Universidade Federal de Ouro Preto
Instituto de Ciências Exatas e Aplicadas
Departamento de Computação e Sistemas

Uma investigação sobre ferramentas para a detecção e o reparo de *flaky tests*

Victor Huggo Duarte Fernandes

TRABALHO DE CONCLUSÃO DE CURSO

ORIENTAÇÃO:
Euler Horta Marinho

Novembro, 2022
João Monlevade–MG

Victor Huggo Duarte Fernandes

Uma investigação sobre ferramentas para a detecção e o reparo de *flaky tests*

Orientador: Euler Horta Marinho

Monografia apresentada ao curso de Engenharia da Computação do Instituto de Ciências Exatas e Aplicadas, da Universidade Federal de Ouro Preto, como requisito parcial para aprovação na Disciplina "Trabalho de Conclusão de Curso II".

Universidade Federal de Ouro Preto

João Monlevade

Novembro de 2022

SISBIN - SISTEMA DE BIBLIOTECAS E INFORMAÇÃO

F363i Fernandes, Victor Huggo Duarte.
Uma investigação sobre ferramentas para a detecção e o reparo de flaky tests. [manuscrito] / Victor Huggo Duarte Fernandes. - 2022.
62 f.: il.: color., gráf., tab..

Orientador: Prof. Me. Euler Horta Marinho.
Monografia (Bacharelado). Universidade Federal de Ouro Preto. Instituto de Ciências Exatas e Aplicadas. Graduação em Engenharia de Computação .

1. Engenharia de software. 2. Software - Confiabilidade. 3. Software - Desenvolvimento. 4. Software - Testes. I. Marinho, Euler Horta. II. Universidade Federal de Ouro Preto. III. Título.

CDU 004.41

Bibliotecário(a) Responsável: Flavia Reis - CRB6-2431

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE OURO PRETO
REITORIA
INSTITUTO DE CIENCIAS EXATAS E APLICADAS
DEPARTAMENTO DE COMPUTACAO E SISTEMAS

FOLHA DE APROVAÇÃO

Victor Huggo Duarte Fernandes

Uma investigação sobre ferramentas para a detecção e o reparo de *flaky tests*

Monografia apresentada ao Curso de Engenharia de Computação da Universidade Federal
de Ouro Preto como requisito parcial para obtenção do título de Bacharel em Engenharia de Computação

Aprovada em 03 de novembro de 2022

Membros da banca

Mestre - Euler Horta Marinho - Orientador (Universidade Federal de Ouro Preto)
Doutor - Diego Zuquim Guimarães Garcia - (Universidade Federal de Ouro Preto)
Doutora - Kattiana Fernandes Constantino - (DELCOM/CEFET-MG)

Euler Horta Marinho, orientador do trabalho, aprovou a versão final e autorizou seu depósito na Biblioteca Digital de Trabalhos de Conclusão de Curso da UFOP em 29/11/2022

Documento assinado eletronicamente por **Euler Horta Marinho**, **PROFESSOR DE MAGISTERIO SUPERIOR**, em 29/11/2022, às 14:11, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site http://sei.ufop.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0 , informando o código verificador **0434053** e o código CRC **4D22744B**.

R. Diogo de Vasconcelos, 122, - Bairro Pilar Ouro Preto/MG, CEP 35402-163
Telefone: (31)3808-0819 - www.ufop.br

Este trabalho é dedicado aos meus pais, Rosilene e Leonardo. Sem eles nada seria possível.

Agradecimentos

Agradeço primeiramente a Deus, por me amparar durante toda a minha vida e por ter me dado forças para superar cada obstáculo durante essa caminhada.

Agradeço a minha mãe, Rosilene, por ser peça fundamental na realização deste sonho. Ao meu pai, Leonardo, e minhas irmãs, Amanda e Maria Clara, pelo apoio durante toda graduação. A toda a minha família que sempre me incentivou e vibrou a cada conquista.

Aos meus amigos e companheiros de turma Bruno, Bryan, Débora, Hellrison, Lucas Henrique, Lucas Miranda, Lucas Rocha, Nathália, Pedro, Thaune, Tiesari e Victor pela parceria durante todo esse percurso. Sem eles seria mais difícil a chegada até aqui.

Ao meu orientador, Euler Horta Marinho, agradeço pelo voto de confiança, parceria, todo conhecimento transmitido, apoio e incentivo. Agradeço também à todos os professores que contribuíram com a minha formação.

Sou muito grato à todos por terem contribuído com a minha caminhada!

Resumo

Flaky tests são testes automatizados de software que possuem comportamento não determinísticos. Esse comportamento dos *flaky tests* implica em diversos problemas no ciclo de vida de um software, isso porque a fase do teste de software é a etapa do processo de desenvolvimento, onde se busca a garantia da qualidade e confiabilidade do produto final. A presença de *flaky tests* em um conjunto de testes pode implicar na degradação da qualidade do software, além de outros impactos que são discutidos neste trabalho. Assim, o tratamento de *flaky tests* é de extrema importância para a Engenharia de Software. Com isso, ferramentas para detecção e reparo de *flaky tests* estão sendo desenvolvidas. Este estudo busca investigar os *flaky tests*, buscando entender também as ferramentas para detecção e reparo de *flaky tests*. Para isso, realizamos o levantamento dessas ferramentas, as quais foram estudadas, configuradas e utilizadas. Além disso, utilizamos uma das ferramentas de detecção *flaky tests* em conjunto com uma ferramenta de detecção de *test smells*, a fim de traçar uma correlação entre os temas. Discutimos a dificuldade na utilização das ferramentas, já que a grande maioria delas estão em estágios iniciais de desenvolvimento. Por fim, constatamos a correlação entre a categoria de *flaky test* causados por espera assíncrona e a categoria *Sleepy Test* de *test smell*.

Palavras-chaves: Teste de software, *Flaky Test*, *Test Smell*.

Abstract

Flaky tests are automated tests that have non-deterministic behavior. This behavior of flaky tests implies several problems in the life cycle of a software, because the software testing phase is the stage of the development process, where the quality and reliability of the software is sought. The presence of flaky tests in a set of tests can imply the degradation of software quality, in addition to other impacts that are discussed in this work. Thus, the treatment of flaky tests is extremely important for Software Engineering. Therefore, tools for detecting and repairing flaky tests are being developed. This study seeks to investigate flaky tests, also seeking to understand the tools for detection and repair of flaky tests. For this, we carried out a survey of these tools, which were studied, configured and used. In addition, we used one of the flaky tests detection tools together with a test smells detection tool, in order to trace a correlation between the themes. We discussed the difficulty in using the tools, since the vast majority of them are in the early stages of development. Finally, we found a correlation between the flaky test category caused by asynchronous waiting and the Sleepy Test category of test smell.

Key-words: Software test, Flaky Test, Test Smell.

Lista de ilustrações

Lista de tabelas

Sumário

1 Introdução

O teste de software é uma etapa imprescindível do ciclo de vida do desenvolvimento de software, o qual tem como objetivo garantir a qualidade e a confiabilidade do produto gerado. Sua principal função é detectar *bugs* para descobri-los e detectá-los (DEVI, 2012). Além disso, os testes de software são utilizados como meio de garantir que o produto entregue esteja de acordo com o que foi especificado, pois como dito por Engholm Junior (2010), a qualidade de software pode ser definida como a entrega de um produto ao cliente que satisfaça suas expectativas com base nos requisitos previamente levantados.

Porém, para que o teste de software garanta qualidade e confiabilidade, é necessário que ele possua comportamento determinístico, para que o desenvolvedor possa tomar as corretas conclusões após executar o conjunto de teste. Ou seja, espera-se que após N execuções de um teste, sem que haja alteração no código testado, ele possua o mesmo comportamento para as N execuções. Caso isso não ocorra, o teste possui um comportamento não determinístico.

Testes de software que possuem comportamento não-determinísticos são conhecidos na literatura como *flaky tests*. Para exemplificar, podemos imaginar um cenário onde existem dois testes (A e B) que utilizam um mesmo recurso, por exemplo um arquivo de texto, sendo que o teste A realiza uma operação de escrita e após isso uma asserção, já o teste B realiza a exclusão do arquivo de texto e após isso, valida se o arquivo foi devidamente excluído. Esse cenário implica em uma dependência de ordem entre os testes, ou seja, é necessário que o teste A seja executado antes do teste B, para que ele possa encontrar o arquivo de texto. Caso numa execução do conjunto de testes essa ordem não seja respeitada, o teste falhará, enquanto em outra execução, onde a ordem foi respeitada, o teste não falhará. Ou seja, um mesmo teste apresentará comportamento diferentes em execuções diferentes. Isso poderá indicar um falso *bug*, já que o teste falhou, mas as operações de escrita e exclusão funcionaram perfeitamente como esperado.

Os *flaky tests* estão ligados a práticas ruins de programação, assim como os *test smells*. *Tests Smells* são indicativos, a nível de código, que apontam pontos fracos, os quais podem ocasionar falhas. Os *test smells* são um desvio de como os testes devem ser escritos, organizados e como os testes devem interagir com os outros (CAMARA et al., 2021). Esse desvio pode indicar problemas de projeto de teste e pode prejudicar o desempenho do teste (PERUMA et al., 2019). Para exemplificar, podemos citar um método de teste que utiliza o método de asserção diversas vezes, sem que seja fornecida uma mensagem de explicação para cada asserção. Se uma dessas asserções falhar, detectar o problema poderá ser uma tarefa custosa para o desenvolvedor.

O tema principal deste estudo são os *flaky tests*. Porém, acredita-se que os *flaky tests* estão correlacionados com os *test smells*. Acredita-se também que a existência de um pode implicar na existência do outro. Logo, estudaremos essa possível correlação e discutiremos os resultados obtidos neste estudo.

1.1 O problema de pesquisa

A Engenharia de Software tem concentrado esforços para combater os *flaky tests* nos últimos anos. Pois eles estão fortemente presentes em diversos projetos de empresas de grande porte, como veremos nos próximos capítulos, o que tem gerado impacto econômico para essas empresas.

Ao executar um conjunto de testes, o desenvolvedor espera identificar se o requisito especificado foi devidamente desenvolvido ou identificar *bugs* no código implementado. Porém, ao executar um teste que possui comportamento *flakiness*, o resultado obtido não é confiável. Logo, um falso *bug* pode ser identificado ou um *bug* verdadeiro pode passar despercebido. Ambos os casos podem gerar grandes impactos. No primeiro caso, serão alocadas horas de desenvolvimento para corrigir um erro que não existe. Já no segundo, o produto pode ser entregue com um problema grave que gerará reclamações ou problemas futuros.

Porém, os esforços nos estudos do *flaky tests* ainda estão em um estágio inicial. Existem poucos referenciais teóricos em comparação à outros temas da Engenharia de Software, como *code smells*, *bad smells*, *clean code*, entre outros.

Acredita-se que um grande passo para lidar com *flaky test* é a criação de ferramentas para detectar, mitigar e solucionar *flaky tests*. Com isso, alocamos esforços para estudar as ferramentas de detecção de *flaky tests* existentes. Pois acredita-se que a detecção é uma etapa de suma importância, já que o grande problema causado pelos *flaky tests* é o fato dos desenvolvedores demorarem para notar que estão trabalhando com *flaky tests*, isso porque um teste pode demorar várias execuções para apresentar um comportamento *flakiness*. Logo, se o desenvolvedor detectar quanto antes a presença de *flaky test* no seu conjunto de testes, ele poderá tomar medidas imediatas para lidar com esse problema.

1.2 Objetivos

Com base no problema apresentado, este trabalho tem como objetivo estudar os *flaky tests*, bem como as ferramentas de detecção e a relação entre *flaky tests* e *test smells*.

Para isso, exploraremos as diversas categorias de *flaky tests* e ferramentas de detecção existentes, com base na literatura. Além disso, trabalharemos com as ferramentas de detecção existentes, onde iremos realizar a configuração delas e utilizá-las em projetos

de domínio público, a fim de detectar *flaky tests* nos conjuntos de teste desses projetos. Destacaremos todas as dificuldades e resultados encontrados nessa etapa.

Por fim, para realizar o estudo da correlação entre *flaky test* e *test smells*, utilizaremos ferramentas de detecção automática para ambos, a fim de comprovar uma possível correlação entre eles. Utilizaremos também ferramentas para correção de *test smells*, pois acredita-se que a correção do *test smell* possa impactar no caráter *flaky* do teste.

1.3 Metodologia

Devido ao fato deste trabalho ser um estudo investigativo, a primeira etapa e a mais importante é a revisão da literatura. Realizamos um levantamento de trabalhos que possuem *flaky tests* como tema principal, levantamos também trabalhos que tratam sobre ferramentas de detecção de *flaky tests*. Foi necessário realizar um levantamento de estudos sobre *test smells* já que temos como objetivo traçar um paralelo entre os temas.

Após explorar esses temas, realizaremos a parte prática deste estudo. Onde utilizaremos as ferramentas encontradas, tanto para detecção de *flaky tests* quanto *test smells*. Por fim, analisaremos e discutiremos os resultados encontrados, primeiro sobre a utilização das ferramentas de detecção de *flaky tests* e por último sobre a utilização da ferramenta de detecção de *test smells* em projetos que apresentaram *flaky tests*.

Outro meio utilizado para comprovar a correlação entre *flaky tests* e *test smells* consiste em utilizar ferramentas para correção automática de *test smells*. Os *test smells* são amplamente estudados e abordados na Engenharia de Software. Logo, existem ferramentas para correção automática que estão em estágios mais avançados do que as ferramentas para correção de *flaky tests*. Assim, realizamos um levantamento de ferramentas disponíveis, as quais utilizaremos a fim de remover os *test smells* de projetos que possuem *flaky tests* e analisar o impacto dessa correção no caráter *flaky* do teste. Porém, como veremos nos próximos capítulos, não foi possível utilizá-las amplamente, isso porque este estudo se concentrou em estudar a correlação entre o *flaky test* causados por espera assíncrona e o *test smell* causado por inserção de atraso (*Sleepy Test*), sendo essa uma categoria de *test smell* de correção complexa, já que exige mais do que uma simples refatoração. Isso se deve ao fato de que, ao inserir um atraso em um método de teste, o usuário espera reproduzir um evento do mundo real, apenas retirar esse atraso não só não resolveria o problema como implicaria em outros. Com isso, nenhuma das ferramentas de correção automática de *test smell* levantada, foi capaz de corrigir o *Sleepy Test*.

Com isso, espera-se discutir a utilização das ferramentas de detecção de *flaky tests* para mitigar os impactos causados por eles e também discutiremos a possível relação entre *flaky tests* e *test smells*.

1.4 Organização do trabalho

Este trabalho está organizado em cinco capítulos. Sendo eles:

- Capitulo 1: introduz o tema e que será abordado neste trabalho, detalhando objetivo e metodologia utilizada;
- Capitulo 2: apresenta o referencial teórico, onde abordaremos os trabalhos relacionados à *flaky tests*, *test smells*, a correlação entre os dois temas anteriores, ferramentas para detecção automática de *flaky tests* e *test smells*;
- Capitulo 3: relata o desenvolvimento do estudo, o processo de configuração das ferramentas de detecção automática de *flaky tests* e *test smells*, e a utilização dessas ferramentas em projetos disponíveis no GitHub;
- Capítulo 4: discussão sobre os resultados obtidos após a execução das ferramentas, relata as dificuldades encontradas no processo de utilização das ferramentas e discute a correlação entre *flaky test* e *test smells*, após a utilização das ferramentas de detecção de ambos;
- Capitulo 5: conclui a dissertação com as considerações finais e trabalhos futuros.

2 Revisão bibliográfica

2.1 Teste de Software

A construção de um software pode se tornar uma tarefa complexa dependendo das características e dimensões do sistema (DELAMARO; JINO; MALDONADO, 2013). É possível que diversos problemas ocorram durante e após a criação do software, como por exemplo, a criação de um sistema diferente do esperado, erros de interface, segurança, entre outros. Logo, tornou-se necessário a criação de ferramentas e metodologias que mitigassem esses problemas. A principal atividade realizada é coletivamente chamada de "Validação, Verificação e Teste", ou "VV&T"(DELAMARO; JINO; MALDONADO, 2013).

