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

Larissa Bicalho, João Eduardo Montandon, Marco Túlio Valente

[1]Universidade Federal de Minas Gerais, Belo Horizonte, MG

larissa.bicalho@dcc.ufmg.br, jemaf@ufmg.br, mtov@dcc.ufmg.br

Resumo. *Os testes de software são peças fundamentais para ajudar a manter a integridade do sistema. Dentre as modalidades existentes, os testes fim-a-fim se destacam por verificar o comportamento integral dos requisitos do sistema. Por outro lado, pouco se tem estudado sobre as más práticas adotadas na implementação deste tipo de teste automatizado. Esta pesquisa investiga essas más práticas e a viabilidade de identificá-las automaticamente. Para isso, foi realizada uma Revisão Sistemática da Literatura e uma Revisão da Literatura Cinza com objetivo de catalogar os smells mais comuns em testes fim-a-fim implementados em Cypress, um popular framework de teste fim-a-fim, resultando em um catálogo de 12 smells. Em seguida, foi conduzido um estudo exploratório onde avaliou-se o desempenho do ChatGPT na detecção desses smells em três sistemas open source. A precisão e cobertura do ChatGPT na detecção dos smells variou entre 15%–31% e 31%–62%, respectivamente. Os resultados mostram variações significativas no desempenho do modelo, destacando a necessidade de estudos mais aprofundados.*

1. Introdução

Considerando que o software é uma das construções humanas mais complexas já realizadas, é natural que uma variedade de erros e inconsistências possam ocorrer [Sommerville 2011]. Para evitar que tais problemas cheguem aos usuários finais e causem prejuízos, são necessárias atividades de teste em projetos de desenvolvimento de software [Barbosa and Hora 2022]. Dentre os tipos existentes, destaca-se o teste fim-a-fim, que visa verificar o comportamento dos requisitos do sistema como um todo [Valente 2020]. 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. Esta pesquisa investigou as más práticas relacionadas ao uso do *framework* Cypress, um *framework* JavaScript para testes fim-a-fim, e estudou a possibilidade de se identificar automaticamente esses problemas. Para isso, realizamos um estudo para identificar os *smells* mais comuns em tais testes por meio de uma Revisão Sistemática da Literatura e uma Revisão da Literatura Cinza. Em seguida, avaliamos o desempenho de um Modelo de Linguagem (ChatGPT) na detecção desses *smells* em testes implementados em três sistemas *open source*.

A revisão sistemática levou a criação de um catálogo com 12 *smells* específicos de testes fim-a-fim implementados com o Cypress. Já a detecção apresentou resultados variados. A precisão na identificação variou entre 15% e 31%, enquanto a cobertura variou entre 31% e 62%. Quando analisado em detalhes, observou-se que o ChatGPT

apresentou desempenho melhor na detecção de alguns *smells* em relação a outros. Tais resultados indicam a necessidade de se aprofundar melhor no assunto.

As principais contribuições deste trabalho são: (a) A criação de um catálogo contendo *smells* específicos para a implementação de testes fim-a-fim utilizando o Cypress; (b) Um estudo exploratório sobre o uso de Modelos de Linguagem para detecção automática dos *smells* catalogados.

Este trabalho está organizado da seguinte forma. A Seção 2 apresenta uma contextualização sobre o tópico investigado. Já a Seção 3 descreve a revisão sistemática da literatura, bem como apresenta o catálogo de *smells* resultante deste processo. O estudo sobre a detecção automática dos *smells*, e os resultados obtidos, estão descritos na Seção 4. As Seções 5 e 6 apresentam os trabalhos relacionados e os riscos à validade do estudo. A Seção 7 conclui o trabalho realizado.

2. Conceitos Básicos

2.1. Testes Fim-a-Fim

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 [Leotta et al. 2016]. 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 [ISTQB 2016]. 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 [Sommerville 2011].

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, detalharemos mais sobre esse *framework*.

