenvolvimento de testes fim-a-fim utilizando o *framework* Cypress. Esse catálogo visa auxiliar os desenvolvedores na identificação e mitigação de problemas comuns durante o processo de desenvolvimento e execução de testes fim-a-fim.

O segundo problema identificado foi a não existência de estudos que usam ferramentas recentes de IA para detecção de *smells*. Para abordar essa questão, foi proposto avaliar o uso Modelos de Linguagem de Grande Escala (*Large Language Models* ou LLMs) [13], para detecção dos *smells* levantados presente no catálogo obtido anteriormente.

A motivação para essa abordagem reside no fato de que esses modelos de linguagem surgiram recentemente e têm sido aplicados com sucesso em diversas áreas. Além

disso, eles são capazes de resolver uma variedade de problemas com baixo esforço, exigindo apenas um *prompt* como entrada. Uma das ferramentas mais proeminentes nesse campo é o ChatGPT, desenvolvido pela OpenAI, que registrou 2,4 bilhões de acessos em janeiro de 2024.⁶ Especificamente, o estudo avaliou o desempenho do ChatGPT como uma ferramenta de detecção de *smells* em testes desenvolvidos com o Cypress, utilizando *prompts* específicos criados para cada um dos *smells* catalogados.

1.3 Contribuições

Esta dissertação apresenta contribuições significativas nas áreas de análise de programas e qualidade de testes automatizados, especificamente no contexto de testes fim-a-fim utilizando o *framework Cypress*. As principais contribuições são:

1. Revisão Sistemática da Literatura (RSL) e Revisão Sistemática da Literatura Cinza (RLC): Foi realizado um levantamento na literatura acadêmica e cinza, permitindo uma compreensão aprofundada do estado da arte sobre *smells* em testes fim-a-fim.
2. Criação de um Catálogo de *Smells* para Testes Fim-a-Fim com Cypress: Com base na RSL e RLC, foi desenvolvido um catálogo composto por 14 *smells* que ocorrem em testes fim-a-fim.
3. Investigação sobre o Uso de LLMs Detecção Automática de *Smells*: Avaliou-se a viabilidade de se utilizar Modelos de Linguagem de Grande Escala (LLMs) para detectar automaticamente os *smells* catalogados. Especificamente, investigou-se a eficiência do uso do ChatGPT como uma ferramenta de detecção dos *smells* catalogados.

1.4 Estrutura da Dissertação

Este trabalho está estruturado da seguinte maneira:

- Capítulo 2: apresenta uma introdução aos conceitos fundamentais relacionados à dissertação, abordando temas como qualidade de sistemas, testes fim-a-fim, Cypress,

⁶[https://www.tudocelular.com/software/noticias/n219370/top-10-brasil-quarto-pais-mais-chatgpt-2024.html#:text=Vale%20dizer%20que%20estes%20n%C3%BAmeros,a%2044%20anos%20\(23.6%25\)](https://www.tudocelular.com/software/noticias/n219370/top-10-brasil-quarto-pais-mais-chatgpt-2024.html#:text=Vale%20dizer%20que%20estes%20n%C3%BAmeros,a%2044%20anos%20(23.6%25))

trabalhos relacionados ao estudo e outras ferramentas que podem ser empregadas no desenvolvimento deste tipo de teste.

- Capítulo 3: Descrição das pesquisas conduzidas na Revisão Sistemática da Literatura e na Revisão da Literatura Cinza. O capítulo inicia com a metodologia empregada nos estudos, detalhando como a pesquisa foi conduzida em cada caso. Em seguida, é apresentado um catálogo contendo 14 *smells* que podem ser identificados ao desenvolver testes fim-a-fim com o uso do Cypress.
- Capítulo 4: Apresentação da avaliação do ChatGPT como ferramenta de detecção dos *smells* catalogados no capítulo anterior. Este capítulo engloba dois tipos de estudos. Primeiramente, um estudo controlado que analisa o software Mantis, com testes específicos conduzidos em um ambiente controlado. A segunda parte deste capítulo descreve um estudo de campo com sistemas de código aberto hospedados no GitHub. Para ambos os casos, foram elaborados *prompts* para serem submetidos ao ChatGPT, a fim de avaliar sua eficácia na detecção dos *smells*. Ao final do capítulo, são descritos e avaliados os resultados obtidos nos estudos.
- Capítulo 5: Resume as conclusões apresentadas ao longo desta dissertação e retrata algumas ideias para trabalhos futuros.

Capítulo 2

Conceitos Básicos

Neste Capítulo, será apresentado os conceitos básicos para o entendimento da Dissertação. Primeiramente é apresentado a definição de qualidade de um sistema (Seção 2.1). Logo após, é descrito o conceito de testes fim-a-fim (Seção 2.2). Em seguida, o texto descreve o que é o *framework* Cypress e apresenta um exemplo de sua aplicação (Seção 2.3). Posteriormente é abordado sobre outras ferramentas que podem ser utilizadas para desenvolvimento de testes fim-a-fim (Seção 2.4). Em seguida, são discutidos os trabalhos relacionados a essa pesquisa (Seção 2.5). Por fim, é apresentado as conclusões deste capítulo (Seção 2.6).

2.1 Qualidade de Software

Qualidade de Software é um conceito complexo. Conforme Pressman [33], qualidade pode ser entendida como uma gestão eficaz aplicada para criar um produto útil que forneça valor mensurável tanto para seus criadores quanto para seus usuários. A qualidade deve estar em conformidade com requisitos funcionais e de desempenho documentados em todo sistema desenvolvido profissionalmente. Segundo Staa [48], “a qualidade de um artefato é um conjunto de propriedades que devem ser satisfeitas em determinado grau, de modo a atender às necessidades de seus usuários e clientes”.

Uma maneira de garantir a qualidade de um sistema é a introdução de atividades de teste no processo de desenvolvimento [46]. Atualmente, o teste é uma das práticas mais valorizadas na programação e está em constante evolução, especialmente no contexto ágil [46]. Anteriormente, no desenvolvimento em cascata, os testes eram realizados em uma fase separada, após o levantamento de requisitos, análise, projeto e codificação. Essa abordagem envolve equipes separadas para verificar a conformidade dos requisitos com a implementação. Muitas das vezes, os testes eram realizados manualmente, com um indivíduo utilizando o sistema fornecendo dados de entrada e verificando as saídas.

Com a adoção do movimento ágil, ocorreu uma revolução nos testes de programas.

A automação tornou-se predominante, e os desenvolvedores passaram a implementar não apenas as classes do sistema, mas também o código de teste correspondente [46]. Os testes passaram a ser implementados antes mesmo da conclusão das classes do sistema [46]. As grandes equipes de testes foram substituídas por equipes menores ou até mesmo pelos próprios desenvolvedores, agora responsáveis por seus próprios testes. Além de identificar defeitos, os testes têm funções adicionais, como verificar se as classes continuam funcionando após correções em outras partes do sistema e servir como documentação do código de produção. Michael Feathers enfatiza que, se um código não for acompanhado por testes, pode ser considerado de baixa qualidade ou até mesmo um código legado [46].

A organização dos tipos de testes segue uma estrutura conhecida como a “pirâmide de testes” concebida por Mike Cohn e descrita por Valente [46]. Essa abordagem classifica os testes com base em sua granularidade. Na base da pirâmide como pode ser observado na Figura 2.1, estão os testes de unidade, que verificam pequenas partes de código, geralmente uma classe individual. Esses testes são simples, fáceis de implementar e executam rapidamente. A segunda camada da pirâmide abrange os testes de integração ou testes de serviços, que verificam funcionalidades ou transações completas de um sistema, frequentemente envolvendo várias classes de pacotes diferentes e, às vezes, componentes externos, como bancos de dados. No topo da pirâmide, estão os testes de sistema, também conhecidos como testes de interface com o usuário ou testes fim-a-fim, que simulam o comportamento de um usuário real. Esses testes são mais caros, mais lentos e menos numerosos. A Seção 2.2 descreve em detalhes este tipo de teste.

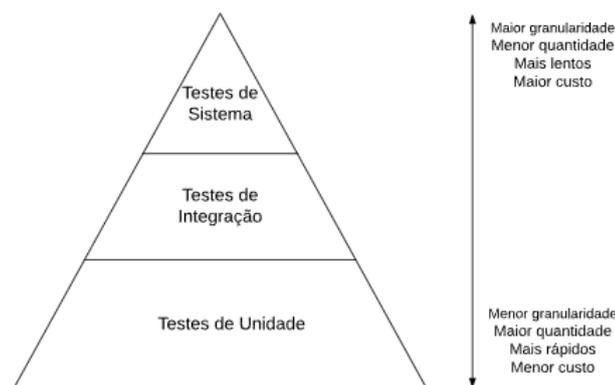


Figura 2.1: Pirâmide de Teste, retirado do livro Engenharia de Software Moderna.

2.2 Testes Fim-a-Fim

Conforme Ricca et al [28], os testes fim-a-fim se baseiam na ideia de cenários de teste e consistem em uma série de etapas ou ações realizadas em sistema. Um exemplo ilustrativo desse conceito é o fluxo de *SignUp* (cadastro de usuário). Nesse cenário de teste, o processo começa na página de cadastro, onde são inseridos todos os dados necessários, levando ao cadastro de um novo usuário. O objetivo central dos testes fim-a-fim é verificar se esse cenário foi executado conforme o esperado. Pode haver um ou mais casos de teste derivados de um cenário, nos quais os dados para cada etapa devem ser definidos, juntamente com os resultados esperados. Esses resultados esperados são usados para verificar se as ações ocorreram conforme o planejado.

Os testes fim-a-fim podem ser comparados ao desenvolvimento de um software, pois seguem um ciclo e requerem boas práticas para serem eficazes [22]. Atualmente, a implementação desses testes se dá por meio de ferramentas de automação. Essas ferramentas tem por objetivo melhorar a qualidade da análise e o tempo de execução dos testes [42].

Neste estudo, o *framework* Cypress foi escolhido como objeto de estudo. Ele é conhecido por sua facilidade de uso e pela capacidade de executar os testes diretamente no navegador. Na próxima seção, serão detalhados mais aspectos sobre esse *framework*.

2.3 Cypress

O framework Cypress¹ foi desenvolvido pela empresa Cypress.io em 2017 e apresenta diversas funcionalidades essenciais para o desenvolvimento de testes fim-a-fim. Dentre suas características mais notáveis destacam-se:

- Esperas automáticas: Em contraste com outros *frameworks*, o Cypress elimina a necessidade de adicionar explicitamente comandos de espera (*wait* ou *sleep*) antes de executar comandos e *asserts*. Por padrão, o Cypress aguarda automaticamente que os elementos estejam visíveis antes de tentar interagir com eles.
- Captura de tela e vídeos: O Cypress oferece a capacidade de capturar automaticamente telas em caso de falhas nos testes ou gravar vídeos de todo o conjunto de testes quando executados na Interface de Linha de Comando (CLI).

¹<https://www.cypress.io/>

- Execução de testes em diferentes navegadores: O Cypress suporta a execução de testes em uma variedade de navegadores, incluindo aqueles da família Firefox e Chrome (incluindo Edge e Electron), tanto localmente como parte de um pipeline de Integração Contínua.

O principal diferencial do Cypress reside na sua interação direta com os navegadores. Ao contrário do Selenium, que requer *drivers* específicos para cada navegador, o Cypress não necessita de um *driver* adicional. A comunicação é estabelecida por meio do envio direto de comandos para o navegador, utilizando o DOM para executar eventos como cliques. Como um *framework* voltado para testes fim-a-fim, o foco principal do Cypress está na verificação de cenários de uso.

A seguir é mostrado um exemplo de um cenário de autenticação de login na ferramenta Mantis Bug Tracker,² um popular rastreador de falhas de código aberto. O fluxo de autenticação no Mantis envolve três etapas principais: preenchimento do campo de nome de usuário, preenchimento da senha e validação do usuário na tela inicial. A seguir, é exibido um exemplo de teste fim-a-fim no Cypress para este cenário.

Para melhor compreensão do teste, é importante entender o conceito de *Page Object*, que é usado para organizar os elementos da página. Nesse contexto, os elementos da tela são agrupados em classes de acordo com as páginas às quais pertencem, detalhando todos os elementos e métodos específicos daquela página [3] [15]. Conforme demonstrado nas Listagens 2 e 3, são utilizados dois *Page Objects*: as classes `LoginPage` e `HomePage`. Além disso, um *Page Object* é composto por Localizadores e Ações. Os Localizadores consistem em variáveis cujo valor identifica elementos na página por meio do seletor CSS. As Ações correspondem a métodos que serão utilizados para realizar interações com os elementos da página da web. Para executar essas interações, as Ações fazem uso dos valores presentes nos Localizadores para identificar os elementos da interface que serão submetidos ao teste do cenário.

²<https://www.mantisbt.org/index.php>

```
1  const el = {
2    campoUsuario: `#usuario`,
3    campoSenha: `#senha`,
4    botaoLogar: `.width-40`
5  }
6  class LoginPage {
7    preencherUser(usuario){
8      cy.get(el.campoUsuario).type(usuario)
9    }
10   preencherSenha(senha){
11     cy.get(el.campoSenha).type(senha)
12   }
13   clicarLogar(){
14     cy.get(el.botaoLogar).click()
15   }
16   clicarEntrar(){
17     cy.xpath("//input[@value='Entrar']").click()
18   }
19 }
20 export default new LoginPage()
21
```

Listagem 2: Login Page

```
1  const el = {
2    tituloHomeText: `.user-info`
3  }
4  class HomePage {
5    validarTituloHome(param){
6      cy.get(el.tituloHomeText).invoke(`text`)
7      .should(`include`, param);
8    }
9  }
10 export default new HomePage();
```

Listagem 3: Home Page

Ainda sobre o exemplo mencionado anteriormente, foram utilizados os seguintes comandos do Cypress para interagir com as páginas do sistema alvo:

- **get**: Responsável por selecionar o elemento no qual uma determinada ação será executada. Por exemplo, nas linhas 9, 13 e 17 de `LoginPage`, foram selecionados os elementos de usuário, senha, botão de próximo e entrar na página.
- **type**: Usado para inserir dados em um elemento. Nas linhas 9 e 13 de `LoginPage`, solicitou-se a inserção dos parâmetros usuário e senha nos campos correspondentes.

- **click**: Utilizado para clicar em um elemento. Por exemplo, nas linhas 17 e 21 de `LoginPage`, foi solicitado um clique nos elementos próximo e entrar.
- **invoke**: Como no Cypress não são executados de forma síncrona, deve se verificar se o elemento especificado por aquele seletor está disponível ou não, utiliza `invoke()` no Cypress para espera até que um elemento esteja disponível para executar a ação, ou um tempo *default*. Por exemplo, na linha 7 de `HomePage`, espera-se que o elemento descrito na propriedade `tituloHomeText` esteja disponível.
- **should**: Utilizado para testar asserções. Por exemplo, na linha 8 de `HomePage`, verifica-se se o *usuario* que está sendo exibido é o mesmo que foi inserido durante o login.