Neste trabalho, o nosso foco será a terceira atividade do "VV&T", a atividade de testes. Segundo Arilo Cláudio Neto (2007), o teste de software é o processo de execução de um produto para determinar se ele atingiu as expecificações e funcionou corretamente no ambiente para o qual foi projetado. Com isso, revelar possíveis falhas, para que as mesmas sejam corrigidas pelos desenvolvedores antes da entrega.

Para revelar possíveis falhas, os testes de software contam com um conjunto de entradas, as quais nos permitem observar o comportamento do sistema. Para cada uma das entradas, tem-se uma saída. Dado uma determinada entrada, caso a saída não seja a esperada, tem-se um erro. Porém, para que seja possível detectar um erro ou defeito por meio da etapa de testes, é necessário que os mesmos sejam determinísticos.

Testes determinísticos são aqueles que possuem um comportamento consistente, ou seja, são testes que dada uma entrada retornam sempre a mesma saída, sendo ela esperada ou não. Testes que possuem comportamento não determinísticos são conhecidos pela literatura como *flaky tests*. Os quais serão tema deste trabalho.

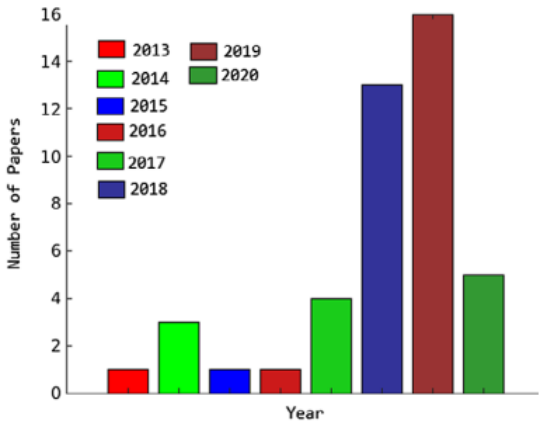
2.2 Flaky Tests

Flaky tests são testes que falham de forma inconsistente, sem alterações no código em teste (PARRY et al., 2021). Isso impede os desenvolvedores de obterem uma indicação clara da presença de bugs de software, limitando a confiabilidade dos conjuntos de testes (PARRY et al., 2021). Nota-se que os *flaky tests* impactam negativamente os desenvolvedores ao fornecer sinais enganosos sobre as mudanças recentes (LAM et al., 2020).

Importante ressaltar que os estudos sobre *flaky tests* estão apenas no início. Como podemos ver na Figura 1, em 2013 apenas uma pesquisa foi realizada sobre o tema. Porém, tem-se uma grande ascensão nos últimos anos. Em 2019, foram realizados 16 estudos que

possuem como tema *flaky tests*. Além disso, é importante pontuar que em 2020 foram relatados apenas cinco estudos, pois o artigo (ZOLFAGHARI et al., 2021) foi redigido no ano de 2020. Logo, espera-se um número de publicações maior ou igual ao de 2019.

Figura 1 – Total de pesquisas sobre *flaky tests* por ano



Fonte: (ZOLFAGHARI et al., 2021)

A ascensão no número de pesquisas na área de *flaky tests* pode ser explicada pelo fato deles terem se tornado um grande problema para grandes indústrias e pesquisas nos últimos anos (LAM et al., 2020). Diversos estudos relatam os impactos dos *flaky tests* nas empresas, como em (MICCO, 2017), onde relata-se que 1,5% de todos os testes executados no Google são *flaky tests* e que quase 16% de seus 4,2 milhões de testes individuais falham independentemente de alteração no código ou nos testes. Behrouz Zolfaghari (2021) relatou que em 2017, o Google listou cerca de 4,2 milhões de testes executados em seu sistema de integração contínua (CI), onde entre esses testes, cerca de 63 mil exibiram pelo menos um flaky run ao longo de uma semana (em média). Isso compõe menos de 2% de seus testes mas ainda é causador de um desgaste significativo aos engenheiros e desenvolvedores da Google.

O ponto de partida para entendermos os *flaky tests* é entender quais são as causas dos mesmos. Em *'A Survey Of Flaky Tests'* (2021) é realizada uma categorização dos *flaky tests* com base nas suas respectivas causas. As categorias levantadas em *'A Survey Of Flaky Tests'* (2021) são divididas em familias, sendo elas: *Intra-test*, *Inter-test* e *External*.

As categorias que compõem a familia *Intra-test* são nomeadas por *Concurrency*, *Randomness*, *Floating Point*, *Unordered Collection*, *Too Restrictive Range* e *Test Case Timeout*. Já da familia *Inter-test* são: *Test Order Dependency*, *Resource Leak* e *Test Suite Timeout*. Por fim, a familia *External* é composta pelas categorias: *Asynchronous Wait*, *I/O*, *Network*, *Time* e *Platform Dependency*. Como podemos ver nas tabelas 1, 2 e 3 respectivamente.

Entender cada uma das categorias foi de extrema importância para a realização deste trabalho. Já que visamos estudar ferramentas para detecção de *flaky testes* e existem ferramentas para identificação de testes de cada uma das categorias citadas.

Tabela 1 – Categorias de *flaky test* da familia *intra-test*.

Categoria	**Descrição**	**Fonte**
Concurrency	Testes que invocam várias threads interagindo de maneira insegura ou inesperada. O *flaky tests* é causado por, por exemplo, condições de corrida resultantes de suposições implícitas sobre a ordem de execução, levando a deadlocks em certas execuções de teste.	(ECK et al., 2019)
Randomness	Testes que usam resultados de geradores de dados aleatórios. Se o teste não considerar todos os casos possíveis, então o teste pode falhar intermitentemente..	(ECK et al., 2019)
Floating Point	Testes que usam resultados de uma operação de ponto flutuante. As operações de ponto flutuante podem sofrer de uma variedade de discrepâncias e imprecisões, como precisão sobre e sob fluxos, adição não associativa, etc., que se não for devidamente contabilizado, pode resultar em resultas de testes inconsistentes, por exemplo, comparando o resultado de uma operação de ponto flutante com um valor real de uma afirmção.	(ECK et al., 2019)
Unordered Collection	O teste assume uma ordem de iteração específica para uma coleção não ordenada. Como nenhuma ordem específica é especificada para tais objetos, os testes que assumem que eles irão iterar em alguma ordem fixa provavelmente serão *flaky tests* devido a uma variedade de razões, por exemplo, implementação da classe de coleção.	(GRUBER et al., 2021)
Too Restrictive Range	Testes onde parte da faixa de saída válida está fora do que é aceito em sus asserções. Estes são considerados *flaky tests*, pois não leva em consideração casos de canto e, portanto, pode falhar intermitentemente quando eles surgirem.	(ECK et al., 2019)
Test Case Timeout	Teste especificado com um limite superior em seu tempo de execução. Já que geralmente não é possível estimar com precisão quanto tempo um teste levará para ser executado, especificando um limite de tempo superior, um desenvolvedor pode correr o risco de criar um *flaky test*, pois pode falhar em certas execuções que são mais lentas do que o previsto.	(ECK et al., 2019)

Fonte: (PARRY et al., 2021) adaptada pelo autor (2022).

Diversos estudos tiveram como foco avaliar *flaky tests* e categorizá-los. Como em *'An empirical analysis of flaky tests'* (2014), onde diz que entre 201 commits realizados em 51 projetos da Apache Software Foundation, 37% foram categorizados por espera assíncrona, 16% da categoria de simultaneidade, 9% na categoria de dependência de ordem de teste e 5% em vazamento de recursos. Já em *'A study on the lifecycle of flaky tests'* (2020), entre os 134 pull requests que possuiam alguma relação com *flaky tests* em projetos internos da Microsoft foram categorizados 78% destes como causados por espera

assíncrona. O que mostra que os testes causados pela espera assíncrona é uma das causas mais proeminentes de *flaky tests*. Além dela, outra categoria muito abordada e estudada é a categoria de dependência de ordem. Segundo Parry (2021), várias fontes consideram especificamente as causas e fatores associados à categoria de dependência de ordem.

Tabela 2 – Categorias de *flaky tests* da família *inter-test*.

Categoria	**Descrição**	**Fonte**
Test Order Dependency	Testes que dependem de algum valor ou recurso compartilhado que é modificado por outro teste que afeta seu resultado. No caso em que o teste é executado e a ordem é alterada, esses testes irregulares podem produzir resultados inconsistentes, uma vez que as dependências de testes previamente executados são quebrados	(ECK et al., 2019)
Resource Leak	Testes que manipulam indevidamente algum recurso externo, por exemplo, falha ao liberar memória alocada. Recursos manipulados incorretamente podem causar falhas em casos de testes executados posteriormente que tentam reutilizar esses recursos.	(ECK et al., 2019)
Test Suite Timeout	O teste faz parte de um conjunto de testes com tempo de execução limitado. Falha de forma intermitente, porque acontece de estar em execução quando o conjunto de testes atinge um nível superior do limite de tempo.	(ECK et al., 2019)

Fonte: (PARRY et al., 2021) adapatada pelo autor (2022).

Em (ECK et al., 2019), foi solicitado a 21 desenvolvedores de software da Mozila que classificassem 200 *flaky tests* que eles haviam corrigido anteriormente, com base nas categorias levantadas em (LUO et al., 2014). Entre esses, 26% dos testes foram classificados por simultaneidade, 22% por espera assíncrona e 16% por intervalo muito restritivo.

Segundo Parry (2021), uma pesquisa realizada com desenvolvedores de software, descobriu que 59% afirmam lidar com testes irregulares mensalmente, semanalmente e diariamente. Segundo Kapfhammer (2004), os desenvolvedores de software dependem de testes para identificar bugs e fornecer um sinal quanto à correção de seu código. Logo, a existência de *flaky tests* faz com que desenvolvedores dediquem tempo para investigar erros inexistentes, o que acaba gerando prejuízos econômicos às empresas de desenvolvimento. Além disso, a existência de *flaky tests* pode fazer com que desenvolvedores ignorem falhas verdadeiras, o que poderá comprometer a qualidade do produto final. Isso motivou diversos pesquisadores a desenvolverem técnicas e ferramentas para detecção, mitigação e reparo de *flaky tests*.

Tabela 3 – Categorias de *flaky tests* da familia *external*.

Categoria	**Descrição**	**Fonte**
Asynchronous Wait	O teste faz uma chamada assíncrona e não espera explicitamente que ela termine antes de avaliar as asserções, geralmente usando um atraso de tempo fixo. Os resultados podem ser inconsistentes em execuções em que a chamada assíncrona leva mais tempo do que o tempo especificado para terminar, levando à falha	(ECK et al., 2019)
I/O	Teste que é instável devido à manipulação de operações de entrada e saída. Por exemplo, um teste que falha quando um disco não tem espaço livre ou fica cheio durante a gravação do arquivo.	(ECK et al., 2019)
Network	Teste que depende da disponibilidade de uma conexão de rede, por exemplo, consultando um servidor web. No caso em que a rede não está disponível ou o recurso necessário está muito ocupado, o teste pode ficar instável.	(ECK et al., 2019)
Time	O teste depende da hora do sistema local e pode ser instável devido a discrepâncias na precisão e no fuso horário, por exemplo, falha quando a meia-noite muda no fuso horário UTC.	(ECK et al., 2019)
Platform Dependency	O teste depende de alguma funcionalidade específica de um sistema operacional específico, versão de biblioteca, fornecedor de hardware etc. Embora esses testes possam produzir um resultado consistente em uma determinada plataforma, eles ainda são considerados irregulares, principalmente com o aumento da integração contínua baseada em nuvem serviços, onde diferentes execuções de teste podem ser executadas em diferentes máquinas físicas de uma maneira que parece não determinística para o usuário.	(ECK et al., 2019)

Fonte: (PARRY et al., 2021) adaptada pelo autor (2022).

2.3 Técnicas para Detecção de *Flaky Tests*

Após entender as categorias de *flaky tests* foi possível dar início ao estudo das técnicas e ferramentas para detecção dos mesmos. Como vimos, a existência de *flaky tests* pode ocasionar em diversos problemas para desenvolvedores, empresas e pesquisadores. Logo, detectar que um teste é *flaky* pode ser muito útil, já que evitará que os desenvolvedores tirem conclusões erradas sobre seus testes e até mesmo percam tempo corrigindo algo que pode não estar errado.

2.3.1 Repetição

A técnica mais utilizada é simples mas também primitiva, se resume a executar o conjunto de teste N vezes, até que um teste apresente um comportamento não determinístico. Porém, essa técnica consome recursos, computação e tempo.

Logo, diversos estudos foram realizados com o objetivo de explorar técnicas para identificação de *flaky tests*. Foi realizada uma análise da literatura em busca de trabalhos que relatassem *flaky tests* e técnicas para detecção dos mesmos. A técnica de repetição, já citada, é relatada em (PARRY et al., 2021) e (ZOLFAGHARI et al., 2021), e é adotada em infraestrutura de testes de grandes empresas, como o Google (MICCO, 2017) e a Microsoft (LAM et al., 2020). Além disso, é utilizada como base para diveras outras técnicas e plugins de detecção.

2.3.2 Cobertura de Código

Outra técnica relatada por Zolfaghari (2021), é a técnica de cobertura de código. Entre os artigos abordados nesse trabalho, os que abordam a técnica de detecção de cobertura de código representam 33,3%. Segundo Zolfaghari (2021), a cobertura de código é uma medida do número de linhas de código do software que são executadas por um teste, e quanto maior for essa cobertura maior a chance de um bug de software ser identificado. Em (BELL et al., 2018), essa técnica é nomeada por "Cobertura Diferencial" e é apresentada como uma solução para o custo computacional da técnica de reexecução de testes, já que para realização da detecção não é necessário reexecutar os testes repetidas vezes. Bell (2018) apresenta uma ferramenta para detecção de testes que utiliza essa técnica, conhecida como DeFlaker. Ferramenta a qual também será abordada neste trabalho. O DeFlaker tem como objetivo detectar *flaky tests* em projetos Java, sem ter que executá-los repetidas vezes. Ele assume que um teste é *flaky* se esse teste passou anteriormente e depois falhou, desde que esse teste não cubra nenhum código que foi alterado recentemente. Essa ferramenta é atrelada a funcionalidades do GitHub, as quais permitem realizar esse monitoramento.

2.3.3 Randomização de elementos

Diferente do DeFlaker, que utiliza a técnica de cobertura diferencial e é capaz de detectar diversas categorias de *flaky tests*, existem outras ferramentas que são focadas em detectar determinadas categorias. Como o NonDex (GYORI et al., 2016), que é focado em detectar *flaky tests* causados por suposições de uma implementação determinística de uma especificação não determinística. Essa categoria é conhecida como **ADINs** (*Assumption of a Deterministic Implementation of a Non-Deterministic Specification*). Esses *flaky tests* ocorrem quando um desenvolvedor espera um comportamento determinístico de um objeto não determinístico. Como por exemplo, testes que esperam uma determinada ordem de iteração de objetos como HashMaps, isso poderá causar instabilidade aos testes pois a implementação do HashMap varia entre plataformas. Segundo Gyori (2016), o NonDex consiste na reimplementação de uma série de métodos e classes da biblioteca padrão Java, que randomizava os elementos não determinísticos de suas respectivas especificações.

2.3.4 Adição de ruído

Outra técnica amplamente utilizada para detecção de *flaky tests* é a adição de ruído ao ambiente de testes. Essa técnica se concentra em detectar testes que possuem dependências cronológicas. Como por exemplo, os *flaky tests* causados por espera assíncrona, simultaneidade e tempo limite do caso de teste. Essa técnica consiste em adicionar estresse à CPU com o objetivo de afetar o tempo de execução e a intercalação de threads. Uma das ferramentas existentes que utiliza essa técnica é apresentada em (SILVA; TEIXEIRA; D'AMORIM, 2020), conhecida como Shaker.

