UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS DE QUIXADÁ

CURSO DE GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

ÉRICA MIRANDA DE SOUSA

INVESTIGANDO AS CAUSAS DE *FLAKY TESTS* EM AUTOMAÇÃO DE TESTES UI EM PROJETOS *OPEN-SOURCE*

QUIXADÁ

2022

ÉRICA MIRANDA DE SOUSA

INVESTIGANDO AS CAUSAS DE *FLAKY TESTS* EM AUTOMAÇÃO DE TESTES UI EM PROJETOS *OPEN-SOURCE*

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Sistemas de Informação do Campus de Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Sistemas de Informação.

Orientadora: Profa. Dra. Carla Ilane Moreira Bezerra

QUIXADÁ

2022

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

S698i Sousa, Érica Miranda de.
Investigando as causa de Flaky Tests em automação de Testes UI em projetos open-source / Érica Miranda de Sousa. – 2022.
53 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Quixadá, Curso de Sistemas de Informação, Quixadá, 2022.
Orientação: Profa. Dra. Carla Ilane Moreira Bezerra.

1. Flaky Tests. 2. Teste de Software. 3. Interfaces de usuário (Sistemas de computação). I. Título.
CDD 005

ÉRICA MIRANDA DE SOUSA

INVESTIGANDO AS CAUSAS DE *FLAKY TESTS* EM AUTOMAÇÃO DE TESTES UI EM PROJETOS *OPEN-SOURCE*

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Sistemas de Informação do Campus de Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Sistemas de Informação.

Aprovada em: ___/___/___

BANCA EXAMINADORA

Profa. Dra. Carla Ilane Moreira Bezerra (Orientadora)
Universidade Federal do Ceará (UFC)

Prof. Dr. Ivan do Carmo Machado
Universidade Federal da Bahia (UFBA)

Profa. Dra. Rainara Maia Carvalho
Universidade Federal do Ceará (UFC)

Dedico esse trabalho aos meus avós, que sempre incentivaram meus sonhos, apoiaram todas as minhas decisões e me ensinaram que o conhecimento é o único bem que não pode nos ser roubado. É graças aos seus esforços que cheguei até aqui e é graças a eles que continuarei caminhando, obrigada!

AGRADECIMENTOS

Primeiramente agradeço a minha família, que sempre me incentivou nos momentos difíceis e compreendeu minha ausência enquanto me dedicava à realização deste trabalho.

Agradeço aos amigos, em especial a Letícia e Marcus, que sempre estiveram ao meu lado, pela amizade incondicional e pelo apoio demonstrado ao longo de todo o período de tempo em que dividimos juntos as angustias e felicidades da graduação.

Sou grata pela confiança depositada na minha proposta de projeto pela professora Carla Ilane, orientadora deste trabalho, e por me manter motivada durante todo o processo.

Também agradeço à Universidade Federal do Ceará e seus docentes que me incentivaram a percorrer o caminho da pesquisa científica.

Por fim, agradeço a todas as pessoas com quem convivi ao longo desses anos de curso, que me incentivaram e que certamente tiveram impacto na minha formação acadêmica.

“Conhecimento não é aquilo que você sabe, mas o que você faz com aquilo que você sabe.”

(Aldous Huxley)

RESUMO

Flaky Tests são casos de testes automatizados que apresentam um comportamento não-determinístico, ou seja, não é possível determinar se o caso de teste irá passar ou falhar a cada execução. Casos de testes com essa característica, tendem a apresentar resultados aleatórios, de sucesso ou falha, a cada nova execução sobre a mesma versão do Sistema em Teste (SUT) e sem alterações no seu código fonte. Tal comportamento tende a ser prejudicial no processo de desenvolvimento de software. Cada falsa falha destrói a relação determinística entre os resultados do teste e a qualidade do software, sendo necessário um trabalho de depuração no código de teste e no código de produção para determinar se a falha foi ocasionada por defeitos no sistema ou pela presença de um *Flaky Test*. Essa depuração consome tempo e mão de obra, podendo causar atrasos nas entregas. Assim, buscando entender mais sobre a ocorrência de *Flaky Tests* em Testes UI, propomos nesse trabalho uma análise empirica sobre *commits* relacionados a *Flaky Tests* de projetos *open-souce* do GitHub, para identificar suas principais causas e estratégias de correção aplicadas. Ao todo foram analisados 123 de 23 projetos, gerando um total de 8 categorias de causa e 7 categorias de correção, onde Adição de Espera, Correção de Lógica e Teste Ignorado figuram como as principais causa de *Flaky Tests* em Tests UI e Condição de Corrida, Problemas de Lógica e Dependência de Teste são apontadas como as principais categorias de correção aplicadas na resolução de *Flaky Tests* em Testes UI. Pode-se ainda identificar quais as estratégias de correção mais utilizadas para cada causa identificada, onde identificou-se que: 89% dos *Flaky Tests* causados por uma Condição de Corrida são corrigidos através de uma Adição de Espera; 100% dos *Flaky Tests* causados por Problemas de Lógica e Dependência de Testes são corrigidos aplicando correções na lógica aplicada no testes. Os resultados apontados nesse trabalho servirão como insumo para o entendimento da ocorrência de *Flaky Testes* em Testes UI e servirão de auxilio para Analistas de Testes e e testadores a desenvolver projetos de automação de Testes UI com maior qualidade, além de fornecer insumos para estudos futuros.

Palavras-chave: *Flaky Tests*. Teste de software. Interfaces de usuário (Sistemas de computação)

ABSTRACT

Flaky Tests are automated test cases that have a non-deterministic behavior, that is, it is not possible to determine if the test case will pass or fail at each execution. Test cases with this characteristic tend to present random results, success or failure, with each new execution on the same version of the System under Test (SUT) and without changes in its source code. Such behavior tends to be detrimental to the software development process. Each false failure destroys the deterministic relationship between test results and software quality, requiring debugging work on test code and production code to determine whether the failure was caused by system defects or the presence of a textitFlaky Test. This debugging consumes time and manpower and can cause delays in deliveries. Thus, seeking to understand more about the occurrence of *Flaky Tests* in UI Tests, we propose in this work an empirical analysis of *commits* related to *Flaky Tests* of *open-souce* projects on GitHub , to identify its main causes and applied correction strategies. In all, 123 out of 23 projects were analyzed, generating a total of 8 categories of cause and 7 categories of correction, where Add Waiting, Correction of Logic and Ignored Test appear as the main causes of *Flaky Tests* in Tests UI and Race Condition, Logic Problems and Test Dependency are pointed out as the main fix categories applied in solving *Flaky Tests* in UI Tests. It is also possible to identify which correction strategies are most used for each identified cause, where it was identified that: 89% of *Flaky Tests* caused by a Race Condition are corrected through a Wait Addition; 100% of *Flaky Tests* caused by Logic Problems and Test Dependency are fixed by applying corrections to the logic applied to the tests. The results pointed out in this work will serve as input for understanding the occurrence of *Flaky Tests* in UI Tests and will help Test Analysts and testers to develop UI Tests automation projects with higher quality, in addition to providing inputs for future studies.

Keywords: Flaky tests. Software testing. User interfaces (Computer systems)

LISTA DE FIGURAS

Figura 1 – Pirâmide de testes 21
Figura 2 – Log do JUnit indicando que o teste “testLoginSuccessfully” foi executo com sucesso 24
Figura 3 – Resumo das diretrizes para MSRs sistemáticos 29
Figura 4 – Visão geral do processo de obtenção dos conjuntos de dados utilizados . . . 31
Figura 5 – Procedimentos metodológicos 35
Figura 6 – Arquitetura do *script* para coleta de *commits* 37
Figura 7 – Processo de escolha dos *commits* para análise 39
Figura 8 – Processo de escolha dos *commits* para análise 40
Figura 9 – Exemplo de teste com condição de corrida 44
Figura 10 – Exemplo de teste com problemas de lógica 44
Figura 11 – Exemplo de teste com problemas de lógica 45
Figura 12 – Exemplo de adição de espera 46
Figura 13 – Exemplo de adição de espera 47
Figura 14 – Exemplo de correção de lógica 47
Figura 15 – Exemplo de correção de lógica 47
Figura 16 – Exemplo de teste ignorado na suíte de execução 48

LISTA DE QUADROS

Quadro 1 – Categorias de manifestação de *flaky tests* em testes baseados em UI 26
Quadro 2 – Categorias de causa correção de *flaky tests* em testes de unidade 27
Quadro 3 – Categorias de causa e correção de *flaky tests* em testes baseados em UI . . . 28
Quadro 4 – Análise comparativa entre os trabalhos relacionados e este trabalho 34
Quadro 5 – Repositórios coletados via GHTorrent 36
Quadro 6 – Dados coletados pelo *script* 38
Quadro 7 – Descrição dos repositórios selecionados via GHTorrent 39
Quadro 8 – Projetos de onde foram extraídos os dados para análise 42
Quadro 9 – Resumo das categorias de causa de *Flaky Tests* 43
Quadro 10 – Resumo das categorias de Estratégias de Correção de *Flak Tests* 45
Quadro 11 – Relação entre as causas e estratégias de correção 49

SUMÁRIO

1 INTRODUÇÃO 15
1.1 Questões de Pesquisa 17
1.2 Organização 17
2 FUNDAMENTAÇÃO TEÓRICA 19
2.1 Teste de Software 19
2.1.1 Técnicas de testes de software 20
2.2 Automação de testes de software 21
2.2.1 Automação de testes de software baseado em interface (Testes UI) 22
2.3 *Flaky Tests* 24
2.3.1 Causas e estratégias de correção 27
2.4 Mineração de repositórios de Software 28
3 TRABALHOS RELACIONADOS 30
3.1 *An Empirical Analysis of Flaky Tests* 30
3.2 *A Study on the Lifecycle of Flaky Tests* 31
3.3 *An Empirical Analysis of UI-based Flaky Tests* 32
3.4 *A Survey on How Test Flakiness Affects Developers and What Support They Need To Address It* 33
3.5 Análise comparativa 33
4 METODOLOGIA 35
4.1 Definição de critérios para seleção de repositórios 35
4.2 Desenvolvimento de *scripts* para coleta de dados 36
4.2.1 GHTorrent 36
4.2.2 API V3 GitHub 37
4.2.3 Script para coleta de commits 37
4.3 Coleta de dados 38
4.3.1 Seleção dos repositórios 38
4.3.2 Extração e filtragem dos commits 38
4.4 Análise dos dados 39
4.4.1 Categorização 40
4.5 Resumo da base de dados 42

5 RESULTADOS 43
5.1 RQ1: Quais as causas mais recorrentes de *Flaky Tests* em Testes UI? . . 43
5.1.1 Condição de Corrida 43
5.1.2 Problemas de Lógica 44
5.1.3 Dependência de Teste 45
5.2 RQ2: Quais as estratégias de correção mais utilizadas no tratamento de *Flaky Tests*? 45
5.2.1 Adição de Espera 46
5.2.2 Correção de Lógica 46
5.2.3 Teste ignorado 47
5.3 RQ3: Quais as estratégias de correção mais utilizadas para cada causa identificada? 48
5.4 Ameaças à validade 49
6 CONCLUSÕES E TRABALHOS FUTUROS 51
6.1 Resultados encontrados 51
6.2 Trabalhos Futuros 51
REFERÊNCIAS 53

1 INTRODUÇÃO

Flaky Tests são casos de testes automatizados que apresentam um comportamento não-determinístico, ou seja, não é possível determinar se o caso de teste irá passar ou falhar a cada execução (ROMANO *et al.*, 2021). Casos de testes com essa característica, tendem a apresentar resultados aleatórios, de sucesso ou falha, a cada nova execução sobre a mesma versão do Sistema em Teste (SUT) e sem alterações no seu código fonte (LUO *et al.*, 2014; ECK *et al.*, 2019; LAM *et al.*, 2019; LAM *et al.*, 2020b; PINTO *et al.*, 2020). Tal comportamento tende a ser prejudicial no processo de desenvolvimento de software (LAM *et al.*, 2020b). Cada falsa falha destrói a relação determinística entre os resultados do teste e a qualidade do software (ROMANO *et al.*, 2021), sendo necessário um trabalho de depuração no código de teste e no código de produção para determinar se a falha foi ocasionada por defeitos no sistema ou pela presença de um *Flaky Test*. Essa depuração consome tempo e mão de obra, podendo causar atrasos nas entregas (LAM *et al.*, 2020b).

Flaky Tests tendem a ser ainda mais prejudiciais quando inseridos dentro de um ambiente de Integração Contínua (IC), onde cada falsa falha impede a criação de novas *builds*, exigindo a intervenção manual no processo, o que, novamente, exige tempo para depuração do problema, causando atrasos nas entregas (LAM *et al.*, 2020b). Os efeitos danosos da ocorrência de *Flaky Tests* foram relatados nos últimos anos por várias empresas, como por exemplo, a relatou Google que 1,5% de todos os testes executados são instáveis e quase 16% de seus 4,2 milhões de testes individuais falham independentemente de alterações no código ou nos testes (MICCO, 2017). Da mesma forma, cerca de 4,6% dos testes de cinco projetos da Microsoft são instáveis (LAM *et al.*, 2020a).