2.2. O *Framework* Cypress

O framework Cypress[1] foi desenvolvido pela empresa Cypress.io em 2017 e apresenta diversas funcionalidades essenciais para o desenvolvimento de testes fim-a-fim. 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.

[1] https://www.cypress.io/

Para ilustrar o funcionamento do Cypress, apresentamos um exemplo de um cenário de autenticação de login na ferramenta Mantis Bug Tracker,[2], um popular rastreador de *bugs open source* descrito na Listagem 1.

```
describe('Realizar Login', ()=>{
    beforeEach(()=>{
        cy.visit(Cypress.config('url'))
    })
    it('RealizarLoginSucesso', ()=>{
        var username = Cypress.config('username')
        var senha = Cypress.config('senha')
        var expectedText = "administrator"
        loginPage.preencherUser(username)
        loginPage.clicarLogar()
        loginPage.preencherSenha(senha)
        loginPage.clicarEntrar()
        homePage.validarTituloHome(expectedText)
    })
})
```

Listing 1: Teste End-To-End para verificar o procedimento de Login.

Para melhor compreensão do teste, é importante entender o conceito de *Page Object*, usado para abstrair as páginas como objetos a serem manipulados. 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. O exemplo em questão faz uso de dois *Page Objects*: as classes `LoginPage` e `HomePage`. Cada uma dessas classes contém métodos para interagir com os elementos da página, como preencher campos de texto, clicar em botões, e validar informações.

O fluxo de autenticação no Mantis envolve três etapas principais: preenchimento do nome de usuário, preenchimento da senha e validação do usuário na tela inicial. Inicialmente, o cenário de teste descrito na Listagem 1 preenche os campos de login e senha para acessar a página inicial do Mantis. Os valores de *username* e senha são recuperados de variáveis presentes no arquivo *Cypress.json* através da função `Cypress.config()`. Em seguida, a classe `LoginPage` é usada para preencher os campos de *username* 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.

3. Revisão Sistemática da Literatura

O objetivo deste trabalho é revelar as más práticas no desenvolvimento de testes utilizando o CYPRESS. Optou-se por realizar uma revisão sistemática da literatura tanto acadêmica—i.e., análise de artigos científicos—quanto cinza—i.e., análise de blogs, fóruns de perguntas e respostas, etc. Essa decisão foi tomada pois a literatura cinza representa uma fonte de evidência relevante para o contexto do trabalho [Garousi et al. 2016, Kamei et al. 2021, Zhang et al. 2020].

[2]https://www.mantisbt.org/index.php

Para ambos os casos, a revisão sistemática foi conduzida em quatro etapas: (a) Busca de Artefatos; (b) Filtragem de Artefatos; (c) Extração dos *smells*; e (d) Validação dos *smells*. Essas etapas estão detalhadas a seguir.

3.1. Busca de Artefatos

Metodologia Nesta etapa definiu-se uma *string* de busca e a base de dados a ser utilizada para cada tipo de revisão. Para a literatura acadêmica, utilizou-se as bases ACM DL, IEEE Xplore, Science Direct, Springer, e Wiley para busca dos artigos, em consonância com trabalhos anteriores da área [Sobrinho et al. 2021]. A *string* de busca foi composta a partir dos termos *end-to-end* e *bad smell*, e está detalhada abaixo.

```
    ("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")
```

Já a revisão da literatura cinza foi realizada utilizando o Google como base de busca. Desta vez, optou-se por compor a *string* de busca a partir dos termos *bad smell* e *Cypress*, conforme detalhado abaixo.

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

Artefatos Selecionados A busca por artefatos resultou em 49 artigos previamente selecionados na literatura acadêmica, e 79 na literatura cinza.

3.2. Filtragem de Artefatos

Metodologia Nesta etapa, foram definidos critérios para a exclusão de artefatos considerados não relacionados ao estudo conduzido. Para isso, a autora realizou a leitura do resumo de cada artigo selecionado na literatura acadêmica, e descartou aqueles que não abordavam *smells* em testes fim-a-fim. De forma análoga, a autora realizou a leitura dos títulos e descrições dos 79 artefatos selecionados na literatura cinza, e descartou aqueles que não abordavam *smells* em testes escritos em *Cypress*.