Os *flaky tests* causados por tempo limite do caso de teste ocorrem pois um limite de tempo é especificado, como por exemplo nos métodos de teste que invocam o método Thread.sleep(). O tempo em que a thread ficará em suspensão é estimado pelo desenvolvedor, porém caso uma execução seja mais lenta que o previsto, o teste acabará falhando. Com isso, o objetivo da técnica de adição de ruído é inserir estresse no ambiente de teste, a fim de fazer com que os testes possuam um tempo de execução superior às execuções normais. Caso algum teste não tenha falhado na execução normal e, após a inserção de ruído, comece a apresentar falhas, pode ser considerado um *flaky test*.

2.4 Test Smells

Code smells é um termo comumente usando para descrever potenciais problemas no projeto de software (SANTOS et al., 2018). Ou seja, *code smells* são indicativos, a nível de código, que apontam pontos fracos que podem ocasionar falhas no código em produção.

No contexto de testes de software, esse indicativo também existe, porém é conhecido como *test smells*. Esse termo foi criado por Deursen (2001), onde definiu-se também um catálogo com 11 categorias de *test smells*. Catálogo o qual foi estendido em (MESZAROS, 2007), onde foram definidos 18 novos *test smells*.

Os *tests smells* têm se mostrando responsáveis por diminuir a qualidade dos sistemas de software em vários aspectos, como dificultar o entendimento mais complexo de manter e mais propenso a erros e *bugs* (MÄNTYLÄ; LASSENIUS, 2006). Além disso, em diversos estudos, como em (ZOLFAGHARI et al., 2021), é relatada uma possível relação entre *test smells* e *flaky tests*, onde diz que a presença do primeiro pode implicar na existência do segundo. Para exemplificar, podemos citar a correlação, que será estudada neste trabalho, entre o *test smell* da categoria *Sleepy Test* e os *flaky tests* causados por espera assíncrona. Isso ocorre devido ao fato de que a espera assíncrona que causa o *flaky test* é exatamente o que configura o *Sleepy Test*, que é a utilização de métodos que realizam atrasos por um tempo fixo, como o *Thread.sleep()*.

Porém, há uma limitação no número de estudos que comprovam essa relação. Logo,

notou-se a importância de tratarmos esse tema neste estudo, pois acredita-se que através dessa correlação, *insights* podem ser gerados visando a criação de novas ferramentas de detecção automática de *flaky tests*s.

Assim como *flaky tests*, existem diversas categorias de *test smells*. Nas próximas subseções abordaremos algumas dessas categorias, com base nas definições de cada uma delas, dadas por Peruma (2020).

2.4.1 Assertion Roulette

Ocorre quando um método de teste tem vários *asserts* não documentados. Várias declarações de *asserts* em um método de teste sem uma mensagem descritiva afetam a legibilidade, compreensão e manutenção, pois não é possível entender o motivo da falha do teste.

No exemplo da Figura 2, o método *testCloneNonBareRepoFromLocalTestServer()*[1] utiliza o método *AssertThat()* três vezes, onde cada um verifica uma condição diferente e não é fornecida nenhuma mensagem de explicação. Caso um desses *asserts* falhe, não será uma tarefa simples identificar qual a causa da falha do teste.

Figura 2 – *Assertion Roulette*

```
@MediumTest
public void testCloneNonBareRepoFromLocalTestServer() throws Exception {
    Clone cloneOp = new Clone(false, integrationGitServerURIFor("small-repo.early.git"), helper().newFolder());

    Repository repo = executeAndWaitFor(cloneOp);

    assertThat(repo, hasGitObject("ba1f63e4430bff267d112b1e8afc1d6294db0ccc"));

    File readmeFile = new File(repo.getWorkTree(), "README");
    assertThat(readmeFile, exists());
    assertThat(readmeFile, ofLength(12));
}
```

Fonte: (PERUMA et al., 2020)

2.4.2 Conditional Test Logic

Os métodos de teste precisam ser simples e executar todas as instruções no método de produção. As condições dentro do método de teste alterarão o comportamento do teste e sua saída esperada e levariam a situações em que o teste não detecta defeitos no método de produção, pois as instruções de teste não foram executadas porque uma condição não foi atendida. Além disso, o código condicional dentro de um método de teste afeta negativamente a facilidade de compreensão pelos desenvolvedores.

[1] https://github.com/rtyley/agit/

No exemplo da figura, 3, o método *testSpinner()*[2] possui vérias instruções de fluxo de controle. O fato dos *asserts* estarem dentro desses bloco de fluxo de controle torna o método mais complexo e impacta negativamente a manutenção do teste.

Figura 3 – *Coditional Test Logic*

```
@Test
public void testSpinner() {
    for (Map.Entry entry : sourcesMap.entrySet()) {

        String id = entry.getKey();
        Object resultObject = resultsMap.get(id);
        if (resultObject instanceof EventsModel) {
            EventsModel result = (EventsModel) resultObject;
            if (result.testSpinner.runTest) {
                System.out.println("Testing " + id + " (testSpinner)");
                //System.out.println(result);
                AnswerObject answer = new AnswerObject(entry.getValue(), "", new CookieManager(), "");
                EventsScraper scraper = new EventsScraper(RuntimeEnvironment.application, answer);
                SpinnerAdapter spinnerAdapter = scraper.spinnerAdapter();
                assertEquals(spinnerAdapter.getCount(), result.testSpinner.data.size());
                for (int i = 0; i < spinnerAdapter.getCount(); i++) {
                    assertEquals(spinnerAdapter.getItem(i), result.testSpinner.data.get(i));
                }
            }
        }
    }
}
```

Fonte: (PERUMA et al., 2020)

2.4.3 Constructor Initialization

Idealmente, o conjunto de testes não deve ter um construtor. A inicialização dos campos deve ser no método setUp. Os desenvolvedores que desconhecem o propósito do método setUp dariam origem a esse *smell*, definindo um construtor para o conjunto de testes. Como podemos ver na classe *TagEncodingTest*[3] na figura 4.

[2] https://github.com/Tyde/TuCanMobile
[3] https://code.briarproject.org/briar/briar/

Figura 4 – *Constructor Initialization*

```
public class TagEncodingTest extends BrambleTestCase {
        private final CryptoComponent crypto;
        private final SecretKey tagKey;
        private final long streamNumber = 1234567890;

        public TagEncodingTest() {
                crypto = new CryptoComponentImpl(new TestSecureRandomProvider());
                tagKey = TestUtils.getSecretKey();
        }

        @Test
        public void testKeyAffectsTag() throws Exception {
                Set set = new HashSet<>();
                for (int i = 0; i < 100; i++) {
                        byte[] tag = new byte[TAG_LENGTH];
                        SecretKey tagKey = TestUtils.getSecretKey();
                        crypto.encodeTag(tag, tagKey, PROTOCOL_VERSION, streamNumber);
                        assertTrue(set.add(new Bytes(tag)));
                }
        }
 ...
}
```

Fonte: (PERUMA et al., 2020)

2.4.4 Default Test

Por padrão, o Android Studio cria classes, como a classe *ExampleUnitTest*[4] da figura 5, como a classe de teste padrão quando um projeto é criado. Essas classes servem como exemplo para os desenvolvedores implementarem testes de unidade e devem ser removidas ou renomeadas. Ter esses arquivos no projeto fará com que os desenvolvedores comecem a adicionar métodos de teste a esses arquivos, tornando a classe de teste padrão um contêiner de todos os casos de teste. Isso também pode causar problemas quando as classes precisarem ser renomeadas no futuro.

Figura 5 – *Default Test*

```
public class ExampleUnitTest {
    @Test
        public void addition_isCorrect() throws Exception {
                assertEquals(4, 2 + 2);
        }

    @Test
    public void shareProblem() throws InterruptedException {
        .....
                Observable.just(200)
                .subscribeOn(Schedulers.newThread())
                .subscribe(begin.asAction());
                begin.set(200);
                Thread.sleep(1000);
                assertEquals(beginTime.get(), "200");
        .....
        }
        .....
}
```

Fonte: (PERUMA et al., 2020)

[4] https://github.com/httpdispatch/MissedNotificationsReminder/

2.4.5 Duplicate Assert

Esse *smell* ocorre quando um método de teste testa a mesma condição várias vezes dentro do mesmo método de teste. Se o método de teste precisar testar a mesma condição usando valores diferentes, um novo método de teste deve ser utilizado; o nome do método de teste deve ser uma indicação do teste que está sendo realizado. Possíveis situações que dariam origem a esse *smell* incluem: (1) desenvolvedores agrupando várias condições para testar um único método; (2) desenvolvedores realizando atividades de depuração; e (3) uma cópia e colagem acidental do código.

No método *testXmlSanitizer()*[5], há diversos *Aserts* duplicados no mesmo método de teste.

Figura 6 – *Duplicate Assert*

```
@Test
public void testXmlSanitizer() {
    boolean valid = XmlSanitizer.isValid("Fritzbox");
    assertEquals("Fritzbox is valid", true, valid);
    System.out.println("Pure ASCII test - passed");

    valid = XmlSanitizer.isValid("Fritz Box");
    assertEquals("Spaces are valid", true, valid);
    System.out.println("Spaces test - passed");

    valid = XmlSanitizer.isValid("Frützbüx");
    assertEquals("Frützbüx is invalid", false, valid);
    System.out.println("No ASCII test - passed");

    valid = XmlSanitizer.isValid("Fritz!box");
    assertEquals("Exclamation mark is valid", true, valid);
    System.out.println("Exclamation mark test - passed");

    valid = XmlSanitizer.isValid("Fritz.box");
    assertEquals("Exclamation mark is valid", true, valid);
    System.out.println("Dot test - passed");

    valid = XmlSanitizer.isValid("Fritz-box");
    assertEquals("Minus is valid", true, valid);
    System.out.println("Minus test - passed");

    valid = XmlSanitizer.isValid("Fritz-box");
    assertEquals("Minus is valid", true, valid);
    System.out.println("Minus test - passed");
}
```

Fonte: (PERUMA et al., 2020)

2.4.6 Eager Test

Ocorre quando um método de teste invoca vários métodos do objeto de produção. Esse cheiro resulta em dificuldades na compreensão e manutenção do teste. Como no método *NmeaSentence_GPGSA_ReadValidValues*[6] da figura abaixo:

[5] https://github.com/openbmap/radiocells-scanner-android/
[6] https://github.com/mendhak/gpslogger/

Figura 7 – *Eager Test*

```
@Test
public void NmeaSentence_GPGSA_ReadValidValues(){
    NmeaSentence nmeaSentence = new NmeaSentence("$GPGSA,A,3,04,05,,09,12,,,24,,,,,2.5,1.3,2.1*39");
    assertThat("GPGSA - read PDOP", nmeaSentence.getLatestPdop(), is("2.5"));
    assertThat("GPGSA - read HDOP", nmeaSentence.getLatestHdop(), is("1.3"));
    assertThat("GPGSA - read VDOP", nmeaSentence.getLatestVdop(), is("2.1"));
}
```

Fonte: (PERUMA et al., 2020)

2.4.7 Empty Test

Ocorre quando um método de teste não contém instruções executáveis. Esses métodos são possivelmente criados para fins de depuração e depois esquecidos ou contêm código comentado. Um teste vazio pode ser considerado problemático e mais perigoso do que não ter um caso de teste, pois o JUnit indicará que o teste passou mesmo se não houver instruções executáveis presentes no corpo do método. Como tal, os desenvolvedores que introduzirem alterações de quebra de comportamento na classe de produção não serão notificados dos resultados alternativos, pois o JUnit relatará o teste como aprovado.

Figura 8 – *Empty Test*

```
public void testCredGetFullSampleV1() throws Throwable{
//        ScrapedCredentials credentials =  innerCredTest(FULL_SAMPLE_v1);
//        assertEquals("p4ssw0rd", credentials.pass);
//        assertEquals("user@example.com",credentials.user);
}
```

Fonte: (PERUMA et al., 2020)

2.4.8 Exception Handling

Esse *smell* ocorre quando um método de teste retorna explicitamente uma aprovação ou falha de um método de teste dependente do método de produção lançando uma exceção. Os desenvolvedores devem utilizar o tratamento de exceção do JUnit para retornar aprovação ou falha no teste, em vez de escrever um código de tratamento de exceção personalizado ou lançar uma exceção.

Como no método *realCase()* [7] da figura 9, onde um *assert* é realizado a partir de uma exceção capturada.

[7] https://github.com/hgdev-ch/toposuite-android/

Figura 9 – *Exception Handling*

```
@Test
public void realCase() {
    Point p34 = new Point("34", 556506.667, 172513.91, 620.34, true);
    Point p45 = new Point("45", 556495.16, 172493.912, 623.37, true);
    Point p47 = new Point("47", 556612.21, 172489.274, 0.0, true);
    Abriss a = new Abriss(p34, false);
    a.removeDAO(CalculationsDataSource.getInstance());
    a.getMeasures().add(new Measure(p45, 0.0, 91.6892, 23.277, 1.63));
    a.getMeasures().add(new Measure(p47, 281.3521, 100.0471, 108.384, 1.63));

    try {
        a.compute();
    } catch (CalculationException e) {
        Assert.fail(e.getMessage());
    }

    // test intermediate values with point 45
    Assert.assertEquals("233.2405",
        this.df4.format(a.getResults().get(0).getUnknownOrientation()));
    Assert.assertEquals("233.2435",
        this.df4.format(a.getResults().get(0).getOrientedDirection()));
    Assert.assertEquals("-0.1", this.df1.format(
        a.getResults().get(0).getErrTrans()));

    // test intermediate values with point 47
    Assert.assertEquals("233.2466",
        this.df4.format(a.getResults().get(1).getUnknownOrientation()));
    Assert.assertEquals("114.5956",
        this.df4.format(a.getResults().get(1).getOrientedDirection()));
    Assert.assertEquals("0.5", this.df1.format(
        a.getResults().get(1).getErrTrans()));

    // test final results
    Assert.assertEquals("233.2435", this.df4.format(a.getMean()));
    Assert.assertEquals("43", this.df0.format(a.getMSE()));
    Assert.assertEquals("30", this.df0.format(a.getMeanErrComp()));
}
```

Fonte: (PERUMA et al., 2020)

O teste acima falha quando ocorre uma exceção especifica. Espera-se que esse teste seja dividido em vários testes que (1) gerem conscientemente a exceção e (2) não gerem uma exceção.

2.4.9 Ignored Test

O JUnit 4 fornece aos desenvolvedores a capacidade de suprimir a execução de métodos de teste, como no código da figura 10. No entanto, esses métodos de teste ignorados adicionam sobrecarga desnecessária em relação ao tempo de compilação e aumentam a complexidade e a compreensão do código.

Figura 10 – *Ignored Test*

```
@Ignore("disabled for now as this test is too flaky")
public void peerPriority() throws Exception {
        final List addresses = Lists.newArrayList(
        new InetSocketAddress("localhost", 2000),
        new InetSocketAddress("localhost", 2001),
        new InetSocketAddress("localhost", 2002)
        );
    peerGroup.addConnectedEventListener(connectedListener);
    .....
}
```

Fonte: (PERUMA et al., 2020)

2.4.10 Lazy Test

Ocorre quando vários métodos de teste invocam o mesmo método do objeto de produção. Como por exemplo, os métodos *testEncrypt()* e *testDecrypt()*[8] que chamam o mesmo método SUT, *Cryptographer.decrypt()*

Figura 11 – *Lazy Test*

```
@Test
public void testDecrypt() throws Exception {
    FileInputStream file = new FileInputStream(ENCRYPTED_DATA_FILE_4_14);
    byte[] enfileData = new byte[file.available()];
    FileInputStream input = new FileInputStream(DECRYPTED_DATA_FILE_4_14);
    byte[] fileData = new byte[input.available()];
    input.read(fileData);
    input.close();
    file.read(enfileData);
    file.close();
    String expectedResult = new String(fileData, "UTF-8");
    assertEquals("Testing simple decrypt",expectedResult, Cryptographer.decrypt(enfileData, "test"));
}

@Test
public void testEncrypt() throws Exception {
    String xml = readFileAsString(DECRYPTED_DATA_FILE_4_14);
    byte[] encrypted = Cryptographer.encrypt(xml, "test");
    String decrypt = Cryptographer.decrypt(encrypted, "test");
    assertEquals(xml, decrypt);
}
```

Fonte: (PERUMA et al., 2020)

2.4.11 Magic Number Test

Ocorre quando as instruções assert em um método de teste contêm literais numéricos (ou seja, números mágicos) como parâmetros. Os números mágicos não indicam o significado/propósito do número. Portanto, eles devem ser substituídos por constantes ou variáveis, fornecendo assim um nome descritivo para a entrada.