Por fim, a Listagem 4 descreve o cenário de teste “RealizarLoginSucesso”, responsável por verificar uma autenticação bem-sucedida. Inicialmente, esse cenário preenche os campos de login e senha para acessar a página inicial do Mantis. Os valores de *usuario* e senha são recuperados de variáveis presentes no arquivo `Cypress.json`. Para acessá-las, utilizou-se a função `Cypress.config`. Em seguida, a classe `LoginPage` é usada para preencher os campos de *usuario* e senha por meio dos métodos `preencherUser` e `preencherSenha`. Essa classe também contém métodos para clicar nos botões próximo e entrar, que são executados por meio dos métodos `clicarLogar` e `clicarEntrar`. Por fim, o método `validarTituloHome` é chamado para verificar se a autenticação foi realizada com sucesso.

```
1 describe('Realizar Login', ()=>{
2   beforeEach(()=>{
3     cy.visit(Cypress.config('url'))
4   })
5   it('RealizarLoginSucesso', ()=>{
6     var usuario = Cypress.config('usuario')
7     var senha = Cypress.config('senha')
8     var expectedText = "administrador" loginPage.preencherUser(usuario)
9     loginPage.clicarLogar()
10    loginPage.preencherSenha(senha)
11    loginPage.clicarEntrar()
12    homePage.validarTituloHome(expectedText);
13  })
14 })
15
```

Listagem 4: Exemplo de Teste Fim-a-Fim.

2.4 Outras Ferramentas

Nesta seção, são analisadas outras ferramentas para automação de testes fim-a-fim, apresentando-se um resumo das principais características de cada uma delas:

- Playwright,³ criado pela Microsoft em 2020, é uma plataforma de automação de testes entre navegadores de código aberto que suporta Chromium, Firefox e WebKit. Ele oferece suporte a várias linguagens de programação, como JavaScript, TypeScript, Python, .NET e Java. Uma característica notável do Playwright é sua capacidade de capturar vídeos, capturas de tela e outros artefatos em caso de falha, simplificando o rastreamento de problemas. Além disso, o Playwright inclui a poderosa ferramenta Codegen, que gera código com base em interações do usuário usando seletores resilientes baseados em texto.
- TestCafe,⁴ desenvolvido pela DevExpress em 2016, é uma ferramenta que oferece suporte a navegadores *desktop*, móveis, ambientes remotos e em nuvem, tanto para testes de interface do usuário quanto para testes *headless*. Os testes podem ser desenvolvidos em JavaScript ou TypeScript. Uma vantagem do TestCafe é sua funcionalidade integrada chamada Visual Test Recorder, que permite aos desenvolvedores criar, editar e executar testes fim-a-fim sem a necessidade de selecionar manualmente os elementos da interface. Além disso, o TestCafe lida automaticamente com tempos de espera durante o carregamento de elementos da página.
- Robot Framework,⁵ criado em 2008 por Pekka Klärck, Janne Härkönen e outros, é um *framework* de automação de teste de código aberto projetada para testes de aceitação. Embora tenha sido escrito em Python, ele oferece suporte a várias outras linguagens, como Java, .NET e Perl. O Robot Framework pode ser facilmente estendido com bibliotecas e palavras-chave personalizadas, o que o torna bastante extensível.
- Selenium,⁶ desenvolvido por Simon Stewart em 2004, é uma das ferramentas mais populares e amplamente utilizadas para automação de testes de aplicações web. Sua suíte de ferramentas oferece suporte a várias linguagens de programação, como C, Java e diversos navegadores, incluindo Chrome e Edge. O Selenium inclui três componentes principais: o primeiro é o Selenium IDE: Usado para testes exploratórios rápidos, permite gravar ações do testador na aplicação web para reproduzi-las posteriormente. Já o segundo é o Selenium WebDriver: Permite o desenvolvimento de

³<https://playwright.dev/>

⁴<https://testcafe.io/>

⁵<https://robotframework.org/>

⁶<https://www.selenium.dev/>

scripts de teste em linguagens de programação de alto nível usando *APIs* do *framework*. Por fim, o Selenium Grid oferece serviços para distribuir *scripts* de teste em vários navegadores e plataformas.

2.5 Trabalhos Relacionados

No estado da arte, diversos trabalhos abordam a ocorrência de *smells* em sistemas, tais como artigos, capítulos e livros [16, 29, 30, 31, 32]. Sendo o principal o capítulo de Fowler e Beck [17], que propõe 22 *code smells* para o paradigma orientado a objetos.

No contexto deste trabalho, foram encontradas pesquisas que também fazem uso da literatura cinza para catalogar *smells* em contextos específicos como JavaScript [14], iOS [19], Android [20], CSS [34], Puppet [38].

Existem também trabalhos que utilizaram a literatura cinza no contexto de *code smells*. Ricca e Stocco [35] investigaram sobre as melhores práticas para automação de testes fim-a-fim para aplicações *Web*. Foram analisadas mais de 2,400 fontes (por exemplo, postagens em blogs, *white-papers*, manuais do usuário, repositórios GitHub) sobre como desenvolver e manter códigos de teste. Os autores destacaram três práticas principais: *Manage the synchronization w/ the web app*, *Create robust/proper locators/selectors*, e *Keep the tests atomic and short*.

2.5.1 Code Smells

O trabalho de Sobrinho et al. [41] consiste em uma revisão sistemática da literatura sobre *bad smells*, abrangendo documentos do período de 1990 a 2017. As principais descobertas do estudo foram:

- **Quais:** Alguns *bad smells* são mais estudados na literatura do que outros, e alguns estão intrinsecamente inter-relacionados.
- **Quando:** O interesse em código duplicado surgiu antes das publicações de referência de Fowler e Beck, enquanto outros tipos de *bad smells* começaram a ser estudados após essas publicações, com uma tendência crescente na última década.

- **O quê:** A variabilidade dos objetivos, descobertas e configurações experimentais pode ser responsável por algumas descobertas contraditórias sobre *bad smells*.
- **Quem:** Autores têm diferentes níveis de interesse no assunto, com alguns publicando esporadicamente e outros continuamente.
- **Onde:** Algumas conferências e publicações são mais propensas a disseminar conhecimento sobre certos tipos de *bad smells*, como o Código Duplicado.

Os autores concluíram que existe uma vasta base de conhecimento sobre *bad smells*. No entanto, ainda existem algumas divergências devido à falta de generalidade dos estudos. Além disso, poucas evidências de relações de causa e efeito foram encontradas, o que limita a aplicação prática dos dados empíricos disponíveis.

Já o trabalho de Vegi e Valente [47] identifica, por meio da revisão na literatura cinza e através de entrevistas com desenvolvedores Elixir, *code smells* identificados no desenvolvimento de sistemas com essa linguagem. Os resultados indicam que 11 dos 22 *code smells* tradicionais catalogados por Fowler e Beck são discutidos pelos desenvolvedores Elixir. Os autores ainda investigaram quais desses *smells* são identificados pelo Credo, uma ferramenta de análise estática de código conhecida para Elixir. Apenas dois *code smells* tradicionais (*Duplicated code* e *Long Parameter List*) e um específico (*Compile-time app configuration*) são detectados automaticamente por esta ferramenta.

2.5.2 Test Smells

Outros trabalhos investigam na literatura cinza *smells* em testes. O artigo de Rwemalika et al. [36] estuda sobre *smells* que ocorrem nos testes interativos do usuário do sistema (SUITs); um SUIT pode ser definido como um teste automático que simula o comportamento de um usuário de um sistema com um *front-end* gráfico para garantir que ele atenda às especificações que foram listadas para o mesmo ser construído. A pesquisa caracterizou a difusão e a frequência com que esses *smells* surgiram. Para responder a essas perguntas, foi utilizado uma revisão da literatura multi-vocal e um estudo empírico envolvendo um grande projeto industrial e 12 repositórios de código aberto. No total, 35 *smells* foram encontrados. Para 16 dos 35 *smells* deste catálogo, foram propostas métricas para detectar a difusão e possibilidade de refatoração nos códigos de teste. Também, foi avaliado a prevalência de *smells* de SUIT e sua remoção em mais de dois milhões de testes, além de um estudo comparativo tanto em código aberto quanto em projetos industriais. Os resultados desta análise exploratória indicam uma tendência de evitar a introdução de

smells pelos engenheiros de automação de teste. No entanto, *smells* tratados de forma incorreta tendem a não ser removidos mais tarde. Assim, é de extrema importância capturar as más práticas assim que são introduzidas.

Seguindo a mesma vertente, Soares et al. [40] destacaram que, apesar da constante evolução dos recursos presentes em *frameworks* de teste, não é conhecido até que ponto os desenvolvedores usam os recursos recém-introduzidos, e se esses recursos de fato ajudam a refatorar o código de teste existente para remover os *smells*. Logo, a pesquisa apresentou um estudo de método misto sobre o uso e más práticas dos recursos presentes no JUnit. O estudo foi dividido em três partes: (i) avaliação de novos recursos do JUnit 5; (ii) proposta de novas refatorações baseadas nos recursos do JUnit 5 para remover *code smells*; e (iii) avaliação das refatorações feitas de acordo com pesquisa online e envio de *Pull Requests* para GitHub. Os autores destacaram que a evolução e os novos recursos do *framework* de testes devem ser mais bem difundidos para desenvolvedores, pesquisadores e profissionais.

2.6 Considerações Finais

É evidente que a qualidade de um é uma das atividades mais cruciais no ciclo de desenvolvimento atualmente. Os testes desempenham um papel fundamental nessa busca pela qualidade e representam uma área em constante crescimento. Os testes fim-a-fim são amplamente empregados pelas empresas para avaliar o comportamento do sistema sob a perspectiva do usuário. Optou-se no presente trabalho pelo uso da ferramenta Cypress devido a sua facilidade de uso e desenvolvimento, e crescente adoção no mercado. No entanto, é importante observar que há outras ferramentas disponíveis, e que podem ser investigadas em trabalhos futuros.

Este trabalho se diferencia dos estudos mencionados ao focar na revisão da literatura acadêmica e complementar essa análise com a literatura cinza para catalogar *smells* em testes fim-a-fim desenvolvidos especificamente no contexto do Cypress, algo que não havia sido explorado anteriormente nas pesquisas existentes.

Uma inovação significativa deste trabalho é a análise do uso de modelos de linguagem natural (LLMs), especificamente o ChatGPT, para detectar *test smells*. A metodologia adotada permitiu a identificação de *test smells* durante o desenvolvimento de testes fim-a-fim com Cypress, proporcionando uma nova perspectiva sobre a detecção e mitigação de *smells*. Essa abordagem inédita amplia o conhecimento atual e contribui significativamente para a literatura, demonstrando a eficácia de LLMs na identificação de *smells* em testes automatizados.

Capítulo 3

Revisões Sistemáticas da Literatura

O estudo descrito neste capítulo tem como objetivo investigar os principais *smells* (problemas ou más práticas) relacionados a testes fim-a-fim, documentados na literatura acadêmica (artigos, livros, etc) e na literatura cinza (constituída por *white-papers*, *magazines*, *postagens em blogs on-line*, *sites*, etc). Para isso, busca-se responder às seguintes questões de pesquisa:

RQ1: Quais são os *smells* relativos a testes fim-a-fim descritos na literatura acadêmica que aplicam-se ao Cypress?

RQ2: Quais são os *smells* relativos a testes fim-a-fim descritos na literatura cinza que aplicam-se ao Cypress?

Este capítulo está estruturado da seguinte forma: Primeiramente é abordada a pesquisa realizada na literatura acadêmica (revisão sistemática da literatura). Especificamente, o protocolo da revisão sistemática da literatura (RSL) está descrito na Seção 3.1. Logo após é apresentado o catálogo de *smells* gerado a partir do estudo conduzido utilizando a RSL (Seção 3.2). Para complementar o estudo da RSL é conduzido um estudo revisão da literatura cinza (RLC) (Seção 3.3). Em seguida, discute-se o catálogo de *smells* identificado na pesquisa da RLC (Seção 3.4). Por fim, é apresentado as considerações finais do capítulo (Seção 3.6).

3.1 Revisão da Literatura Científica

A Figura 3.1 apresenta uma síntese do procedimento realizado na revisão sistemática da literatura para a seleção de artigos científicos e identificação dos *smells*. Ao todo, esse procedimento é dividido em quatro etapas principais: (1) pesquisa nas bibliotecas digitais, (2) filtragem de artigos, (3) extração de *smells* e (4) validação dos *smells*. Cada uma destas etapas estão descritas em mais detalhes a seguir.



Figura 3.1: Protocolo da Revisão da Literatura Científica

Pesquisa nas bibliotecas digitais: Para obter uma lista inicial de artigos, foram selecionadas as seguintes fontes de dados: *ACM DL*, *IEEE Xplore*, *ScienceDirect*, *Springer* e *Wiley*. A escolha dessas bibliotecas foi baseada no artigo de Sobrinho et al. [41], que serviu como referência para a elaboração desta pesquisa. Uma vez selecionadas as bibliotecas, foi criada uma *query* de pesquisa para identificar potenciais artigos científicos relacionados a *smells* em testes fim-a-fim. Especificamente, foi construída uma *query* para buscar artigos que mencionem em seu título ou *abstract* os seguintes termos:

```

("end-to-end testing" OR "end-to-end test"
OR "gui testing" OR "gui test" OR
"system testing" OR "system test" OR
"frontend testing" OR "frontend test")
AND ("code smell" OR "test smell"
OR "bad smell" OR "anti-pattern" OR
"bad practice" OR "code smells" OR
"test smells" OR "bad smells" OR
"anti-patterns" OR "bad practices")

```

Naturalmente, cada portal adota sua própria estrutura de busca, resultando na adaptação da *string* de busca para atender a essas limitações. Por exemplo, foi necessário segmentar a *string* de busca em quatro *queries* distintas no portal *ScienceDirect*, devido a restrição de caracteres e conectores, e.g., “Use menos conectores *booleanos* (máximo de 8 por campo)”.

Além disso, o portal *Springer* obteve um maior número de artigos coletados por não possuir uma busca por *abstract*. Uma busca por título poderia ter sido realizada, no entanto, esta abordagem se mostra bastante restritiva e poderia excluir artigos relevantes. Diante disso, optou-se por uma pesquisa mais abrangente, onde as palavras-chave da *string* foram procuradas em todo o conteúdo dos artigos.

As *queries* descritas acima foram executadas em Novembro de 2023. A tabela 3.1 apresenta a quantidade de artigos encontrados em cada biblioteca. Ao todo, 49 artigos distintos foram identificados. É importante notar que dois dos artigos encontrados na *ACM* também foram localizados na *IEEE Xplore*, resultando em uma discrepância na soma total dos artigos apresentados na tabela 3.1.

Biblioteca	URL	Artigos
ACM	https://dl.acm.org	3
IEEEExplore	https://ieeexplore.ieee.org/Xplore/home.jsp	3
Science Direct	https://www.sciencedirect.com/search	1
Wiley	https://onlinelibrary.wiley.com	0

Tabela 3.1: Quantidade de artigos encontrados em cada biblioteca

Filtragem de artigos: Nesta etapa, foram definidos critérios específicos para exclusão dos documentos encontrados na etapa anterior. A autora realizou uma leitura do *abstract* de cada artigo, com intuito de checar se os artigos encontrados possuíam relevância para o objetivo da pesquisa, isto é, catalogar *smells* em testes fim-a-fim implementados usando o Cypress. Foram então excluídos alguns artigos que não eram de relevância para o estudo. A Tabela 3.2 apresenta a relação de artigos excluídos em cada biblioteca e o critério de exclusão utilizado.

Biblioteca	Motivos para Exclusão	Quantidade
ACM	artigos que não possuem relação com testes	(1)
	artigos sobre testes de sistemas móveis	(1)
IEEEExplore	artigos que não possuem relação com testes	(1)
	artigos sobre testes de sistemas móveis	(1)
Science Direct	artigos que não possuem relação com testes	(1)
	artigos sobre testes de unidade	(1)
Springer	artigos sobre evolução de testes smells	(2)
	artigos que não possuem relação com testes	(32)
	artigos sobre ferramentas para detectar smells	(1)
	artigos sobre testes de sistemas móveis	(3)
	artigos sobre testes de microsserviços	(2)
	artigos editoriais	(1)
	artigos sobre design smells	(1)
Wiley	nenhum artigo excluído	(0)

Tabela 3.2: Quantidade de artigos excluídos em cada biblioteca digital. A terceira coluna apresenta a quantidade de artigos excluídos por motivo

Ao total, restaram três artigos, os quais estão descritos na Tabela 3.3, junto do local em que foram publicados e o ano da publicação.