Artefatos Filtrados Ao final desta etapa, restaram três artigos na literatura acadêmica e 28 na literatura cinza.

3.3. Extração dos *Smells*

Metodologia Para a extração dos *smells*, a autora realizou a leitura completa dos três artigos selecionados na literatura acadêmica, e dos 28 artefatos selecionados na literatura cinza. Durante a leitura, a autora identificou e marcou os trechos que descrevem explicitamente os *smells* de cada artefato.

Smell	Descrição
Espera Desnecessária	Um teste que aguarda por um período arbitrário usando `cy.wait()`
Utilizar force: true para Interagir com Elementos	Ocorre quando o parâmetro `force:true` é usado para interagir com elementos da página, por exemplo, `cy.get('elem').click({force:true})`.
Visitar Páginas Externas	Visita ou interage com servidores externos, ou seja, sites fora do domínio do sistema, por exemplo, `cy.visit('https://example.com')`.
Inicializar Serviços Web	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')`.
Seletores Frágeis	Ocorre em seletores que tem 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 `cy.get('[id="submit"]')`.
Verificações de Visibilidade Desnecessárias	Um teste que utiliza verificações redundantes de visibilidade ao lidar com elementos acionáveis, tais como `should("be.visible")` ou `should("exist")` antes de ações.
Atribuir Retorno de Funções Cypress em Variáveis	Um teste que tenta atribuir o valor retornado por chamadas de métodos assíncronos a uma variável.
Não Utilizar URLs Globais	Um teste que chama `cy.visit()` sem definir um baseUrl.
Campos Sujos	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.
Limpar Estado com after() ou afterEach()	Um teste que usa `after()` ou `afterEach()` para limpar o estado em vez de `before()` ou `beforeEach()`.
Acessar Diretamente Elementos de Lista	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).
Utilizar URLs Absolutas	Ocorre quando uma URL absoluta é usada como parâmetro na chamada de `cy.visit()`, por exemplo, `cy.visit('http://example.com/courses');`

Tabela 1. *Smells* resultantes da revisão sistemática da literatura.

***Smells* Descobertos** Foram identificados 69 *smells* a partir dos artigos selecionados na literatura acadêmica, e 88 a partir dos artefatos selecionados na literatura cinza.

3.4. Validação dos *Smells*

Metodologia Por fim, cada um dos *smells* identificados anteriormente foi analisado em dois passos. Primeiramente, a autora agrupou os *smells* semelhantes, i.e., aqueles que possuem o mesmo significado mas apresentavam nomes distintos; isto reduziu a quantidade de *smells* para 60. Em seguida, a autora categorizou os *smells* conforme seu escopo de aplicação, e selecionou aqueles que são específicos para testes fim-a-fim desenvolvidos com o *framework* Cypress. Por exemplo, descartou-se *smells* considerados tradicionais, válidos para qualquer tipo de teste, e relacionados a outros aspectos do sistema.

***Smells* Selecionados** Ao final desta etapa, pôde-se compor um catálogo com 12 *smells* específicos para testes fim-a-fim desenvolvidos com o *framework* Cypress. A Tabela 1 lista cada um dos *smells* catalogados, junto da sua descrição.

4. Detecção de *Smells*

Na segunda parte deste trabalho, foi conduzido um estudo empírico para avaliar a possibilidade de se identificar os *smells* catalogados automaticamente. Para isso, optou-se por selecionar o ChatGPT como ferramenta de detecção. Essa escolha se justifica pelo desempenho que esse tipo de modelo (e.g., modelos GPT) tem apresentado na automação de tarefas voltadas para o desenvolvimento de software [Chen et al. 2024, Schäfer et al. 2024]. Basicamente, esta avaliação verificou se o ChatGPT é capaz de identificar os 12 *smells* reportados na Tabela 1 em projetos *open source* presentes no GitHub que fazem uso do Cypress para desenvolver seus testes fim-a-fim. O restante desta seção detalha a metodologia e os resultados obtidos.