Ao falar sobre testes automatizados, é necessário diferenciar os tipos de automação existentes dentro do processo de desenvolvimento: Testes de Unidade e Testes Baseados em Interface de Usuário (Testes UI). Testes de Unidade, são responsáveis por testar as unidades do sistema, ou seja, os métodos que compõem o código fonte do mesmo, que normalmente são mantidos pelos próprios desenvolvedores (SOMMERVILLE, 2011). Testes UI consistem em *scripts* que, através de *frameworks*, são capazes de simular o uso do sistema na perspectiva do usuário, interagindo com a interface do SUT, validando os resultados obtidos (ROMANO *et al.*, 2021).

Tanto Testes de Unidade quanto Testes UI estão suscetíveis à ocorrência de *Flaky Tests*, ou seja, casos de testes que apresentam resultados de sucesso e falha para execuções sobre o

mesmo SUT e sem alterações em seu código fonte (LAM *et al.*, 2020a). Nos últimos anos, alguns autores se dedicaram a estudar os *Flaky Tests*. Em seu trabalho, Luo *et al.* (2014) analisaram 201 *commits* que corrigiam a ocorrência de *Flaky Tests* em 51 projetos *open-source*. Através dessa análise foi possível gerar uma lista de causas para *Flaky Tests*, bem como indicar abordagens para sua identificação e descrever estratégias de correção. Assim como Luo *et al.* (2014), Lam *et al.* (2020a) dedicaram-se a estudar o ciclo de vida de *Flaky Tests* em projetos da Microsoft, concentrando-se em identificar seus impactos, estratégias de correções e abordagens para a detecção de *Flaky Tests*. Já Eck *et al.* (2019) concentraram-se em entender a perspectiva dos desenvolvedores sobre *Flaky Tests* conduzindo um estudo onde os desenvolvedores classificaram testes anteriormente corrigidos por eles. No entanto, esses e outros estudos concentram-se em estudar a ocorrência de *Flaky Tests* em Testes de Unidade.

Comparado com os Teste de Unidade, o ambiente de execução e o processo de automação do Teste de UI são significativamente diferentes. Primeiro, muitos dos eventos nesses testes, como lidar com a entrada do usuário, fazer chamadas de API do sistema operacional (ou navegador), baixar e renderizar vários recursos (como imagens e scripts) exigidos pela interface são altamente assíncronos por natureza. Isso significa que os eventos do usuário e várias outras tarefas serão acionados em uma ordem não determinística (ROMANO *et al.*, 2021). Segundo, é mais difícil reproduzir *Flaky Tests* em Testes UI, pois os Testes UI trabalham com duas camadas (usuário interface e interface código) (ROMANO *et al.*, 2021).

Dessa forma, este trabalho propõe identificar as principais causas de *Flaky Tests* em Testes UI de projetos *open-source*, através da análise de *commits* de correção. Sendo o objetivo geral deste trabalho investigar as causas de *Flaky Tests* em testes automatizados baseados em interface em projetos *open-source*, bem como identificar as principais estratégias de correção de *Flaky Tests* aplicadas.

Para comprender um pouco mais sobre a ocorrência de *Flaky Tests* em Testes UI, realizamos um estudo exploratório em 123 *commits* extraídos de projetos *open-source* do GitHub. Os resultados dessa análise apontam que Condição de Corrida, Problemas de Lógica e Dependência de Teste são a causa raiz de aproximadamente 91%, dos *Flaky Tests* analisados. Também identificamos que Adição de Espera, Correção de Lógica e Ignorar o Teste Aleatório são as estratégias de correção mais utilizadas para correção de *Flaky Tests* nos *commits* analisados. Identificamos ainda qual a estratégia de correção mais utilizada para causa de *Flaky Tests* identificada, espera-se que o conhecimento sobre esses pontos possa auxiliar analistas de testes

e testadores a desenvolver projetos de automação de Testes UI com maior qualidade, além de fornecer insumos para estudos futuros.

1.1 Questões de Pesquisa

As seguintes questões de pesquisa nortearam a execução deste trabalho:

RQ1: Quais as causas mais recorrentes de *Flaky Tests* em Testes UI?

Ao identificar as causas recorrentes de *Flaky Tests* em Testes UI, espera-se fornecer insumos para conscientização para Analistas de Testes, que ao conhecerem essas causas, evitem inseri-lás em seus testes. Foram examinados *commits* relacionados a *Flaky Tests* em Teste UI, para determinar as principais causas desse comportamento. As causas identificadas foram agrupadas em 8 categorias, sendo 3 delas as causa mais recorrentes: Condição de Corrida, Problemas de Lógica e Dependência de Teste.

RQ2: Quais as estratégias de correção mais utilizadas no tratamento de *Flaky Tests*?

Para cada *commits* de *Flaky Test* analisado, identificamos a correção utilizada para corrigir o comportamento instável. As estratégias de correção foram agrupadas em estratégias semelhantes, resultando em 7 categorias de estratégias de correção, sendo 3 delas recorrentes: Adição de Espera, Correção de Lógica e Teste Ignorado.

RQ3: Quais as estratégias de correção mais utilizadas para cada causa identificada?

Ao analisar um *commit* é possível identificar qual a causa raíz do problema que está sendo corrigido. Após mapear todas as causas, é possível identificar as relações entre as causas básicas dos problemas e como o problema foi corrigido, bem como qual estratégia de correção foi a mais utilizada para corrigir cada categoria de problema. Analisando essa relações, conseguimos determinar que: Problemas causados por *Condição de Corrida* são em grande maioria corrigidos com a Adição de Espera; Problemas de lógica são sempre corrigidos com atividade de refatoração da lógica implementada no teste e Dependência de Teste é corrigida através da remoção dessa dependência.

1.2 Organização

Este trabalho está organizado da seguinte forma: no Capítulo 2 são apresentados os termos e conceitos necessários para o entendimento do trabalho, onde explicamos o conceito de Teste de Software e detalhamos suas principais técnicas de teste, contextualizando em que

momento a automação de testes de software faz sentido e por quê utilizá-la. Neste capítulo também é feita uma introdução ao conceito de *Flaky Tests*, explicando seus impactos no processo de desenvolvimento de software. No Capítulo 3, são apresentados trabalhos relacionados, ressaltando suas semelhanças e diferenças ao trabalho aqui proposto. No Capítulo 4, são detalhados os procedimentos metodológicos que foram empregados na realização deste estudo. Por fim, no Capítulo 5, os resultados encontrados são detalhados bem como as ameaças a validade, fechando esse trabalho com o Capítulo 6 apresentando uma breve discussão sobre os resultados, trabalhos futuros e considerações finais.

2 FUNDAMENTAÇÃO TEÓRICA

Nesta Seção serão apresentados os conceitos mais relevantes necessários para o entendimento deste trabalho. Na Seção 2.1 é apresentada a definição básica de teste de software. A Seção 2.2 contém os principais conceitos relacionados a automação de testes de software. Na subseção 2.2.1 é apresentado o conceito de automação de teste de software baseado em Interface (Teste de UI). A Seção 2.3 introduz o conceito *Flaky Tests*, bem como suas implicações no mercado e academia. Por fim, a seção 2.4 apresenta os principais conceitos de Mineração de Repositórios de software.

2.1 Teste de Software

Software são construções complexas que estão sujeitas a erros e inconsistências. Para evitar que os erros cheguem ao usuário final, e causem prejuízos, é necessário introduzir atividades de testes (VALENTE, 2020). Sommerville (2011) define quatro atividades básicas no processo de desenvolvimento de *software*: especificação, desenvolvimento, validação e evolução. As atividades de validação e verificação tem como objetivo mostrar que um software se adequa a suas especificações e satisfaz os requisitos do cliente.

Segundo Sommerville (2011), a Verificação é uma atividade a qual, envolve a análise de um sistema para certificar se este atende aos requisitos funcionais e não funcionais. Já a Validação, é a certificação de que o sistema atende as necessidades e expectativas do cliente.

O conceito de teste de software pode ser compreendido através de uma visão intuitiva ou mesmo de uma maneira formal (WHITTAKER, 2000). Existem atualmente várias definições para esse conceito. De uma forma simples, testar um software significa verificar através de uma execução controlada se o seu comportamento corre de acordo com o especificado (WHITTAKER, 2000). O objetivo principal desta tarefa é revelar o número máximo de falhas dispondo do mínimo de esforço, ou seja, mostrar aos que desenvolvem se os resultados estão ou não de acordo com os padrões estabelecidos (PRESSMAN, 2005).

Atualmente existem muitas maneiras de se testar um software. Mesmo assim, existem técnicas que sempre foram muito utilizadas em sistemas desenvolvidos sobre linguagens estruturadas (PINTO *et al.*, 2020). Apesar de os paradigmas de desenvolvimento serem diferentes, o objetivo principal destas técnicas continua a ser o mesmo: encontrar falhas no software (PRESSMAN, 2005).

2.1.1 Técnicas de testes de software

O objetivo principal do processo de teste de software é detectar a presença de erros no sistema testado. Sendo o teste bem sucedido, aquele que consegue determinar situações nas quais o software falhe. Para alcançar esse objetivo, podem ser aplicadas diferentes técnicas. A seguir são descritas algumas dessas técnicas.

Técnica Estrutural (ou teste caixa-branca). Técnica de teste que avalia o comportamento interno do componente de software. Essa técnica trabalha diretamente sobre o código fonte do componente de software para avaliar aspectos tais como: teste de condição, teste de fluxo de dados, teste de ciclos e teste de caminhos lógicos (PRESSMAN, 2005).

Este tipo de teste é desenvolvido analisando-se o código fonte e elaborando-se casos de teste que cubram todas as possibilidades do componente de software. Um exemplo bem prático desta técnica de teste é o uso da ferramenta livre JUnit[1] para desenvolvimento de casos de teste para avaliar classes ou métodos desenvolvidos na linguagem Java (WHITTAKER, 2000). A técnica de teste de estrutural é recomendada para os níveis de Testes de Unidade e Testes de Integração.

Teste Funcional (ou teste caixa-preta). Técnica de teste em que o componente de software a ser testado é abordado como se fosse uma caixa-preta, ou seja, não se considera o comportamento interno do mesmo. Dados de entrada são fornecidos, o teste é executado e o resultado obtido é comparado a um resultado esperado previamente conhecido. A técnica de teste funcional é aplicável a todos os níveis de teste (PRESSMAN, 2005).

As técnicas de teste são classificadas de acordo com a origem das informações utilizadas para estabelecer os requisitos de teste. Elas contemplam diferentes perspectivas do software e impõe-se a necessidade de se estabelecer uma estratégia de teste que contemple as vantagens e os aspectos complementares dessas técnicas (ZHANG *et al.*, 2014). As técnicas existentes são: técnica funcional e estrutural (PRESSMAN, 2005).

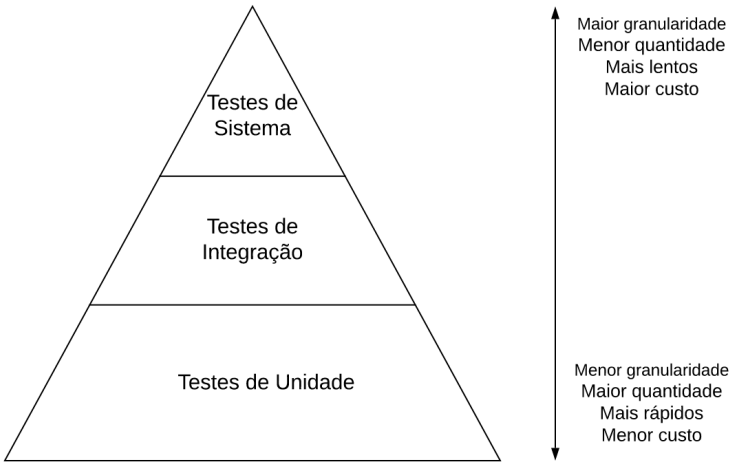
Para auxiliar no processo de validação, as equipes de desenvolvimento e testes têm recorrido à automação de casos de testes, que permitem a execução rápida de um grande número de casos de testes gerando economia de tempo e recursos (ZHANG *et al.*, 2014).

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

2.2 Automação de testes de software

Testes automatizados são *scripts* desenvolvidos utilizando *frameworks* e executam o SUT sem intervenção manual. Um conjunto de testes automatizados de qualidade auxiliam na rotina de testes de um sistema e na garantia de um produto final de qualidade. Os diferentes tipos de automação utilizados durante o processo de desenvolvimento de software foram representadas de forma gráfica na Pirâmide de Testes (Figura 1), proposta por Cohn (2009). Ela ilustra os tipos de testes, seus níveis, velocidade de implementação e complexidade dos testes realizados, particionando-os de acordo com sua granularidade (VALENTE, 2020).

Figura 1 – Pirâmide de testes



Fonte: Valente (2020)

Na base da pirâmide de testes, encontram-se os testes de unidade. Sua posição indica que esses testes possuem menor custo de implementação e são mais rápidos de executar e, por esse motivo, devem ser os testes de maior número dentro da suíte de testes de um sistema (VALENTE, 2020). Teste de unidade tem por objetivo testar uma única funcionalidade, aplicação ou característica do produto. São os testes realizados na menor parte testável de uma aplicação, independentemente da sua interação com outras partes do código (COHN, 2009). Por testarem os menores pedaços possíveis do código de forma isolada, os testes de unidade tendem a ser extremamente pequenos e de rápida criação e execução. Essa característica permite que caso um dos testes falhe, seja possível saber com precisão o local da falha (SOMMERVILLE, 2011).