Biblioteca	Artigo	Local	Mês/Ano
ACM	Guidelines for GUI testing maintenance: a linter for test smell detection	13th International Workshop on Automating Test Case Design, Selection and Evaluation	Novembro 2022
IEEEExplore	Bad Smells and Refactoring Methods for GUI Test Scripts	13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing	Agosto 2012
Springer	Smells in system user interactive tests	Empirical Software Engineering Journal	Dezembro 2021

Tabela 3.3: Artigos selecionados da RSL

Extração dos Smells: Nesta etapa foi feita leitura e análise de cada artigo para extrair os *smells*. Após a leitura dos artigos, identificou-se os trechos que mencionam explicitamente os *smells*, e extraiu-se essa informação. No final desta etapa, foram identificados 69 candidatos a *smells*. A Tabela 3.4 cita a quantidade de *smells* encontrados em cada artigo selecionado.

Artigo	Smells
Guidelines for GUI testing maintenance: a linter for test smell detection	23
Bad Smells and Refactoring Methods for GUI Test Scripts	11
Smells in system user interactive tests	35

Tabela 3.4: Smells identificados em cada artigo

Validação dos Smells: Por fim, cada candidato a *smell* foi analisado sob duas perspectivas. Primeiro, um agrupamento manual de *smells* que possuíam o mesmo significado, mas apresentavam nomes distintos, reduzindo o número de candidatos para 60. Foi então verificado se esses *smells* eram de fato voltados a testes fim-a-fim e aplicados ao Cypress, ou se poderiam ser adaptados para testes implementados usando essa ferramenta. Explicando melhor, foram descartados *smells* das seguintes categorias:

- **Smell Tradicional:** São *smells* que podem ser identificados em outros módulos do sistema, i.e., não são voltados para testes. Os *smells* desse categoria foram descartados por serem comuns em qualquer código desenvolvido.

- **Test Smell Genérico:** São *smells* que valem para qualquer tipo de teste, incluindo testes de unidade, testes de integração, testes fim-a-fim, etc. Os *smells* dessa categoria foram descartados pois não são específicos para testes fim-a-fim portanto fora do escopo deste estudo.
- **Não relacionado com código:** São *test smells* que estão relacionados a outros aspectos do sistema como, ambientes de execução. Os *smells* dessa categoria foram descartados, pois não podem ser detectados por meio do código.
- **Smell de Teste Fim-a-Fim mas não específico do Cypress:** São *test smells* que são de outros *frameworks* de testes fim-a-fim, e portanto não podem ser aplicados ao Cypress. Os *smells* dessa categoria foram descartados devido à ênfase deste estudo estar centrada no Cypress.

Em geral, a decisão de remover esses *smells* foi tomada com base na necessidade de manter o foco nas questões mais pertinentes ao contexto desta pesquisa. Como resultado, restaram apenas cinco *smells*, os quais serão catalogados na Seção 3.2. O Apêndice A.1 apresenta a lista dos *smells* excluídos neste procedimento.

3.2 Catálogo de Smells RSL

Por fim, foi compilado um catálogo de *smells* específicos de testes fim-a-fim aplicáveis ao Cypress, obtidos através da RSL. As definições dos *smells* foram traduzidas para português, a fim de ter mais coerência com idioma adotado no texto, as definições originais dos *smells* estão em inglês e podem ser encontrados no Apêndice B.1. Abaixo estão os cinco *smells* selecionados, juntamente com suas respectivas descrições:

3.2.1 Unnecessary Waiting

Descrição: Um teste que aguarda por um período arbitrário usando `cy.wait()`, por exemplo, `cy.wait(2000)`. Porém, o uso de *wait* é necessário, quando é preciso sincronizar a aplicação com os testes, quando há atrasos devido ao carregamento de dados ou operações assíncronas, com isso evita falhas prematuras nos testes.

Razão: Este é considerado um *smell* porque pode tornar o teste mais lento ou mascarar uma lentidão no sistema.

Artigos: A1

3.2.2 Brittle Selectors

Descrição: Como o próprio nome sugere, esses seletores têm grandes chances de serem renomeados no futuro, por exemplo, usando uma sequência de classes específicas como `cy.get('.btn.btn-large-890')` em vez de identificadores mais estáveis como `cy.get('[id="submit"]')`.

Razão: Este é considerado um *smell* porque, o teste pode ser tornar frágil se houver um seletor mal nomeado na interface. Se o seletor for alterado, o teste precisará de manutenção cada vez que isso acontecer.

Artigos: A1, A2, A3

3.2.3 Adopt Page Object Pattern

Descrição: Esse *smell* se refere ao uso de *Page Objects* ao escrever testes. A introdução desse tipo de estrutura de dados pode tornar os testes difíceis de manter.

Razão: Este é considerado *smell* no Cypress, devido à natureza natureza assíncrona do Cypress. Pois, *Page Object* foca na estrutura de elementos da interface diferentemente do *App Actions*, uma alternativa mais viável devido a melhoria da manutenibilidade ao encapsular comportamentos da aplicação.

Artigos: A1

3.2.4 Use Relative URLs Locator Instead Of Absolute Ones

Descrição: Esse *smell* ocorre quando uma URL absoluta é usada como parâmetro na chamada de `cy.visit()`, por exemplo, `cy.visit('http://example.com/courses')`; A melhor prática é usar uma URL relativa como parâmetro, como mostrado abaixo: `cy.visit('/courses')`;

Razão: É considerado um *smell* devido ao uso de URLs absolutas poderem levar a testes mais frágeis. Uma vez que URLs absolutas contemplam o domínio completo da aplicação, a mudança de domínio também levará a mudança da URL no teste.

Artigos: A1

3.2.5 Unnecessary Navigation

Descrição: Esse *smell* está relacionado a testes que realizam ações que não estão diretamente ligadas ao foco do teste. Por exemplo, um teste para verificar o mecanismo de busca de um livro realizando uma série de ações prévias que não estão diretamente relacionadas ao teste em si, como autenticação, redirecionamento e acesso à página de busca.¹ Devido a isso, é possível encontrar falhas que não necessariamente deveriam ser capturados com o teste realizado.

Razão: Este é um *smell* devido a adição de complexidade desnecessária aos testes que podem levar a falsos positivos. O *Unnecessary Navigation*, pode tornar os testes mais lentos e trazer a chance de encontrar falhas que não estão no escopo do teste.

Artigos: A3

¹<https://mattarcherblog.wordpress.com/2010/11/29/how-test-automation-with-selenium-or-watir-can-fail/>

3.3 Revisão da Literatura Cinza

A revisão sistemática da literatura evidenciou a escassez de estudos científicos formais sobre o assunto, com conhecimento limitado apenas à catalogação de alguns *smells* na documentação do Cypress. Essa ausência motivou a opção pela revisão da literatura cinza para responder à questão de pesquisa RQ2.

De acordo com Ricca e Stocco [35], a literatura cinza “abrange uma ampla gama de fontes, como *white-papers*, revistas, postagens em *blogs* online, sites de perguntas e respostas, resultados de pesquisas e informações técnicas”. Essa diversidade a torna uma valiosa fonte nos quais profissionais compartilham suas experiências e propõem melhores práticas, diretrizes e dicas relacionadas a diversos aspectos da qualidade do código.

Dessa forma, recomenda-se o uso da literatura cinza como fonte de evidência, como um complemento das revisões formais da literatura. Isso se justifica pelo fato de que os profissionais são os principais autores ao compartilhar suas experiências em *blogs*, fóruns de perguntas e respostas, entre outras fontes [2, 18, 24, 51]. No contexto da pesquisa em questão, centrada em uma tecnologia emergente sem informações relevantes na literatura acadêmica, a literatura cinza se torna crucial.

A Figura 3.2 resume o procedimento adotado para a seleção de artigos e identificação de *smells*, também definido em quatro etapas: (1) pesquisa no Google, (2) filtragem de artigos, (3) extração de *smells* e (4) validação dos *smells*. As próximas seções detalham cada uma dessas etapas.

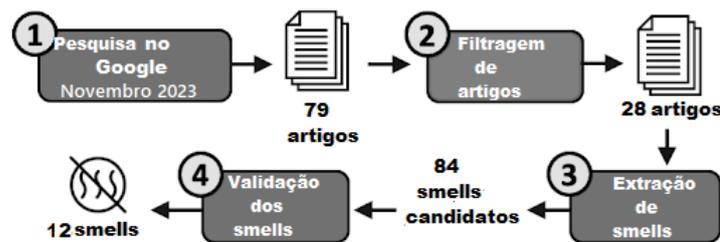


Figura 3.2: Processo Literatura Cinza

Pesquisa no Google: Para obter uma lista inicial de artigos relacionados a *code smells* no contexto do Cypress, uma pesquisa foi realizada no Google em Novembro de 2023. A consulta foi formulada incluindo a palavra-chave “cypress” para restringir os resultados à ferramenta específica. Além disso, termos como *code smell*, *bad smell*, *anti-pattern*, *bad practice* e *test smell* foram adicionados para identificar documentos relacionados a esses conceitos. Ao todo, 79 artigos foram obtidos por meio da seguinte *query* de busca:

```
("code smell" OR "bad smell" OR "anti-pattern" OR "bad practice" OR "test smell")
AND ("cypress")
```

Filtragem de Artigos: Os 79 artigos obtidos como resultado da pesquisa inicial foram submetidos a 11 critérios de exclusão, vistos na Tabela 3.5. Do total, 51 artigos se encaixaram em um dos critérios, e portanto foram descartados.

Motivo para Exclusão	Quantidade
artigos que não são relacionados com teste	(21)
artigos que fazem comparação entre Cypress e ferramentas	(5)
artigos que relata sobre Cypress mas não sobre smells	(9)
artigos que aborda sobre testes fim-a-fim mas não sobre smells e Cypress	(4)
artigos que aborda sobre testes fim-a-fim e smells mas não sobre Cypress	(1)
artigos que citam sobre testes unitários	(2)
artigos que trazem solução de smells	(2)
artigos que discorrem sobre condições que acontecem no teste	(3)
artigos que abordam sobre plugin	(1)
artigos que abordam sobre ferramentas	(1)
artigos que relatam sobre desenvolvimento de código	(2)

Tabela 3.5: Critérios utilizados para excluir artigos durante a filtragem. A segunda coluna apresenta a quantidade de artigos excluídos por motivo

Extração de Smells: Uma análise minuciosa dos 28 artigos que restaram da etapa anterior foi realizada para extrair informações sobre os *smells*. Cada artigo foi lido com foco nas partes que descrevem os *smells*. A etapa resultou em 84 ocorrências de candidatos a *smells*.

Validação dos Smells: Na etapa de validação, os candidatos a *code smells* foram submetidos a uma análise mais aprofundada. Os artigos foram lidos e classificados em 4 categorias: *Smell Tradicional*, *Test Smell Genérico* e *Não Relacionado com Código*. *Smells* classificados nessas categorias foram excluídos da pesquisa, seguindo os mesmos critérios descritos anteriormente na seção 3.1. Já os *smells* classificados na Categoria de Testes *Fim-a-Fim* foram acrescentados no catálogo.

Ainda nesta etapa optou-se por excluir um *smell* devido à falta de clareza em sua explicação e à ausência de exemplos para caracterizá-lo. Ao final desta etapa, doze *smells* foram selecionados, e estão descritos na Seção 3.4. Os *smells* excluídos podem ser visualizados no Apêndice A.2.

3.4 Catálogo de Smells RLC

Esta seção contém os *smells* retirados da RLC e que estão de acordo com o objetivo da pesquisa proposta, i.e., *smells* de testes fim-a-fim desenvolvidos com o Cypress. Os *smells* descritos nas seções 3.4.8 a 3.4.12 incluem exemplos devido a ter uma maior complexidade de entendimento. Esses exemplos são fornecidos para facilitar a compreensão e a detecção desses *smells*, tornando mais clara a identificação dos padrões e problemas associados. Assim como reportado na Seção 3.2, optou-se por traduzir esses *smells* para português para manter coerência com o idioma da dissertação. O texto em inglês pode ser visualizado no Apêndice B.

3.4.1 Unnecessary Waiting

Descrição: Um teste que aguarda por um período arbitrário usando `cy.wait()`, por exemplo, `cy.wait(2000)`. Porém, o uso de *wait* é necessário, quando é preciso sincronizar a aplicação com os testes, quando há atrasos devido ao carregamento de dados ou operações assíncronas, com isso evita falhas prematuras nos testes.

Razão: Este é considerado um *smell* porque pode tornar o teste mais lento ou mascarar uma lentidão no sistema.

Artigos:A1, A2, A3, A4, A5, A7, A9, A13, A18, A19, A20, A22, A27

3.4.2 Using Force: True When Interacting With Elements

Descrição: Esse *smell* ocorre quando o parâmetro `force:true` é usado para interagir com elementos da página, por exemplo, `cy.get('coveredOrDisabledButton').click({force:true})`.

Razão: Este é considerado um *smell* devido espera nativa da apresentação do componente não ser realizada, levando a uma eventual falta de sincronização entre teste e interface, o que faz com que elementos não estejam acessíveis no momento da interação.

Artigos: A6

3.4.3 Visiting External Sites

Descrição: Um teste que visita ou interage com servidores externos, ou seja, sites fora do domínio do sistema, por exemplo,

```
cy.visit('https://example.com',{chromeWebSecurity:false}).
```

Razão: Este é um *smell* porque utiliza dependências externas o que podem levar a falhas inesperadas e aumentar a fragilidade dos testes.

Artigos: A1, A4, A5, A13

3.4.4 Starting Web Servers

Descrição: Um teste que inicia um servidor web usando `cy.exec()` ou `cy.task()`, por exemplo,

```
cy.exec('http-server ./path-to-your-app -p 3000', {timeout:60000}).
```

Razão: É considerado um *smell*, pois iniciar um servidor Web dentro do ambiente de testes faz uso de variáveis externas e aumenta o tempo de execução dos testes.

Artigos: A1, A4, A5, A13, A26

3.4.5 Brittle Selectors

Descrição: Como o próprio nome sugere, esses seletores têm grandes chances de serem renomeados no futuro, por exemplo, usando uma sequência de classes específicas como `cy.get('.btn.btn-large-890')` em vez de identificadores mais estáveis como `cy.get('[id="submit"]')`.

Razão: Este é considerado um *smell* porque, o teste pode ser tornar frágil se houver um seletor mal nomeado na interface. Se o seletor for alterado, o teste precisará de manutenção cada vez que isso acontecer.

Artigos: A1, A2, A4, A5, A7, A13, A19, A20, A22, A28

3.4.6 Unnecessary Visibility Checks For Previously Actionable Elements

Descrição: Um teste que utiliza verificações redundantes de visibilidade ao lidar com elementos acionáveis, tais como `should("be.visible")` ou `should("exist")` antes de ações (como clicar ou digitar), por exemplo, `cy.get("seletor").should("be.visible").click()`.

Razão: É considerado um *smell* porque há presença de verificações redundantes para visibilidade ou existência de um determinado elemento.

Artigos: A24

3.4.7 Adopt Page Object Pattern

Descrição: Esse *smell* se refere ao uso de *Page Objects* ao escrever testes. A introdução desse tipo de estrutura de dados pode tornar os testes difíceis de manter.