4.1. Metodologia

A metodologia empregada para a detecção de *smells* no ChatGPT foi dividida em três etapas: (a) seleção de projetos no GitHub; (b) criação do oráculo de teste; e (c) requisição ao *prompt* do ChatGPT. Essas etapas estão detalhadas a seguir.

Seleção de Projetos no GitHub A seleção de projetos no GitHub utilizou como critério principal a declaração do *framework* Cypress como dependência do projeto. Para buscar sistemas que se encaixem nessa restrição, utilizou-se a ferramenta *ghtopdep*[3], responsável por buscar repositórios populares que dependam de alguma biblioteca a ser fornecida. Neste caso, configuramos a ferramenta para que listasse os projetos mais populares que dependem do *framework* Cypress. Em seguida, filtramos os projetos com mais de 1.000 estrelas e que possuíssem uma quantidade significativa de testes fim-a-fim. A Tabela 2 apresenta os sistemas selecionados para a avaliação.

Projeto	Descrição	# Estrelas	# Testes
Pigallery	Site otimizado para galeria de fotos.	1,4K	6
Powergrid	Manipulador de *datatables* para uso em conjunto no *framework* Laravel.	1,2K	8
GlobaLeaks	Permite a qualquer usuário configurar e manter uma plataforma segura para denúncias.	1,1K	20

Tabela 2. Sistemas Selecionados para Avaliação.

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 da Tabela 1. Os *smells* identificados manualmente neste estudo podem ser vistos na Tabela 3.

Requisição ao *prompt* do ChatGPT Para cada arquivo mapeado no oráculo de teste, foi construído um *prompt* específico para ser enviado ao ChatGPT, conforme exemplificado na Tabela 4. O texto do *prompt* começa com uma contextualização sobre o problema e

[3]`https://pypi.org/project/ghtopdep/0.1.0/`

Smells	Pigallery	Livewire	GlobaLeaks
Espera Desnecessária	2	1	1
Utilizar force: true ao interagir com elementos	5	0	2
Seletores Frágeis	6	8	19
Verificações de Visibilidade Desnecessárias	0	0	1
Acessar Diretamente Elementos de Lista	0	0	2

Tabela 3. Quantidade de Smells por Sistema.

uma breve descrição de cada um dos doze *smells* levantados. Em seguida, é informado ao modelo o formato no qual ele deverá apresentar sua resposta, seguido de diretivas algumas diretivas gerais a respeito de quando ele deve reportar ou não os *smells*. Finalmente, é repassado ao final do *prompt* o código-fonte do arquivo de teste.

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. Prompt enviado ao ChatGPT durante o Segundo Experimento

O *prompt* foi executado no ChatGPT 3.5—versão mais recente do modelo disponível gratuitamente na época da execução—utilizando sua configuração padrão, sem ajustes adicionais. Cada *prompt* foi enviado uma vez, e os resultados foram compilados manualmente pela autora do trabalho.

Métricas de Avaliação Os resultados foram avaliados conforme as métricas de precisão e *recall* em relação às ocorrências reportadas na Tabela 3. Para o cálculo de ambas as métricas, é necessário quantificar as ocorrências consideradas *Verdadeiro Positivo*, *Falso Positivo*, e *Falso Negativo*. Dado um mesmo arquivo, consideramos a ocorrência como *Verdadeiro Positivo* quando um determinado *smell* é reportado tanto pelo Oráculo quanto pelo ChatGPT. Já ocorrências do tipo *Falso Positivo* ocorrem quando um *smell* é reportado pelo ChatGPT, mas não foi identificado no Oráculo. Por fim, ocorrências são consideradas *Falso Negativo* quando o *smell* está presente no Oráculo mas não foi reportado pelo ChatGPT.

4.2. Resultados

Uma avaliação de precisão total e *recall* total em todos os sistemas pode ser vista na Tabela 5. 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.