No método *testGetLocalTimeAsCalendar()*[9], os número 15 e 30 são passados como parâmetro nos asserts sem qualquer indicação do significado desses valores.

Figura 12 – *Magic Number Test*

```
@Test
public void testGetLocalTimeAsCalendar() {
    Calendar localTime = calc.getLocalTimeAsCalendar(BigDecimal.valueOf(15.5D), Calendar.getInstance());
    assertEquals(15, localTime.get(Calendar.HOUR_OF_DAY));
    assertEquals(30, localTime.get(Calendar.MINUTE));
}
```

Fonte: (PERUMA et al., 2020)

[8] https://github.com/MarmaladeSky/aRevelation/
[9] https://github.com/Cbsoftware/PressureNet/

2.4.12 Mystery Guest

Ocorre quando um método de teste utiliza recursos externos (por exemplo, arquivos, banco de dados, etc.). O uso de recursos externos em métodos de teste resultará em problemas de estabilidade e desempenho. Os desenvolvedores devem usar objetos simulados no lugar de recursos externos.

Figura 13 – *Mystery Guest*

```
public void testPersistence() throws Exception {
    File tempFile = File.createTempFile("systemstate-", ".txt");
    try {
        SystemState a = new SystemState(then, 27, false, bootTimestamp);
        a.addInstalledApp("a.b.c", "ABC", "1.2.3");

        a.writeToFile(tempFile);
        SystemState b = SystemState.readFromFile(tempFile);

        assertEquals(a, b);
    } finally {
        //noinspection ConstantConditions
        if (tempFile != null) {
            //noinspection ResultOfMethodCallIgnored
            tempFile.delete();
        }
    }
}
```

Fonte: (PERUMA et al., 2020)

No método *testPersistence()*[10] da figura 13, onde o arquivo *tempFile* é criado e em seguida uilizado no processo de teste.

2.4.13 Redundant Print

As instruções de impressão em testes de unidade são redundantes, como no método *testTransform10mNEUAndBack()*[11], pois os testes de unidade são executados como parte de um processo automatizado com pouca ou nenhuma intervenção humana. As instruções de impressão são possivelmente usadas pelos desenvolvedores para fins de rastreabilidade e depuração e depois esquecidas.

Figura 14 – *Redundant Print*

```
@Test
public void testTransform10mNEUAndBack() {
    Leg northEastAndUp10M = new Leg(10, 45, 45);
    Coord3D result = transformer.transform(Coord3D.ORIGIN, northEastAndUp10M);
    System.out.println("result = " + result);
    Leg reverse = new Leg(10, 225, -45);
    result = transformer.transform(result, reverse);
    assertEquals(Coord3D.ORIGIN, result);
}
```

Fonte: (PERUMA et al., 2020)

[10] https://github.com/walles/batterylogger/
[11] https://github.com/richsmith/sexytopo/

2.4.14 Redundant Assertion

Esse *smell* ocorre quando os métodos de teste contêm declarações que são sempre verdadeiras ou sempre falsas, como por exemplo na figura 15. Esse cheiro é introduzido pelos desenvolvedores para fins de depuração e depois esquecido.

Figura 15 – *Redundant Assertion*

```
@Test
public void testTrue() {
    assertEquals(true, true);
}
```

Fonte: (PERUMA et al., 2020)

2.4.15 Resource Optimism

Esse *smell* ocorre quando um método de teste faz uma suposição otimista de que existe um determinado recurso externo (por exemplo, Arquivo).

Como no método *saveImage_noImageFile_ko*[12] da figura 16, o qual acessa um arquivo sem verificar se o arquivo existe antes de utilizá-lo.

Figura 16 – *Resource Optimism*

```
@Test
public void saveImage_noImageFile_ko() throws IOException {
File outputFile = File.createTempFile("prefix", "png", new File("/tmp"));
ProductImage image = new ProductImage("01010101010101", ProductImageField.FRONT, outputFile);
Response response = serviceWrite.saveImage(image.getCode(), image.getField(), image.getImguploadFront(), image.getImguploadIngredients(),
assertTrue(response.isSuccess());
assertThatJson(response.body())
    .node("status")
        .isEqualTo("status not ok");
}
```

Fonte: (PERUMA et al., 2020)

2.4.16 Sensitive Equality

Ocorre quando o método toString é usado em um método de teste, como por exemplo no método *test1()*[13] na figura 17. Os métodos de teste verificam os objetos invocando o método toString() padrão do objeto e comparando a saída com uma string específica. Alterações na implementação de toString() podem resultar em falha. A abordagem correta é implementar um método personalizado dentro do objeto para realizar essa comparação.

[12] https://github.com/Freeyourgadget/Gadgetbridge/
[13] https://github.com/liveplayergames/UFP/

Figura 17 – *Sensitive Equality*

```
@Test
public void test1() throws UnknownHostException {

    String peersPacket = "F8 4E 11 F8 4B C5 36 81 " +
        "CC 0A 29 82 76 5F B8 40 D8 D6 0C 25 80 FA 79 5C " +
        "FC 03 13 EF DE BA 86 9D 21 94 E7 9E 7C B2 B5 22 " +
        "F7 82 FF A0 39 2C BB AB 8D 1B AC 30 12 08 B1 37 " +
        "E0 DE 49 98 33 4F 3B CF 73 FA 11 7E F2 13 F8 74 " +
        "17 08 9F EA F8 4C 21 B0";

    byte[] payload = Hex.decode(peersPacket);

    byte[] ip = decodeIP4Bytes(payload, 5);

    assertEquals(InetAddress.getByAddress(ip).toString(), ("/54.204.10.41"));
}
```

Fonte: (PERUMA et al., 2020)

2.4.17 Sleepy Test

Fazer com que uma thread entre em suspensão explicitamente pode levar a resultados inesperados, pois o tempo de processamento de uma tarefa pode diferir em dispositivos diferentes. Os desenvolvedores introduzem esse cheiro quando precisam pausar a execução de instruções em um método de teste por uma determinada duração (ou seja, simular um evento externo) e continuar com a execução.

O método *testEdictExternSearch()*[14] utiliza o método *Thread.sleep(500)* explicitamente a fim de realizar um atraso para estimular uma atividade do mundo real, causando um *flakiness*.

Figura 18 – *Sleepy Test*

```
public void testEdictExternSearch() throws Exception {
    final Intent i = new Intent(getInstrumentation().getContext(), ResultActivity.class);
    i.setAction(ResultActivity.EDICT_ACTION_INTERCEPT);
    i.putExtra(ResultActivity.EDICT_INTENTKEY_KANJIS, "空白");
        tester.startActivity(i);
        assertTrue(tester.getText(R.id.textSelectedDictionary).contains("Default"));
        final ListView lv = getActivity().getListView();
        assertEquals(1, lv.getCount());
        DictEntry entry = (DictEntry) lv.getItemAtPosition(0);
        assertEquals("Searching", entry.english);
        Thread.sleep(500);
        final Intent i2 = getStartedActivityIntent();
        final List result = (List) i2.getSerializableExtra(ResultActivity.INTENTKEY_RESULT_LIST);
        entry = result.get(0);
        assertEquals("(adj-na,n,adj-no) blank space/vacuum/space/null (NUL)/(P)", entry.english);
        assertEquals("空白", entry.getJapanese());
        assertEquals("くうはく", entry.reading);
        assertEquals(1, result.size());
}
```

Fonte: (PERUMA et al., 2020)

[14] https://github.com/mvysny/aedict/

2.4.18 Unknown Test

Um *assert* é usado para declarar uma condição booleana esperada para um método de teste. Ao examinar a declaração de *assert*, é possível entender o propósito do método de teste. No entanto, é possível que um método de teste seja escrito sem uma declaração de *assert*, em tal instância o JUnit mostrará o método de teste como aprovado se as declarações dentro do método de teste não resultarem em uma exceção, quando executadas. Novos desenvolvedores do projeto terão dificuldade em entender o propósito de tais métodos de teste (mais ainda se o nome do método de teste não for suficientemente descritivo).

Figura 19 – *Unknown Test*

```
@Test
public void hitGetPOICategoriesApi() throws Exception {
    POICategories poiCategories = apiClient.getPOICategories(16);
    for (POICategory category : poiCategories) {
      System.out.println(category.name() + ": " + category);
    }
}
```

Fonte: (PERUMA et al., 2020)

O método *hitGetPOICategoriesApi()*[15] é um exemplo de *Unknown Test*, já que não há um método de asserção, fazendo com que o não seja possível saber o que está sendo testado.

2.5 Relação entre *flaky tests* e *test smells*

Durante a revisão bibliográfica deste trabalho, foi possível observar que diversos estudos fazem uma corelação entre *flaky tests* e *test smells*. Entre os trabalhos relacionados, o artigo "*On the use of test smells for prediction of flaky tests*" (CAMARA et al., 2021) foi um dos estudos que trataram esse tema de forma mais contundente. Como dito anteriormente, acredita-se que seja possível utilizar a presença de *test smells* para detectar *flaky tests*, exatamente o que é proposto em (CAMARA et al., 2021).

Em (CAMARA et al., 2021) algumas relações diretas entre *flaky tests* e *test smells* são traçadas, como as categorias de *flaky tests Concurrency* e *Assync Wait* que são causados por injeção de atrasos como *Thread.Sleep()*, o que está diretamente relacionado a categoria de *Sleepy Test* de *smell*. Com isso, acredita-se que a existência de *Sleepy Test* é capaz de implicar na existência de *Concurrency* e *Assync Wait*.

Além disso, Camara (2021) se preocupa em responder algumas questões relacionados à esse tema. Como a precisão em que é possível prever *flaky tests* com base em *test smells*. Para responder a essa pergunta, um modelo de previsão baseado em *test smells* foi construído, treinado e testado e apresentou uma precisão em torno de 70%.

[15] https://github.com/cyclestreets/android/

Foi realizado também uma análise *test smells* que estão mais fortemente associados à previsão de *flaky tests*. Como resultado, obteve-se que os *test smells* mais associados à *flaky tests* são *Sleepy Test* e *Constructor Initialization*.

2.5.1 Presença de *Tets Smells* como uma forma de detecção de *Flaky Tests*

Além das diversas técnicas de detecção de *flaky tests* relatadas na literatura, as técnicas que utilizam *tests smells* como indicador de *flaky tests* ainda são pouco trabalhadas. O que nos motivou a tratar esse ponto de forma aprofundada neste trabalho.

Segundo Zolfaghari (2021), o aumento de *test smells* é um indicador para detectar *flaky tests*. Garousi (2018) definiu-se cheiros de teste como "testes mal projetados e sua presença pode afetar negativamente o conjunto de testes e código de produção". Além disso, diversos outros estudos, como em (PALOMBA; ZAIDMAN, 2017), afirmam que um aumento nos *smell tests* leva a um aumento de *flaky tests*.

Porém, mesmo com todas essas afirmações, não foi encontrada, até o momento do desenvolvimento deste trabalho, nenhuma ferramenta que utilize *test smells* como um indicador da presença de *flaky tests*. Com isso, notamos a necessidade de estudarmos essa correlação, a fim de comprovar que é possível utilizar a presença de *test smells* como indicador para detectar *flaky tests*.

3 Desenvolvimento

Os *flaky tests* são testes instáveis que implicam em diversas dores, como limitação da confiabilidade do conjunto de teste e tempo gasto com refatoração desnecessária. Existem algumas formas para se tratar *flaky tests*, como detectar, corrigir e mitigar. É possível prever que para corrigir e mitigar *flaky tests* é necessário em um primeiro momento detectá-los. Com isso, o objetivo principal deste trabalho é explorar as ferramentas de detecção de *flaky tests*.

3.1 Levantamento de ferramentas de detecção de *flaky tests*

Mesmo em um estágio inicial, já existem diversas ferramentas e técnicas para detecção de *flaky tests*. Logo, o primeiro passo foi realizar um levantamento das ferramentas que serão exploradas durante o desenvolvimento deste trabalho.

Para isso, utilizamos um levantamento prévio realizado em (PARRY et al., 2021), onde foram citadas diversas ferramentas capazes de detectar diferentes categorias de *flaky tests*. Com base nesse levantamento prévio, realizamos um segundo levantamento com o objetivo de filtrar as ferramentas disponíveis para utilização, pois entre a listagem realizada em (PARRY et al., 2021) existiam ferramentas de domínio privado, descontinuadas e outras que eram apenas conceitos e não uma ferramenta propriamente dita. As ferramentas levantadas estão presentes na Tabela 4.

Nota-se que há um número limitado de ferramentas, porque, após realizar um levantamento inicial, deu-se início à etapa de configuração e utilização das ferramentas. Porém, algumas ferramentas foram descartadas por dificuldades no processo de configuração. Esses problemas foram causados por questões como versionamento de Java Development Kit, limitação de sistemas operacionais etc.

Entre as ferramentas levantadas, a Shaker se destacou no processo de configuração, por ser extremamente simples e intuitiva. A ferramenta Shaker (CORDEIRO et al., 2021) é utilizada para detectar *flaky tests* causados por restrição de tempo. Ela será detalhada na seção a seguir.

3.1.1 Shaker

Shaker é uma ferramenta, desenvolvida por Marcello Cordeiro (2021), para detecção de *flaky tests* causados por restrição de tempo, adicionando ruído ao ambiente de execução. Segundo Cordeiro (2021), a ideia principal por trás do Shaker é adicionar tarefas estressoras que competem com a execução do teste pelo uso de recursos (CPU ou memória).

Tabela 4 – Ferramentas de detecção de flaky tests levantadas.

Ferramenta	**Descrição**	**Categoria**	**Fonte**
NonDex[1]	Utilizada para manifestar ADINs (flaky tests decorrentes da suposição de uma implementação determinística de uma especificação não deterministica). Consiste em uma reimplementação de uma série de métodos e classes na biblioteca padrão Java, que randomiza os elementos não deterministicos de susas respectivas especificações	ADINs ou coleção não ordenada.	(GYORI et al., 2016)
iDFlakies[2]	Classifica os testes flaky como dependentes da ordem ou não com base na comparação dos resultados de execuções repetidas de testes com falha em uma ordem de execução de teste modificada com a ordem de execução de teste original.	Testes dependentes de ordem	(LAM et al., 2019)
Shaker[3]	Uma ferramenta de código aberto para detectar flakiness em testes com restrição de tempo adicionando ruído no ambiente de execução. A idéia principal por trás do SHAKER é adicionar tarefas estressantes que competem com a execução do teste pelo uso de recursos (CPU ou memória)	Testes com restrição de tempo	(CORDEIRO et al., 2021)
DeFlaker[4]	O DeFlaker monitora a cobertura das últimas alterações de código e marca como irregular qualquer teste com falha recente que não executou nenhuma das alterações	Diversas	(BELL et al., 2018)

Fonte: (PARRY et al., 2021) adaptada pelo autor (2022).

Em (CORDEIRO et al., 2021), são citadas outras ferramentas de detecção de *flaky tests* que possuem estratégias de detecção diferentes. Como a DeFlaker (BELL et al., 2018), que também foi estudada neste trabalho, que utiliza a estratégia de monitoração de cobertura de código, como veremos melhor nas próximas seções.

Além disso, Cordeiro (2021) cita e faz um paralelo entre a técnica de detecção *ReRun*, técnica mais popular e simples para detecção de *flaky testes*. O *ReRun* consiste basicamente na reexecução do conjunto de testes inúmeras vezes, onde um teste com falha que passa em uma execução subsequente é considerado *flaky test*.