Razão: Este é considerado *smell* no Cypress, devido à natureza assíncrona do Cypress. Pois, *Page Object* foca na estrutura de elementos da interface diferentemente do *App Actions*, uma alternativa mais viável devido a melhoria da manutenibilidade ao encapsular comportamentos da aplicação.

Artigos: A14

3.4.8 Using Cypress Calls In Assignments

Descrição: Um teste que tenta atribuir o valor retornado por chamadas de métodos assíncronos a uma variável. Como todos os métodos do Cypress são assíncronos, eles não retornam valores que podem ser diretamente e imediatamente atribuídos a variáveis. O trecho de código abaixo ilustra uma instância desse problema:

```
1 describe('Test', () => {
2     ...
3     cy.visit('your-test-page');
4     let x = cy.get("selector");
5     x.click();
6     cy.get('selector').invoke('text').should('include', textExpect);
7
8 })
```

Listagem 5: Using Cypress Calls In Assignments example

Na linha 4, há uma chamada para `cy.get()` e o resultado está sendo atribuído a uma variável `x`. Isso não é permitido no Cypress, pois `cy.get()` é uma chamada de método assíncrono.

Razão: É considerado um *smell* uma vez que a variável atribuída não retém o resultado da chamada da função por esta ser assíncrona.

Artigos: A1, A4, A5, A13

3.4.9 Missing Global baseUrl

Descrição: Um teste que chama `cy.visit()` sem definir um *baseUrl*. Este parâmetro deve ser definido em um arquivo de configuração (geralmente, `cypress.config.js`). O trecho de código abaixo ilustra uma instância desse problema:

```
1 describe('Test', () => {
2     ...
3     cy.visit('https://example.com');
4     cy.get('h1').should('contain', 'Your Page Title');
5
6 })
```

Listagem 6: Trecho de código contendo o smell Missing Global baseUrl

Na linha 4, há uma chamada para `cy.visit()` com um valor de uma URL fixa como parâmetro. Isso não é recomendado porque a melhor prática é definir essa URL externamente em um arquivo de configuração.

Razão: Este é um *smell* uma vez que mudanças na URL base causam alterações em todos os testes, aumentando o risco de erros e torna o teste mais difícil de manter.

Artigos: A1, A4, A5, A7, A13, A18, A19

3.4.10 Dirty Inputs

Descrição: Esse *smell* se refere a uma situação em que um teste interage com um campo de entrada sem limpar seu conteúdo existente, o que pode levar a comportamentos inesperados ou indesejados. Isso ocorre tipicamente quando um campo já possui um valor, e o teste continua a digitar um novo valor no campo sem primeiro limpá-lo. Assim, antes de preencher o campo, devemos usar `clear()` para apagar valores previamente existentes. O trecho de código abaixo ilustra uma instância desse problema:

```
1 describe('Test', () => {
2     ...
3     cy.visit('your-test-page');
4     let value = 'Larissa Bicalho';
5     cy.get('#username').type(value);
6     const newValue = 'newTestValue';
7     // need to user clear() before type()
8     cy.get('#username').type(newValue);
9     ...
10 }
```

Listagem 7: Trecho de código contendo o smell Dirty inputs

Na linha 9, um novo valor está sendo atribuído ao campo de nome de usuário sem utilizar a função `clear()`. Como resultado, o valor atribuído ao campo na linha 6 será concatenado com o valor atribuído na linha 9, enquanto o comportamento recomendado deveria envolver a substituição do valor existente.

Razão: É considerado um *smell* pois a ausência de comandos de limpeza podem resultar em problemas de validação e conflitos de dados, tornando o teste inconsistente e difícil de rastrear.

Artigos: A19

3.4.11 Cleaning up state with `after()` or `afterEach()`

Descrição: Um teste que usa `after()` ou `afterEach()` para limpar o estado em vez de `before()` ou `beforeEach()`. Não é recomendado limpar o estado do teste usando `afterEach()`. O uso destes métodos levam a perda de uma funcionalidade do Cypress voltada para rastrear o estado de execução de cada teste, tornando a depuração e a compreensão do resultado do teste mais difícil. O trecho de código abaixo ilustra uma instância desse problema:

```
1 describe('Test', () => {
2   ...
3   afterEach(() => {
4     cy.logout();
5   })
6   ...
7   cy.get('h1').should('contain', 'Your Page Title');
8
9 })
```

Listagem 8: Trecho de código contendo o smell Cleaning up state with `after()` or `afterEach()` exemplo

Usar `afterEach()` para realizar logout nas linhas 3 e 4 significa limpar o estado do teste, o que não é considerado uma boa prática. Especificamente, se for feito logout, o testador não consegue recuperar o estado caso o teste falhe.

Razão: Este é um *smell* pois torna a depuração e a compreensão do resultado do teste mais difícil.

Artigos: A1, A4, A5, A13, A19

3.4.12 Direct Access to List Elements

Descrição: Quando um teste precisa clicar em um elemento de lista, deve primeiro iterar sobre esses elementos (em vez de acessar diretamente o elemento de interesse), porque existem chances de obter um elemento desanexado do DOM (Document Object Model). Portanto, o seguinte código não é recomendado:

```
1 describe('Test', () => {
2   ...
3   cy.get('ul[role="listbox"]').each(( $btn, index ) => {
4     if(index === 0) {
5       cy.get( $btn ).click();
6     }
7     ...
8
9 })
```

Listagem 9: Trecho de código contendo o smell Direct Access to List Elements

Na linha 3, é feita uma chamada para `cy.get()` para interagir com a lista de elementos `` sem percorrer seus elementos filhos ``. Consequentemente, na linha 5, pode existir a possibilidade do elemento não estar no DOM.

Em vez disso, deve ser usado:

```
1 describe('Test', () => {
2     ...
3     cy.get('ul[role="listbox"]').find('li').eq(0).click();
4     ...
5
6 })
7
8
```

Listagem 10: Trecho de código contendo como evitar o smell Avoid Direct Access to List Elements

Na linha 3, é feita uma chamada para `cy.get()` para interagir com a lista de elementos `` percorrendo seus elementos filhos ``, com isso, evitando que o problema ocorra.

Razão: Este é um *smell*, pois este tipo de acesso pode levar a um elemento que não esteja mais no DOM posteriormente, levando a uma falha no teste.

Artigos: A16

3.5 Ameaças à Validade

Este estudo possui algumas limitações que podem ameaçar os resultados apresentados nesta seção. Essas ameaças podem ser divididas em quatro categorias: validade interna, validade externa, validade de construção e validade das conclusões. A seguir, será discutido em detalhes cada ameaça à validade e como elas podem impactar nas conclusões.

Validade Interna: As ameaças à validade interna referem-se à extensão na qual pode-se atribuir com precisão os resultados observados ao tratamento utilizado no experimento. A seleção dos resultados da metodologia, como artigos e *smells* selecionados pode ser enviesada devido os critérios de exclusão e inclusão escolhidos. Neste estudo foi aplicado critérios de inclusão e exclusão claramente definidos e realização de uma revisão cruzada dos documentos por mais de um revisor. Além da fundamentação da categorização baseadas discussões e revisões sobre cada *smell* catalogado, contando ainda com a experiência de um dos autores no desenvolvimento de testes fim-a-fim.

Validade Externa: Essa validade trata-se da generalização dos resultados para outros contextos. Os resultados podem não ser generalizáveis para todas as práticas de

testes fim-a-fim ou outras ferramentas além do Cypress. Para mitigar esse problema foi feita delimitação do escopo da pesquisa apenas para o testes fim-a-fim desenvolvidos com Cypress.

Validade de Construção: Essa validade trata-se a respeito à adequação das definições e categorias usadas para avaliar os *smells*. A estrutura da *string* de busca utilizada nas abordagens RSL e RLC apresenta um risco potencial relacionado à validade construtiva. Se a *string* de busca não for bem formulada, pode resultar em uma cobertura inadequada da literatura existente ou trazer documentos não pertinentes. Para mitigar essa ameaça, foram conduzidas buscas para refinar a consulta, incluindo a utilização de sinônimos e a aplicação de estudos anteriores para melhorar a precisão e a abrangência da *string* de busca. Adicionalmente, os documentos recuperados foram revisados cuidadosamente para garantir que os critérios de inclusão e avaliação fossem seguidos.

Validade das Conclusões: Essa validade refere-se à medida na qual pode-se determinar com precisão se o tratamento utilizado em um experimento é responsável pelo resultado observado. Neste estudo a ameaça de conclusão é a limitação nas definições de *smells* e a relevância das fontes utilizadas na RSL e RLC. A precisão das conclusões pode ser comprometida se as definições de *smells* não cobrirem todos aspectos relevantes desses problemas ou se as fontes de literatura selecionadas não forem representativas o suficiente para uma compreensão completa do tema. Ou seja, os *smells* e suas definições podem ser influenciados por perspectivas e práticas que não refletem a área de testes fim-a-fim. Para lidar com essa limitação, as definições de *smells* foram baseadas na revisão da literatura, abrangendo tanto fontes acadêmicas quanto cinzas. Além disso, a autora utilizou da experiência na área para validar as definições adotadas.

3.6 Considerações Finais:

O estudo sistemático descrito na Seção 3.1 ressalta ausência de estudos dentro do escopo proposto neste trabalho. Foram encontrados três artigos relacionados a teste fim-a-fim e apenas cinco dos *smells* encontrados podem ser aplicados ao Cypress. Para contornar essa limitação, adotou-se uma revisão da literatura cinza, onde foram selecionados 28 documentos relevantes. Ao final desta revisão, foi possível criar um catálogo contendo doze *smells* para testes fim-a-fim, voltados especificamente para a Plataforma Cypress. A detecção desses *smells* serão avaliadas no Capítulo 4.

Capítulo 4

Detecção de Smells

4.1 Introdução

Os capítulos anteriores desta dissertação se dedicaram à catalogação dos *smells* em testes fim-a-fim desenvolvidos com Cypress. Para solucionar a segunda questão levantada sobre o uso de ferramentas de IA para detecção de *smells* foi utilizado o ChatGPT. Essa escolha se justifica não apenas pelo crescente reconhecimento e popularidade da ferramenta [45]. Além disso, sua aplicação na detecção de *smells* ainda não foi amplamente explorada, com apenas alguns artigos investigando seu uso no desenvolvimento de testes [8] [21]. Este capítulo descreve o presente estudo, dividido em duas partes:

- **Estudo Controlado:** Consiste em um experimento com a ferramenta ChatGPT com o objetivo de avaliar sua habilidade em detectar os doze *smells* catalogados. Utilizando o sistema Mantis como objeto de estudo, a escolha do Mantis foi devido ao ser uma ferramenta de código aberto, bem utilizada na área por QA's e de ser de domínio da autora, facilitando o desenvolvimento dos testes. Foram implementados casos de testes injetados com os *smells* identificados para, em seguida, serem submetidos a análise do ChatGPT. Na Seção 4.2, é apresentada a metodologia adotada para a condução do estudo, bem como os resultados obtidos e sua análise subsequente.
- **Estudo com Aplicações GitHub:** Neste segundo estudo, foi utilizado o GitHub para buscar testes em sistemas de código aberto relevantes construídos com o Cypress. O objetivo deste estudo, semelhante ao anterior, é avaliar a capacidade do ChatGPT em detectar os doze *smells* identificados, porém em um ambiente não controlado. Ou seja, busca-se avaliar a eficácia de sua detecção em testes que são rotineiramente utilizados. A Seção 4.3 segue uma estrutura semelhante à do primeiro estudo, detalhando a metodologia empregada, os resultados obtidos e as conclusões da análise realizada.

4.2 Estudo Controlado - Mantis App

Nesta parte do estudo foi utilizado o sistema Mantis, o qual é um sistema de rastreamento de falhas de código aberto baseado na web. Ele é projetado para ajudar equipes a gerenciar efetivamente projetos de desenvolvimento de software, permitindo que eles acompanhem e priorizem falhas e problemas ao longo do ciclo de vida do desenvolvimento. Foi escolhido para este estudo devido à sua natureza de código aberto e à familiaridade da autora com a ferramenta.

Com objetivo de avaliar a eficácia do ChatGPT em detectar *smells* em testes desenvolvidos com Cypress, a autora desenvolveu testes fim-a-fim para testar as funcionalidades principais do sistema, tais como página de login. A princípio esses testes foram desenvolvidos sem nenhum *smell*.

4.2.1 Metodologia

A metodologia empregada para a detecção dos *smells* no ChatGPT foi dividida em três etapas: (1) avaliação e injeção dos *smells* encontrados no catálogo; (2) construção do *prompt* para o ChatGPT; Por fim, a etapa (3) consistiu na execução do *prompt* no ChatGPT. Abaixo pode ser visto com detalhes sobre cada uma.



Figura 4.1: Metodologia do Estudo do Mantis

Avaliação e Injeção dos Smells: Esta etapa é responsável pela avaliação e injeção dos *smells* encontrados. Alguns *smells* precisaram ser excluídos nessa fase, notadamente o *Adopt Page Object Pattern* e o *Unnecessary Navigation*. A exclusão dos *smells* ocorreu devido à dificuldade da detecção dos mesmos pelo ChatGPT. Especificamente, o primeiro

smell está relacionado à organização do código, uma vez que o *smell* envolve múltiplos arquivos. Já o segundo foi retirado dessa fase, pois o ChatGPT não é alimentado com as informações contextuais necessárias para entender o fluxo de navegação ideal para a funcionalidade sob teste.

A fim de facilitar a injeção dos *smells*, optou-se por reutilizar o mesmo teste para detecção de vários deles. Contudo, é importante ressaltar que cada *smell* foi testado isoladamente.

A Tabela 4.1 mostra em qual teste foi incorporado manualmente cada *smell*.

Teste	Smells Injetados Manualmente
loginTest.cy.js	Unnecessary Waiting/Starting Web Servers/ Brittle Selectors/Dirty Inputs/Using Cypress Calls In Assignments/Direct Access to List Elements/ Cleaning up state with after() or afterEach()/ Unnecessary Visibility Checks For Previously Actionable Elements/Missing Global baseUrl/ Use relative URLs locator instead of absolute ones/
visitExternalSite.cy.js	Visiting External Sites

Tabela 4.1: Smells injetados em cada teste

Construção do Prompt para o ChatGPT: Esta etapa consiste na criação de um *prompt* que foi executado no ChatGPT (versão 3.5) para cada *smell* descrito no Apêndice B. Esse *prompt* foi iniciado com uma contextualização sobre cada um dos doze *smells* e encerrou-se com a referência aos arquivos Cypress que continham as instâncias dos *smells* catalogados. O padrão temperatura, que serve como um parâmetro para regular o nível de aleatoriedade e variação encontrado nas respostas produzidas, não foi alterado neste estudo.

ChatGPT Prompt

First, I will give you a list of 12 code smells that can occur in end-to-end tests implemented using the Cypress framework. The list includes a short description of each smell; in some cases, there is also a simple code example.

Smell 1: Unnecessary Waiting

A test that waits for an arbitrary period using `cy.wait()`, e.g., `cy.wait(2000)`.

* [other smell descriptions]

Suppose you are a cypress developer. Could you please identify the smell that occurs in the following test? The file is delimited using markdown code blocks. Please, in your answer mention at most one smell from the list described above.

[source code of the tests with the smells]

Tabela 4.2: Prompt enviado ao ChatGPT

Execução do Prompt no ChatGPT: Para cada *smell*, o *prompt* é executado até que o *smell* seja detectado corretamente ou, no máximo, 5 vezes. Se após essas 5 tentativas o *smell* não for identificado, considera-se que o ChatGPT não conseguiu detectar o *smell* [39].