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

Tabela 5. Resultado geral para precisão e *recall*.

Para uma compreensão mais aprofundada das possíveis causas dessa avaliação, uma análise individual de cada sistema é descrita na Tabela 6. Ao analisar resultados do Pigallery, observa-se que tanto os *smells Seletores Frágeis* e *Utilizar force: true ao interagir com elementos* atingiram uma precisão de 100%. Em outras palavras, a detecção desses *smells* pelo ChatGPT refletiu sua presença nos testes. Em contrapartida, o *Espera Desnecessária* registrou uma precisão de apenas 33%. Já em relação ao *recall*, apenas o *smell Seletores Frágeis* apresentou valor abaixo do total (17%).

Sistema	Smell	Precisão (%)	*Recall (%)*
Pigallery	Seletores Frágeis	100	17
	Utilizar force: true ao interagir com elementos	100	100
	Espera Desnecessária	33	100
Livewire	Seletores Frágeis	100	38
	Espera Desnecessária	20	100
GlobaLeaks	Verificações de Visibilidade Desnecessárias	11	50
	Seletores Frágeis	100	22
	Utilizar force: true ao interagir com elementos	33	50
	Espera Desnecessária	22	100

Tabela 6. Precisão e *recall* individual por *smell* em cada sistema.

Com relação ao Livewire, observa-se que o *smell Seletores Frágeis* apresentou alta precisão (100%) e baixo *recall* (38%). Um efeito oposto é percebido junto ao *Espera Desnecessária*, no qual apresentou 20% de precisão e 100% de *recall*.

Já em relação ao GlobaLeaks, observamos uma tendência de menor desempenho se comparado com os resultados dos demais sistemas. Apenas um dos quatro *smells* detectados pelo ChatGPT alcançou 100% de precisão (*Seletores Frágeis*). A taxa para os demais *smells* varia entre 11% e 33%. Comportamento similar pode ser observado para o *recall*. Apenas *Espera Desnecessária* alcançou 100% de cobertura; os demais apresentaram desempenho entre 22% e 50%. O ChatGPT não identificou nenhuma das duas ocorrências do *smell Acessar Diretamente Elementos de Lista*; este é o único caso de um *smell* que foi mapeado no Oráculo mas não foi encontrado pelo modelo.

5. Trabalhos Relacionados

A literatura sobre *smells* é vasta e abrange diversos tópicos relacionados tanto ao levantamento quanto à detecção de más práticas no desenvolvimento de software. Alguns trabalhos investigam a ocorrência de *smells* em sistemas [Palomba et al. 2013, Moha et al. 2009, Liu et al. 2019, Nguyen et al. 2012]. Por outro lado, o trabalho apresentado tem como objetivo catalogar sistematicamente *smells* voltados para o desenvolvimento de testes end-to-end. Alguns trabalhos também fazem uso da literatura cinza para catalogar *smells* em contextos específicos como nas plataformas Android [Hecht et al. 2015] e iOS [Habchi et al. 2017], nas linguagens JavaScript [Fard and Mesbah 2013], Elixir [Vegi and Valente 2022], e CSS [Punt et al. 2016], e nas bibliotecas Puppet [Sharma et al. 2016] e ReactJS [Ferreira and Valente 2023]. Descrevemos abaixo os principais trabalhos relacionados ao nosso estudo.

5.1. *Smells* em Testes

Soares et al. [Soares et al. 2022] apresentaram 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.

Ricca e Stocco [Ricca and Stocco 2021] 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: *Gerenciar a sincronização com o aplicativo web, Criar localizadores/seletores robustos/corretos e Manter os testes atômicos e curtos.*

Rwemalika et al. [Rwemalika et al. 2021] estudaram *smells* em testes interativos do usuário do sistema (SUITs). A pesquisa caracterizou a difusão e a frequência com que esses *smells* surgiram. Para isso, 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 teste. No entanto, *smells* tratados de forma incorreta tendem a não ser removidos mais tarde.