Apesar da simplicidade, o custo de execução da *ReRun* é alto. O que é de se esperar, já que os conjuntos de testes a serem reexecutados podem ser extensos e, além disso, pode ser necessário um número grande de reexecuções para que um *flaky test* seja identificado. Para reforçar isso, Zolfaghari ((ZOLFAGHARI et al., 2021)) afirma que, na Microsoft, 4,6% dos testes foram identificados como *flaky tests* após o monitoramento de cinco projetos durante o período de um mês. A Google relata que entre 2-16% de seu orçamento de testes é utilizado apenas para repetição de testes (MICCO, 2017). Isso reforça a necessidade de novas técnicas de detecção de *flaky tests*.

O Shaker é uma ferramenta para detecção de *flaky tests* causados por restrição de tempo. Ou seja, as categorias de *flaky tests* que podem ser detectados pelo Shaker são as categorias *Asynchronous Wait*, que, como vimos, são testes que fazem uma chamada

assíncrona e não esperam explicitamente que ela termine antes de avaliar as asserções, geralmente usando um atraso de tempo fixo com métodos como *Thread.sleep()*.

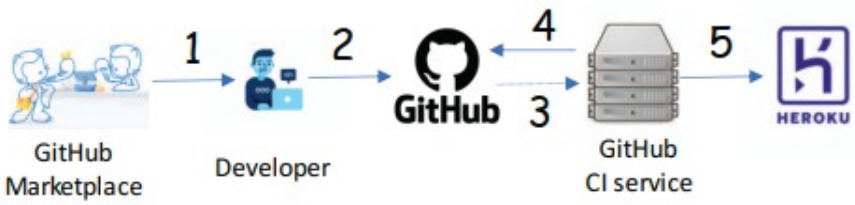
O *Asynchronous Wait* é causado quando o desenvolvedor necessita pausar a execução de instruções por um determinado tempo para simular eventos externos. Para isso, o *Thread.sleep()* é utilizado. Ao introduzir ruído ao ambiente de execução, o Shaker faz com que a execução do teste demore mais do que era esperado pelo desenvolvedor, fazendo com que o teste falhe em algumas execuções e passe em outras.

3.1.1.1 Configuração

O principal fator em que a ferramenta Shaker se destacou foi na facilidade de configuração e utilização. Esta ferramenta foi desenvolvida em Python e é integrada ao GitHub. Para isso, ela conta com o GitHub Actions, um serviço do GitHub que permite aos desenvolvedores automatizarem tarefas dentro do ciclo de vida de desenvolvimento de software (GITHUB, 2022). O código fonte da Shaker está disponível publicamente no GitHub em https://star-rg.github.io/shaker.

A Figura 20 (CORDEIRO et al., 2021) ilustra o fluxo de trabalho associado à execução do GitHub Action da Shaker. Onde na primeira etapa, o desenvolvedor copia o modelo da Action GitHub Shaker para o arquivo .github/workflow/main.yml no repositório do projeto. Na segunda etapa, o desenvolvedor faz uma solicitação *push* ou *pull* para o repositório GitHub. Na terceira etapa, o GitHub notifica seu serviço de CI sobre esse evento. Na etapa 4, o serivço de CI extrai as alterações do *commit* correspondente do GitHub e executa a ação Shaker em um contêiner LINUX que é preparado com uma ferramenta para inserir ruído ao ambiente de execução. Já na quinta etapa, a Shaker notifica um serviço *Web*, hospedado no Heroku para armazenar dados de telemetria sobre a execução.

Figura 20 – Fluxo de trabalho do GitHub Action para o Shaker



Fonte: (CORDEIRO et al., 2021)

O modelo da Action GitHub Shaker que deverá ser copiado para o arquivo .github/workflow/main.yml é ilustrado na figura 21. Onde o usuário pode alterar os parâmetros da cláusula *with*: *tool*, *runs* e *no_stress_run*.

Figura 21 – Modelo da Action GitHub Shaker

```yaml
shaker:
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v2
    - name: Shaker
      uses: STAR-RG/shaker@main
      with:
        tool: maven
        runs: 3
        no_stress_runs: 1
```

Fonte: (CORDEIRO et al., 2021)

Sendo que:

- *Tool*: a ferramenta utilizada para executar os casos de teste. Atualmente, a Shaker suporta projetos Java (baseados em Maven) e Python (baseados em pytest) (CORDEIRO et al., 2021);
- *Runs*: o número de execuções sem estresse no ambiente de execução. Este é um parâmetro opcional e se o usuário não fornecer um valor, nenhuma execução sem estresse será executada (CORDEIRO et al., 2021);
- *No_stress_runs*: o número de execuções para cada configuração de *stress-ng*. Por exemplo, se o usuário digitar 3, como existem quatro configurações diferentes de stress-ng, os testes serão executados 12 vezes no total (CORDEIRO et al., 2021).

Como dito acima, a Shaker conta com quatro configurações de *stress-ng* (KING, 2017). Sendo elas:

- -cpu n: n estressadores são iniciados em uma CPU trabalhando sequencialmente por meio de diferentes métodos de estress, como função de Ackermann ou sequência de Fibonacci (CORDEIRO et al., 2021).
- -cpu-load p: a porcentagem de carregamento p para o comando -cpu (CORDEIRO et al., 2021).
- -vm n: São iniciados n estressadores para alocar e desalocar continuamente a memória (CORDEIRO et al., 2021).

- -vm-bytes *p*: a porcentagem *p* é definida do total de memória disponível para o uso pelas tarefas criadas com a opção -vm (CORDEIRO et al., 2021).

3.1.1.2 GitHub Actions

A integração da Shaker é realizada através do GitHub Actions, plataforma desenvolvida pelo GitHub de integração e entrega contínua (CI/CD) que permite automatizar seu pipeline de compilação, teste e implementação. Ela permite criar fluxos de trabalho que constroem e testam cada *pull request* para seu repositório ou implementam *pull requests* mesclados para produção (GITHUB, 2022).

O GitHub Actions permite a execução dos fluxos de trabalho quando outros eventos acontecem em seu repositório (GITHUB, 2022). No caso da Shaker, o fluxo de trabalho configurado na Figura 21 é acionado sempre que um *commit* é realizado no repositório. A ideia é verificar, a cada *commit*, se a alteração realizada no código inseriu um comportamento *flaky* em algum teste do conjunto de testes.

A utilização do GitHub Actions é ainda não muito explorada. Segundo Kinsman (2021), entre um conjunto de 416.266 repositórios do GitHub, apenas 0,7% utilizam o GitHub Actions. Estes estão distribuídos em 20 categorias, incluindo integração contínua, utilitários e implantação. Além disso, a literatura apresenta poucas evidências sobre o uso do GitHub Actions (GOLZADEH et al., 2021).

Porém, acreditamos que este cenário mudará, já que o GitHub Actions se apresenta como uma maneira fácil, reutilizável e portátil para automatizar os fluxos de trabalho dos desenvolvedores (KINSMAN et al., 2021). Ao utilizarmos a Shaker foi possível observar que esta ferramenta apresentou maior simplicidade na etapa de configuração e isso se deve à utilização do GitHub Actions.

3.1.2 NonDex

NonDex é uma ferramenta desenvolvida por Gyori (2016) para detecção de *flaky tests* causados por suposições erradas em APIs JAVA. Segundo Gyoori (2016), isso acontece, por exemplo, quando a ordem de iteração de um *HashSet* é indeterminada e o código assume alguma ordem de iteração específica da implementação.

Para detectar esse tipo de *flaky tests*, o NonDex explora aleatoriamente diferentes comportamentos de APIs indeterminadas durante a execução do teste e quando um teste falha durante a exploração, o NonDex procura a instância de invocação da API que causou a falha (GYORI et al., 2016).

3.1.2.1 Configuração

O NonDex é disponibilizado como um plugin Maven ou por linha de comando. Devido a maior simplicidade, optamos por utilizá-lo como plugin Maven. O plugin está disponível no Maven Central Repository[5].

É necessário que o projeto em que a ferramenta detectará *flaky tests* utilizem o Maven. Com isso, o primeiro passo é utilizar o comando *mvn install* no repositório do projeto. Após isso, é necessário adicionar o plugin na seção de plugins da seção de compilação no arquivo pom do projeto, como ilustrado na Figura 22.

Figura 22 – Plugin NonDex adicionado ao arquivo pom do repositório.

```
<project>
  ...
  <build>
    ...
    <plugins>
      ...
      <plugin>
        <groupId>edu.illinois</groupId>
        <artifactId>nondex-maven-plugin</artifactId>
        <version>1.1.2</version>
      </plugin>
    </plugins>
  </build>
</project>
```

Fonte: (GYORI et al., 2016)

Após isso, basta executar o comando *mvn nondex:nondex* para descobrir se o seu projeto possui *flaky tests*. Após isso, se existirem *flaky tests* no conjunto de teste a saída no console reportará na seção marcada *"NonDex SUMMARY"*. Além disso, os *flaky tests* são registrados em arquivos chamados *"failure"* no diretório .nondex/.

3.1.3 iDFlakies

O iDFlakies é uma ferramenta desenvolvida por Wing Lam (2019) para detecção de *flaky tests* causados por dependência de ordem. Essa ferramenta tem como objetivo classificar testes como OD (*Order Dependent*) ou NOD (*Non-Order Dependent*). Sendo que os classificados como OD são considerados *flaky tests*.

A estratégia do iDFlakies consiste em executar repetidamente o conjunto de testes com base na configuração especificada pelo usuário (LAM et al., 2019). Existem cinco diferentes configurações, sendo elas:

[5] https://repo1.maven.org/maven2/edu/illinois/nondex-maven-plugin/

- Original-Order: executa repetidamente os testes na ordem original e classifica qualquer teste com falhar como NOD. A configuração não pode detectar testes de OD, porque a ordem é sempre a mesma.
- Random-Class (RandomC): executa repetidamente classe de teste em ordem aleatória, mas mantém os métodos em cada classe na mesma ordem que na ordem original. O Maven Surefire já pode randomizar a ordem das classes de teste, mas não executa o conjunto de teste repetidamente nem classifica os *flaky tests* como OD ou NOD.
- Random-Class-Method (RandomC+M): executa repetidamente métodos de teste em uma ordem aleatória, hierarquicamente randomizando primeiro a ordem das classes de teste e depois os métodos dentro das classes de teste, mas não intercalando métodos de classes diferentes.
- Reverse-Class (ReverseC): inverte a ordem de todas as classes de teste da ordem original, mas mantém os métodos de teste na mesma ordem da ordem original; o iDFlakies executa essa configuração apenas uma vez para limitar o tempo dos experimentos (embora execuções repetidas possam detectar mais alguns testes NOD, mas nenhum novo teste OD).
- Reverse-Class-Method (ReverseC+M): inverte a ordem de todas as classes e métodos de teste da ordem original; semelhante à classe reversa, essa configuração é executada apenas uma vez.

Após executar a ferramenta, ela classificará os testes como OD ou NOD.

3.1.3.1 Configuração

O iDFlakies (LAM et al., 2019) é capaz de trabalhar em projetos baseados em Maven ou Gradle. Além disso, até o momento deste trabalho, está disponível de forma plena apenas para sistemas baseados em Linux.

Neste trabalho, utilizamos o iDFlakies através de uma *Virtual Machine* com uma ISO Ubuntu 18.04 LTS devido à limitação de sistemas operacionais imposta pela ferramenta. Optamos também por trabalhar com a ferramenta em projetos baseados em Maven.

Assim como a NonDex, para configurar a ferramenta é necessário configurar o arquivo pom.xml do projeto. É necessário adicionar o plugin na seção de plugins da seção de compilação no arquivo pom do projeto, como ilustrado na Figura 23.

Após configurar o arquivo pom.xml, é possível executar o iDFlakies através da linha de comando no repositório do projeto. Para executar, executa-se a seguinte linha de comando da Figura 24.

Figura 23 – Plugin iDFlakies adicionado ao arquivo pom do repositório.

```
<build>
    ...
    <plugins>
        ...
        <plugin>
            <groupId>edu.illinois.cs</groupId>
            <artifactId>idflakies-maven-plugin</artifactId>
            <version>2.0.1-SNAPSHOT</version>
        </plugin>
    </plugins>
</build>
```

Fonte: (LAM et al., 2019)

Figura 24 – Linha de comando para execução do iDFlakies.

```
C:\flow>mvn iDFlakies:detect -Ddetector.detector_type=random-class-method -Ddt.randomize.rounds=10 -Ddt.detector.original_order.all_must_pass=false
```

Fonte: Produzida pelo autor (2022)

Onde cada parâmetro na linha acima é uma opção de configuração do iDFlakies. Sendo:

- detector.detector_type: Configuração do iDFlakies.
- dt.randomize.rounds: Número de vezes para executar o conjunto de testes.
- dt.detector.original_order.all_must_pass: Controla se o iDFlakies deve usar uma ordem original de testes onde todos eles passam ou não.
- dt.original.order: Permite especificar ao iDFlakies a lista exata de testes que devem ser executados.

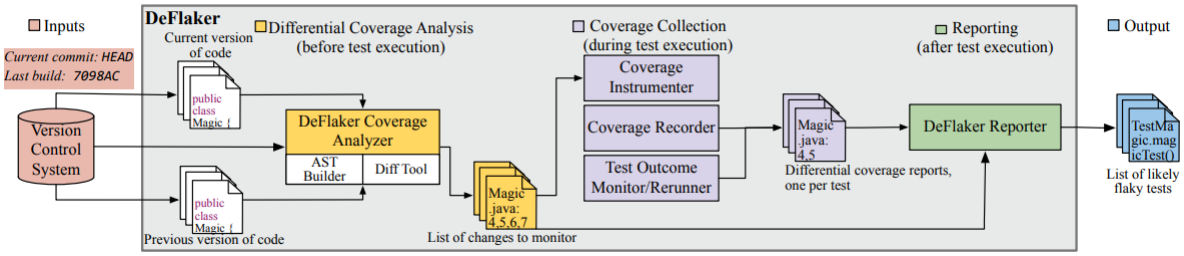
3.1.4 DeFlaker

O DeFlaker (BELL et al., 2018) é a única ferramenta, entre as levantadas, capaz de detectar diferentes categorias de *flaky tests*. Isso porque ela utiliza uma técnica que não é voltada apenas pra uma categoria, como todas as outras ferramentas citadas neste estudo. O DeFlaker utiliza uma estratégia conhecida como monitoramento de cobertura de código, a qual consiste em monitorar a cobertura das últimas alterações de código e marcar como *flaky tests* qualquer teste com falha recente que não executou nenhuma das alterações.

Segundo Bell (2018), para detectar *flaky tests* o DeFlaker utiliza três etapas ilustradas na Figura 25. Onde na primeira fase o DeFlaker utiliza uma diferença sintática

do VCS e um construtor AST para identificar uma lista de alterações a serem rastreadas para cada arquivo de origem do programa. Na segunda etapa, o DeFlaker injeta no processo de execução do teste, monitorando a cobertura de cada mudança na fase anterior. Por fim, depois que os testes terminam a execução, o DeFlaker analisa as informações de cobertura e os resultados do teste para determinar o conjunto de falhas de teste que provavelmente são irregulares.

Figura 25 – Arquitetura de alto nível do DeFlaker, com três fases: antes, durante e depois da execução dos testes.



Fonte: (BELL et al., 2018)

3.1.4.1 Configuração

Assim como as ferramentas anteriores (iDFlakies e NonDex) o DeFlaker é disponibilizado como um plugin Maven. Para configurá-lo no repositório basta adicionar a extensão Maven ao arquivo de compilação, como demonstrado na Figura 26.

Após adicionar a extensão Maven ao arquivo pom.xml, basta executar o comando *mvn test* no repositório do projeto. A saída será gerada através de um arquivo de texto $[basedri}/target/diffcov.log.

Figura 26 – Plugin Maven do DeFlaker adicionado ao arquivo de compilação.

```
<extensions>
 <extension>
  <groupId>org.deflaker</groupId>
  <artifactId>deflaker-maven-extension</artifactId>
  <version>1.4</version>
 </extension>
</extensions>
```

Fonte: (BELL et al., 2018)

3.1.5 Dificuldades encontradas

As principais dificuldades encontradas na configuração das ferramentas foram incompatibilidade de versões, tanto de JDK quanto de sistemas operacionais. As ferramentas que são executadas por linha de comando (iDFlakies, DeFlaker e NonDex) foram inicialmente construídas para sistemas baseados em Linux. Com isso, para utilizá-las foi necessário utilizarmos uma Virtual Machine (VM) com uma imagem do Ubuntu 18.04 LTS.