4.2.2 Resultados

A Tabela 4.3 apresenta os *smells* submetidos ao ChatGPT, acompanhados dos resultados correspondentes. Na coluna Cypress Smells, estão listados os *smells* enviados como *prompt* ao ChatGPT. A coluna Round é dividida em cinco partes, representando a quantidade de rodadas em que o *prompt* foi executado para cada *smell* listado na coluna Cypress Smells. O símbolo (·) indica que o *smell* não foi encontrado até o momento naquela rodada. Por fim, o marcador (●) indica a rodada em que o *smell* foi identificado.

Por exemplo, o *smell Unnecessary Waiting* foi identificado na primeira rodada, enquanto *Missing Global BaseUrl* foi identificado na quarta rodada.

Cypress Smells	Round #				
	1st	2nd	3rd	4th	5th
Unnecessary Waiting	●				
Using force: true when interacting with elements	·	●			
Visiting External Sites	●				
Starting Web Servers	●				
Brittle Selectors	●				
Unnecessary Visibility Checks For Previously Actionable Elements	●				
Using Cypress Calls In Assignments	●				
Missing Global BaseUrl	·	·	·	●	
Dirty Inputs	·	●			
Cleaning up state with after() or afterEach()	●				
Direct Access to List Elements	●				
Use relative URLs locator instead of absolute ones	·	·	●		

Tabela 4.3: Lista de *smells* solicitados ao ChatGPT. O símbolo de marcador (●) indica qual rodada o *smell* foi identificado pelo ChatGPT.

No geral, o ChatGPT foi capaz de detectar os doze *smells* identificados na revisão da literatura cinza. Destes, oito foram encontrados após a primeira solicitação (67%). Dos quatro *smells* restantes, dois foram identificados na segunda, um na terceira, e um na quarta. Este resultado reforça a importância de executar os *prompts* diversas vezes conforme pode ser confirmado na Tabela 4.3.

4.2.3 Discussão

Nesta seção, serão discutidos alguns aspectos observados durante a análise dos resultados. O primeiro trata do *output* do ChatGPT ao identificar os *smells* e o outro aspecto é sobre a dificuldade na identificação dos *smells* *Missing GlobalBaseUrl*

4.2.3.1 Descrição dos Smells

Em alguns casos o ChatGPT não apenas identificou os *smells*, mas forneceu um texto descritivo justificando a necessidade de corrigi-lo. Por exemplo, a Listagem 11 mostra o código-fonte que foi utilizado para detecção do *smell Unnecessary Waiting*. Por uma questão de simplicidade, foi necessário mostrar apenas o método que contém o *smell*, considerando que o arquivo de código-fonte inclui outros arquivos.

```
1 //File : loginTest.cy.js
2 describe('TEST', () => {
3     const expectedText = {
4         textMenu: 'administrator'
5     };
6
7     it('Login Test', { tags: '@smoke' }, () => {
8         const username = Cypress.config('username');
9         const password = Cypress.config('password');
10        cy.visit('/login_page.php');
11        cy.get('[id="username"]').clear().type(username);
12        cy.wait(100000);
13        cy.get('input[type="submit"]').click();
14        cy.get('[id="password"]').clear().type(password);
15        cy.get('input[type="submit"]').click();
16        cy.get('.user-info').invoke('text').should('include', expectedText.textMenu);
17    });
18
19 })
```

Listagem 11: Exemplo de teste contendo o smell Unnecessary Waiting

Para este *smell*, o ChatGPT apontou corretamente a linha com o *smell* e justificou detalhadamente as razões para ser considerado como tal. A seguinte resposta foi fornecida ao *prompt* enviado:

The test exhibits the “Unnecessary Waiting” smell. This is indicated by the line ‘cy.wait(100000);’, where an arbitrary wait time of 100000 milliseconds (100 seconds) is introduced. Such waits are generally unnecessary and can lead to

longer test execution times without providing any significant benefit. Waiting for fixed durations can also make tests flaky, as it may not guarantee that the element or action being waited for will occur within that time frame.

Outro exemplo em que o ChatGPT identificou corretamente um *smell* e explicou a necessidade de sua correção foi na detecção do *smell Starting Web Servers*. A análise fornecida pelo ChatGPT destacou os pontos críticos e as implicações desse *smell*. Segue abaixo o trecho de código 12 que foi submetido à análise.

```
1 //File : loginTest.cy.js
2 describe('TEST', () => {
3   const expectedText = {
4     textMenu: 'administrator'
5   };
6   it('Login Test', { tags: '@smoke' }, () => {
7     const username = Cypress.config('username');
8     const password = Cypress.config('password');
9     before(() => {
10      cy.exec(' docker compose up -d ');
11    });
12    cy.visit('/login_page.php');
13    cy.get('[id="username"]').clear().type(username);
14    cy.get('input[type="submit"]').click();
15    cy.get('[id="password"]').clear().type(password);
16    cy.get('input[type="submit"]').click();
17    cy.get('.user-info').invoke('text').should('include', expectedText.textMenu);
18  });
19 })
20
```

Listagem 12: Exemplo de teste contendo o *smell Starting Web Servers*

Neste código na linha 10, pode ser encontrado o *smell Starting Web Servers*. O seguinte *output* foi retornado após a execução:

The smell occurring in the provided test is “Starting web servers”(Smell 4). This smell is identified by the usage of `cy.exec()` within a test to start a web server. In this case, the command `cy.exec('docker compose up -d')` is starting a Docker container with a web server. Starting web servers within tests can introduce dependencies and potential instability to the tests, as they rely on external resources that may not always be available or consistent. It's generally recommended to avoid starting web servers within Cypress tests and instead ensure that the application is in the desired state before running the tests.

Este texto destaca que a utilização de recursos externos pode gerar instabilidade nos testes, uma observação que reflete a realidade. Além disso, ressalta a importância de garantir que a aplicação esteja no estado desejado antes da execução dos testes, uma prática recomendada. Adicionalmente, é importante notar que essas informações são inferidas pelo ChatGPT e não são explicitamente fornecidas no *prompt*. Isso evidencia o potencial do ChatGPT em identificar melhorias no código ao sugerir práticas recomendadas com base em conhecimento prévio e padrões estabelecidos. O *prompt* submetido e o resultado a execução pode ser visto com mais detalhes¹.

4.2.3.2 Dificuldade na Identificação de Missing Global URL e Use Relative URLs

Em outros casos, o ChatGPT não obteve a mesma precisão para identificar o *smell*, por exemplo, *Missing Global BaseUrl*, identificado na quarta rodada. A Listagem 13 contém o trecho que foi utilizado para detecção deste *smell*.

```
1  const { defineConfig } = require('cypress')
2  module.exports = defineConfig({
3    username: 'administrator',
4    password: 'root',
5    ...
6  })
7  describe('TEST', () => {
8    const expectedText = {
9      textMenu: 'administrator'
10   };
11   it('Login Test', { tags: '@smoke' }, () => {
12     const username = Cypress.config('username');
13     const password = Cypress.config('password');
14     cy.visit('127.0.0.1:8989/login_page.php');
15     cy.get('[id="username"]').clear().type(username);
16     cy.get('input[type="submit"]').click();
17     cy.get('[id="password"]').clear().type(password);
18     cy.get('input[type="submit"]').click();
19     cy.get('.user-info').invoke('text').should('include', expectedText.textMenu);
20   });
21 }
```

Listagem 13: Exemplo de teste contendo o *smell Missing Global BaseUrl*

¹<https://chat.openai.com/share/c3b6337e-ad6a-4474-b986-48960b9ecd4c>

Neste exemplo, o *smell Missing Global BaseUrl* foi introduzido na linha 29. Na resposta do ChatGPT, a identificação correta da linha com o *smell* ocorreu apenas na quarta tentativa. Nas três rodadas anteriores, o modelo identificou o *smell Use Relative URLs Locator Instead Of Absolute Ones*. Tais *smells* são semelhantes, pois ambos se referem à introdução incorreta da URL do sistema no código, especificamente no comando `cy.visit()`. No primeiro caso, a URL é codificada diretamente no *script* de teste. No segundo caso, a URL completa é utilizada em vez de apenas o caminho necessário para o *script* de teste. Abaixo na Tabela 4.4 estão as respostas ao *prompt* enviado:

Round	ChatGPT
1	Use Relative URLs Locator Instead Of Absolute Ones
2	Use Relative URLs Locator Instead Of Absolute Ones
3	Use Relative URLs Locator Instead Of Absolute Ones
4	Missing Global BaseUrl

Tabela 4.4: Respostas do ChatGPT na tentativa de identificar o *smell Missing Global BaseUrl*

Um padrão semelhante ocorreu ao detectar o *smell Use Relative URLs Locator Instead Of Absolute Ones*, identificado na terceira tentativa. Nas duas tentativas anteriores, o ChatGPT inferiu erroneamente o *smell Missing Global BaseUrl*. Segue na Tabela 4.5 as respostas ao *prompt* enviado:

Round	ChatGPT
1	Missing Global BaseUrl
2	Missing Global BaseUrl
3	Use Relative URLs Locator Instead Of Absolute Ones

Tabela 4.5: Respostas do ChatGPT na tentativa de identificar o *smell Use Relative URLs Locator Instead Of Absolute Ones*

Em resumo, o resultado obtido para este estudo controlado são promissores uma vez que, 67% dos *smells* foram detectados pelo ChatGPT na primeira execução da ferramenta e todos foram detectados após cinco execuções.

4.3 Estudo com Aplicações GitHub

Após avaliar o catálogo e a eficácia do ChatGPT como uma ferramenta de detecção de *smells* em um cenário controlado, optou-se por realizar um estudo de campo com sistemas de código aberto para ampliar o estudo anterior. Dado que o GitHub é amplamente utilizado pela comunidade de profissionais de desenvolvimento de software [6][10][11], a plataforma foi escolhida como a ferramenta de busca para este estudo. Mais detalhes sobre a metodologia adotada bem como seus resultados são apresentados nas seções a seguir.

4.3.1 Metodologia

A metodologia empregada para a detecção de *smells* no ChatGPT foi dividida em cinco etapas como pode ser visto na Figura 4.2: (1) seleção de projetos no GitHub; (2) criação do oráculo de teste; (3) construção do *prompt* para o ChatGPT; (4) execução do *prompt*; e (5) avaliação dos resultados. Abaixo, cada etapa é detalhada:



Figura 4.2: Metodologia do Estudo das Aplicações GitHub

Seleção de Projetos no GitHub: Para esta etapa, o critério de busca foi selecionar sistemas que tenham uma dependência do Cypress. Para isso, utilizou-se a ferramenta *ghstopdep*², responsável por buscar repositórios populares que dependam de alguma biblioteca a ser fornecida.

Os parâmetros da ferramenta foram configurados para que ela pudesse retornar os projetos mais populares que declararam Cypress em suas dependências. Após esta etapa, foram descartados sistemas conforme os seguintes critérios:

²<https://pypi.org/project/ghstopdep/0.1.0/>

- Repositórios com menos de 1.000 estrelas, pois a quantidade de estrelas em um repositório pode servir como um indicador preliminar da qualidade e confiabilidade de seus testes [5].
- Quantidade de testes fim-a-fim disponíveis: A escassez de testes pode comprometer a abrangência das análises realizadas, limitando a validade dos resultados obtidos.

Após a aplicação dos critérios mencionados anteriormente apenas três sistemas se adequaram a eles, com isso foram selecionados, e podem ser vistos na Tabela 4.6.

Projeto	URL	Sistema	Estrelas	Número de Testes Fim-a-Fim
Pigallery	https://github.com/bpatrik/pigallery2	Site de galeria de fotos, baseado em diretórios otimizados para rodar em servidores de baixos recursos.	1.4K	6
Livewire Powergrid	https://github.com/Power-Components/livewire-powergrid	Cria datatables modernos, poderosos e fáceis de personalizar com base na biblioteca Laravel Livewire.	1.2K	8
GlobaLeaks	https://github.com/globaleaks/GlobaLeaks	Software gratuito e de código aberto que permite a qualquer pessoa configurar e manter facilmente uma plataforma segura para denúncias.	1.1K	20

Tabela 4.6: Tabela de Sistemas Selecionados do GitHub

Criação do Oráculo de Teste: Após concluir a fase anterior, a autora—que possui cinco anos de experiência na implementação de automatizados—avaliou manualmente o código fonte de cada teste fim-a-fim presente nos três sistemas, procurando por ocorrência dos *smells* reportados no catálogo levantado nas Seções 3.2 e 3.4. Os *smells* identificados manualmente neste estudo podem ser vistos na Tabela 4.7.

Smells	Pigallery	Livewire	GlobaLeaks
Unnecessary Waiting	2	1	1
Using Force:True When Interacting With Elements	5	0	2
Brittle Selectors	6	8	19
Unnecessary Visibility Checks For Previously Actionable Elements	0	0	1
Direct Access to List Elements	0	0	2

Tabela 4.7: Quantidade de Smells por Sistema

Construção do Prompt para o ChatGPT: O *prompt* a ser utilizado com o ChatGPT é fundamentalmente o mesmo em relação ao que foi usado na Seção 4.2, também não foi alterado o padrão temperatura, a diferença desse *prompt* deve-se a modificação da última pergunta. Como neste experimento um mesmo arquivo pode ter mais de um *smell* distinto, requisitou-se ao ChatGPT que identifique todos os *smells* presentes em cada teste, em contraste com o estudo realizado anteriormente.

```
ChatGPT Prompt

First, I will give you a list of 12 code smells that can occur in end-to-end tests implemented using the Cypress framework. The list includes a short description of each smell; in some cases, there is also a simple code example.

# Smell 1: Unnecessary Waiting
A test that waits for an arbitrary period using cy.wait(), e.g., cy.wait(2000).
* [other smell descriptions]

Please give your answer in the following format:

This code has the following smells:
1. Smell name
2. Smell name
3. etc

IMPORTANT:
1. Only respond when you are absolutely certain that the smell occurs in the code. The goal is to avoid false positives. Otherwise, just respond: there are no smells!
2. Moreover, you do not need to explain your answer.

[source code of the tests with the smells]
```

Tabela 4.8: Prompt enviado ao ChatGPT durante o Segundo Experimento

Execução do Prompt: Em todos os conjuntos de testes dos três sistemas selecionados do GitHub, o *prompt* descrito anteriormente foi executado uma vez para cada arquivo. O objetivo consistiu em obter do ChatGPT uma resposta para cada teste, identificando os possíveis *smells* que o modelo avaliava estar presentes. Nesta etapa, dois testes do GlobalLeaks foram desconsiderados, pois mesmo estando nas pasta de testes fim-a-fim eles não podem ser considerados como tais, pois não verificam o fluxo comportamental de alguma funcionalidade, sendo eles: *test-init.js* (trata-se de um teste que apenas acessa uma página web) e *test-authentication-redirects.js* (trata-se apenas de um teste de redirecionamento de links).

Avaliação dos Resultados: Os resultados foram avaliados conforme as métricas de precisão e *recall*. Este método compara o Oráculo de Teste criado na Etapa 2 (que é a referência que indica os *smells* presentes nos arquivos de teste analisados) com os resultados da Execução do *Prompt* (que são os *smells* detectados nos arquivos pelo ChatGPT).

$$Precisão = \frac{VerdadeiroPositivo}{VerdadeiroPositivo + FalsoPositivo} \quad (4.1)$$

$$Recall = \frac{VerdadeiroPositivo}{VerdadeiroPositivo + FalsoNegativo} \quad (4.2)$$

Para um arquivo específico, foi considerado como VP (Verdadeiro Positivo) todo *smell* que foi identificado tanto pelo ChatGPT quanto pelo Oráculo. Já as ocorrências do tipo FP (Falso Positivo) ocorrem quando um *smell* é reportado pelo ChatGPT, mas não foi identificado no Oráculo. Por fim, ocorrências do tipo FN (Falso Negativo) ocorrem quando o *smell* foi identificado pelo Oráculo mas não foi reportado pelo ChatGPT.