O presente trabalho se diferencia dos estudos da literatura ao focar no levantamento de *smells* em testes fim-a-fim desenvolvidos especificamente para o contexto do *framework* Cypress, algo que não havia sido explorado anteriormente. Para avaliar a possibilidade de se identificar os *smells* catalogados automaticamente, o estudo explorou o uso de Modelos de Linguagem (LLMs)—especificamente o ChatGPT—para detectar *smells* em testes fim-a-fim implementados em sistemas reais.

6. Riscos a Validade

Validade Interna Existe a possibilidade de viés na classificação adotada durante a categorização dos *smells*. No entanto, essa categorização foi fundamentada em 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 Os projetos utilizados na avaliação empírica foram selecionados a partir de critérios específicos, o que pode limitar a generalização dos resultados. Contudo, a popularidade é um indicador frequentemente utilizado para selecionar projetos utilizados pela comunidade de desenvolvimento [Borges and Valente 2022].

Validade de Construção A estrutura da *string* de consulta utilizada durante a revisão sistemática da literatura apresenta um risco potencial. Para lidar com isso, foram conduzidas buscas preliminares para refinar a consulta, incluindo sinônimos, seguindo recomendações anteriores. Além disso, foi revisado cuidadosamente os documentos recuperados, e estratégias similares adotadas em estudos anteriores obtiveram resultados igualmente interessantes.

Validade das Conclusões As métricas de precisão e *recall* utilizadas para avaliar o desempenho do ChatGPT podem não refletir completamente sua capacidade. Por outro lado, essas métricas são utilizadas para medir o desempenho de modelos de linguagem [Silva et al. 2024].

7. Conclusão

Este trabalho apresentou um estudo sobre os *smells* encontrados no desenvolvimento de testes fim-a-fim utilizando o *framework* Cypress. O estudo foi conduzido em duas etapas. Primeiro, foram realizadas duas revisões sistemáticas da literatura—uma acadêmica e uma cinza—com o intuito de catalogar o *smells* em testes fim-a-fim. Este levantamento permitiu a composição de um catálogo com 12 *smells*, específicos para testes desenvolvidos com Cypress. Em seguida, foi conduzida uma avaliação empírica para verificar a possibilidade de se identificar os *smells* catalogados automaticamente utilizando Modelos de Linguagem (i.e., ChatGPT). Em linhas gerais, os resultados obtidos indicam a necessidade de se aprimorar a configuração destes modelos para uma melhor detecção dos *smells* catalogados.

Como trabalhos futuros, pretendemos nos aprofundar nas técnicas de treinamento e configuração dos Modelos de Linguagem com intuito de melhorar a detecção dos *smells* catalogados. Também pretendemos reproduzir o estudo com outros *frameworks* de testes fim-a-fim (e.g., Playwright, Selenium, etc) para avaliar a generalização dos *smells* catalogados. Por fim, planejamos incluir novos sistemas no processo de avaliação, além de envolver profissionais da área de testes para avaliar a relevância dos *smells* catalogados.

Disponibilidade de Artefatos O conjunto de dados com as planilhas e artigos base para a construção desse artigo está disponível publicamente em: `https://doi.org/10.5281/zenodo.12547333`.

Referências

Barbosa, L. and Hora, A. (2022). How and Why Developers Migrate Python Tests. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 538–548.

Borges, H. S. and Valente, M. T. (2022). 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. Association for Computing Machinery.

Chen, Y., Hu, Z., Zhi, C., Han, J., Deng, S., and Yin, J. (2024). ChatUniTest: A Framework for LLM-Based Test Generation. In *ACM International Conference on the Foundations of Software Engineering (FSE)*.

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

Ferreira, F. and Valente, M. T. (2023). Detecting code smells in React-based web apps. *Information and Software Technology*, 155:1–16.

Garousi, V., Felderer, M., and Mäntylä, M. V. (2016). 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.

Habchi, S., Hecht, G., Rouvoy, R., and Moha, N. (2017). 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.