No nível intermediário, encontram-se os testes de integração, que possuem como objetivo testar um conjunto de unidades interagindo entre si (COHN, 2009). Alguns casos comuns de cobertura de testes de integração são testes realizados na comunicação com o

banco de dados, comunicação de interfaces, APIs, microserviços (VALENTE, 2020). Por isso, demandam mais esforço para implementação e executam de forma mais lenta que os testes de unidade (VALENTE, 2020).

No topo da pirâmide encontram-se os testes de sistema ou testes de interface, que testam a aplicação na perspectiva do usuário final (ROMANO *et al.*, 2021), executados depois da aplicação estar completamente integrada e concluída para serem validados os requisitos funcionais (VALENTE, 2020). São testes que simulam o ambiente real, navegando dentro da aplicação, clicando em botões, preenchendo formulários e, ao fim, validam se o comportamento apresentado condiz com o esperado para à funcionalidade (COHN, 2009). Estes testes costumam ser mais complexos de manter e possuem uma alta complexidade de automação, por isso encontram-se em menor número na suíte de testes de um software (VALENTE, 2020).

2.2.1 Automação de testes de software baseado em interface (Testes UI)

Como mencionado anteriormente, Testes UI são *scripts* que interagem com a interface do SUT (Sistema em Teste), testando o sistema na perspectiva do usuário final. Esses testes, validam se a camada de interface com o usuário está sendo exibida conforme elicitado nos requisitos e se as funcionalidades estão funcionando corretamente, sem apresentar falhas (VALENTE, 2020). Ao testar a interface do usuário, é possível interagir com o aplicativo exatamente como o usuário. Não temos acesso a nenhum código interno, não podemos verificar solicitações de rede e não podemos consultar nossa camada de persistência ou gravar em uma variável interna (EL-MORABEA; EL-GAREM, 2021).

Apesar do objetivo comum, Testes UI apresentam características diferentes dos testes de unidade, como o ambiente de execução e o processo de automação (ROMANO *et al.*, 2021). Testes UI lidam com eventos de entrada do usuário, chamadas de API, *download* e renderização dos componentes da interface, esses eventos são assíncronos e acionados em ordem não determinística (EL-MORABEA; EL-GAREM, 2021). Além disso, *Flaky Tests* nesse contexto são mais difíceis de reproduzir, pois é difícil cobrir todos os casos de uso simulando eventos de usuário (ROMANO *et al.*, 2021).

Como são testes de ponta a ponta (*end-to-end*), Tests UI tendem a ser mais caros, mais lentos e menos numerosos. Testes UI costumam ser também frágeis, pois mínimas alterações nos componentes da interface podem demandar modificações nesses testes (VALENTE, 2020).

O trecho de código abaixo é um exemplo de Teste UI - utilizando Java, JUnit e

Selenium WebDriver - que realiza o teste da funcionalidade de *login* do sistema web Moodle[2]:

```
public class TestLogin {

  private WebDriver driver;

  @Before
  public void openBrowser() {
    WebDriverManager.chromedriver().setup();
    driver = new ChromeDriver();
    driver.get("https://moodle2.quixada.ufc.br/login/index.php");
  }

  @Test
  public void testLoginSuccessfully(){
    driver.findElement(By.id("username")).sendKeys("01234567890");
    driver.findElement(By.id("password")).sendKeys("passtest");
    driver.findElement(By.id("loginbtn")).click();

    WebElement menuBar = driver.findElement(By.xpath("//*[@id=\"
       page-site-index\"]/nav"));

    assertTrue(menuBar.isDisplayed());
  }

  @After
  public void closeBrowser() {
    driver.quit();
  }
}
```

Código-fonte 1 – Código fonte de Teste de UI em Java

A classe *TestLogin* é um *script* que realiza o teste da funcionalidade de *login* na

[2] https://moodle2.quixada.ufc.br/login/index.php

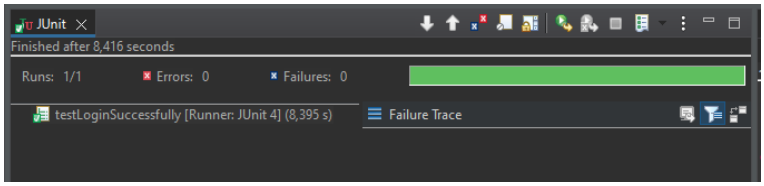
perspectiva do usuário. Seu código é capaz de abrir o navegador, interagir com a tela e completar o fluxo testado, validando se tudo ocorreu conforme o esperado.

A função *openBrowser* é responsável por abrir o navegador, neste exemplo está sendo utilizado o Chrome. Para acessar a página, é utilizado o método *driver.get()*, passando como parâmetro a URL que desejamos acessar, no caso a página de *login* do Moodle. A tag *@Before* indica que a função *openBrowser* deve ser sempre executada antes de cada teste.

A função *testLoginSucessfully* realiza o teste em si. Para interagir com os campos de entrada de texto CPF e login, é utilizado o objeto driver - do tipo WebElement -, que através do método *driver.findElement()*, identifica os elementos através dos seus respectivos Ids e Xpaths. os campos de *input* são localizados na Árvore Dom da página renderizada, através de seus Ids, e utiliza o método *sendKeys()* para simular a ação de digitação e então inserir nos campos de *input* os valores passados como parâmetro. Já o método *click()* clica no botão "Acessar"e realiza o envio do formulário.

Por último, é realizada a verificação do sucesso no processo de login, se o processo tiver sido concluído com sucesso, o Moodle exibe uma página principal contendo uma barra superior - capturado através de um Xpath -. A função *isDisplayed()* apenas retorna *true* ou *false* identificando se a barra está sendo exibida ou não. A função *assertTrue* válida se o resultado retornado por *isDisplayed()* é *true*, se sim, significa que o login foi realizado com sucesso e considera o teste como aprovado (Figura 2). Se o retorno da função *isDisplayed()* for false, significa que a barra não está sendo exibida e houve erro no processo de login.

Figura 2 – Log do JUnit indicando que o teste “testLoginSuccessfully” foi executo com sucesso



Fonte: Elaborado pela autora.

2.3 *Flaky Tests*

A definição exata de *Flaky Tests* podem variar dependendo do autor, mas geralmente são considerados *Flaky Test* os testes que passam e falham para mesma versão do SUT e sem alterações no código de teste (PARRY *et al.*, 2021).

Alguns autores classificam como *Flaky Tests*, aqueles testes que falham devido a

problemas de simultaneidade (VAHABZADEH *et al.*, 2015). Porém, a grande parte dos trabalhos relacionados definem como *Flaky Tests* aqueles testes que apresentam resultados de falha e sucesso para casos de testes sem alterações em seu código fonte e que executam sobre a mesma versão do SUT (PARRY *et al.*, 2021). Essa definição é amplamente aceita por autores e inclui *Flaky Tests* causados por fatores externos ao teste, como a execução anterior de outros testes ou a própria plataforma de execução (PARRY *et al.*, 2021).

O tipo mais comum de *Flaky Test* relacionado a ordem de execução dos testes, são conhecidos como Testes Dependentes de Ordem, cujo seu resultado depende da ordem de execução do teste (ZHANG *et al.*, 2014). Quanto à plataforma de execução, um teste com dependência de plataforma pode ser considerado *Flaky*, pois pode passar em uma máquina, mas falhar em outra (ECK *et al.*, 2019).

Flaky Tests costumam ser prejudiciais durante os testes de regressão, pois desafiam a suposição de que uma falha implica em um *defeito* (LAM *et al.*, 2020a). Para desenvolvedores, *Flaky Tests* destroem a confiança na suíte de testes e levam a desperdício de tempo na depuração do defeito (PARRY *et al.*, 2021).

Eck *et al.* (2019) em uma pesquisa recente, com desenvolvedores industriais, mostraram a prevalência e a ocorrência de *Flaky Tests* na perspectiva dos desenvolvedores. Os resultados desse estudo, indicaram que *Flaky Tests* é um problema encontrado com frequência, pois 20% dos entrevistados afirmam lidar mensalmente com eles, 24% citam lidar semanalmente e 15% afirmam lidar com *Flaky Tests* diariamente. Em termos de gravidade, 91% dos desenvolvedores que afirmam lidar com *Flaky Tests* pelo menos uma vez por ano, 56% os descreveram como um problema moderado e 23% acharam que eram um problema sério.

Luo *et al.* (2014) listaram alguns problemas intrínsecos decorrentes da presença de *Flaky Tests* durante os testes de regressão: Primeiro, as falhas de teste causadas por *Flaky Tests* podem ser difíceis de reproduzir devido ao seu não determinismo. Em segundo lugar, os testes fragmentados causam atrasos nas entregas, pois o desenvolvedor pode gastar um tempo substancial depurando apenas para descobrir que a falha não é devido às mudanças recentes, mas devido a um teste irregular. Terceiro, *Flaky Tests* também podem ocultar bugs reais: se um teste fragmentado falhar com frequência, os desenvolvedores tendem a ignorar suas falhas e, portanto, podem perder bugs reais.

A presença de testes instáveis impõe uma carga significativa aos desenvolvedores que usam pipelines de IC (LAM *et al.*, 2019). Em uma pesquisa realizada com 58 desenvolvedores da

Microsoft, Lam *et al.* (2019) descobriram que os *Flaky Tests* são considerados a segunda razão, dentre 10, que desacelera as implantações de software. Uma pesquisa detalhada adicional com 18 dos desenvolvedores mostrou que eles valorizam a depuração e correção de erros existentes.

Normalmente identificados durante os testes de regressão, os *Flaky Tests* causam atrasos na liberação de novas versões de sistemas pois demandam esforço e tempo de análise e refatoração do código de produção e testes para determinar se a falha ocorreu devido a falhas no sistema ou no código de testes (PARRY *et al.*, 2021).

Devido à natureza não determinística dos *Flaky Tests*, detectá-los é bastante desafiador e tem se tornado uma área de pesquisa cada vez mais ativa (LAM *et al.*, 2020b). A forma mais comum de identificação de *Flaky Tests* utilizada por desenvolvedores é a execução do caso de teste *n* vezes, normalmente 10 vezes, se o caso de teste passar ou falhar pelo menos uma vez, o mesmo será classificado como não-determinístico (LAM *et al.*, 2020b). Claramente, essa não é forma mais vantajosa para identificação, pois a mesma demanda tempo e mão de obra para execução repetida dos casos de testes, tal cenário torna-se insustentável para grandes sistemas com uma grande suíte de testes automatizados (PINTO *et al.*, 2020). Alguns autores chegaram a propor ferramentas que realizam a identificação de *Flaky Tests*, como as ferramentas Flakiness e Time Balancer (FaTB) (LAM *et al.*, 2020a), mas nenhuma apresenta grande adesão pelos times de desenvolvimento e qualidade de software (PARRY *et al.*, 2021).

Luo *et al.* (2014) propõem em seu trabalho categorias de correções de *Flaky Tests* em Testes de Unidade, onde explicam que a maioria dos testes *Async Wait* (85%) não esperam por recursos externos e envolvem apenas um pedido, podendo ser detectados adicionando um atraso de tempo em uma determinada parte do código, sem a necessidade de controlar o ambiente externo. Da mesma forma, Romano *et al.* (2021) propõe estratégias de correção para *Flaky Tests* de Testes UI (Quadro 1).

Quadro 1 – Categorias de manifestação de *flaky tests* em testes baseados em UI

Categoria de manifestação	**Total**
Não especificado	141
Especificar a plataforma problemática	38
Reordenar suíte	12
Redefinir configuração entre os testes	9
Fornecer trecho de código	14
Forçar condição do ambiente	15

Fonte: Romano *et al.* (2021)

2.3.1 Causas e estratégias de correção

Os fatores que levam a ocorrência de *Flaky Tests* ainda estão sendo estudados, porém, nos estudos já existentes, os autores atribuem sua ocorrência a presença de esperas implícitas e dependências de testes como as causas mais comuns para ocorrência de *Flaky Tests* (ROMANO *et al.*, 2021).

Devido a falta de informações concretas, os desenvolvedores ainda não chegaram a um consenso sobre como tratar *Flaky Tests*, estando focados na sua refatoração a medida que vão sendo identificados, sem apoio de ferramentas automatizadas ou diretrizes nesse processo. Alguns desenvolvedores simplesmente ignoram a presença dos *Flaky Tests*, removendo-os da suíte de testes de regressão, através de comentários ou utilizando tags dos JUnit, o que não pe uma boa opção, pois apenas ignorar *Flaky tests* pode ser prejudicial para o sistema, pois os mesmos podem estar ocultando falhas reais no sistema (PINTO *et al.*, 2020).

Alguns trabalhos, como Luo *et al.* (2014) e Romano *et al.* (2021) apontam em seus trabalhos estratégias de correções mais utilizadas na correção de *Flaky Tests*. Romano *et al.* (2021) categorizou as principais estratégias utilizadas para correção de *Flaky Tests* em Testes de Unidade (Quadro 2).