4.3.2 Resultados

Esta seção oferece uma análise detalhada dos resultados de precisão e *recall* obtidos durante a avaliação dos sistemas investigados neste estudo.

Primeiro em linhas gerais, uma avaliação de precisão total e *recall* total em todos os sistemas pode ser vista na Tabela 4.9.

Sistema	Precisão Total	Recall Total
Pigallery	31%	62%
Livewire	24%	44%
GlobaLeaks	15%	31%

Tabela 4.9: Avaliação Geral da Precisão e Recall dos Sistemas

O Pigallery foi o sistema que apresentou a melhor precisão na detecção dos *smells*, sendo ela 31%. Por outro lado, o GlobaLeaks foi o sistema que apresentou a menor precisão dentre os avaliados, sendo ela 15%. Já sobre *recall*, Pigallery apresentou um *recall* de 62%, enquanto o GlobaLeaks teve a performance, mais baixa com um *recall* de 31%.

Logo, os resultados gerais obtidos mostram que o desempenho do modelo foi insatisfatório. Para uma compreensão mais aprofundada das possíveis causas dessa avaliação, uma análise individual de cada sistema é realizada nas próximas seções.

4.3.2.1 Resultado Pigallery

A Figura 4.3 mostra os resultados de precisão por *smell* no Pigallery. Ao analisar os *smells* detectados no Pigallery, observa-se que os *smells* *Brittle Selectors* e *Using Force: Interacting With Elements* atingiram uma precisão de 100%. Em outras palavras, a detecção desses *smells* pelo ChatGPT refletiu sua presença nos testes. Em contrapartida, o *smell* *Unnecessary Waiting* registrou uma precisão de apenas 33%.

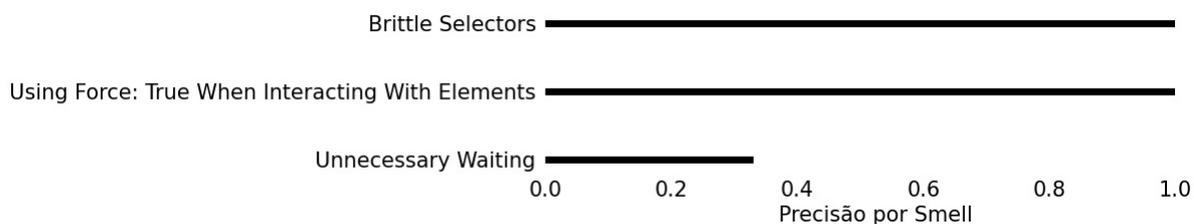


Figura 4.3: Taxas de Precisão de cada smell para o sistema Pigallery

A Figura 4.4 mostra os resultados de *recall* por *smell* no Pigallery. Ao examinar os *smells* *Unnecessary Waiting* e ao *Using Force: Interacting With Elements*, foi constatado um *recall* de 100%. Isso sugere que esses *smells* foram identificados com uma frequência considerável. Por outro lado, com o *smell* *Brittle Selectors* obteve-se um *recall* de 17%.

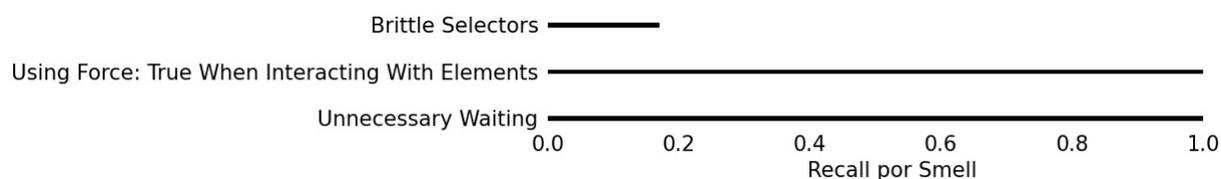


Figura 4.4: Taxas de Recall de cada smell para o sistema Pigallery

4.3.2.2 Resultado Livewire

A Figura 4.5 mostra os resultados de precisão por *smell* no Livewire. Ao analisar os *smells* identificados no Livewire, o *smell* *Brittle Selectors* atingiu uma precisão de 100%. No entanto, o *smell* *Unnecessary Waiting* mostrou uma precisão consideravelmente mais baixa, marcando apenas 20%.

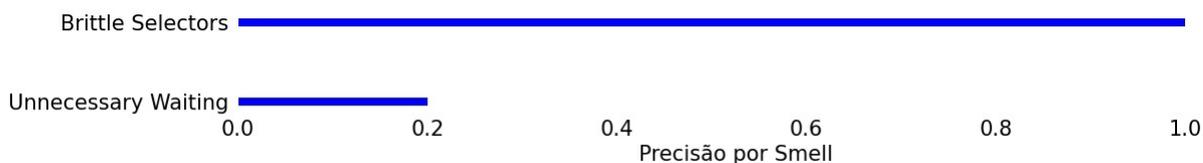


Figura 4.5: Taxas de Precisão de cada smell para o sistema Livewire

A Figura 4.6 mostra os resultados de *recall* por *smell* no Livewire. Observa-se um *recall* de 100% para o *smell Unnecessary Waiting*. Em contraste, o *smell Brittle Selectors* alcançou um *recall* de apenas 38%.

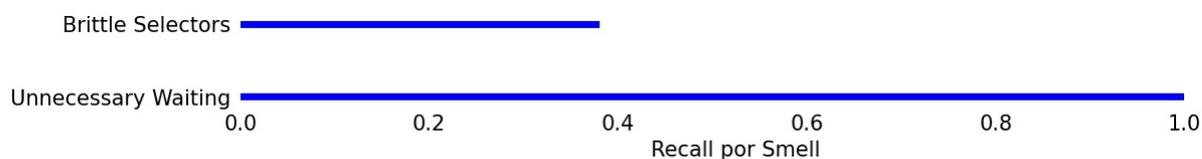


Figura 4.6: Taxas de Recall de cada smell para o sistema Livewire

4.3.2.3 Resultado GlobaLeaks

A Figura 4.7 mostra os resultados de precisão por *smell* no GlobaLeaks. Considerando os *smells* encontrados no GlobaLeaks, observa-se que o *smell Brittle Selectors* alcançou uma precisão de 100%. Em contrapartida, o *smell Unnecessary Visibility Checks For Previously Actionable Elements* apresentou a menor precisão, sendo ela de 11%.

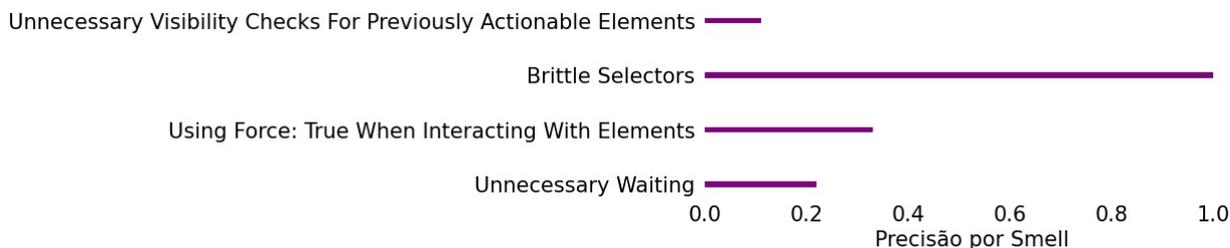


Figura 4.7: Taxas de Precisão de cada smell para o sistema GlobaLeaks

Por fim, a Figura 4.8 apresenta os resultados de *recall* por *smell* no GlobaLeaks. Primeiramente, observa-se um *recall* de 100% para o *smell Unnecessary Waiting*. Em contraste, o *recall* para o *smell Brittle Selectors* é de apenas 22%.

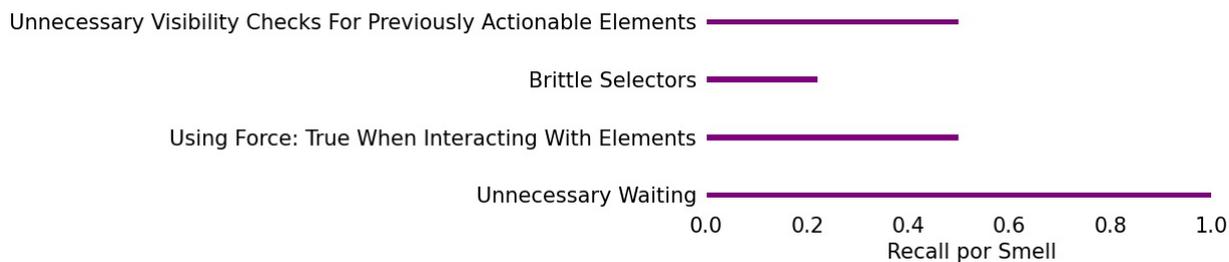


Figura 4.8: Taxas de Recall de cada smell para o sistema GlobaLeaks

4.3.2.4 Discussão

Nesta subseção, são apresentadas observações gerais sobre os resultados de precisão e *recall* nos sistemas. É evidente que o ChatGPT não alcançou uma precisão satisfatória na avaliação de *smells* em sistemas de código aberto. Nesse contexto, os *smells* não foram artificialmente inseridos para este estudo; em vez disso, foi realizada uma análise para determinar se os testes continham esses *smells*. Análise revela que os valores de precisão foram baixos, como mostrado na Tabela 4.9, e isso pode ser atribuído a diversos fatores. Essa limitação pode ser atribuída à natureza de funcionamento do ChatGPT, que não está adequadamente treinado para lidar com esse tipo específico de tarefa. No entanto, é esperado que com iterações e execuções futuras, o ChatGPT seja capaz de aprimorar sua capacidade de detectar de forma mais precisa os *smells* apresentados.

Primeiramente, o *Unnecessary Waiting* foi detectado pelo ChatGPT em todos os testes do PigGallery2, bem como em cinco dos oito testes do Livewire e em oito dos vinte testes do GlobaLeaks.

Além disso, o *smell Dirty Inputs* foi identificado pelo ChatGPT em quatro dos seis testes do Pigallery, em cinco dos oito testes do Livewire e em onze dos vinte testes do GlobaLeaks. Os resultados obtidos pelo oráculo indicam que esse elemento não estava presente em nenhum dos casos analisados.

O *Unnecessary Visibility Checks For Previously Actionable Elements* foi detectado pelo ChatGPT em sete testes, mas ocorreu apenas em dois testes no GlobaLeaks. Além disso, no mesmo sistema, o *Direct Access to List Element* foi detectado pelo ChatGPT mas, de acordo com o oráculo, esse elemento não está presente em nenhum dos testes.

Por fim, *Brittle Selectors* foi o *smell* que mais contribuiu para os resultados negativos, como evidenciado na Tabela 4.10 em cada sistema:

Sistema	Oráculo de Teste	ChatGPT
Pigallery	6	1
Livewire	8	3
GlobaLeaks	19	4

Tabela 4.10: Quantidade do smell Brittle Selector no Oráculo vs Quantidade desse smell detectada pelo ChatGPT

Por fim, observa-se um caso de alucinação acontecido no exemplo de teste *docs-screens.cy.ts* do sistema Pigallery que ilustra a inconsistência na resposta do ChatGPT. O código abaixo foi enviado junto com o *prompt* 4.8.

```

1 describe('Docs generation', () => {
2   beforeEach(() => {
3     cy.viewport(1600, 900);
4     cy.visit('/');
5     cy.get('.card-body'); // Brittle Selectors
6     cy.get('.col-sm-12').contains('Login'); // Brittle Selectors
7     /* ==== Generated with Cypress Studio ==== */
8     ....
9     it.skip('blog', () => {
10      cy.wait('@getContent');
11      cy.wait(1000); // Unnecessary Waiting
12      cy.get('app-gallery-blog .btn-blog-details').click({multiple: true,
13        ↪ scrollBehavior: false, force: true}); // Using force: true when interacting
14        ↪ with elements // Brittle Selectors
15      cy.window().screenshot('blog', {capture: 'viewport'});
16    });
17  });
18 }

```

O ChatGPT detectou os seguintes *smells*:

ChatGPT Resultado

This code has the following smells:

1. Unnecessary Waiting
2. Using force: true when interacting with elements
3. Starting web servers
4. Brittle Selectors
5. Unnecessary Visibility Checks For Previously Actionable Elements
6. Using Cypress Calls In Assignments
7. Missing global baseUrl
8. Dirty inputs
9. Cleaning up state with after() or afterEach()
10. Direct Access to List Elements

Tabela 4.11: Resultado do ChatGPT

Como observado, na resposta do ChatGPT, o modelo identificou que praticamente todos os *smells* listados nesta Tabela 4.11 estavam presentes no teste. No entanto, conforme determinado pelo oráculo de teste, apenas os *smells Unnecessary Waiting; Using force: true when interacting with elements e Brittle Selectors* realmente ocorriam neste teste.

4.4 Ameaças à Validade

Este estudo possui algumas limitações que podem ameaçar os resultados apresentados nesta seção. Essas ameaças podem ser divididas em quatro categorias: validade interna, validade externa, validade de construção e validade das conclusões. A seguir, será discutido em detalhes cada ameaça à validade e como elas podem impactar nas conclusões.

Validade Interna: As ameaças à validade interna referem-se à extensão na qual pode-se atribuir com precisão os resultados observados ao tratamento utilizado no experimento. Este estudo aborda a natureza não determinística do ChatGPT, portanto os resultados dos *prompts* podem variar entre as execuções, afetando a consistência dos resultados obtidos no estudo. Para mitigar essa ameaça, foram realizados no máximo cinco execuções dos *prompts* para obter uma média dos resultados e verificar a consistência.

Validade Externa: Essa validade trata-se da generalização dos resultados para outros contextos. A escolha do Mantis pode dificultar a replicação dos estudos a outras ferramentas e linguagens. Porém, foi realizada uma discussão sobre como os resultados poderiam ser aplicados a outras ferramentas e linguagens. Além disso, no Estudo com Aplicações GitHub, a seleção de projetos específicos pode não refletir o comportamento do ambiente real do dia-a-dia. Para abordar a limitação da escolha de projetos específicos no GitHub, foi realizada uma busca para selecionar projetos relevantes que utilizassem o Cypress.

Validade de Construção: Essa validade trata-se a respeito da adequação das definições e categorias usadas para avaliar os *smells*. A possibilidade de que as definições utilizadas não capturem todas os conceitos dos *smells* catalogados, pode afetar interpretação do ChatGPT, com isso dificultando detecção dos *smells*. Para mitigar essa validade foi estudado a literatura existente para garantir que as definições estão corretas e avaliado por especialista da área.

Validade das Conclusões: Essa validade referem-se à medida na qual pode-se determinar com precisão se o tratamento utilizado em um experimento é responsável pelo resultado observado. Neste estudo, a precisão das conclusões pode ser comprometida por uma interpretação inadequada dos resultados ou pelo uso de métricas que não refletem

completamente a capacidade do ChatGPT. Para mitigar o risco foi utilizado métricas de precisão e *recall*, frequentemente utilizadas para avaliar de problemas de classificação.

4.5 Considerações Finais

Foram realizados estudos utilizando o ChatGPT para a detecção de *test smells* em duas abordagens distintas: um estudo controlado e um estudo com aplicações do GitHub. O Estudo Controlado demonstrou que o ChatGPT possui capacidade para identificar *test smells* em um ambiente controlado, apresentando resultados relativamente precisos na detecção dos *smells* catalogados. Por outro lado, o Estudo com Aplicações GitHub evidenciou que a detecção de *smells* pelo ChatGPT em um ambiente não controlado é mais desafiadora devido à complexidade e à variabilidade dos testes existentes em projetos reais.