Algumas dificuldade com versões do Java e do Maven também foram encontradas. O DeFlaker por exemplo, só foi executado com sucesso ao utilizarmos o Java 8. Em (BELL et al., 2018), diz que o DeFlaker está disponível apenas para projetos que utilizam a versão 1.19.1 do Maven Surefire, o que impôs um grande obstáculo no momento de utilizá-la. Pois foi necessário encontrar um projeto que também utilize o Java 8 e o Maven Surefire 1.19.1.

Por isso, os esforços foram voltados de forma majoritária para a ferramenta Shaker, pois devido ao fato de ser uma ferramenta integrada ao GitHub, não impôs nenhuma dificuldade relacionada à sistema operacional e versionamento do Java, Maven ou JUnit, como as outras três ferramentas (DeFlaker, iDFlakies e NonDex) impuseram.

3.2 Levantamento de projetos disponíveis para análise

Após o levantamento de ferramentas de detecção automática de *flaky tests*, foi necessário realizar também um levantamento de projetos a serem utilizados na análise das ferramentas.

Realizar esse levantamento sem um direcionamento seria uma tarefa árdua, pois o GitHub hospeda cerca de 100 milhões de repositórios[6]. Logo, utilizou-se *datasets* fornecidos pelos autores das próprias ferramentas de detecção. Pois todas elas possuem estudos base, os quais relatam toda a construção da ferramenta até a etapa de utilização em projetos reais. A maior parte dos projetos utilizados estão disponíveis em repositórios públicos do GitHub e puderam ser utilizados neste trabalho.

O principal *dataset* utilizado, foi fornecido em (CORDEIRO et al., 2021). O qual é composto por 20 projetos Android e 57 projetos Java. Neste trabalho, focamos os nossos esforços nos projetos Java. Na Figura 27, tem-se alguns repositórios disponibilizados em (CORDEIRO et al., 2021).

Configuramos o Shaker para todos os repositórios da Figura 27, os resultados obtidos serão discutidos no Capitulo 4.

[6] https://www.hostinger.com.br/tutoriais/o-que-github

Figura 27 – Repositórios utilizados em (CORDEIRO et al., 2021).

Repository	Ref	Tests	Stars
Azure/azure-iot-sdk-java	a9226a5	4563	153
CorfuDB/CorfuDB	b99ecff	954	541
OpenHFT/Chronicle-Queue	bec195b	408	2.3k
soabase/exhibitor	d345d2d	52	1.7k
vaadin/flow	6.0.6	4679	304
apache/hbase	d50816f	6024	4k
intuit/karate	09bc49e	529	4.7k
killbill/killbill	killbill-0.22.21	1828	2.3k
mock-server/mockserver	b1093ef	3532	3.2k
apache/ozone	dfd2aaf	1900	359
RipMeApp/ripme	19ea20d	247	2.4k

Fonte: (CORDEIRO et al., 2021)

3.3 Relação entre flaky tests e test smells

Segundo Zolfaghari (2021), o aumento de *test smells* é um indicador para detectar *flaky tests*. Em (GAROUSI; KÜÇÜK, 2018), *test smells* são definidos como "testes mal projetados e a sua presença pode afetar negativamente a qualidade dos conjuntos de teste e código de produção". De acordo com alguns artigos de pesquisa, como em (PALOMBA; ZAIDMAN, 2017), um aumento nos *test smells* pode levar ao aumento de *flaky tests*.

Com isso, notamos a importância de acrescentarmos o estudo dessa relação neste trabalho. Para isso, contamos com o tsDetect (PERUMA et al., 2020), ferramenta de detecção automática de *test smells*, com o objetivo de identificar *test smells* em repositórios que encontrássemos *flaky tests*. A fim de comprovar a relação entre ambos os temas.

3.3.1 tsDetect

O tsDetect é implementado como uma ferramenta de código aberto baseada em linha de comando que está disponível como um arquivo Java autônomo (PERUMA et al., 2020). Na Figura 28, tem-se uma representação da arquitetura do tsDetect.

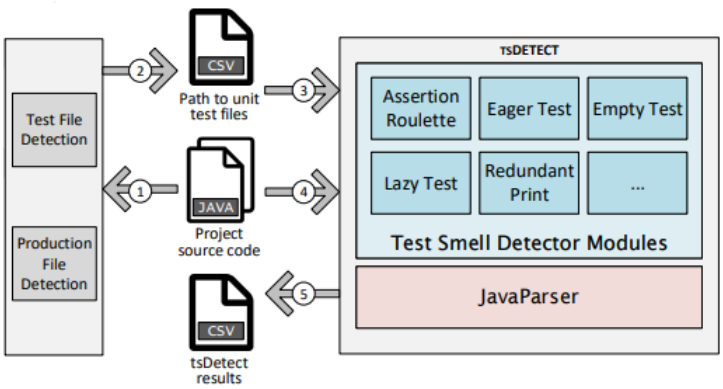
Na primeira e na segunda etapa, os arquivos de teste e produção são identificados a partir da estrutura do projeto. Já na terceira e na quarta, o tsDetect verifica se os arquivos de teste exibem *test smells*. Na última etapa, os resultados do processo de detecção de *test smells* são salvos.

3.3.2 Configuração

Para utilizar o tsDetect, basta realizar o download do arquivo .jar disponibilizado no repositório GitHub do tsDetect[7]. Após realizar o download, para facilitar, movemos o arquivo .jar para o repositório do projeto no qual tentaremos identificar *test smells*.

[7] https://github.com/TestSmells/TestSmellDetector/releases

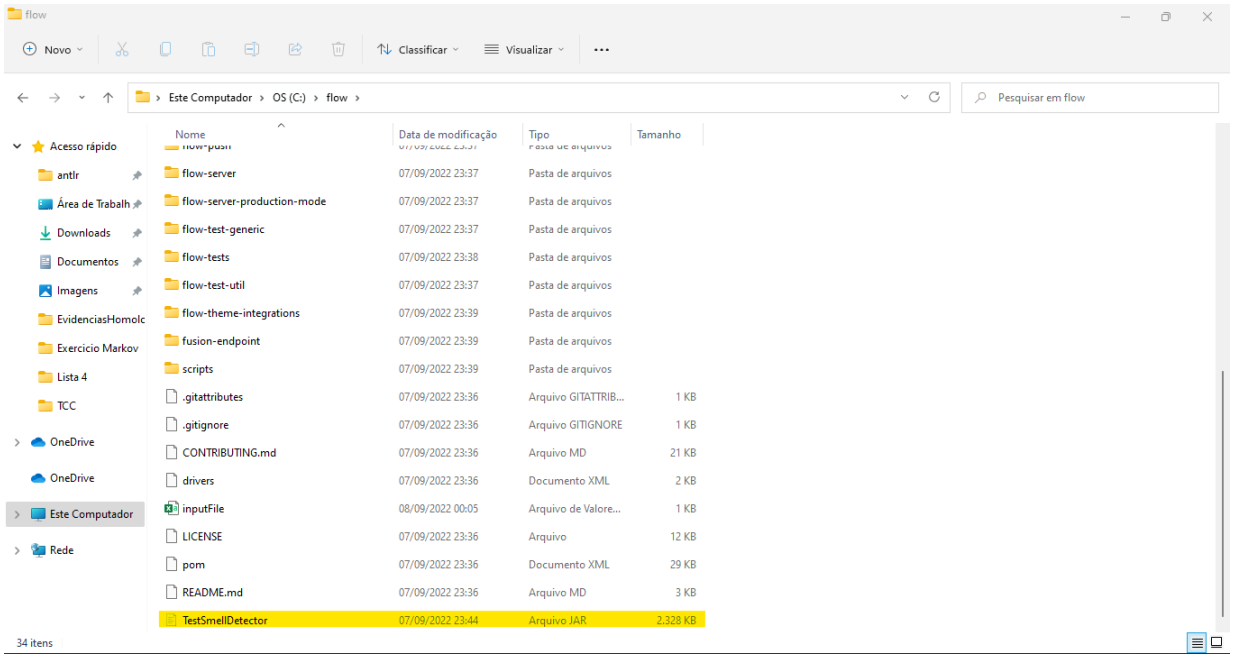
Figura 28 – Arquitetura em alto nível do tsDetect.



Fonte: (PERUMA et al., 2020)

Utilizamos o tsDetect no projeto Vaadin Flow[8]. O primeiro passo foi clonar o repositório. Após isso, movemos o arquivo TestSmellDetector.jar para o repositório do projeto, como podemos observar na Figura 29.

Figura 29 – Arquivo TestSmellDetector.jar no diretório do projeto.



Fonte: Produzida pelo autor (2022)

Após isso, é necessário configurar um arquivo CSV que será utilizado como entrada do tsDetect. O arquivo CSV deverá conter a seguinte estrutura:

[8] https://github.com/vaadin/flow

appName,pathToTestFile,pathToProductionFile

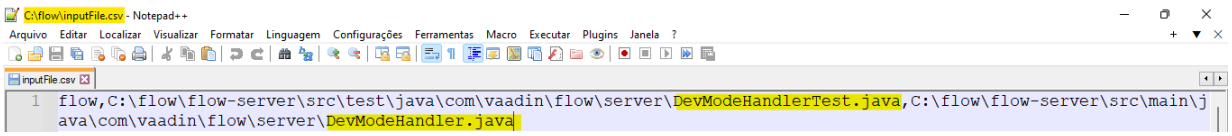
Para o projeto Vaadin/Flow, configuramos o arquivo inputFile.csv também presente no diretório do projeto, como podemos ver na Figura 29. O conteúdo do arquivo inputFile.csv é descrito na Figura 30. Nota-se que o arquivo inputFile.csv está configurado para analisar o módulo de teste DevModeHandlerTest.java.

Após configurar o arquivo CSV, é necessário executar a ferramenta através de linha de comando. O comando para executar o tsDetect segue o seguinte padrão:

java -jav .\TestSmellDetector.jar pathToInputFile.csv

Após executar o tsDetect, um arquivo CSV contendo os resultados da execução é gerado e armazenado no repositório. O arquivo contém o caminho dos arquivos de teste junto com o status de detecção de cada *smell*.

Figura 30 – Arquivo inputFile.csv.



Fonte: Produzida pelo autor (2022)

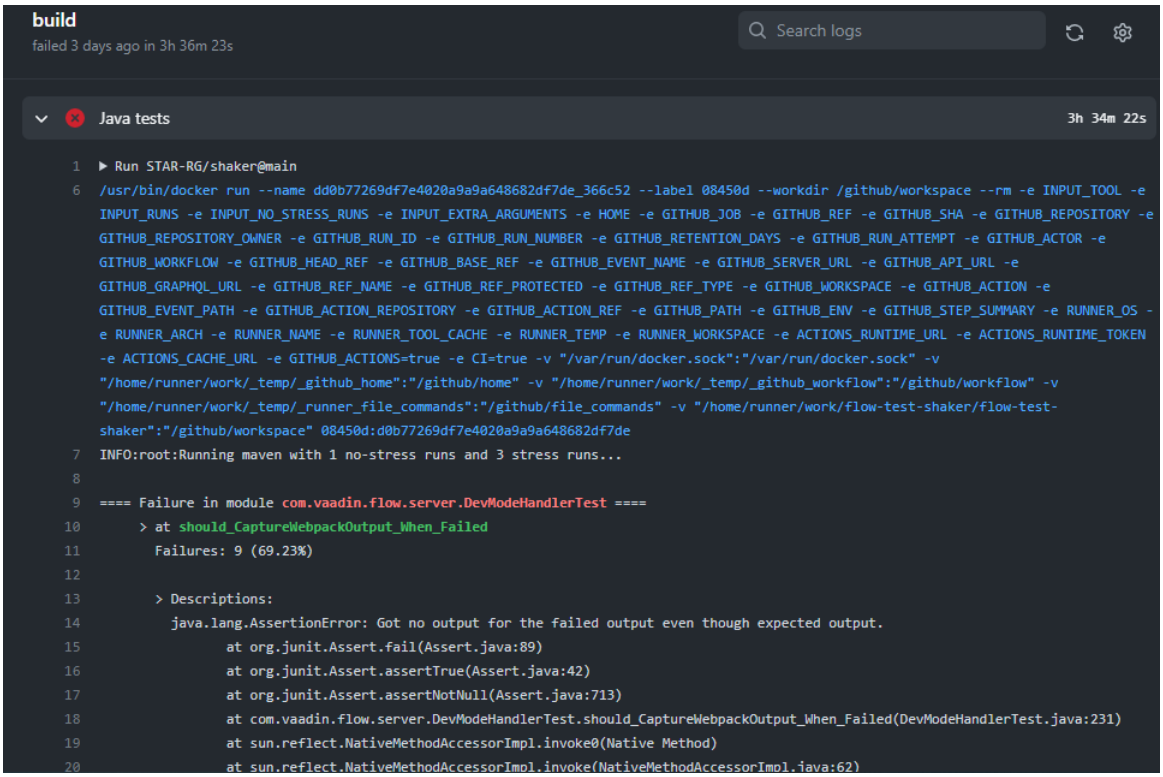
Com isso, utilizamos o tsDetect como uma forma de detectar *test smells* em projetos que já detectamos *flaky tests*, a fim de comprovar a correlação entre ambos. O módulo **DevModeHandlerTest.java** apresentou *flaky tests* quando executamos a ferramenta Shaker e, após isso, apresentou um número considerável de *test smells* quando executamos o tsDetect. Os resultados obtidos serão discutidos no próximo capítulo.

4 Resultados

Realizamos o estudo de quatro ferramentas para detecção de *flaky tests*, sendo elas: NonDex, iDFlakies, Shaker e DeFlaker. Entre elas, focamos os nossos esforços na Shaker, devido a simplicidade na configuração e utilização. Utilizamos a Shaker em diversos projetos disponíveis no GitHub, os quais foram previamente levantados em (CORDEIRO et al., 2021). Já nas outras ferramentas não nos aprofundamos, devido à complexidade da utilização. Nos limitamos à configurá-las e utilizá-las de forma não extensiva em um número limitado de projetos.

O primeiro projeto a apresentar a presença de *flaky tests*, foi o vaadin/flow [1]. Como podemos ver na Figura 31. Nota-se que o Shaker foi configurado para realizar quatro execuções, sendo elas: três com inserção de estresse e uma sem. Como já foi dito anteriormente, o Shaker possui quatro diferentes configurações de estresse, logo espera-se que quatro execuções pra cada uma com estresse seja realizada mais uma sem, o que resulta em um total de 13 execuções.

Figura 31 – Resultado da execução do Shaker no projeto Vaadin/flow.



Fonte: Produzida pelo autor (2022).

[1] https://github.com/vaadin/flow

Nota-se que entre as 13 execuções realizadas, nove falharam (69,23%) e quatro não (30,77%). Esse comportamento não determinístico é exatamente o que se espera de um *flaky test*. Logo, o módulo **should_CaptureWebpackOutput_When** é categorizado como um *flaky test*.

Um ponto importante a ressaltar é o fato de que a saída dada pelo Shaker precisa ser interpretada, para concluir se um teste é *flaky* ou não. No exemplo acima, temos um *flaky test* pois ele possui um comportamento não determinístico. Já no módulo **org.corfudb.infrastructure.LogUnitServerTeste** do projeto CorfuDB/CorfuDB[2], todas as 13 execuções falharam, logo não é possível afirmarmos que esse módulo possui *flaky test*, pois mesmo que ele tenha falhado, o seu comportamento é determinístico.