Quadro 2 – Categorias de causa correção de *flaky tests* em testes de unidade

Categoria de causa	**Categoria de correção**
Espera Assíncrona	Adicionar/Modicar espera Adicionar/Modificar Sleep Reordenar execução Outros
Simultaneidade	Operação Atômica de Bloqueio Tornar determinístico Mudança de Condição Asserção de mudança Outros
Dependência de Teste	Estado de Configuração/Limpeza Remover depedência Testes de mesclagem

Fonte: Luo *et al.* (2014)

Da mesma forma, Romano *et al.* (2021) propõe estratégias de correção para *Flaky Tests* de Testes UI (Quadro 3).

As categorias de Causa de *Flaky Tests* propostas por Luo *et al.* (2014) e Romano *et al.* (2021) são definidas como:

Quadro 3 – Categorias de causa e correção de *flaky tests* em testes baseados em UI

Categoria de causa	**Categoria de correção**
Espera Assíncrona	Adicionar/aumentar atraso Corrigir mecanismos de espera
Ambiente	Corrigir acesso à API Alterar versão da biblioteca
Problema de Lógica	Implementação de lógica de refatoração
Problemas de Plataforma	Desativar recursos

Fonte: Romano *et al.* (2021)

Esperas Assíncronas ocorrem quando a execução do teste faz uma chamada assíncrona e não espera adequadamente o resultado da chamada estar diponível antes de usá-lo (LUO *et al.*, 2014; ROMANO *et al.*, 2021).

Problemas de Simultaneidade ocorrem quando o não determinismo do teste é devido a diferentes *threads* interagindo de uma maneira não desejável, devido a corridas de dados, violações de atomicidade ou impasses (LUO *et al.*, 2014).

Dependência de Teste ocorre quando o resultado do teste depende da ordem em que os testes são executados. Idealmente, casos de teste devem isolados e independentes uns dos outros, onde a ordem em que os testes são executados não deve afetar seus resultados (LUO *et al.*, 2014).

Problemas de Plataforma ocorrer quando testes sofrem com problemas oriundos de uma plataforma particular que faz com que os resultados obtidos ou ações realizadas sejam diferentes entre execuções consecutivas dentro da mesma plataforma (ROMANO *et al.*, 2021). Já problema de Lógica ocorrem devido à lógica incorreta implementada nos scripts de teste (ROMANO *et al.*, 2021).

Problemas de ambiente ocorrem devido diferenças na plataforma subjacente usada para executar os testes, por exemplo: teste executado em diferentes navegadores (ROMANO *et al.*, 2021).

2.4 Mineração de repositórios de Software

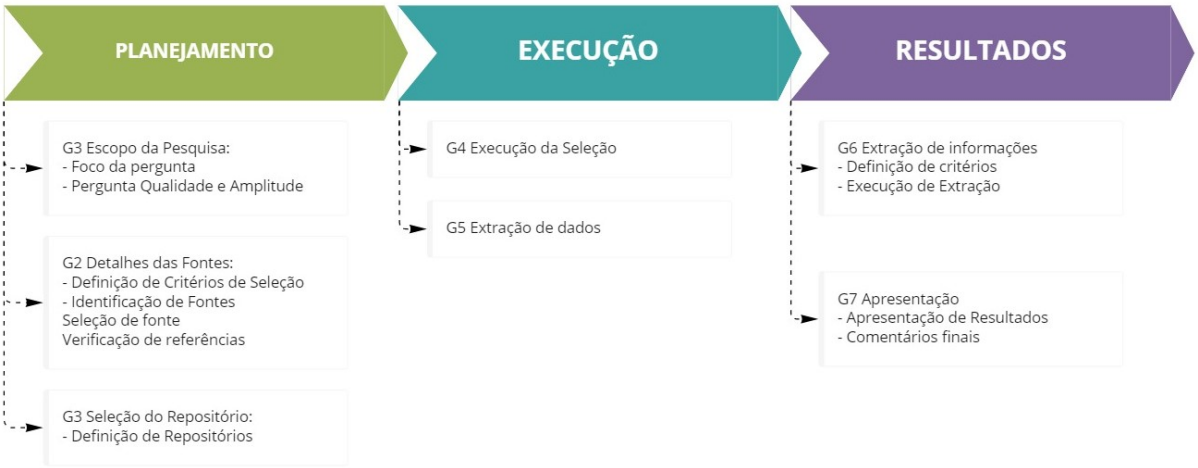
Mineração de Repositórios de Software (MSR) é uma área da Engenharia de Software Empírica que se concentra na extração de informações de repositórios de software utilizando técnicas e ferramentas de mineração de dados (HAN *et al.*, 2011). Desde seu surgimento em 2004, o campo MSR tem sido uma ferramenta importante para diversos estudos, crescendo com o avanço da tecnologia, fornecendo novas técnicas, mapeamentos e um conjunto de dados rico

de informações (BARROS *et al.*, 2021).

Nos estudos que aplicam MSR são selecionados repositórios de acordo com critérios pré-estabelecidos, é realizada a extração de seus dados para obter as informações necessárias para responder às questões de pesquisa (VIDONI, 2022). As informações armazenadas em Repositórios de Software permitem um melhor entendimento do processo de desenvolvimento e evolução do software e fornece apoio a tomada de decisões importantes por parte dos desenvolvedores e gerentes (MSR, 2022).

Um MSR é dividido em três fases: Planejamento, Execução e Resultados (VIDONI, 2022), seguindo as diretrizes da Figura 3.

Figura 3 – Resumo das diretrizes para MSRs sistemáticos



Fonte: Vidoni (2022)

Na fase de Planejamento devem ser definidos os objetivos e questões de pesquisa. Além de selecionar as fontes dos repositórios que serão utilizadas na pesquisa, para isso é necessário definir os critérios de seleção de fontes (VIDONI, 2022). Definidas as fontes, é necessário descrever em detalhes o processo e os critérios de seleção e avaliação dos repositórios, especificando os critérios de inclusão e exclusão dos mesmos (VIDONI, 2022).

Na Execução, deve ser realizada a extração dos dados, registrando em detalhes os critérios de inclusão e exclusão de dados e a execução da extração (VIDONI, 2022). Na fase de Análise deve ser realizada a extração das informações, descrevendo os critérios pelos quais as informações obtidas dos dados devem ser avaliadas. Por último, deve ser realizada a apresentação dos resultados obtidos de forma clara e objetiva (VIDONI, 2022).

3 TRABALHOS RELACIONADOS

Foram identificados na literatura estudos que se relacionam com a identificação de causas comuns para ocorrência de *Flaky Tests*. Nesta seção, são descritos as contribuições desses estudos e como eles se relacionam com o presente trabalho.

3.1 *An Empirical Analysis of Flaky Tests*

Em seu trabalho, Luo *et al.* (2014) apresentam os detalhes de um estudo aplicado sobre 51 projetos *open-source* extraídos do repositório SVN da *Apache Software*. Os autores analisaram um total de 201 *commits* que corrigem *Flaky Tests* com o intuito de: identificar e classificar as causas raiz, maneiras de detecção e estratégias de correção de *Flaky Tests* em Testes de Unidade.

Na primeira fase do estudo foi realizada a montagem da base de dados. Para isto, foi realizada a extração do histórico de *commits* de todos os projetos da *Apache Software Foundation*. Os autores justificam a opção por utilizar *commits*, e não relatórios de *bugs* (comumente utilizados em outros estudos), já que através dos *commits* é possível identificar todos os *Flaky Tests fixed*, pois toda correção é refletida no sistema de controle de versão, mas nem sempre é refletido na base de dados de relatórios de *bugs*. A análise aplicada sobre os *commits* garante que não haja perda de dados.

Para identificar todos os *commits* que possivelmente tratam sobre *Flaky Tests*, os autores aplicaram uma filtragem na base de dados obtida, utilizando os termos *"intermit"* e *"flak"* como palavras-chave para sua pesquisa, que retornou um total de 1.129 *commits* candidatos à análise. Cada *commit* candidato passou por uma inspeção manual, realizada por dois autores, que analisaram suas mensagens e, se disponível, relatórios de bugs, com o objetivo de identificar e descartar *commits* falsos positivos, que não tratam a ocorrência de *Flaky Tests*. Os 486 *commits* restantes dessa filtragem foram rotulados como:

1. *Commits about flaky tests*: *commits* que relatam e/ou corrigem *Flaky Tests*.
2. *Commits LDFFT*: *commits* que tentam corrigir *Flaky Tests* distintos.

Para garantir que o estudo não fosse aplicado sobre uma base de dados tendenciosa para projetos com maior número de *commits*, os autores dividiram ainda os 486 *commits* em dois grupos: *Small* (projetos com menos de 6 *commits* LDFFT) e *Large* (projetos com pelo menos 6 *commits* LDFFT). Assim, foram selecionados para a análise final todos os *commits* pertencentes

ao grupo *Small* e um terço dos *commits* pertencentes ao grupo *Large*.

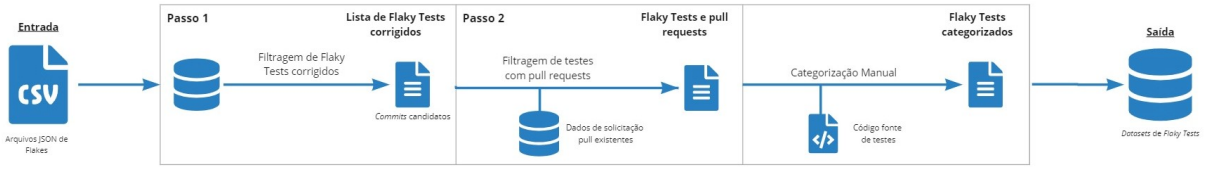
3.2 *A Study on the Lifecycle of Flaky Tests*

Lam *et al.* (2020a) realizaram um estudo sobre o ciclo de vida geral dos *Flaky Tests* em seis projetos de larga escala da Microsoft, revelando o impacto negativo que os *Flaky Tests* têm sobre esses projetos. Ao analisar as correções aplicadas, os autores descobriram que a categoria de *Flaky Tests* mais comum nos projetos analisados é a categoria *Async Wait* (testes que fazem chamadas assíncronas sem esperar o retorno da chamada). Assim, os autores confirmam que categorizações em *Flaky Tests* propostas por estudos anteriores aplicados em projetos *open-source*, também se aplicam para os projetos proprietários da Microsoft estudados.

Após entender o ciclo de vida dos *Flaky Tests* Lam *et al.* (2020a) propõem uma ferramenta para equilibrar a instabilidade do teste e do tempo de execução. Chamada de *Flakiness and Time Balancer(FaTB)*, a ferramenta proposta alivia o impacto negativo dos testes *Async Wait*, identificando as chamadas de método no código de teste que estão relacionadas a tempos limite ou esperas de encadeamento e calculando a frequência com que o *Flaky Test* ocorre.

Os estudos de Lam *et al.* (2020a) foram conduzidos sobre três bases de dados, obtidas diretamente da ferramenta *CloudBuild* (ferramenta para construção de código e execução de testes, que permite o gerenciamento de *Flaky Tests*), através do módulo *Flakes*, que identifica e armazena os dados dos *Flaky Tests* identificados em relatórios. Os dados das bases foram categorizados através de análise manual, como: *All-Fixed* (*Flaky Tests* que já foram corrigidos), *Pull-Requests* (*Flaky Tests* corrigidos que possuem a solicitação de *pull* vinculada ao relatório do bug), e *Categorized* (todos os *Flaky Tests* disponíveis). A Figura 4 fornece uma visão geral de como os autores obtiveram os conjuntos de dados utilizados.

Figura 4 – Visão geral do processo de obtenção dos conjuntos de dados utilizados



Fonte: Lam *et al.* (2020a)

3.3 *An Empirical Analysis of UI-based Flaky Tests*

Em seu trabalho, Romano *et al.* (2021) se propõem a identificar as principais causas e técnicas de correção de *Flaky Tests* em projetos *open-source* de Testes UI do GitHub. Utilizando uma metodologia parecida com a utilizada por Luo *et al.* (2014), foram coletados e analisados 235 *commits* relacionados à ocorrência de *Flaky Tests* em Testes UI, encontradas em 62 projetos de ambientes WEB e Android. Nesse estudo, os autores identificam as causas básicas comuns da instabilidade nos Testes de UI, as estratégias usadas para identificar o comportamento instável e as estratégias de correção usadas para corrigir os *Flaky Tests* em Testes UI.

A coleta dos dados utilizados no estudo foi realizada através da API V3[1], API de pesquisa do GitHub. Inicialmente foram selecionados repositórios de estruturas UI populares, usando as palavras-chave *"react", "angular", "vue", "emberjs", "d3", "svg", "web", "bootstrap"*, identificados assim 7.037 repositórios distintos, totalizando 2.613.420 *commits*. Para filtragem dos *commits* candidatos, os autores seguiram o processo definido por Luo *et al.* (2014), utilizando as palavras-chave *"flaky", "intermit"* adicionando a palavra "test". A filtragem retornou um total de 3.516 *commits* candidatos, que provavelmente relatam e ou/corrigem *Flaky Tests*. Uma segunda fase do processo de filtragem, foi aplicar nos *commits* candidatos uma segunda filtragem, dessa vez utilizando as palavras-chave *"ui", "gui", "visual", "window", "button", "display", "click"* e *"animation"* para identificar *commits* que se referem a *Flaky Tests* de Testes UI. Uma terceira etapa da fase de filtragem é a análise manual dos *commits* candidatos, para remoção de duplicatas e *commits* que não se referem a Testes UI. Esse processo resultou em um total de 254 *commits* para condução do estudo.