Esses resultados indicam que, na sua versão atual, o ChatGPT não pode ser utilizado como uma ferramenta para a detecção de *smells* em ambientes não controlados. O estudo controlado revelou a viabilidade do ChatGPT em contextos bem definidos, enquanto o estudo com aplicações GitHub destacou a necessidade de aprimoramentos.

Capítulo 5

Conclusão

Nesta dissertação, foi conduzido um estudo sobre os *smells* encontrados no desenvolvimento de testes fim-a-fim utilizando o *framework* Cypress.

Com o surgimento do manifesto ágil, a prática de testes de software passou por uma reestruturação significativa nas equipes de desenvolvimento de software, com uma ênfase maior na automação dos testes. Essas transformações elevaram os testes a uma das práticas de programação mais valorizadas no desenvolvimento moderno de software. Como afirmou Michael Feathers, “se um código não é acompanhado por testes, ele pode ser considerado de baixa qualidade ou até mesmo um código legado” [46]. Enquanto é crucial adotar boas práticas para garantir a implementação de testes confiáveis, a literatura acadêmica tem negligenciado, em certa medida, o levantamento de melhores práticas no contexto de testes automatizados de software, especificamente testes fim-a-fim.

Por exemplo, muitas pesquisas atuais estão focadas em soluções para mitigar problemas específicos, como fragilidade de teste [28] ou reparo automatizado [43]. Embora investiguem diferentes aspectos dos testes automatizados, essas pesquisas tendem a se concentrar em testes de menor granularidade, como os testes unitários e de regressão. Em outras palavras, há ainda muitas questões a serem exploradas na literatura acadêmica para identificar as melhores práticas ao desenvolver testes fim-a-fim [23].

Diante desse cenário, esta dissertação realizou um estudo sobre os *smells* encontrados no desenvolvimento de testes fim-a-fim utilizando o *framework* Cypress. A escolha deste *framework* se deu por sua facilidade de uso e ampla adoção na indústria. Durante o desenvolvimento do estudo, foram identificados dois problemas principais, e este trabalho investigou soluções para ambos.

Problema 1: Ausência de catálogos de *smells* para Cypress

Problema 2: Ainda não existem estudos que usam ferramentas recentes de IA para detecção de *smells*.

Para abordar essas questões, esta dissertação de mestrado foi estruturada em duas partes principais. A primeira parte consistiu em uma revisão sistemática da literatura (RSL), seguida por uma revisão geral da literatura (RLC), visando complementar a análise. A partir dessas revisões, foi possível elaborar um catálogo contendo 14 *smells* específicos para testes desenvolvidos com Cypress.

Na segunda parte, a eficácia do ChatGPT em detectar os *smells* catalogados foi avaliada. Esta etapa do estudo foi dividida em duas fases:

1. Estudo Controlado - Um estudo realizado com a ferramenta Mantis, onde os testes foram desenvolvidos pela autora da dissertação, garantindo um ambiente controlado e uniforme.
2. Estudo com Sistemas GitHub - Uma investigação realizada em sistemas de código aberto hospedados no GitHub, onde foram analisados testes elaborados para esses sistemas em particular.

5.1 Contribuições

Nesta seção, são resumidas as principais contribuições oferecidas por esta dissertação, abordando de forma mais detalhada os avanços e *insights* gerados ao longo do estudo.

5.1.1 Catálogo de Smells

Neste estudo, foram empregadas tanto a Revisão Sistemática da Literatura (RSL) quanto a Revisão de Literatura Cinza (RLC) para abordar o Problema 1, resultando na criação de um catálogo de *smells* específico para este tipo de teste. Ao longo desta investigação, foi catalogado um total de 14 *smells*. Dentre estes, 3 foram identificados tanto na RSL quanto na RLC, enquanto 2 foram exclusivos da RSL. Adicionalmente, 9 *smells* foram detectados somente através da RLC.

5.1.2 Detecção de Smells

Para resolver o Problema 2, avaliou-se a possibilidade de criar uma nova ferramenta ou utilizar uma já existente para a detecção dos *smells* catalogados. Para isso, recorreu-se aos Modelos de Linguagem de Grande Escala (LLMs). Uma das ferramentas mais empregadas

nesse campo é o ChatGPT. Os estudos envolvendo o ChatGPT foram divididos em duas etapas, as quais serão detalhadas nas subseções seguintes.

5.1.2.1 Estudo Controlado - Mantis

Neste primeiro estudo de caso, foram usados os testes do Mantis, onde *smells* foram introduzidos nos testes desenvolvidos pela autora principal da dissertação. Nesta primeira etapa, dois dos 14 *smells* foram removidos da pesquisa devido à inviabilidade de serem detectados pelo ChatGPT. Os resultados desse primeiro estudo foram considerados bem promissores. De forma geral, o ChatGPT foi capaz de identificar doze dos *smells* avaliados. A maioria desses *smells* foi detectada na primeira execução do modelo, representando 67% do total. Os demais foram identificados nas execuções subsequentes.

5.1.2.2 Estudo com Sistemas GitHub

Na segunda etapa, foi conduzido um estudo de campo utilizando o GitHub para encontrar testes em sistemas de código aberto relevantes desenvolvidos com o Cypress. Três sistemas foram escolhidos para análise: Pigallery, Livewire e GlobaLeaks.

Pigallery mostrou o melhor desempenho na detecção dos *smells*, com uma precisão de 0.31 e um *recall* de 0.62. Por outro lado, GlobaLeaks teve o desempenho mais baixo, com uma precisão de apenas 0.15 e um *recall* de 0.31. Uma dificuldade na detecção do *smell* de *Brittle Selectors* nos sistemas contribuiu significativamente para esses resultados.

Com base nessas descobertas, conclui-se que o ChatGPT não se revelou eficaz como um detector de *smells* em sistemas de código aberto.

5.1.3 Trabalhos Futuros

Nesta dissertação, foi conduzido um estudo sobre testes fim-a-fim desenvolvidos com Cypress e sugerimos algumas extensões para este trabalho:

1. Estudo com outras linguagens: Além de focar na identificação de *smells* para testes fim-a-fim desenvolvidos com Cypress, uma extensão valiosa seria replicar o estudo

utilizando outras linguagens e *frameworks* de desenvolvimento de testes fim-a-fim, como Selenium, PlayWright, entre outros. Isso permitiria uma comparação mais ampla e uma compreensão mais abrangente dos *smells* em diferentes ambientes de teste.

2. Estudo com mais sistemas e avaliação com mais profissionais de QA: A detecção de *smells* foi limitada a três sistemas e a avaliação dos resultados foi conduzida por um único profissional de QA (isto é, a autora desta dissertação). Uma direção para futuras pesquisas seria ampliar o estudo para incluir um número maior de sistemas e envolver mais profissionais de QA na avaliação. Isso aumentaria a diversidade e representatividade dos dados coletados, proporcionando uma análise mais abrangente dos *smells* em diferentes contextos de aplicação e perspectivas de profissionais com experiências variadas.
3. Criação de uma ferramenta específica: O ChatGPT foi utilizado com o objetivo de detectar os *smells* presentes nos testes fim-a-fim, mas ficou evidente que ele não é eficaz para esse propósito. Portanto, uma possível extensão seria a criação de uma ferramenta específica para a detecção de *smells* em testes fim-a-fim. Isso poderia aumentar a confiabilidade dos resultados da detecção e proporcionar uma abordagem mais direcionada e eficiente para a identificação de problemas nos testes.
4. Refatoração de *Smells*: Investigar como os *smells* podem ser removidos, ou seja, identificar e aplicar refatorações apropriadas para eliminar os *smells* catalogados. Este processo envolve a análise de cada *smell* individualmente, determinando a causa raiz e escolhendo a técnica de refatoração mais adequada.

Referências

- [1] Amazon Web Services. O que é JavaScript? https://aws.amazon.com/pt/what-is/javascript/?trk=faq_card, 2023. Acesso em: 16 mar. 2024.
- [2] Titus Barik, Brittany Johnson, and Emerson Murphy-Hill. I heart hacker news: expanding qualitative research findings by analyzing social news websites. In *10th Joint Meeting on Foundations of Software Engineering (FSE)*, pages 882–885, 2015.
- [3] Blog Reddit. The Page Objects anti-pattern. https://www.reddit.com/r/softwaretesting/comments/7pvx5r/the_page_objects_antipattern/, 2017. Acesso em: 09 dez. 2023.
- [4] Blog Voidr. Teste E2E: Como garantir a qualidade do seu produto em todas as etapas. <https://voidr.co/blog/teste-e2e/>, 2024. Acesso em: 16 abr. 2024.
- [5] Hudson Borges, Andre Hora, and Marco Tulio Valente. Understanding the factors that impact the popularity of github repositories. In *2016 IEEE international conference on software maintenance and evolution (ICSME)*, pages 334–344. IEEE, 2016.
- [6] Hudson Silva Borges and Marco Tulio Valente. Github proxy server: A tool for supporting massive data collection on github. In *Proceedings of the XXXVI Brazilian Symposium on Software Engineering, (SBES)*, page 370–375, New York, NY, USA, 2022. Association for Computing Machinery.
- [7] William H Brown, Raphael C Malveau, Hays W” Skip” McCormick, and Thomas J Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., 1998.
- [8] BrowserStack. Mastering Test Automation with ChatGPT. <https://www.browserstack.com/guide/mastering-test-automation-with-chatgpt>, 2023. Acesso em: 16 mar. 2024.
- [9] Suelen Goularte Carvalho. *Anomalias na Camada de Apresentação de Aplicativos Android*. PhD thesis, Instituto de Matemática e Estatística, Universidade de São Paulo, 2018.
- [10] Valerio Cosentino, Javier L Cánovas Izquierdo, and Jordi Cabot. A systematic mapping study of software development with github. *Ieee access*, 5:7173–7192, 2017.

- [11] Valerio Cosentino, Javier Luis, and Jordi Cabot. Findings from github: methods, datasets and limitations. In *Proceedings of the 13th International Conference on Mining Software Repositories*, (MSR 16), page 137–141, New York, NY, USA, 2016. Association for Computing Machinery.
- [12] Cypress. Cypress - what cypress is and why you should use it. <https://docs.cypress.io/guides/overview/why-cypress>, 2022. Acesso em: 14 jan. 2024.
- [13] DS Academy. O Que São Large Language Models (LLMs). <https://blog.dsacademy.com.br/o-que-sao-large-language-models-llms/>, 2021. Acesso em: 16 abr. 2024.
- [14] Amin Milani Fard and Ali Mesbah. Jsnose: Detecting javascript code smells. In *2013 IEEE 13th international working conference on Source Code Analysis and Manipulation (SCAM)*, pages 116–125. IEEE, 2013.
- [15] Filip Hric. Page objects vs. App actions in Cypress. <https://applitools.com/blog/page-objects-app-actions-cypress/>, 2021. Acesso em: 09 dez. 2023.
- [16] Francesca Arcelli Fontana, Vincenzo Ferme, Marco Zanoni, and Aiko Yamashita. Automatic metric thresholds derivation for code smell detection. In *6th International Workshop on Emerging Trends in Software Metrics (WETSoM)*, pages 44–53, 2015.
- [17] Martin Fowler and Kent Beck. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, S.l., 1999.
- [18] Vahid Garousi, Michael Felderer, and Mika V. Mäntylä. The need for multivocal literature reviews in software engineering: complementing systematic literature reviews with grey literature. In *20th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 1–6, 2016.
- [19] Sarra Habchi, Geoffrey Hecht, Romain Rouvoy, and Naouel Moha. Code smells in ios apps: How do they compare to android? In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 110–121. IEEE, 2017.
- [20] Geoffrey Hecht, Omar Benomar, Romain Rouvoy, Naouel Moha, and Laurence Duchien. Tracking the Software Quality of Android Applications Along Their Evolution (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 236–247, 2015.
- [21] ICterra. Using ChatGPT for REST Endpoint Testing. <https://www.icterra.com/using-chatgpt-for-rest-endpoint-testing/>, 2021. Acesso em: 09 dez. 2023.

- [22] ISTQB. Certified Tester Syllabus Test Automation Engineer. https://bcr.bstqb.org.br/docs/syllabus_ct_tae_1.0br.pdf, 2016. Acesso em: 09 dez. 2023.
- [23] ISTQB. Certified Tester Syllabus Foundation Level. https://istqb-main-web-prod.s3.amazonaws.com/media/documents/ISTQB-CTFL_Syllabus_2018_v3.1.1.pdf, 2018. Acesso em: 09 dez. 2023.
- [24] Fernando Kamei, Igor Wiese, Crescencio Lima, Ivanilton Polato, Vilmar Nepomuceno, Waldemar Ferreira, Márcio Ribeiro, Caroline Pena, Bruno Cartaxo, Gustavo Pinto, and Sérgio Soares. Grey literature in software engineering: A critical review. *Information and Software Technology*, 138:106609, 2021.
- [25] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer Science & Business Media, S.l., 2006.
- [26] Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Paolo Tonella. Capture-replay vs. programmable web testing: An empirical assessment during test case evolution. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 272–281. IEEE, 2013.
- [27] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. Automated generation of visual web tests from dom-based web tests. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 775–782, 2015.
- [28] Maurizio Leotta, Andrea Stocco, Filippo Ricca, and Paolo Tonella. Robula+: An algorithm for generating robust xpath locators for web testing. *Journal of Software: Evolution and Process*, 28(3):177–204, 2016.
- [29] H. Liu, J. Jin, Z. Xu, Y. Bu, Y. Zou, and L. Zhang. Deep learning based code smell detection. *IEEE Transactions on software Engineering*, pages 1–28, 2019.
- [30] Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Francoise Le Meur. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, 2009.
- [31] Hung Viet Nguyen, Hoan Anh Nguyen, Tung Thanh Nguyen, Anh Tuan Nguyen, and Tien N. Nguyen. Detection of embedded code smells in dynamic web applications. In *27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 282–285, 2012.
- [32] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. Detecting bad smells in source code using change

- history information. In *28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 268–278, 2013.
- [33] R.S. Pressman. *Engenharia de Software - 7.ed.* McGraw Hill Brasil, 2009.
- [34] Leonard Punt, Sjoerd Visscher, and Vadim Zaytsev. The a?b*a pattern: Undoing style in css and refactoring opportunities it presents. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 67–77, 2016.
- [35] Filippo Ricca and Andrea Stocco. Web test automation: Insights from the grey literature. In Tomáš Bureš, Riccardo Dondi, Johann Gamper, Giovanna Guerrini, Tomasz Jurdziński, Claus Pahl, Florian Sikora, and Prudence W.H. Wong, editors, *SOFSEM 2021: Theory and Practice of Computer Science*, pages 472–485, Cham, 2021. Springer International Publishing.
- [36] Renaud Rwemalika, Sarra Habchi, Mike Papadakis, Yves Le Traon, and Marie-Claude Brasseur. Smells in system user interactive tests. *arXiv preprint arXiv:2111.02317*, 2021.
- [37] Wickramasinghe Shanika. Selenium vs. cypress- a complete comparison between the two testing frameworks. <https://www.blazemeter.com/blog/selenium-vs-cypress-a-complete-comparison-between-the-two-testing-frameworks>, 2022. Acesso em: 09 dez. 2023.
- [38] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. Does your configuration code smell? In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 189–200. IEEE, 2016.
- [39] Mohammed Latif Siddiq, Abdus Samee, Sk Ruhul Azgor, Md. Asif Haider, Shehabul Islam Sawraz, and Joanna C. S. Santos. Zero-shot prompting for code complexity prediction using github copilot. In *2023 IEEE/ACM 2nd International Workshop on Natural Language-Based Software Engineering (NLBSE)*, pages 56–59, 2023.
- [40] Elvys Soares, Marcio Ribeiro, Rohit Gheyi, Guilherme Amaral, and Andre Medeiros Santos. Refactoring test smells with junit 5: Why should developers keep up-to-date. *IEEE Transactions on Software Engineering*, 2022.
- [41] Elder Vicente de Paulo Sobrinho, Andrea De Lucia, and Marcelo de Almeida Maia. A systematic literature review on bad smells–5 w’s: Which, when, what, who, where. *IEEE Transactions on Software Engineering*, 47(1):17–66, 2021.
- [42] I. Sommerville. *Engenharia de software*. Pearson Prentice Hall, 2011.