Figura 32 – Resultado da execução do Shaker no projeto CorfuDB/CorfuDB.

```
==== Failure in module org.corfudb.infrastructure.LogUnitServerTest ====
    > at cantOpenReadOnlyLogFiles
      Failures: 13 (100.00%)

      > Descriptions:
        java.lang.AssertionError: Should have failed to startup in read-only mode
              at org.junit.Assert.fail(Assert.java:88)
              at org.corfudb.infrastructure.LogUnitServerTest.cantOpenReadOnlyLogFiles(LogUnitServerTest.java:186)
              at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
              at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
              at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
              at java.lang.reflect.Method.invoke(Method.java:498)
              at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.java:50)
              at org.junit.internal.runners.model.ReflectiveCallable.run(ReflectiveCallable.java:12)
              at org.junit.runners.model.FrameworkMethod.invokeExplosively(FrameworkMethod.java:47)
              at org.junit.internal.runners.statements.InvokeMethod.evaluate(InvokeMethod.java:17)
              at org.junit.internal.runners.statements.RunBefores.evaluate(RunBefores.java:26)
              at org.junit.internal.runners.statements.RunAfters.evaluate(RunAfters.java:27)
              at org.corfudb.AbstractCorfuTest$1$1.lambda$evaluate$0(AbstractCorfuTest.java:95)
              at java.lang.Thread.run(Thread.java:750)
```

Fonte: Produzida pelo autor (2022).

Executamos o Shaker em nove projetos que foram previamente levantados em (CORDEIRO et al., 2021). Todos estão presentes na Tabela 5.

Entre os respositórios da Tabela 5, o Shaker só foi capaz de identificar *flaky tests* no projetos intuit/karate e vaadin/flow, diferente do que ocorreu em (CORDEIRO et al., 2021), onde foi detectado *flaky tests* em todos os respositórios. Isso se deve a configuração utilizada nos dois estudos. Na Tabela 6, é possível observar que para todos os projetos o número de execuções com estresse estabelecida é alto. O que não foi possível refletir nos nossos estudos.

A configuração utilizada para todos os projetos da Tabela 5 seguiu o padrão de três execuções com estresse e uma sem. Isso porque o Shaker foi configurado para ser utilizado de forma integrada ao GitHub, através do GitHub Actions. A execução do Shaker

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

Tabela 5 – Projetos em que o Shaker foi executado.

Repositório	**Ref**	**Testes**	**Estrelas**
CorfuDB/CorfuDB	a9226a5	4563	153
soabase/exhibitor	d345d2d	52	1.7k
vaadin/flow	6.0.6	4679	304
apache/hbase	d50816f	6024	4k
intuit/karate	09bc49e	529	4.7k
killbill/killbill	killbill-0.22.21	1828	2.3k
RipMeApp/ripme	19ea20d	247	2.4k

Fonte: (CORDEIRO et al., 2021)

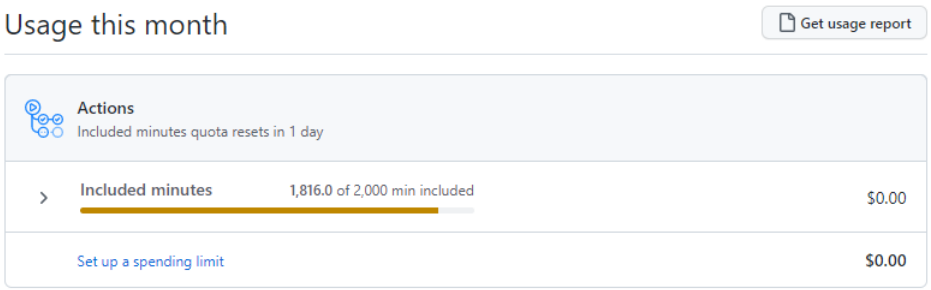
Tabela 6 – *Flaky tests* identificados para cada configuração do Shaker.

Repositório	**Flakies**	**sem estresse**	**com estresse**
soabase/exhibitor	3	1	34
apache/hbase	3	1	19
RipMeApp/ripme	9	2	55

Fonte: (CORDEIRO et al., 2021)

é responsabilidade do GitHub, que utiliza alocação de conteiners LINUX para simular o ambiente de execução de testes com e sem ruído. Ou seja, O GitHub precisa alocar recursos internos para executar a ferramenta. O GitHub fornece recursos limitados às contas básicas (gratuitas), incluindo apenas 2.000 minutos mensais de execução de GitHub Actions.

Figura 33 – Limite de utilização do GitHub Actions.



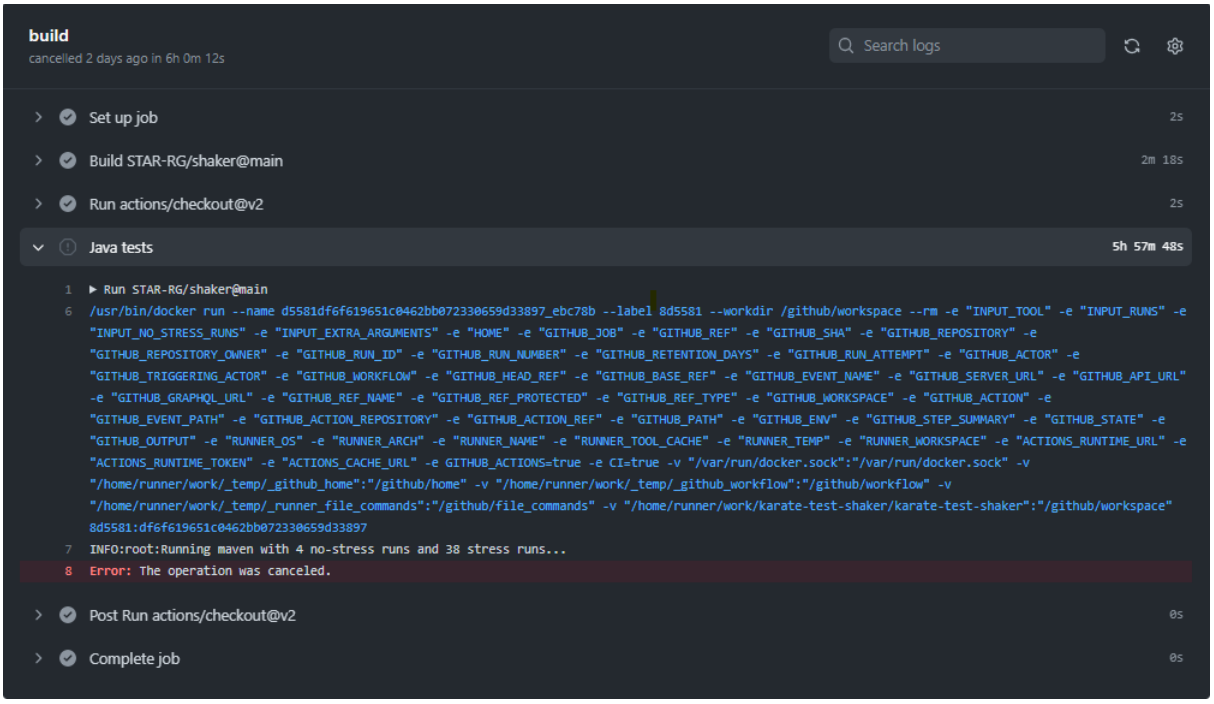
Fonte: Produzida pelo autor (2022).

Além disso, uma action do GitHub Action pode ter um tempo de execução de no máximo 6 horas. Se ele atingir esse limite, ele será encerrado e não concluído[3]. Como

[3] https://docs.github.com/en/actions/learn-github-actions/usage-limits-billing-and-administration

aconteceu para o projeto intuit/karate, quando configuramos o Shaker para realizar 38 execuções com inserção de estresse e 4 sem. Como podemos ver na Figura 34.

Figura 34 – Action cancelada devido o limite de tempo de execução.



Fonte: Produzida pelo autor (2022).

Com isso, não foi possível configurar o Shaker para realizar uma quantidade de execuções similar à quantidade executada em (CORDEIRO et al., 2021). Nos limitamos a 4 execuções, sendo 3 com estresse e 1 sem. Acredita-se que devido à isso, nem todos os *flaky tests* identificados em (CORDEIRO et al., 2021) foram identificados neste estudo.

Após realizar a detecção de *flaky tests* através do Shaker, a fim de comprovar uma possível corelação entre *flaky tests* e *test smells*, realizamos a detecção de *test smells* nos projetos que apresentaram *flaky tests*. Para isso, utilizamos a ferramenta tsDetect (PERUMA et al., 2020).

O primeiro projeto foi o vaadin/flow. Para realizar a detecção de test smells, criamos um arquivo *inputFile.csv* com a seguinte linha de configuração da Figura 35. Esta configuração indica o nome do projeto, o módulo de teste que desejamos detectar *test smells* e a classe que está sendo testada.

Após isso, realizou-se a execução do tsDetect. Ao fim da execução, um arquivo csv é gerado no mesmo repositório em que o arquivo TestSmellDetector.jar está armazenado. Na Tabela 7, tem-se a relação de todos os test smells detectados no módulo *con.vaadin.flow.server.DevModeHandlerTest*. Existem 218 *test smells* presente nesse mó-

Figura 35 – Configuração do arquivo de entrada do tsDetect.

```
flow,C:\flow\flow-server\src\test\java\com\vaadin\flow\server\DevModeHandlerTest.java,C:\flow\flow-server\src\main\j
ava\com\vaadin\flow\server\DevModeHandler.java
```

Fonte: Produzida pelo autor (2022).

dulo para apenas 46 métodos. Isso vai de encontro com a ideia de (PALOMBA; ZAIDMAN, 2017), que diz que um aumento de *test smells* pode levar ao aumento de *flaky tests*.

Tabela 7 – *Test Smells* detectados.

Test Smell	**Quantidade**
Assertion Roulette	6
Conditional Test Logic	4
Exception Catching Throwing	26
General Fixture	33
Mystery Guest	1
Sleepy Test	3
Eager Test	23
Lazy Test	74
Duplicate Assert	2
Unknown Test	3
Ignored Test	1
Resource Optimism	8
Magic Number Test	34

Fonte: (CORDEIRO et al., 2021)

Além disso, acredita-se que essa relação entre *flaky tests* e *test smells* pode ser mais direta e determinística, ou seja, acredita-se que a presença de determinados *test smells* pode implicar na presença de determinados *flaky tests*.

Sabemos que as categorias de *flaky tests Concurrency* e *Async Wait* são causados por injeção de atrasos através do *Thread.Sleep*(), o que está diretamente ligado ao *Sleepy Test*. Logo, acredita-se que a existência de *Sleepy Test* é capaz de implicar na existência de *Concurrency* e *Async Wait*.

Como dito anteriormente, o Shaker utiliza uma estratégia para detecção de *flaky tests* capaz de detectar apenas a categoria *Async Wait*. Logo, o *flaky test* detectado no módulo *DevModeHandlerTest* do projeto vaadin/flow é um *flaky test* da categoria *Async Wait*. Ao realizarmos a detecção de *test smells* neste módulo, foram detectados três *Sleepy Tests*, o que reforça a nossa ideia. Foram detectados também diversas outras categorias, porém, elas não estão relacionadas à tempo de execução assim como a *Sleepy Test*.

O *test smell* é um tema antigo na Engenharia de Software, já foi amplamente estudado e é abordado em diversos estudos na literatura. Existem diversas formas e até

mesmo ferramentas para correção de *test smells*, todas elas baseadas em refatoração de código. Com isso, acredita-se que se os *test smells* realmente implicam na existência de *flaky test*, a correção de *test smell* poderá implicar também na correção de um *flaky test*.

Para comprovar essa ideia, buscamos ferramentas de correção automática ou semi-automática de *test smells*. Em (ALJEDAANI et al., 2021) encontramos uma relação dessas ferramentas. Na Tabela 8, tem-se todas as ferramentas levantadas e estudadas neste trabalho.

Tabela 8 – Ferramentas para correção automática de *test smells* .

Ferramenta	**Linguagem**	**Interface**	**Fonte**
DARTS	Java	IntelliJ Plugin	(LAMBIASE, 2022)
Raide	Java	Eclipse Plugin	(SANTANA, 2022)
RTj	Java	Linha de comando	(MARTINEZ, 2022)
TestHound	Java	Aplicação	(ZAIDMAN, 2022)

Fonte: (ALJEDAANI et al., 2021)

Porém nenhuma das ferramentas acima é capaz de refatorar *Sleepy Test*. Isso porque ele é causado pelo uso de atrasos para simular uma atividade do mundo real e refatorá-lo não é uma tarefa simples. Logo, não foi possível aplicar nenhuma das ferramentas da Tabela 8 para refatorar os *Sleepy Tests*.

A primeira hipótese levantada, para comprovar a relação entre o *Sleepy Test* e o *Async Wait* consiste em aumentar o tempo de atraso utilizado no teste, afim de fazer com que mesmo após a inserção de ruído pelo Shaker, a caraterística *flakiness* do teste não seja detectada.

O método de teste do projeto vaadin/flow que apresentou presença de *flaky test* após a execução do Shaker está representado na Figura 36. Nota-se que na linha 230 há a presença de *Sleepy Test*, isso porque um atraso de 350ms é forçado através do método *Thread.sleep()*.

Ao executarmos o Shaker, temos o resultado da Figura 31. Onde o método **should_CaputreWebpackOutput_When_Failed** falha em 69,23% das execuções. Após isso, alteramos o valor de atraso no método *Thread.sleep()* para 1.000ms e realizamos o *commit* no repositório para ativar a *action* do Shaker.

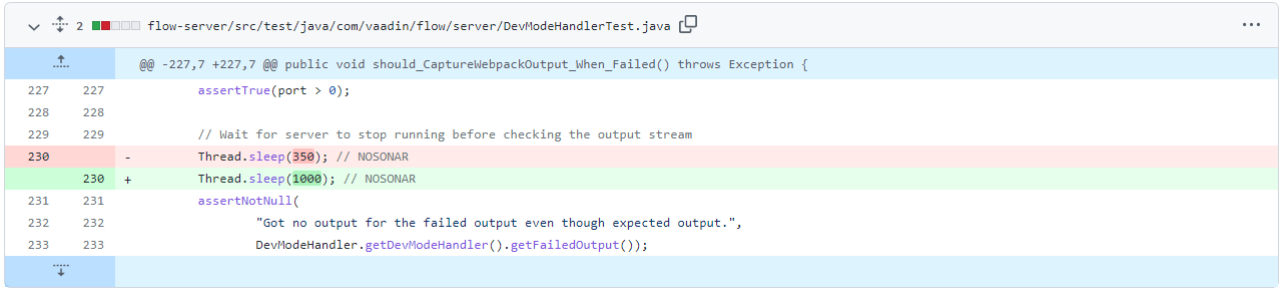
Após essa alteração, o Shaker não foi capaz de detectar nenhum *flaky test*, como podemos observar na Figura 38. Isso comprova a nossa ideia de que o *flaky test* da categoria *Async Wait*, os quais podem ser detectados pelo Shaker, está diretamente ligado ao *Sleepy Test*. Porém, demonstra também uma fragilidade da ferramenta, já que a característica *flakiness* do método **should_CaputreWebpackOutput_When_Failed** não deixa de existir ao aumentarmos o valor do atraso, porém o Shaker não é capaz de detectá-lo.

Figura 36 – Método de teste com presença de *flaky test* do tipo *Async Wait* e *test smell* do tipo *Sleepy Test*.

```
217     @Test
218     public void should_CaptureWebpackOutput_When_Failed() throws Exception {
219         configuration.setApplicationOrSystemProperty(
220                 SERVLET_PARAMETER_DEVMODE_WEBPACK_TIMEOUT, "100");
221         createStubWebpackServer("Failed to compile", 300, baseDir);
222         DevModeHandler handler = DevModeHandler.start(createDevModeLookup(),
223                 npmFolder, CompletableFuture.completedFuture(null));
224         assertNotNull(handler);
225         handler.join();
226         int port = DevModeHandler.getDevModeHandler().getPort();
227         assertTrue(port > 0);
228
229         // Wait for server to stop running before checking the output stream
230         Thread.sleep(350); // NOSONAR
231         assertNotNull(
232                 "Got no output for the failed output even though expected output.",
233                 DevModeHandler.getDevModeHandler().getFailedOutput());
234
235     }
```

Fonte: Produzida pelo autor (2022).

Figura 37 – Alteração realizada no método.



Fonte: Produzida pelo autor (2022).