Após realizada a coleta de *commits* e relatórios, os *commits* foram inspecionados manualmente. Nos projetos WEB, foram analisados a mensagem do *commit*, o código alterado e os problemas vinculados. Para os relatórios de problemas, foram inspecionados os comentários dos desenvolvedores e os *commits* vinculados. Quando disponíveis, também foram inspecionados os *logs* de execução do Integração Contínua (CI).

[1] https://docs.github.com/en/rest

3.4 *A Survey on How Test Flakiness Affects Developers and What Support They Need To Address It*

Em seu trabalho Gruber e Fraser (2022), aplicaram um *survey* com 335 desenvolvedores e testadores de software em diferentes domínios buscando compreender como os desenvolvedores percebem os *Flaky Tests*, quão prevalentes e o quanto afetam a rotina diária dos desenvolvedores, e o que eles desejam que a academia faça sobre o assunto.

O público alvo do estudo foram desenvolvedores e testadores de software que já vivenciaram uma rotina com presença de *Flaky Tests*. O questionário aplicado na execução do estudo é composto por 12 perguntas, divididas entre perguntas demográficas e perguntas sobre a visão do participante sobre *Flaky Tests*, com opções de resposta em texto livre ou escala Likert.

Para processar as respostas em escala *Likert*, os dados foram transformados em números inteiros e calculada sua média, o que permitiu compactar as respostas para cada afirmação em um único número. Para processar as respostas em texto livre, foi realizada uma análise de dados quantitativa utilizando um conjunto de códigos. A codificação foi realizada por dois pesquisadores, para comparação das mesmas.

Como resultado da pesquisa, Gruber e Fraser (2022) concluíram que os desenvolvedores vêem os *Flaky Tests* como um problema comum e sério e que estão mais preocupados com a perda de confiança nos resultados dos testes do que com os custos computacionais da reexecução do teste instável. Por último, os desenvolvedores externaram seu desejo por *plugins* para identificar trechos de códigos irregulares diretamente em suas IDEs, também desejam mais treinamentos e informações sobre *Flaky Tests*.

3.5 Análise comparativa

Assim como ocorre em Luo *et al.* (2014) e Lam *et al.* (2020a), este trabalho também aborda como tema geral as principais causas, estratégias de identificação e correção de *Flaky Tests*. No entanto, diferente do que ocorre nos trabalhos citados, que se dedicam a estudar *Flaky Tests* em Testes de Unidade, este trabalho propõe um estudo sobre a ocorrência de *Flaky Tests* em Testes UI. Lam *et al.* (2020a) conduziram seu estudo sobre o ciclo de vida dos *Flaky Tests* analisando *commits* em projetos proprietários das Microsoft. Desta forma, este trabalho diverge dos acima citado porque estuda a ocorrência de *Flaky Tests* em Testes UI de projetos *open-source* do GitHub.

Por fim, assim como feito por Romano *et al.* (2021), este trabalho tem como objetivo analisar a ocorrência de *Flaky Tests* em Testes UI de projetos *open-source* do Github através da análise de *commits* de correção.

Enquantos os trabalhos citados e o trabalho proposto buscam entender os *Flaky Tests* em nível de código, Gruber e Fraser (2022) buscam entender os impactos que estes causam no processo de desenvolvimento através da visão dos desenvolvedores, buscando informações sobre como eles lidam com o problema e quais informações esses desenvolvedores buscam ter sobre o assunto. O estudo de Gruber e Fraser (2022) fornece um *backgroud* sobre que informações devemos buscar no trabalho aqui proposto. O Quadro 4 compara o presente trabalho com os trabalhos relacionados descritos anteriormente.

Quadro 4 – Análise comparativa entre os trabalhos relacionados e este trabalho

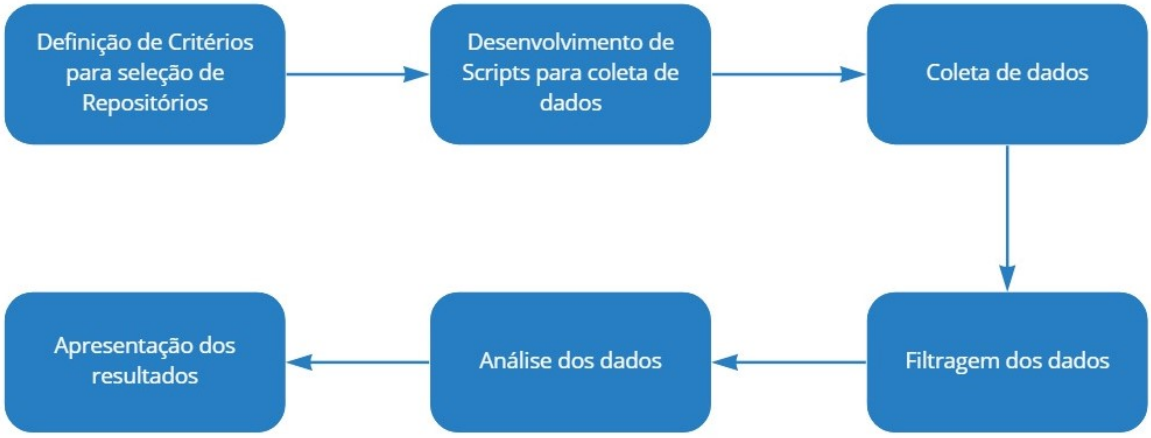
	Luo *et al.* (2014)	**Lam *et al.* (2020a)**	**Romano *et al.* (2021)**	**Gruber e Fraser (2022)**	Trabalho proposto
Identificar as principais causas, estratégias de identificação e correção de *Flaky Tests*	X	X	X		X
Utilizar projetos *open-source*	X		X		X
Analisar projetos de automação baseados em UI			X	X	X

Fonte: Elaborado pela autora.

4 METODOLOGIA

Este capítulo apresenta as etapas seguidas para alcançar os objetivos deste trabalho. A Figura 5 resume as etapas e as próximas Seções descrevem cada uma delas.

Figura 5 – Procedimentos metodológicos



Fonte: elaborado pelo autor

4.1 Definição de critérios para seleção de repositórios

Os dados utilizados para conduzir esta pesquisa foram coletados de repositórios hospedados no GitHub[1]. O GitHub foi escolhido como fonte de pesquisa devido a sua alta popularidade entre os desenvolvedores, conter um grande volume de dados e por ser referência na mineração de repositórios *open-source* (VIDONI, 2022).

Os repositórios selecionados para a condução desta pesquisa, foram selecionados de acordo com as diretrizes propostas por Vidoni (2022) (mais detalhes em 2.4. Assim, os repositórios selecionados obedecem aos seguintes critérios:

- Ser um repositório de Testes UI ou possuir módulos de Testes UI que utilizem os *framworks* Selenium WebDriver[2], Robot[3] ou Cypress[4]. Assim garantimos estar trabalhando com repositórios que possuam código de automação de Testes UI;
- Possuir no mínimo 100 *commits*. Isso reduz o número de repositórios candidatos, pois com isso descartamos um grande número de projetos pessoais e/ou que não estejam voltados ao

[1] https://github.com/
[2] https://www.selenium.dev/documentation/webdriver/
[3] https://robotframework.org/
[4] https://www.cypress.io/

desenvolvimento.

4.2 Desenvolvimento de *scripts* para coleta de dados

Para coleta dos dados foram utilizadas duas ferramentas: o GHTorrent e um *script* Python que consome dados da API V3 do GitHub para extração dos *commits* candidatos.

4.2.1 GHTorrent

Neste trabalho, a ferramenta GHTorrent[5] foi utilizada para identificar projetos candidatos - que provavelmente contenham Testes UI - . O GHTorrent trata-se de espelho *offline* consultável dos dados da API do GitHub, onde todos os dados recuperados são ofertados como serviço (GODFREY MICHAEL W. E WHITEHEAD,). Os mais de 15 TB de dados do GHTorrent estão disponíveis para download dos bancos de dados correspondentes como também podem ser acessados online via Google BigQuery[6].

Para este trabalho utilizamos os dados do GHTorrent via Google BigQuery para recuperar os projetos que provavelmente possuam Testes de UI utilizando consultas simples SQL (*Standard Query Language*), linguagem padrão para manipulação de registros em bancos de dados relacionais. Para isso foi elaborada uma *query* simples - disponível no repositório githubDataHandler[7] - que realiza os seguintes passos:

- Filtra dentre os repositórios disponíveis, aqueles que possuem mais de 100 *commits*;
- Seleciona do resultado do passo anterior, todos os projetos que contenham em sua descrição referência aos termos definidos na subseção 4.3.1.

Os resultados dessa *query* foram exportado em arquivos JSON - que podem ser consultados no repositório githubDataHandler [8] - contendo as seguintes informações:

Quadro 5 – Repositórios coletados via GHTorrent

name	nome do repositório
url	referência para api do repositório
description	descrição do repositório disponível no GitHub
NumDeCommits	quantidade de *commits* do repositório

Fonte: Elaborado pelo autor

[5] https://ghtorrent.org/
[6] https://ghtorrent.org/gcloud.html
[7] https://github.com/EriccaSousa/githubDataHandler/blob/main/sql/findProjects.sql
[8] https://github.com/EriccaSousa/githubDataHandler/tree/main/resource/GHTorrent/Projetos

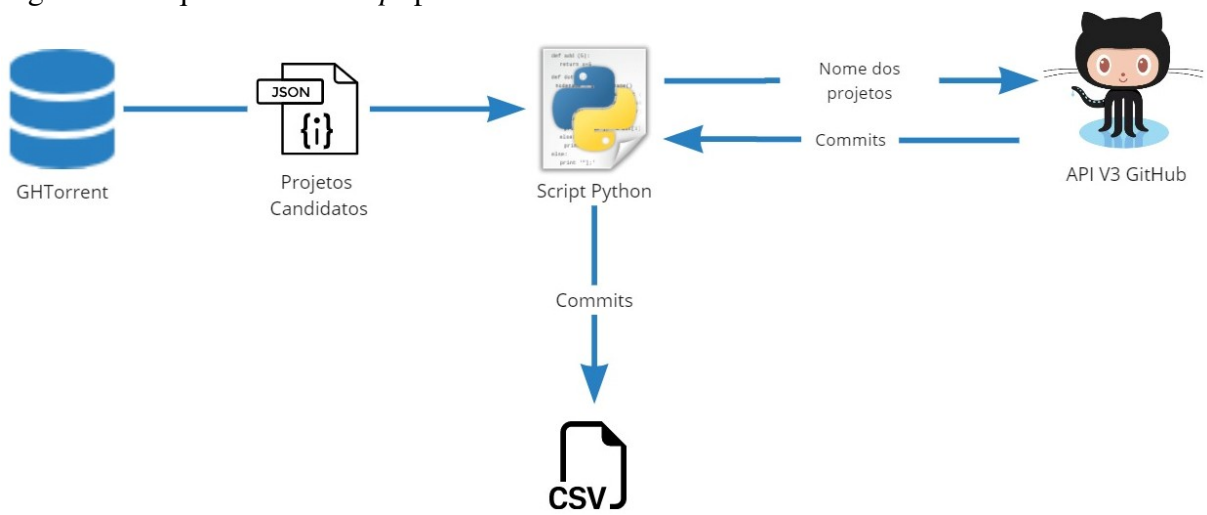
4.2.2 API V3 GitHub

O GitHub fornece a API v3[9], uma API REST que permite recuperar *commits* de repositórios dos projetos e os eventos gerados por meio de ações do usuário nos recursos do projeto (GODFREY MICHAEL W. E WHITEHEAD,). Sendo possível acessar e manipular os dados do GitHub através de qualquer linguagem de programação que forneça suporte a esse tipo de ação.

4.2.3 Script para coleta de commits

Para coleta dos *commits* utilizados no estudo, foi necessário a criação de um *script* - disponível no repositório gitHubDataHandler[10] - que realiza comunicação com a API v3 do GitHub para extrair os dados necessários, utilizando como entrada os arquivos JSON oriundos do GHTorrent. A Figura 6 representa a arquitetura do *script* utilizado.

Figura 6 – Arquitetura do *script* para coleta de *commits*



Fonte: elaborado pelo autor.

Os arquivos JSON oriundos do GHTorrent foram utilizados como entrada para o *script*, que identifica nos arquivos o nome dos projetos e busca via API v3 todos os seus *commits*. Para cada repositório identificado é realizada a extração dos seus *commits*. Como saída, todos os *commits* selecionados foram salvos em arquivos CSV, que podem ser acessados no repositório githubDataHandler. O Quadro 6 descreve as informações armazenadas nos arquivos CSV.

[9] https://docs.github.com/pt/rest
[10] https://github.com/EriccaSousa/githubDataHandler

Quadro 6 – Dados coletados pelo *script*

Nome da Coluna	**Conteúdo**
repo_name	nome do repositório
repo_url	link para o repositório
url_commit	link para o *commit*
message	mensagem de confirmação do *commit*

Fonte: Elaborado pela autora

4.3 Coleta de dados

A coleta dos dados foi dividida em 3 fases: Seleção dos repositórios, Extração dos *Commits* candidatos e Filtragem dos *commits* que fizeram parte do estudo.