- [43] Andrea Stocco, Rahulkrishna Yandrapally, and Ali Mesbah. Visual web test repair. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 503–514, New York, NY, USA, 2018. Association for Computing Machinery.
- [44] Girish Suryanarayana, Ganesh Samarthiyam, and Tushar Sharma. *Refactoring for Software Design Smells: Managing Technical Debt*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2014.
- [45] The Indian Express. ChatGPT emerges as the most popular AI tool: Study by Bard Midjourney Trail. <https://indianexpress.com/article/technology/artificial-intelligence/chatgpt-most-popular-ai-tool-study-bard-midjourney-trail-9026116/>, 2021. Acesso em: 09 dez. 2023.
- [46] Marco Tulio Valente. *Engenharia de Software Moderna: Princípios e Práticas para Desenvolvimento de Software com Produtividade*. Editora: Independente, 2020.
- [47] Lucas Vegi and Marco Tulio Valente. Code smells in Elixir: Early results from a grey literature review. In *30th International Conference on Program Comprehension (ICPC)*, pages 1–5, 2022.
- [48] A. von Staa. *Programação modular: desenvolvendo programas complexos de forma organizada e segura*. Campos, 2000.
- [49] R. Wazlawick. *Engenharia de software: Conceitos e práticas*. Elsevier Editora Ltda., 2019.
- [50] Claes Wohlin, Per Runeson, Pedro Antônio Martins Silveira Neto, Emelie Engström, Ivanilton Polato Machado, and Eduardo Alves Almeida. On the reliability of mapping studies in software engineering. *Journal of Systems and Software*, 86(10):2594–2610, 2013.
- [51] He Zhang, Xin Zhou, Xin Huang, Huang Huang, and Muhammad Ali Babar. An evidence-based inquiry into the use of grey literature in software engineering. In *42nd International Conference on Software Engineering (ICSE)*, pages 1422–1434, 2020.

Apêndice A

Tabelas de Exclusão de Smells Candidatos ao Catálogo

A.1 Tabela de Exclusão de Smells da RSL

Smell	Motivo para Exclusão
Army Clones/Duplicated Code/ Inconsistent Hierarchy/Lack of Macro Events/ Large Classes/Lack of Macro Events/ Lack of Encapsulation/Long Method/ Long Parameter List/Middle Man/ Shotgun Surgery/Unsuitable Naming/ For elements and variables use name that mirrors functionalities use Ids which identifies the method when there is no specific functional purpose/ Keep names of variables clear to everyone/ Separate words in Ids and class names by a hyphen/ Minimize the number of external libraries/	Smell Tradicional
Keep test cases as simple and short as possible/ Test cases name should contain three sections: what is being tested, the circumstances and expected results/Do not use global variables in test cases/Devote separate database data to each test case/Tag test cases to run just a subset of them when necessary/Do not perform visual actions to setup the test case scenario/A test case must not continue the workflow of other test cases/	Test Smell Genérico

Smell	Motivo para Exclusão
Conditional Assertions/Complicated Setup Scenarios Conspiracy Of Silence/Data Creep/Duplicate Check/ Dependencies between tests/Duplicate Check/ Eager Test/Hardcoded Environment/Lifeless/ Hiding Test Data/Implementation Dependent/ Inconsistent Wording/Lack Of Early Feedback/ Missing Assertion/Narcissistic/Noisy Logging/ Obscure Test/On the Fly/Over-Checking/ Pointless Scenario Descriptions/Test Data Loss/ Incorrect Use of Asserts/Stinky Synchronization/ Testing Data Not Code/Unrealistic Data/ Unsecured Test Data/	Test Smell Genérico
Directly Executing UI Scripts/Run linters to detect anti-pattern/Run test cases with multiple browser's driver/Keep the number of unit tests greater than the number of end-to-end tests/End-to-end tests and integration tests should be developed before unit tests/End-to-end tests and integration tests should be developed before unit tests/	Não Relacionado com Código
Create a separate web driver for each test case	Test Smell mas não específico do Cypress

Tabela A.1: Smells Excluídos RSL

A.2 Tabela de Exclusão de Smells da RLC

Smell	Motivo para Exclusão
Large classes/Duplicated Code/ Long Methods/Not using comments/	Smell Tradicional
Incorrect Use of Asserts/Element is not on screen/ Conditional Overuse/Using random data in mocks/ Test without tags/Dependent it() blocks/ Run independent tests without the CI environment/	Test Smell Genérico

Smell	Motivo para Exclusão
Outdated testing scenarios/Not defining “scripts” in package.json/Incorporate Cypress into your (CI/CD)/Incorrect Cypress configuration/	Não relacionado com Código

Tabela A.2: Smells Excluídos RLC

Apêndice B

Catálogo de Smells

B.1 Catálogo de Smells RSL em Inglês

B.1.1 Unnecessary Waiting

Description: A test that waits for an arbitrary period using `cy.wait()`, for example, `cy.wait(2000)`. However, the use of *wait* is necessary when synchronizing the application with tests, especially when delays occur due to data loading or asynchronous operations, thereby preventing premature test failures.

Reason: This is considered a smell because it can slow down the test or mask slowness in the system.

Articles: A1

B.1.2 Brittle Selectors

Description: As the name suggests, these selectors are prone to renaming in the future, for example, using a specific sequence of classes like `cy.get('.btn.btn-large-890')` instead of more stable identifiers such as `cy.get('[id="submit"]')`.

Reason: This is considered a smell because tests can become fragile if there are poorly named selectors in the interface. If the selector is changed, the test will require maintenance every time this happens.

Articles: A1, A2, A3

B.1.3 Adopt Page Object Pattern

Description: This *smell* refers to using *Page Objects* when writing tests. Introducing this type of data structure can make tests difficult to maintain.

Reason: This is considered a smell in Cypress due to its asynchronous nature. Page Objects focus on the structure of interface elements rather than App Actions, a more viable alternative that improves maintainability by encapsulating application behaviors.

Articles: A1

B.1.4 Use Relative URLs Locator Instead Of Absolute Ones

Description: This *smell* occurs when an absolute URL is used as a parameter in `cy.visit()` call, for example, `cy.visit('http://example.com/courses')`; . The best practice is to use a relative URL as a parameter, as shown below: `cy.visit('/courses')`;

Reason: It is considered a smell because using absolute URLs can lead to more fragile tests. Since absolute URLs include the entire domain of the application, a domain change will also change the URL in the test.

Articles: A1

B.1.5 Unnecessary Navigation

Description: This *smell* is related to tests that perform actions not directly related to the test's focus. For example, a test to verify a book's search mechanism performs a series of preliminary actions not directly related to the test itself, such as authentication, redirection, and accessing the search page.¹ Because of this, it is possible to encounter failures that should not necessarily be captured by the test being conducted.

¹<https://mattarcherblog.wordpress.com/2010/11/29/how-test-automation-with-selenium-or-watir-can-fail/>

Reason: This is a smell due to adding unnecessary complexity to tests, potentially leading to false positives. Unnecessary Navigation can slow down tests and increase the chance of finding failures that are not within the test scope.

Articles: A3

B.1.6 Catálogo de Smells RLC em Inglês

B.1.6.1 Unnecessary Waiting

Description: A test that waits for an arbitrary period using `cy.wait()`, for example, `cy.wait(2000)`. However, the use of *wait* is necessary when synchronizing the application with tests, especially when delays occur due to data loading or asynchronous operations, thereby preventing premature test failures.

Reason: This is considered a smel because it can slow down the test or mask slowness in the system.

Articles: A1, A2, A3, A4, A5, A7, A9, A13, A18, A19, A20, A22, A27

B.1.6.2 Using Force: True When Interacting With Elements

Description: This smell occurs when the `force:true` parameter is used to interact with page elements, e.g., `cy.get('coveredOrDisabledButton').click({force:true})`.

Reason: This is considered a smell because the native wait for the component's presentation is not performed, leading to potential asynchronization between testing and interface, causing elements to be inaccessible at the time of interaction.

Article: A6

B.1.6.3 Visiting External Sites

Description: A test that visits or interacts with external servers, i.e., sites outside the system's domain, e.g., `cy.visit('https://example.com', {chromeWebSecurity:false})`.

Reason: This is a smell because it uses external dependencies, which can lead to unexpected failures and increase the fragility of the tests.

Articles: A1, A4, A5, A13

B.1.6.4 Starting Web Servers

Description: A test that starts a web server using `cy.exec()` or `cy.task()`, e.g., `cy.exec('http-server ./path-to-your-app -p 3000', {timeout:60000})`.

Reason: This is considered a smell because starting a web server within the testing environment relies on external variables and increases the test execution time.

Articles: A1, A4, A5, A13, A26

B.1.6.5 Brittle Selectors

Description: As they are named, these selectors have high chances of being renamed in the future, e.g., using a specific class sequence like `cy.get('.btn.btn-large-890')` instead of more stable identifiers like `cy.get('[id="submit"]')`.

Reason: This is considered a smell because tests can become fragile if there are poorly named selectors in the interface. If the selector is changed, the test will require maintenance every time this happens.

Articles: A1, A2, A4, A5, A7, A13, A19, A20, A22, A28

B.1.6.6 Unnecessary Visibility Checks For Previously Actionable Elements

Description: A test that uses redundant visibility checks when dealing with actionable elements such as `should("be.visible")` or `should("exist")` before actions (click or type), e.g., `cy.get("selector").should("be.visible").click()`.

Reason: This is considered a smell because there are redundant checks for the visibility or existence of a particular element.

Article: A24

B.1.6.7 Adopt Page Object Pattern

Description: This smell refers to the use of Page Objects when writing tests. Introducing this type of data structure can make tests difficult to maintain.

Reason: This is considered a smell in Cypress due to its asynchronous nature. Page Objects focus on the structure of interface elements rather than App Actions, a more viable alternative that improves maintainability by encapsulating application behaviors.

Article: A14

B.1.6.8 Using Cypress Calls In Assignments

Description: A test that attempts to assign the value returned by asynchronous method calls to a variable. Since all Cypress methods are asynchronous, they do not return values that can be directly and immediately assigned to variables. The code snippet below illustrates an instance of this smell:

```
1 describe('Test', () => {
2   ...
3   cy.visit('your-test-page');
4   let x = cy.get("selector");
5   x.click();
6   cy.get('selector').invoke('text').should('include', textExpect);
7
8 })
```

Listagem 14: Code snippet containing the Using Cypress Calls In Assignments smell

In line 4, there is a call to `cy.get()` and the result is being assigned to a variable `x`. This is not permitted in Cypress, as `cy.get()` is an asynchronous method call.

Reason: It is considered a smell because the assigned variable does not retain the result of the function call as it is asynchronous.

Articles: A1, A4, A5, A13

B.1.6.9 Missing Global baseUrl

Description: A test that calls `cy.visit()` without setting a `baseUrl`. This parameter must be defined in a configuration file (usually, `cypress.config.js`). The code snippet below illustrates an instance of this smell:

```
1 describe('Test', () => {
2   ...
3   cy.visit('https://example.com');
4   cy.get('h1').should('contain', 'Your Page Title');
5
6 })
```

Listagem 15: Code snippet containing the Missing Global baseUrl smell

In line 4, there is a call to `cy.visit()` with a fixed URL value as a parameter. This is not recommended because the best practice is to define this URL externally in a configuration file.

Reason: This is a *smell* because changes in the base URL cause alterations in all tests, increasing the risk of errors and making the test more difficult to maintain.

Articles: A1, A4, A5, A7, A13, A18, A19

B.1.6.10 Dirty Inputs

Description: This smell refers to a situation where a test interacts with an input field without clearing its existing content, potentially leading to unexpected or undesired behavior. This typically occurs when a field has a value, and the test proceeds to type a new value into the field without resetting it first. Thus, before filling the field, we should use `clear()` to delete previously existing values. The code snippet below illustrates an instance of this smell:

```
1 describe('Test', () => {
2   ...
3   cy.visit('your-test-page');
4   let value = 'Larissa Bicalho';
5   cy.get('#username').type(value);
6   const newValue = 'newTestValue';
7   // need to user clear() before type()
8   cy.get('#username').type(newValue);
9   ...
10  })
```

Listagem 16: Code snippet containing the Dirty inputs smell

In line 9, a new value is being assigned to the username field without utilizing the `clear()` function. As a result, the value assigned to the field in line 6 will be concatenated with the value assigned in line 9, whereas the recommended behavior should involve replacing the existing value.

Reason: It is considered a smell because the absence of cleanup commands can result in validation issues and data conflicts, making the test inconsistent and difficult to trace.

Articles: A19

B.1.6.11 Cleaning up state with `after()` or `afterEach()`

Description: A test that uses `after()` or `afterEach()` to clear state instead of `before()` or `beforeEach()`. It is not recommended to clean up the test state using `afterEach()` the use of these methods leads to the loss of a functionality of Cypress aimed at tracking each test execution state, making debugging and understanding the test result more difficult. The code snippet below illustrates an instance of this smell:

```
1 describe('Test', () => {
2   ...
3   afterEach(() => {
4     cy.logout();
5   })
6   ...
7   cy.get('h1').should('contain', 'Your Page Title');
8
9 })
```

Listagem 17: Code snippet containing the Cleaning up state with `after()` or `afterEach()` smell

Using `afterEach()` to perform logout in lines 3 and 4 means clearing the test state, which is not considered a good practice. Particularly, if we logout, the tester cannot recover the state if the test fails.

Reason: This is a smell because it makes debugging and understanding the test results more difficult.

Articles: A1, A4, A5, A13, A19

B.1.6.12 Direct Access to List Elements

Description: When a test needs to click on a list element, it should first iterate on these elements (instead of directly accessing the element of interest), because there are chances of getting an element detached from the DOM (Document Object Model). Thus, the following code is not recommended:

```
1 describe('Test', () => {
2   ...
3   cy.get( 'ul[role="listbox"]' ).each(( $btn, index ) => {
4     if(index === 0 ) {
5       cy.get( $btn ).click();
6     }
7     ...
8   })
9 }
```

Listagem 18: Code snippet containing the Direct Access to List Elements smell

In line 3, a call to `cy.get()` is made to interact with the list of `` elements without traversing its children ``. Consequently, on line 5, there is a possibility that the element is not in the DOM.

Instead, we should use:

```
1 describe('Test', () => {
2   ...
3   cy.get( 'ul[role="listbox"]' ).find('li').eq(0).click();
4   ...
5
6 }
```

Listagem 19: Code snippet containing how avoid Direct Access to List Elements smell

In line 3, a call to `cy.get()` is made to interact with the list of `` elements traversing its children ``, with this, preventing the smell from occurring.

Reason: This is a *smell* because this type of access can lead to an element that may no longer be in the DOM later, resulting in a test failure.

Articles: A16