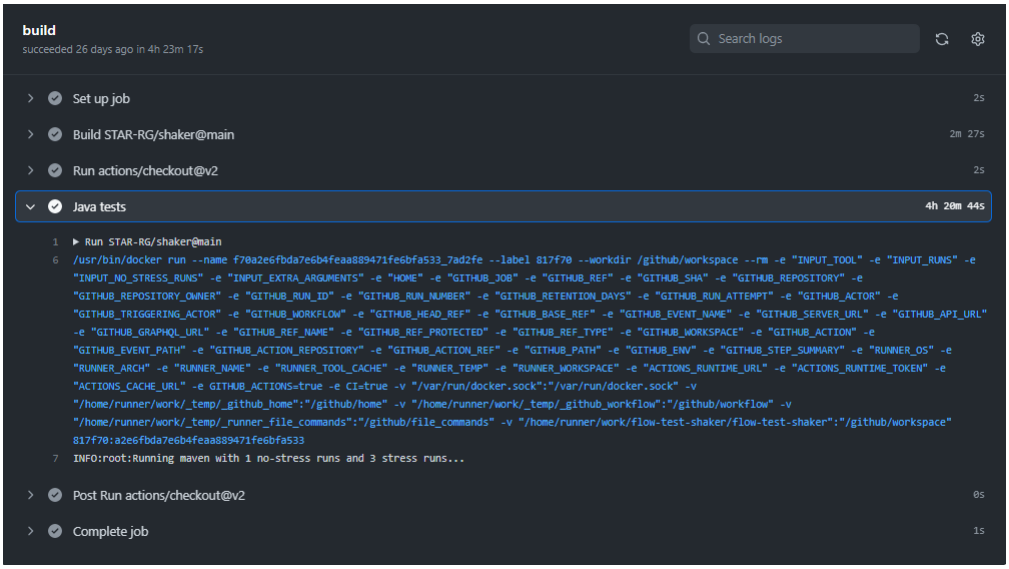
A segunda hipótese consiste em refatorar o *Sleepy Test* e após isso validar se a característica *flakiness* do método permanece. Espera-se que não, pois se se o *flaky test* e o *test smell* estão diretamente ligados, solucionar um implica em solucionar o outro.

Não encontramos nenhuma ferramenta disponível, capaz de refatorar de forma automática ou semi-automática o *Sleepy Test*. Com isso, foi necessário realizar a refatoração de forma manual. Para isso, utilizamos uma DSL (Domain-specific language) conhecida como *Awaility*[4], a qual permite expressar expectativas de sistemas assíncronos de maneira concisa e fácil de ler. A alteração realizada está representada na Figura 39.

Nota-se que o *Thread.sleep()* foi substituído pelo método *Awaitly.await.atMost()*. O qual repete a verificação do *assert* várias vezes até que a condição seja verdadeira. Com

[4] http://www.awaitility.org/

Figura 38 – Execução do Shaker após aumento no tempo de suspensão da thread.



Fonte: Produzida pelo autor (2022).

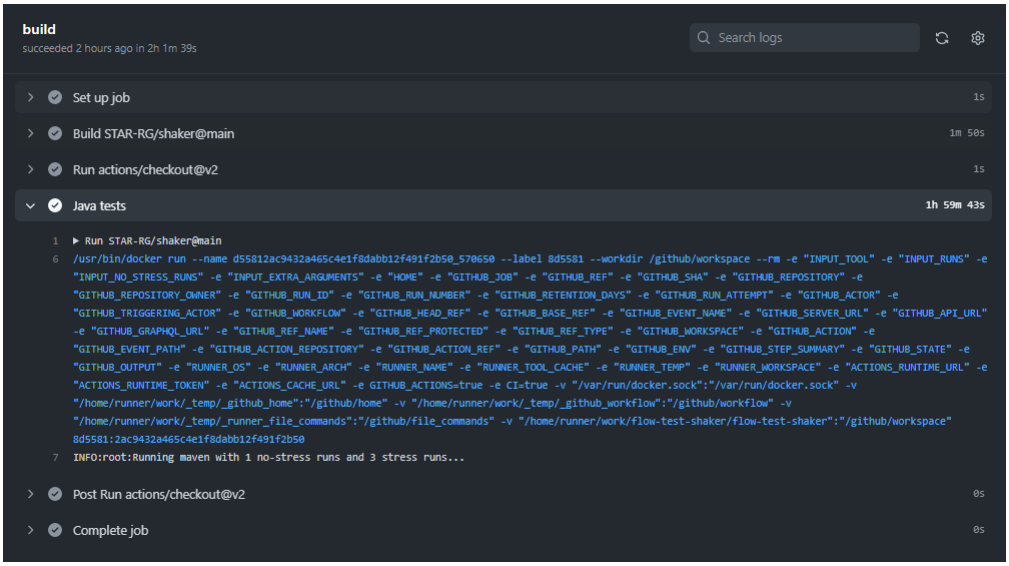
Figura 39 – Correção do *Sleepy Test*.

Fonte: Produzida pelo autor (2022).

isso, ao invés de fazer com que o método entre em suspensão por um determinado tempo antes de realizar a asserção, realiza-se várias asserções durante um determinado período de tempo, se a asserção for verdadeira durante esse tempo o teste é realizado com sucesso, caso nenhuma execução do *assert* seja verdadeira dentro do tempo delimitado, o teste falha.

Após realizar a alteração e efetuar o commit, o Shaker foi executado e nenhum *flaky test* foi detectado. Como podemos ver na Figura 40. Com isso, podemos afirmar que o fator causador do *flaky test* neste projeto é a utilização de esperas estáticas, o que configura o *Sleepy Test*.

Figura 40 – Execução do Shaker após a correção do *Sleepy Test*.



Fonte: Produzida pelo autor (2022).

Acredita-se que existem outras relações fortes entre *flaky tests* e *test smells*, assim como a relação entre *Async Wait* e *Sleepy Test*. Essa correlação entre os temas ainda é pouco explorado, acreditamos que isso possa ser tema de trabalhos futuros.

5 Considerações Finais

É essencial garantir a qualidade e a confiabilidade do conjunto de testes de software. Para isso, é necessário garantirmos que não haja a presença de *flaky tests* neste conjunto. Neste estudo, buscamos estudar os *flaky tests*, apresentar as diversas categorias existentes e explorar também ferramentas para a detecção e o reparo desses testes.

Importante ressaltar que tínhamos como objetivo também realizar um estudo comparativo entre ferramentas para detecção de *flaky tests*. Porém, ao realizar o levantamento de ferramentas disponíveis, notamos que elas possuem propósitos e estratégias muito diferentes, logo, seria inviável realizarmos uma comparação direta entre elas. Por exemplo, existem ferramentas que utilizam a estratégia de re-execução do conjunto de testes, o que demanda tempo e esforço computacional, mas existem também ferramentas que utilizam a análise de cobertura de código que é uma alternativa para solucionar o custo computacional das ferramentas que utilizam reexecução. Assim, as métricas coletadas para cada uma das ferramentas divergiriam, o que não nos traria conclusões concretas. Além disso, como o estudo sobre *flaky tests* está em uma fase inicial, as ferramentas existentes ainda são difíceis de encontrar, configurar e utilizar, o que dificultaria o nosso estudo.

Por fim, durante a realização do estudo acrescentamos a análise da correlação entre *flaky tests* e *test smells*. Para constatar essa correlação, utilizamos as ferramentas Shaker e tsDetect para detecção automática de *flaky tests* e *test smells*, respectivamente. Com isso, conseguimos comprovar que o *test smell Sleepy Test* é capaz de implicar no *flaky test Async Wait*.

5.1 Trabalhos Futuros

Como vimos, por meio deste estudo foi possível constatar a correlação entre *Sleep Test* e *flaky tests* da categoria *Async Wait*. Acredita-se que essa correlação é mais ampla e outras categorias de *flaky tests* e *test smells* estão interligadas. Porém, para comprovar isso seria necessário expandir este estudo, buscando investigar ferramentas de detecção de *flaky tests* capazes de detectar outras categorias.

Neste trabalho, ficamos limitados à categoria *Async Wait* de *flaky test*, pois a ferramenta utilizada possui uma técnica de detecção, a qual é capaz de detectar apenas essa categoria. Logo, para sermos capazes de trabalhar com outras categorias, será necessário buscar outras ferramentas. Como por exemplo, ferramentas que utilizam a análise de cobertura de código, pois essa técnica é uma das poucas capazes de detectar diversas categorias de *flaky tests*. Com isso, será possível relacionar outras categorias de *flaky tests*

à outros tipos de *test smells*.

Neste estudo, tivemos indícios da comprovação de que *test smells* são capazes de implicar *flaky tests*. Com isso, acredita-se que possam ser desenvolvidas ferramentas que utilizam a detecção de *test smells* como meio de detecção de *flaky tests*. Para isso, é necessário um estudo mais aprofundado dessa correlação, pois como dito, este estudo foi capaz de constatar apenas a correlação entre o *flaky tests* da categoria *Async Wait* e o *test smell* da categoria *Sleepy Test*.

Além disso, acredita-se que seja possível e necessário realizar uma revisão sistemática e comparativa entre os estudos que abordam as ferramentas de detecção de *flaky tests*. Neste estudo, analisamos separadamente cada um desses trabalhos abordando separadamente o processo de configuração e resultados obtidos de cada uma das ferramentas. Uma revisão sistemática e comparativa desses trabalhos pode nos trazer conclusões e *insights* importantes no estudo das ferramentas.

Referências

ALJEDAANI, W. et al. Test smell detection tools: A systematic mapping study. *Evaluation and Assessment in Software Engineering*, p. 170–180, 2021. Citado na página 54.

BELL, J. et al. Deflaker: Automatically detecting flaky tests. In: IEEE. *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. [S.l.], 2018. p. 433–444. Citado 5 vezes nas páginas 22, 37, 43, 44 e 45.

CAMARA, B. et al. On the use of test smells for prediction of flaky tests. In: *Brazilian Symposium on Systematic and Automated Software Testing*. [S.l.: s.n.], 2021. p. 46–54. Citado 2 vezes nas páginas 13 e 34.

CORDEIRO, M. et al. Shaker: a tool for detecting more flaky tests faster. In: IEEE. *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. [S.l.], 2021. p. 1281–1285. Citado 13 vezes nas páginas 8, 36, 37, 38, 39, 40, 45, 46, 49, 50, 51, 52 e 53.

DELAMARO, M.; JINO, M.; MALDONADO, J. *Introdução ao teste de software*. [S.l.]: Elsevier Brasil, 2013. Citado na página 17.

DEURSEN, A. V. et al. Refactoring test code. In: CITESEER. *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*. [S.l.], 2001. p. 92–95. Citado na página 23.

DEVI, T. R. Importance of testing in software development life cycle. *International Journal of Scientific & Engineering Research*, v. 3, n. 5, p. 1–5, 2012. Citado na página 13.

ECK, M. et al. 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*. [S.l.: s.n.], 2019. p. 830–840. Citado 3 vezes nas páginas 19, 20 e 21.

GAROUSI, V.; KÜÇÜK, B. Smells in software test code: A survey of knowledge in industry and academia. *Journal of systems and software*, Elsevier, v. 138, p. 52–81, 2018. Citado 2 vezes nas páginas 35 e 46.

GITHUB. *GitHub Actions*. 2022. Http://github.com/features/actions. Citado 2 vezes nas páginas 38 e 40.

GOLZADEH, M. et al. A ground-truth dataset and classification model for detecting bots in github issue and pr comments. *Journal of Systems and Software*, Elsevier, v. 175, p. 110911, 2021. Citado na página 40.

GRUBER, M. et al. An empirical study of flaky tests in python. In: IEEE. *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. [S.l.], 2021. p. 148–158. Citado na página 19.

GYORI, A. et al. Nondex: A tool for detecting and debugging wrong assumptions on java api specifications. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. [S.l.: s.n.], 2016. p. 993–997. Citado 4 vezes nas páginas 22, 37, 40 e 41.

JR, H. E. *Engenharia de software na prática*. [S.l.]: Novatec Editora, 2010. Citado na página 13.

KAPFHAMMER, G. *The Computer Science Handbook, chapter Software Testing*. [S.l.]: CRC Press, Boca Raton, FL„ 2004. Citado na página 20.

KING, C. I. Stress-ng. *URL: http://kernel. ubuntu. com/git/cking/stressng. git/(visited on 28/03/2018)*, 2017. Citado na página 39.

KINSMAN, T. et al. How do software developers use github actions to automate their workflows? In: IEEE. *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. [S.l.], 2021. p. 420–431. Citado na página 40.

LAM, W. et al. A study on the lifecycle of flaky tests. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. [S.l.: s.n.], 2020. p. 1471–1482. Citado 4 vezes nas páginas 17, 18, 19 e 22.

LAM, W. et al. idflakies: A framework for detecting and partially classifying flaky tests. In: IEEE. *2019 12th ieee conference on software testing, validation and verification (icst)*. [S.l.], 2019. p. 312–322. Citado 4 vezes nas páginas 37, 41, 42 e 43.

LAMBIASE, S. *DARTS*. 2022. Https://github.com/StefanoLambiase/DARTS. Access date: 14 ago. 2021. Citado na página 54.

LUO, Q. et al. An empirical analysis of flaky tests. In: *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*. [S.l.: s.n.], 2014. p. 643–653. Citado 2 vezes nas páginas 19 e 20.

MÄNTYLÄ, M. V.; LASSENIUS, C. Subjective evaluation of software evolvability using code smells: An empirical study. *Empirical Software Engineering*, Springer, v. 11, n. 3, p. 395–431, 2006. Citado na página 23.

MARTINEZ, M. *RTj*. 2022. Https://github.com/UPHF/RTj. Access date: 14 ago. 2021. Citado na página 54.

MESZAROS, G. *xUnit test patterns: Refactoring test code*. [S.l.]: Pearson Education, 2007. Citado na página 23.

MICCO, J. The state of continuous integration testing@ google. 2017. Citado 3 vezes nas páginas 18, 22 e 37.

NETO, A.; CLAUDIO, D. Introdução a teste de software. *Engenharia de Software Magazine*, v. 1, p. 22, 2007. Citado na página 17.

PALOMBA, F.; ZAIDMAN, A. Notice of retraction: Does refactoring of test smells induce fixing flaky tests? In: IEEE. *2017 IEEE international conference on software maintenance and evolution (ICSME)*. [S.l.], 2017. p. 1–12. Citado 3 vezes nas páginas 35, 46 e 53.

PARRY, O. et al. A survey of flaky tests. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, ACM New York, NY, v. 31, n. 1, p. 1–74, 2021. Citado 8 vezes nas páginas 17, 18, 19, 20, 21, 22, 36 e 37.

PERUMA, A. et al. Tsdetect: An open source test smells detection tool. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2020. (ESEC/FSE 2020), p. 1650–1654. ISBN 9781450370431. Disponível em: <https://doi.org/10.1145/3368089.3417921>. Citado 14 vezes nas páginas 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 46, 47 e 52.

PERUMA, A. et al. On the distribution of test smells in open source android applications: An exploratory study. 2019. Citado na página 13.

SANTANA, R. *Raide*. 2022. Https://github.com/arieslab/raide. Access date: 14 ago. 2021. Citado na página 54.

SANTOS, J. A. M. et al. A systematic review on the code smell effect. *Journal of Systems and Software*, Elsevier, v. 144, p. 450–477, 2018. Citado na página 23.

SILVA, D.; TEIXEIRA, L.; D'AMORIM, M. Shake it! detecting flaky tests caused by concurrency with shaker. In: IEEE. *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.], 2020. p. 301–311. Citado na página 23.

ZAIDMAN, A. *TestHound*. 2022. Https://github.com/SERG-Delft/TestHoundj. Access date: 14 ago. 2021. Citado na página 54.

ZOLFAGHARI, B. et al. Root causing, detecting, and fixing flaky tests: State of the art and future roadmap. *Software: Practice and Experience*, Wiley Online Library, v. 51, n. 5, p. 851–867, 2021. Citado 6 vezes nas páginas 18, 22, 23, 35, 37 e 46.