4.3.1 Seleção dos repositórios

Nesta etapa foi realizada a seleção dos repositórios utilizados no estudo. Os repositórios foram selecionados de acordo com os critérios definidos na Seção 4.1, utilizando o GHTorrent e o *script* GitHub-DataHandler (mais detalhes nas Seções 4.2.1 e 4.2.3).

Inicialmente, foram selecionados os projetos presentes na base de dados do GHTorrent, utilizando a query definida na Seção 4.2.1, utilizando como entrada os valores: "Cypress", "Robot", "Selenium Webdriver", e termos populares de estruturas front-end: *“react”*, *“angular”*, *“vue”*, *“emberjs”*, *“d3”*, *“svg”*, *“web”*, *“bootstrap”*, que foram utilizados por Romano *et al.* (2021) e Luo *et al.* (2014) em suas pesquisas. Para garantir que fossem selcionados o maior número de repositórios, foi utilizada uma variação na escrita os termos. Por exemplo, para capturar os repositórios relacionados a Robot foi utilizada as seguintes *strings* na *query* SQL: "Robot"e "robot"; Para Selenium: "Selenium Webdriver", "selenium webdriver", "Selenium-Webdriver"e "selenium-webdriver".

Ao todo foram selecionados 2.263 e repositórios no GHTorrent, descritos em detalhes no Quadro 7. Todos os repositórios coletados pode ser encontrados no repositório Github DataHandler[11].

4.3.2 Extração e filtragem dos commits

Primeiramente, foi realizada a extração das mensagens de confirmação de todos os *commits* dos repositórios selecionados e armazenadas em um arquivo csv. Para identificar

[11] https://github.com/EriccaSousa/githubDataHandler/tree/main/resource/GHTorrent/Projetos

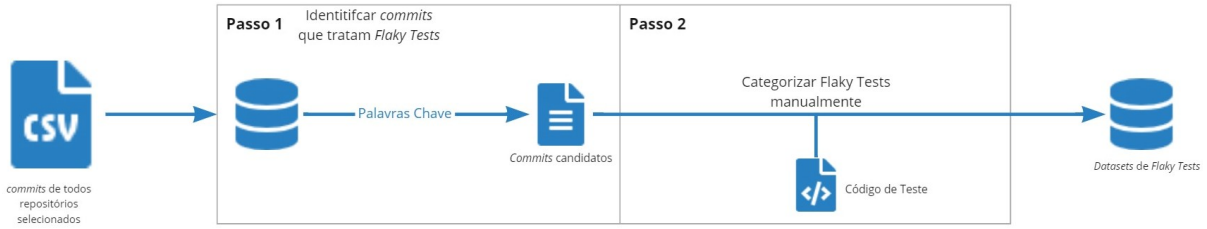
Quadro 7 – Descrição dos repositórios selecionados via GHTorrent

String de Pesquisa	Quantidade de Repositórios
Cypress	9
Robot	1.915
Selenium Webdriver	339
Estruturas front-end	490

Fonte: Elaborado pelo autor

os *commits* que provavelmente corrigem *Flaky Tests*, foi utilizado um processo parecido aos utilizados por Luo *et al.* (2014) e Romano *et al.* (2021). A Figura 7 ilustra o processo de filtragem dos *commits* para análise.

Figura 7 – Processo de escolha dos *commits* para análise



Fonte: elaborado pelo autor.

Ao todo foram coletados 34.572, *commits* dos 2.263 repositórios candidatos. O primeiro passo para filtragem foi identificar dentre esses 34.572 *commits*, aqueles que provavelmente tratem sobre *Flaky Tests* (identificação e/ou correção). Para isto, foi aplicada uma pesquisa nas mensagens de confirmação dos *commits* utilizando as palavras-chave propostas por Romano *et al.* (2021): *"flak"*, *"intermit"*. Este passo reduziu o número de *commits* candidatos para 1.146 *commits*.

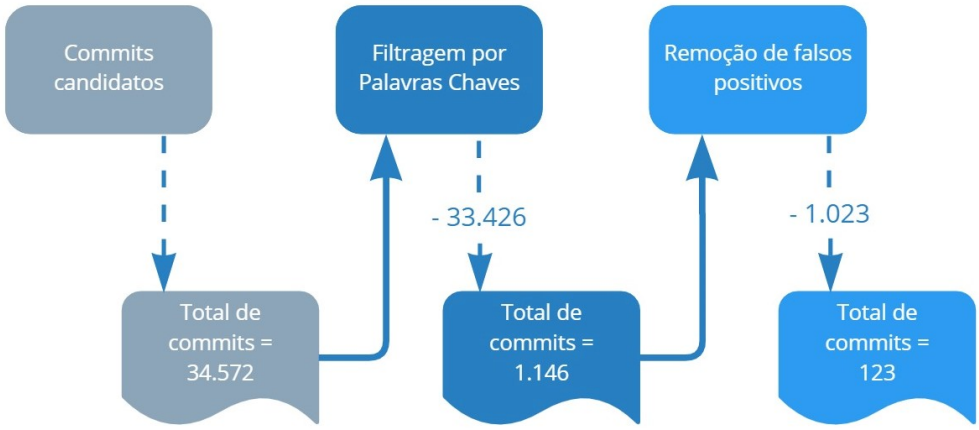
No segundo passo, foi realizada uma análise manual das mensagens de confirmação dos *commits* selecionados no passo anterior, buscando identificar *commits* falsos positivos (i.e., que possuem pelo menos uma das palavras-chave mas não tratam de *Flaky Tests* no código de Teste UI). Ao final desse processo, obteve-se um *dataset* com 123 *commits* para realização do estudo. A Figura 8 detalha os números de *commits* a cada etapa da filtragem dos dados.

4.4 Análise dos dados

Cada *commit* selecionado passou por uma análise manual, para identificar informações relevantes para nossas questões de pesquisa. Para cada *commit* coletado, foram inspecionados os seguintes pontos:

– **Mensagem dos *commits*:** as mensagens de confirmação oferecem detalhes sobre como

Figura 8 – Processo de escolha dos *commits* para análise



Fonte: elaborado pelo autor.

Flaky Test foi corrigido, causas e possíveis detalhes de identificação.

- **Relatórios dos problemas:** os comentários dos desenvolvedores nos *commits* fornecem informações mais detalhadas sobre como o *Flaky Test* foi inicialmente identificado, reproduzido e implicações de possíveis correções.
- **Código alterado:** ao examinar os trechos de código alterado em cada *commit*, é possível identificar padrões de fixação e, posteriormente, agrupá-los em categorias para geração do catálogo de causas e correções de *Flaky Tests* em Testes UI.

Ao final desse processo, foram obtidos insumos para responder às nossas questões de pesquisa. Com os dados colhidos até aqui, foi possível identificar e mapear as causas recorrentes de *Flaky Tests* na base de dados, bem como mapear as estratégias de correções mais utilizadas.

4.4.1 Categorização

Após a inspeção manual dos *commits* relacionados a testes de IU instáveis, foram identificadas 8 categorias, baseadas nas categorias propostas por Luo *et al.* (2014) e Romano *et al.* (2021) (Seção 2.3.1) nas quais as causas raiz da instabilidade nesses testes podem se enquadrar: (1) Condição de Corrida, (2) Problemas de Lógica, (3) Dependência de Teste, (4) Falha da Ferramenta, (5) Problemas no Seletor DOM, (6) Aleatoriedade, (7) Interferência Externa e (8) Problemas no tempo de animação. Cada categoria de Causa está explicada abaixo:

- 1) Condição de Corrida: Ocorre quando existe a tentativa de manipular dados ou outros recursos antes de serem totalmente carregados na página
- 2) Problemas de Lógica: Ocorre quando existe erros na lógica implementada no caso de teste, como por exemplo execução de uma sequência errada dos passos necessários para

finalizar o teste, manipulação errada de dados de testes, etc.

- 3) Dependência de Teste: Ocorrem quando casos de testes não está isolados na suíte de testes e seus resultados são influenciados por outros casos de testes.
- 4) Falha da Ferramenta: Falhas existentes no próprio framwork utilizado para escrita dos testes ou navegador utilizado para executar os testes. Normalmente essas falhas estão além do poder de solução dos desenvolvedores dos testes já que estão atrelados as ferramentas utilizadas.
- 5) Problemas no Seletor DOM: problemas de lógica presentes nas estruturas utilizadas para localizar o elementos, por exemplo a utilização de um *xpath* que retorna dois elementos quando deveria retornar apenas um, fazendo com que o teste interaja com o elemento incorreto
- 6) Aleatoriedade: ocorre quando o caso de teste trabalha com valores gerados de maneira aleatória e apresenta falha para determinados valores gerados.
- 7) Interferência Externa: corresponde a qualquer tipo de interferência externa no SUT que esteja fora do controle da execução do teste. Por exemplo, os dados do SUT sofrem alteração de, oriundas de fontes desconhecidas, durante a execução do caso de teste, fazendo com que existam inconsistências entre os dados presentes no SUT e os dados esperados pelo caso de teste, fazendo com que o teste falhe.
- 8) Problemas no tempo de animação: ocorre em testes que dependem da execução de uma animação para realizar o teste ou que realizam testes diretamente sobr a animação, já que os mesmos são sensíveis às diferenças de tempo no ambiente de execução.

Também foram identificadas 7 categorias de estratégias de correção de *Flaky Tests*, baseadas nas categorias propostas por Luo *et al.* (2014) e Romano *et al.* (2021) (Seção 2.3.1): (1) Adição de Espera, (2) Correção de Lógica, (3) Teste Ignorado, (4) Aumento no tempo de Espera, (5) Alterar forma de acesso ao Elemento, (6) Teste removido da suíte, (7) Alterar configuração do Ambiente. Cada categoria de Correção está explicada abaixo:

- 1) Adição de Espera: Ocorre quando é adicionado um atraso entre as ações qu envolvem buscar ou interagir com elementos da interface.
- 2) Correção de Lógica: Ocorre quando são realizadas alterações na lógica do testes.
- 3) Teste Ignorado: Ocorre quando testes testes são marcados como aleatórios e consequentemente são ignorados automaticamente durante a execução dos testes.
- 4) Aumento no Tempo de Espera: Ocorre quando é realizado um aumento no tempo de

esperas já existentes.

– 5) Alterar forma de acesso ao Elemento: Ocorre quando existe alteração no tipo de mecanismo utilizado para acessar o elemento.
– 6) Teste removido da suíte: Ocorre quando o teste é totalmente deletado ou comentado.
– 7) Alterar configuração do Ambiente: Ocorre quando é necessário alterar consfigurações do ambiente de execução.

4.5 Resumo da base de dados

Ao todo foram analisados 123 *commits* na execução deste estudo, consistindo em um conjunto diversificado de amostras de Teste UI instáveis para sistemas WEB. As linguagens dos testes de UI instáveis analisados são Java (61,8%), TypeScript (13,8%), JavaScript (12,2%), Ruby (3,3%) e outras (1,6%). O dataset[12] pode ser acessado no repositório do GitHub. O Quadro 8 mostra as informações dos projetos utilizados em nossa análise.

Quadro 8 – Projetos de onde foram extraídos os dados para análise

Projetos	**Total de Flaky Tests**	**Total de Commits**	**LOC**
selenium	73	28 461	803 475
influxdb	11	35 142	384 589
acceptance-test-harness	10	5 307	33 016
selenide	4	3 908	64 004
cypress-example-kitchensink	3	924	42 193
metabase	2	23 148	687 463
platform	2	25 101	1 209 905
Selenium2	2	2 738	770 393
cypress-documentation	1	6 238	73 282
e2e-testsuite-platform	1	272	6 522
tor-browser-selenium	1	330	1 455
php-webdriver	1	1 065	12 263
server	1	529	16 199
ionic-framework	1	11 685	364 559
se34euca	1	545	21 912
next.js	1	11 690	462 341
angular	1	24 832	611 781
gatsby	1	20 358	859 260
react-styleguidist	1	2 162	179 734
react-cosmos	1	1 949	31 864
playground	1	516	12 322
opencrvs-farajaland	1	1 925	34 967
node.js	1	8 703	76 740
SwagPaymentPayPal	1	951	69 618

Fonte: Elaborado pelo autor

[12] https://github.com/EriccaSousa/githubDataHandler/blob/main/resource/Flakys_Planilha_Oficial.csv

5 RESULTADOS

5.1 RQ1: Quais as causas mais recorrentes de *Flaky Tests* em Testes UI?

Para responder nossa RQ1, foram avaliados durante a análise dos *Flak Tests* coletados o código alterado e os problemas relacionados aos *commits*, em busca de informações que pudessem indicar a causa da instabilidade do teste. Dos 123 *commits* analisados, 21 não continham informações suficientes para definir sua causa. Os resultados da categorização estão resumidos no Quadro 9.