Hecht, G., Benomar, O., Rouvoy, R., Moha, N., and Duchien, L. (2015). 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.

ISTQB (2016). Certified Tester Syllabus Test Automation Engineer. `https://bcr.bstqb.org.br/docs/syllabus_ct_tae_1.0br.pdf`. Acesso em: 09 dez. 2023.

Kamei, F., Wiese, I., Lima, C., Polato, I., Nepomuceno, V., Ferreira, W., Ribeiro, M., Pena, C., Cartaxo, B., Pinto, G., and Soares, S. (2021). Grey literature in software engineering: A critical review. *Information and Software Technology*, 138:106609.

Leotta, M., Stocco, A., Ricca, F., and Tonella, P. (2016). Robula+: An algorithm for generating robust xpath locators for web testing. *Journal of Software: Evolution and Process*, 28(3):177–204.

Liu, H., Jin, J., Xu, Z., Bu, Y., Zou, Y., and Zhang, L. (2019). Deep learning based code smell detection. *IEEE Transactions on software Engineering*, pages 1–28.

Moha, N., Guéhéneuc, Y.-G., Duchien, L., and Le Meur, A.-F. (2009). Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1):20–36.

Nguyen, H. V., Nguyen, H. A., Nguyen, T. T., Nguyen, A. T., and Nguyen, T. N. (2012). Detection of embedded code smells in dynamic web applications. In *27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 282–285.

Palomba, F., Bavota, G., Penta, M. D., Oliveto, R., Lucia, A. D., and Poshyvanyk, D. (2013). Detecting bad smells in source code using change history information. In *28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 268–278.

Punt, L., Visscher, S., and Zaytsev, V. (2016). 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.

Ricca, F. and Stocco, A. (2021). Web test automation: Insights from the grey literature. In Bureš, T., Dondi, R., Gamper, J., Guerrini, G., Jurdziński, T., Pahl, C., Sikora, F., and Wong, P. W., editors, *SOFSEM 2021: Theory and Practice of Computer Science*, pages 472–485, Cham. Springer International Publishing.

Rwemalika, R., Habchi, S., Papadakis, M., Traon, Y. L., and Brasseur, M.-C. (2021). Smells in system user interactive tests. *arXiv preprint arXiv:2111.02317*.

Schäfer, M., Nadi, S., Eghbali, A., and Tip, F. (2024). An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering*, 50(1).

Sharma, T., Fragkoulis, M., and Spinellis, D. (2016). Does your configuration code smell? In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 189–200. IEEE.

Silva, L. L., Silva, J., Montandon, J. E., and Valente, M. T. (2024). Detecting code smells using chatgpt: Initial insights. In *International Symposium on Empirical Software Engineering and Measurement - Emerging Results, Vision and Reflection (ESEIW-ERVR)*, pages 1–7.

Soares, E., Ribeiro, M., Gheyi, R., Amaral, G., and Santos, A. M. (2022). Refactoring test smells with junit 5: Why should developers keep up-to-date. *IEEE Transactions on Software Engineering*.

Sobrinho, E. V. d. P., De Lucia, A., and Maia, M. d. A. (2021). A systematic literature review on bad smells–5 w's: Which, when, what, who, where. *IEEE Transactions on Software Engineering*, 47(1):17–66.

Sommerville, I. (2011). *Engenharia de software*. Pearson Prentice Hall.

Valente, M. T. (2020). *Engenharia de Software Moderna: Princípios e Práticas para Desenvolvimento de Software com Produtividade*. Editora: Independente.

Vegi, L. and Valente, M. T. (2022). Code smells in Elixir: Early results from a grey literature review. In *30th International Conference on Program Comprehension (ICPC)*, pages 1–5.

Zhang, H., Zhou, X., Huang, X., Huang, H., and Babar, M. A. (2020). An evidence-based inquiry into the use of grey literature in software engineering. In *42nd International Conference on Software Engineering (ICSE)*, pages 1422–1434.