Quadro 9 – Resumo das categorias de causa de *Flaky Tests*

Causa	**Total**
Condição de Corrida	74
Problemas de Lógica	15
Dependência de Teste	4
Falha da Ferramenta	3
Problemas no Seletor DOM	2
Aleatoriedade	2
Interferência externa	1
Problema no tempo de animação	1

Fonte: Elaborado pelo autor

5.1.1 Condição de Corrida

Sistemas web possuem uma natureza assíncrona, pois estão estritamente ligados às ações dos usuários, tempo de resposta da API e a velocidade da conexão. Por esse motivo, casos de testes automatizados estão expostos ao que chamamos de Condição de Corrida, tal condição ocorre quando existe a tentativa de manipular dados ou outros recursos antes de serem totalmente carregados na página. A Condição de Corrida aparece como a principal causa de *Flaky Tests* em nossa base de dados, estando presente em 60% dos *commits* analisados.

Nos *commits* analisados, a Condição de Corrida normalmente foi causada pelo agendamento inadequado da busca e execução de ações sobre os elementos da interface, causando problemas com a tentativa de interagir com elementos que não foram carregados completamente. Essa ordem inadequada de eventos resulta em uma ação inválida e faz com que uma exceção seja lançada.

Um exemplo desse caso pode ser visto na Figura 9, a Condição de Corrida ocorre na linha 320, onde é feita a busca pelo elemento de id “prompt”, que provavelmente trata-se de

um botão, e em seguida realizada uma interação de *click* no elemento. Porém, nada garante que botão estará carregado na interface quando a linha 320 for executada. Imagine que a renderização do botão seja dependente de um processamento da API, que por vezes demora 10 segundos para ser concluído, um tempo de renderização fora do comum. Nos casos em que o botão demora mais que o comum a ser renderizado, o teste , que não possui consciência desse tempo de espera, realizará a busca pelo botão e não será encontrado, gerando assim uma *exception* que deixará o caso de teste em questão com *status* de erro.

Figura 9 – Exemplo de teste com condição de corrida

```
317       public void testPromptShouldUseDefaultValueIfNoKeysSent() {
318         driver.get(promptPage("This is a default value"));
319
320   -     driver.findElement(By.id("prompt")).click();
321         Alert alert = wait.until(alertIsPresent());
322         alert.accept();
```

Fonte: *Print* retirado do projeto selenium no GitHub.

5.1.2 Problemas de Lógica

Dos *commits* analisados, 12% de seus *Flaky Tests* foram ocasionados por problemas na lógica dos *scripts* de teste. Os *flaky Tests* ocorreram por motivos como: realizar afirmações incorretas durante o teste, carregado recursos em uma ordem incorreta, dentre outros fatores.

No exemplo da Figura 10, nas linhas 187 e 188 são realizados *clicks* desnecessários na seleção do elemento, a execução desses *clicks* causam comportamentos inesperados no sistema, que por vezes leva o teste a falha, evidenciando a presença de um *Flaky Test*. Já na Figura 11, o *Flaky Test* é causado pelo nome do arquivo especificado como uma *string* fixa, quando na verdade o mesmo pode conter variações.

Figura 10 – Exemplo de teste com problemas de lógica

```
186       public void selectDropdownMenuAlt(Class type) {
187   -       click();
188   -       findCaption(type,findDropDownMenuItemBySelector).click();
189           elasticSleep(1000);
190       }
```

Fonte: Print retirado do projeto acceptance-test-harness no GitHub.

Figura 11 – Exemplo de teste com problemas de lógica

```
153      @Test
154      void downloadsPotentiallyHarmfulWindowsFiles() throws IOException {
155  -       File downloadedFile = $(byText("Download EXE
         file")).download(withNameMatching("tiny\\.exe.*"));
156
157          assertThat(downloadedFile.getName()).startsWith("tiny.exe");
158          assertThat(Files.size(downloadedFile.toPath())).isEqualTo(43);
```

Fonte: Print retirado do projeto selenide no GitHub.

5.1.3 Dependência de Teste

Dos dados analisados, 3% dos *Flaky Tests* foram causados por Dependência de Teste ocorrem quando casos de testes são influenciados por testes outros casos de testes. Essa interferência pode ser feita por meio de armazenamentos de dados compartilhados que não são bem limpos entre as execuções de teste. Como resultado, o armazenamento de dados pode conter valores de testes anteriores e produzir valores incorretos como resultado.

5.2 RQ2: Quais as estratégias de correção mais utilizadas no tratamento de *Flaky Tests*?

Para responder nossa RQ2, também foram avaliados durante a análise dos *Flak Tests* coletados o código alterado e os problemas relacionados aos *commits*, em busca de informações que pudessem indicar a causa da instabilidade do teste. Os resultados da categorização estão resumidos no Quadro 10.

Quadro 10 – Resumo das categorias de Estratégias de Correção de *Flak Tests*

Causa	**Total**
Adição de Espera	66
Correção de Lógica	27
Teste Ignorado	19
Aumento no tempo de espera	6
Alterar forma de acesso ao elemento	2
Teste removido da suíte	1
Alterar configuração de ambiente	1

Fonte: Elaborado pelo autor

5.2.1 Adição de Espera

Para reduzir a chance de encontrar um comportamento instável ocasionado por uma Condição de Corrida, 53% dos testes adicionaram um atraso entre as ações que envolvem buscar ou interagir com elementos da interface, essas esperas são de dois tipos: esperas implícitas e explícitas.

Ao utilizar uma espera implícita dizemos ao teste para sondar o DOM por um determinado período de tempo ao tentar encontrar qualquer elemento não imediatamente disponível, exemplificado na Figura 12, onde podemos ver que na linha 73 foi adicionado um *Thread.sleep* antes de capturar informações para realização de uma asserção.

Figura 12 – Exemplo de adição de espera

```
67 +   public void
       testShouldFocusOnTheReplacementWhenAFrameFollowsALinkToA_TopTargettedPage() throws
       Exception {
68         driver.get(framesetPage);
69
70         driver.findElement(By.linkText("top")).click();
71
72 +       // TODO(simon): Avoid going too fast when native events are there.
73 +       Thread.sleep(1000);
74 +
75         assertThat(driver.getTitle(), equalTo("XHTML Test Page"));
76         assertThat(driver.findElement(By.xpath("/html/head/title")).getText(),
77                    equalTo("XHTML Test Page"));
```

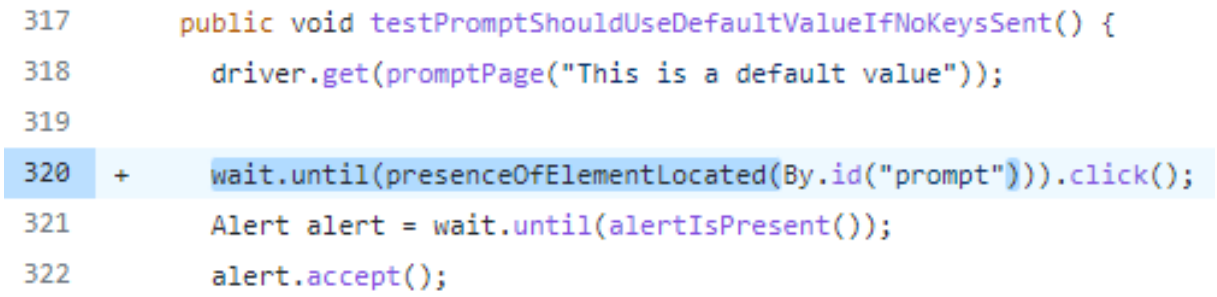
Fonte: Print retirado do projeto selenium no GitHub.

Já as esperas explícitas permitem que você expecifique uma condição para o teste, fazendo com que o mesmo pause sua execução até que a condição seja atendida, exemplicado na Figura 13, onde foi adicionada uma espera com a condição de apenas realizar o *click* quando o elemento de id "prompt"estiver presente. Essa estratégia evita que o resto do script de teste seja executado até que o atraso termine, dando à chamada assíncrona tempo adicional para ser concluída antes de prosseguir.

5.2.2 Correção de Lógica

Alguns testes apresentaram comportamentos aleatórios devido a erros na lógica de implementação, para contornar esse problema, em 21% dos *commits* analisados, foram realizadas

Figura 13 – Exemplo de adição de espera

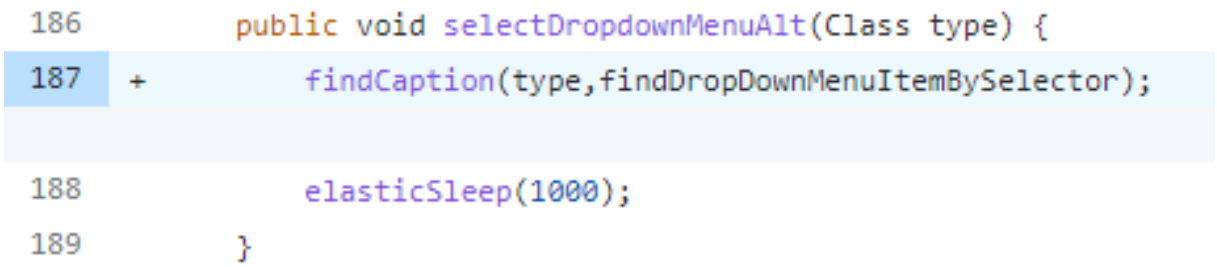


```
317       public void testPromptShouldUseDefaultValueIfNoKeysSent() {
318         driver.get(promptPage("This is a default value"));
319
320   +     wait.until(presenceOfElementLocated(By.id("prompt"))).click();
321         Alert alert = wait.until(alertIsPresent());
322         alert.accept();
```

Fonte: Print retirado do projeto selenoid-containers-tests no GitHub.

alterações diversas na lógica dos testes, para melhorar o propósito pretendido do teste, removendo o *Flaky Test*. Essas alterações vão desde a correção da forma de acesso aos elementos, coreção da ordem de execução dos passos da interação com a interface, gerção de dados, dentre outros.

Figura 14 – Exemplo de correção de lógica

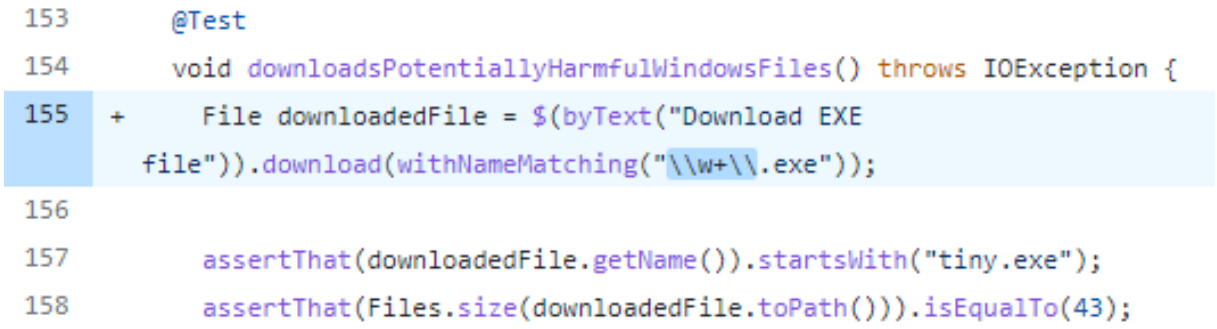


```
186         public void selectDropdownMenuAlt(Class type) {
187   +           findCaption(type,findDropDownMenuItemBySelector);

188             elasticSleep(1000);
189         }
```

Fonte: Print retirado do projeto acceptance-test-harness no GitHub.

Figura 15 – Exemplo de correção de lógica



```
153       @Test
154       void downloadsPotentiallyHarmfulWindowsFiles() throws IOException {
155   +     File downloadedFile = $(byText("Download EXE
          file")).download(withNameMatching("\\w+\\.exe"));
156
157         assertThat(downloadedFile.getName()).startsWith("tiny.exe");
158         assertThat(Files.size(downloadedFile.toPath())).isEqualTo(43);
```

Fonte: Print retirado do projeto selenide no GitHub.

5.2.3 Teste ignorado

Para lidar com a ocorrência de *Flaky Tests*, 15% dos *commits* analisados, removeram esses de suas suítes de execução, marcando os mesmos como testes instáveis. A remoção do teste da suíte de execução reduz a cobertura do código.

Alguns testes não são totalmente removidos do conjunto de testes. Em vez disso, eles são marcados como fragmentados, o que significa que, se o teste falhar, todo o conjunto de testes não falhará. Um exemplo dessa estratégia de correção pode ser visualizado na Figura 16, observe que na coluna da direita a tag "@CategorySmokeTest.clas", que indica que o teste pertence a classe de execução SmokeTest, foi comentada, impedindo assim que o caso de teste seja executado junto a suíte SmokeTest.

Figura 16 – Exemplo de teste ignorado na suíte de execução

```
62          @Test
63  +       //TODO Re enable category once JENKINS-49524 is done
64  +       //@Category(SmokeTest.class)
65          public void login_and_logout() {
66
67              User user =
        realm.signup().fullname(FULL_NAME).email(EMAIL).password(PWD).signup(NAME);
```

Fonte: Print retirado do projeto acceptance-test-harness no GitHub.

5.3 RQ3: Quais as estratégias de correção mais utilizadas para cada causa identificada?

Por meio da análise dos dados, é possível identificar algumas relações entre as Causas básicas de *Flaky Tests* e como estes foram corrigidos. Esses relacionamentos é apresentado no Quadro 11.

Para estabelecer essa relação entre Causa e Correção utilizamos uma contagem simples manual de quantas vezes cada estratégia de correção foi utilizada para solucinar cada ocorrência de *Flaky Test*, com isso foi possível identificar qual estratégia de correção mais utilizada para cada Causa identificada.

Ao analisar o Quadro 11 é possível verificar que os *Flaky Tests* causados por Interferência Externa, Aleatoriedade. Problemas de Lógica e Tamanho, Tamanho da Tela e Dependencia de Teste foram corrigidos em 100% de sua ocorrência por operações de Correção de Lógica. Já 89% dos *Flaky Tests* causados por Condição de Corrida foram corrigidos através da Adição de Espera e 8% por Aumento no tempo de espera já existentes; *Flaky Tests* causados por Problemas no seletor DOM foram corrigidos através da alteração na forma de acesso ao elemento e correção de lógica, apresentando número de 50% cada.

Quadro 11 – Relação entre as causas e estratégias de correção

Causa	**Nº ocorrências**	**Estratégia de correção**	**(%)**
Condição de Corrida	74	Adição de Espera Aumento no tempo de espera Teste removido Alterar forma de acesso ao elemento	89% 8% 1% 1%
Problema de Lógica	15	Correção de lógica	100%
Dependencia de Teste	4	Correção de Lógica	100%
Falha da ferramenta	3	Teste Ignorado	100%
Problema no seletor DOM	2	Alterar forma de acesso ao elemento Correção de Lógica	50% 50%
Aleatoriedade	2	Correção de lógica	100%
Interferência Externa	1	Correção de lógica	100%
Problema de Configuração	1	Alterar cnfiguração do ambiente	100%

Fonte: Elaborado pelo autor

5.4 Ameaças à validade

A classificação de ameaças à validade descrita por Wohlin *et al.* (2012) foi utilizada para discutir as ameaças deste trabalho. Esta classificação define quatro tipos de ameaças de validade, sendo elas, ameaças de conclusão, internas, de construção e de validade externa.

Ameaças à validade de construção: Este tipo de validade diz respeito à generalização do resultado para o conceito ou teoria por trás da execução do estudo. Com o objetivo de minimizar ameaças dessa natureza, foi utilizado sinônimos e variações de escrita para as às palavras-chaves utilizadas na seleção dos repositórios fonte de dados para esse estudo.

Ameaças de validade interna: estão relacionadas a uma possível conclusão erradas sobre as categorias de causa e correção aqui apresentadas (WOHLIN et al., 2000). Decisões subjetivas podem ocorrer durante a seleção de repositórios e análise dos dados extraídos, uma vez que a análise dos dados foi realizada por apenas uma pessoa. A fim de minimizar erros de seleção e análise, o processo de seleção foi realizado de forma iterativa de forma que quando ocorreu dúvida na aplicação de algum critério, o estudo não foi eliminado e passou para a próxima fase, a análise foi realizada com máxima atenção e cada *commit* foi revisado no mínimo três vezes para garantir plena confiança na categorização proposta.

Ameaças à validade externa: está relacionada a validade dos repositórios selecionados para o estudo. Esta ameaça foi mitigada devido a utilização do critérios para a seleção dos

repositórios, garantindo a inclusão no estudo apenas de repositórios relacionados a projetos reais, excluindo repositórios dedicados a estudos pessoais ou de outra característica que não forneça dados validos ao estudo realizado.

Ameaças de conclusão: Em relação ao tamanho reduzido do dataset formado, a generalização dos resultados obtidos neste estudo se restringe aos projetos open-source que possuem Testes automatizados baseados em interface, que utilizem os frameworks Sellenium-Webdriver, Cypress e Robot, que possuem características semelhantes aos que foram selecionados para este trabalho. As implicações númericas apresentadas são limitadas ao conjunto de dados coletados.

6 CONCLUSÕES E TRABALHOS FUTUROS

6.1 Resultados encontrados

Neste trabalho propomos inestigar as principais causas e estratégias de correção de *Flaky Tests* em testes automatizados de UI de 26 projetos *open-source* do GitHub. Foram analisados 123 *commits* relacionados à *Flaky Tests*. Os resultados obtidos apontam que Condição de Corrida (60%), Problemas de Lógica (12%) e Dependência de Teste (3%) são as principais causas dos *Flaky Tests*. Acompanhados de Problemas no Seletor DOM, Aleatoriedade, Interferência Externa e Problema no Tempo de Animação, que juntos representam aproximadamente 7% dos *Flaky Tests* analisados.

Ao analisar em detalhes as estratégias de correção aplicadas nos *commits* de fixação dos *Flaky Tests* analisados, observamos que Adição de Espera (53%), Correção de Lógica (21%) e Teste Ignorado (15%) são as principais estratégias de correção dos *Flaky Tests*. Acompanhados por Aumento no tempo de Espera, Alterar Forma de Acesso ao Elemento, Deletar Teste e Alterar Configuração do Ambientes, que juntos correspondem a 8% dos *Flaky Tests* analisados.

Apontamos ainda, relações entre as Causas de *Flaky Tests* e quais Estratégias de Correção mais foram utilizadas em cada causa. Os resultados apresentados podem apoiar as relações entre Causas e a Estratégias de correção durante o tratamento de *Flaky Tests* em projetos reais. Se a causa raiz de um teste de IU instável for conhecida, os relacionamentos do Quadro 11 podem ser utilizados para selecionar uma estratégia de correção apropriada.

Por fim, este trabalho oferece para estudos futuros uma base de dados contendo amostras de 123 *commits* relacionados a *Flaky Tests* em Testes UI. Embora o conjunto de dados não seja exaustivo, acreditamos que os resultados podem fornecer uma base para trabalhos futuros.

6.2 Trabalhos Futuros

Como trabalhos futuros, propõe-se: (i) replicar este estudo em projetos industriais, para comparar os resultados e garantir uma maior generalização; (ii) realizar uma comparação entre as ferramentas existentes para identificação de *Flaky Tests*, afim de analisar sua acurácia e capacidade de utilização surante o processo de desenvolvimento e (iii) desenvolver uma ferramenta de apoio a revisão de código, para ser utilizada em plataformas de gerenciamento de

versão, que utilize as principais categorias de Causa de *Flaky Tests* apresentadas neste trabalho e em outros trabalhos disponíveis na literatura, para identificar nos *commits* trechos de código que possam vir a causar *Flaky Tests* e indicar estratégias de correção que possam ser aplicados aos mesmos.

REFERÊNCIAS

BARROS, D.; HORITA, F.; WIESE, I.; SILVA, K. A mining software repository extended cookbook: lessons learned from a literature review. In: **Brazilian Symposium on Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2021. p. 1–10. ISBN 9781450390613. Disponível em: https://doi.org/10.1145/3474624.3474627. Acesso em: 14 jul. 2008.

COHN, M. **Succeeding with agile**: software development using scrum. 1st. ed. [*S. l.*]: Addison-Wesley Professional, 2009. ISBN 0321579364.

ECK, M.; PALOMBA, F.; CASTELLUCCIO, M.; BACCHELLI, A. Understanding flaky tests: the developer's perspective. In: **Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2019. (ESEC/FSE 2019), p. 830–840. ISBN 9781450355728. Disponível em: https://doi.org/10.1145/3338906.3338945. Acesso em: 14 jul. 2022.

EL-MORABEA, K.; EL-GAREM, H. Testing pyramid. In: **Modularizing legacy projects using TDD**. Springer, 2021. p. 65–83. ISBN 978-1-4842-7428-6. Disponível em: https://doi.org/10.1007/978-1-4842-7428-6_4. Acesso em: 14 jul. 2022.

GHTorrent: GitHub's data from a firehose: proceedings of the 9th working conference on mining software repositories. In: GODFREY MICHAEL W. E WHITEHEAD, J. (Ed.). **MSR '12**. [*S. n.*]. ISSN 2160-1852. Disponível em: /pub/ghtorrent-githubs-data-from-a-firehose.pdf. Acesso em: 14 jul. 2022.

GRUBER, M.; FRASER, G. A survey on how test flakiness affects developers and what support they need to address it. In: **2022 IEEE Conference on Software Testing, Verification and Validation (ICST)**. [*S. l.*: *s. n.*], 2022. p. 82–92.

HAN, J.; PEI, J.; KAMBER, M. **Data mining**: concepts and techniques. [*S. l.*]: Elsevier, 2011.

LAM, W.; GODEFROID, P.; NATH, S.; SANTHIAR, A.; THUMMALAPENTA, S. Root causing flaky tests in a large-scale industrial setting. In: **Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis**. New York, NY, USA: Association for Computing Machinery, 2019. (ISSTA 2019), p. 101–111. ISBN 9781450362245. Disponível em: https://doi.org/10.1145/3293882.3330570. Acesso em: 14 jul. 2022.

LAM, W.; MUsLU, K.; SAJNANI, H.; THUMMALAPENTA, S. A study on the lifecycle of flaky tests. In: **Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2020. (ICSE '20), p. 1471–1482. ISBN 9781450371216. Disponível em: https://doi.org/10.1145/3377811.3381749. Acesso em: 14 jul. 2022.

LAM, W.; WINTER, S.; WEI, A.; XIE, T.; MARINOV, D.; BELL, J. A large-scale longitudinal study of flaky tests. **Proc. ACM Program. Lang.**, Association for Computing Machinery, New York, NY, USA, v. 4, n. OOPSLA, nov 2020. Disponível em: https://doi.org/10.1145/3428270. Acesso em: 14 jul. 2022.

LUO, Q.; HARIRI, F.; ELOUSSI, L.; MARINOV, D. An empirical analysis of flaky tests. In: **Proceedings of the 22nd ACM SIGSOFT International Symposium on**

Foundations of Software Engineering. New York, NY, USA: Association for Computing Machinery, 2014. (FSE 2014), p. 643–653. ISBN 9781450330565. Disponível em: https://doi.org/10.1145/2635868.2635920. Acesso em: 14 jul. 2022.

MICCO, J. **The state of continuous integration testing @Google**. 2017. Disponível em: https://research.google/pubs/pub45880/. Acesso em: 14 jul. 2022.

MSR. **The 2022 mining software repositories conference**. 2022. Disponível em: https://conf.researchr.org/home/msr-2022. Acesso em: 14 jun. 2022.

PARRY, O.; KAPFHAMMER, G. M.; HILTON, M.; MCMINN, P. A survey of flaky tests. **ACM Trans. Softw. Eng. Methodol.**, Association for Computing Machinery, New York, NY, USA, v. 31, n. 1, oct 2021. ISSN 1049-331X. Disponível em: https://doi.org/10.1145/3476105. Acesso em: 14 jul. 2022.

PINTO, G.; MIRANDA, B.; DISSANAYAKE, S.; D'AMORIM, M.; TREUDE, C.; BERTOLINO, A. What is the vocabulary of flaky tests? In: **Proceedings of the 17th International Conference on Mining Software Repositories**. New York, NY, USA: Association for Computing Machinery, 2020. (MSR '20), p. 492–502. ISBN 9781450375177. Disponível em: https://doi.org/10.1145/3379597.3387482. Acesso em: 14 jul. 2022.

PRESSMAN, R. S. **Software engineering**: a practitioner's approach. [*S. l.*]: Palgrave macmillan, 2005.

ROMANO, A.; SONG, Z.; GRANDHI, S.; YANG, W.; WANG, W. An empirical analysis of ui-based flaky tests. In: **2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)**. [*S. l.*: *s. n.*], 2021. p. 1585–1597. ISSN 1558-1225.

SOMMERVILLE, I. **Engenharia de software**. Pearson Prentice Hall, 2011. ISBN 9788579361081. Disponível em: https://books.google.com.br/books?id=H4u5ygAACAAJ. Acesso em: 14 jul. 2022.

VAHABZADEH, A.; FARD, A. M.; MESBAH, A. An empirical study of bugs in test code. In: **Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)**. USA: IEEE Computer Society, 2015. (ICSME '15), p. 101–110. ISBN 9781467375320. Disponível em: https://doi.org/10.1109/ICSM.2015.7332456. Acesso em: 14 jul. 2022.

VALENTE, M. T. **Engenharia de software moderna**. [*S. n.*], 2020. ISBN 6500019504. Disponível em: https://engsoftmoderna.info/. Acesso em: 14 jul. 2022.

VIDONI, M. A systematic process for mining software repositories: results from a systematic literature review. **Information and Software Technology**, v. 144, p. 106791, 2022. ISSN 0950-5849. Disponível em: https://www.sciencedirect.com/science/article/pii/S0950584921002317. Acesso em: 14 jul. 2022.

WHITTAKER, J. What is software testing? and why is it so hard? **IEEE Software**, v. 17, n. 1, p. 70–79, 2000.

WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLÉN, A. **Experimentation in software engineering**. [*S. l.*]: Springer Science & Business Media, 2012.

ZHANG, S.; JALALI, D.; WUTTKE, J.; MUsLU, K.; LAM, W.; ERNST, M. D.; NOTKIN, D. Empirically revisiting the rest independence assumption. In: **Proceedings of the 2014 International Symposium on Software Testing and Analysis**. New York, NY, USA: Association for Computing Machinery, 2014. (ISSTA 2014), p. 385–396. ISBN 9781450326452. Disponível em: https://doi.org/10.1145/2610384.2610404. Acesso em: 14 jul. 2022.