UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
CURSO DE ENGENHARIA DA COMPUTAÇÃO

THALISSON MOURA TAVARES

UM ESTUDO SOBRE A RELAÇÃO ENTRE TEST SMELLS E FLAKY TESTS

Recife
2023

Thalisson Moura Tavares

UM ESTUDO SOBRE A RELAÇÃO ENTRE TEST SMELLS E FLAKY TESTS

Trabalho de Conclusão de Curso apresentado ao Curso de Engenharia da Computação da Universidade Federal de Pernambuco, Centro de Informática, como requisito parcial para obtenção do título de Bacharel em Engenharia da Computação.

Orientador: Leopoldo Motta Teixeira
Coorientador: Rodrigo dos Santos Lima

Recife
2023

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Tavares, Thalisson Moura.
Um estudo sobre a relação entre test smells e flaky tests / Thalisson Moura Tavares. - Recife, 2023.
19 p. : il., tab.

Orientador(a): Leopoldo Motta Teixeira
Cooorientador(a): Rodrigo dos Santos Lima
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Engenharia da Computação - Bacharelado, 2023.

1. Test smells. 2. Flaky test. 3. Flakiness. I. Teixeira, Leopoldo Motta. (Orientação). II. Lima, Rodrigo dos Santos. (Coorientação). IV. Título.

000 CDD (22.ed.)

Um estudo sobre a relação entre *test smells* e *flaky tests*

Thalisson Tavares[1], **Leopoldo Teixeira**[1], **Rodrigo Lima**[1]

[1]Centro de Informática - Universidade Federal de Pernambuco (UFPE)
Recife - PE - Brazil

tmt2@cin.ufpe.br, lmt@cin.ufpe.br, rsl@cin.ufpe.br

Resumo. *Test smells são indicadores de que pode haver problemas com o design ou implementação de um teste. Testes são considerados flaky quando produzem resultados diferentes ao executá-los várias vezes. Isto geralmente é devido a fatores externos, como conectividade de rede ou estado do sistema. Existe uma relação potencial entre test smells e flaky test, pois alguns test smells, como testes que dependem de dados de teste frágeis ou testes que possuem lógica complexa, podem contribuir para um comportamento flaky. O objetivo deste estudo é investigar as alterações que ocorrem no teste após a correção do flaky test e entender o impacto dessa correção na redução de test smells. Este estudo destaca as categorias de smells que sofreram alterações após a correção de flakiness nos testes, apresentando como sua principal contribuição uma análise qualitativa dos testes que sofreram modificações na quantidade de smells. Isso proporcionou uma compreensão mais profunda das causas e soluções para o flakiness, bem como a explicação das razões por trás das variações na detecção de smells e sua correlação com o flakiness.*

1. Introdução

Para entregar software de forma contínua, com qualidade e minimizar a ocorrência de falhas após alterações nos códigos de produção, é essencial o desenvolvimento de testes. Os testes são uma fase importante no desenvolvimento de software, e alguns dos principais fatores para essa importância e que motivam o uso de testes são, a prevenção no aparecimento de *bugs*, confiança no código de produção desenvolvido e testar situações de sucesso e de falha. Entretanto, semelhante aos códigos de produção, os testes podem ser desenvolvido com más práticas, *test smells*, o que afeta a manuntenção e qualidade do teste. Além disso, a presença de testes inconsistentes, *flaky test*, tornam o teste pouco confiável e que frustam o desenvolvedor.

Nesse contexto, esse estudo tem como principal objetivo encontrar uma relação entre *test smells* e *flaky test*, bem como investigar as alterações dos *test smells* após a eliminação da instabilidade dos testes, ou seja, após correção do *flakiness*. Para atingir essa meta, desenvolvemos um script para extrair resultados quantitativos das categorias de *smells* que sofreram modificação após a correção do *flakiness*. Além disso, realizamos uma análise qualitativa abrangente em todos os testes que passaram por essas alterações na quantidade de *smells*. Como resultado, notamos que as categorias *Assertion Roulette* e *Sensitive Equality* foram as mais afetadas, registrando reduções significativas após a remoção do *flakiness* nos testes. Em todos os testes analisados nessas categorias, constatou-se que a causa do *flakiness* estava relacionada a comparações entre strings fixas que levavam em consideração a ordem dos elementos na string. Embora a categoria *Assertion*

Roulette tenha apresentado a maioria das modificações no resultado quantitativo, a análise qualitativa revelou que, entre as duas, apenas a categoria *Sensitive Equality* demonstrou uma correlação significativa entre a correção dos *smells* e a eliminação do *flakiness*.

As seções seguintes deste trabalho estão organizadas da seguinte forma. A Seção 2 apresenta os conceitos fundamentais para o entendimento desse estudo. A Seção 3 explica qual metodologia foi utilizada e o script do projeto desenvolvido. A Seção 4 contém os resultados obtidos. Por fim, a Seção 5 contém as conclusões obtidas com o estudo e aponta sugestões para trabalhos futuros.

2. Background

Nesta seção, será apresentado uma visão geral dos principais tópicos relacionados a este estudo: *test smells* e *flakiness*.

2.1. Test smells

Assim como nos códigos de produção, os testes também podem ser afetados por más práticas de programação, o que pode comprometer sua qualidade, compreensão e facilidade de manutenção [Peruma et al. 2020]. Essas más práticas no design e desenvolvimento de testes são chamados de *test smells* [Van Deursen et al. 2001], [Meszaros 2007] e [Mens et al. 2008]. Um conjunto de *test smells* foi proposto inicialmente por [Van Deursen et al. 2001], sendo eles: *Assertion Roulette*, *Eager Test*, *General Fixture*, *Lazy Test*, *Mystery Guest*, *Resource Optimism* e *Sensitive Equality*. Como extensão dos *test smells* existentes e definidos na literatura, [Peruma et al. 2019] propôs um novo conjunto de *test smells* inspirados em práticas ruins na programação de testes mencionadas na literatura baseada em testes unitários. O novo conjunto proposto é utilizado nesse estudo para classificar os tipos de *test smells* e sua relação com *flaky test*.

O Código 1 mostra um exemplo de um tipo de *test smell*, o *Sleepy Test*. O *Sleepy Test* ocorre quando adicionamos no nosso teste um tempo de espera fixo com o objetivo de parar a execução por determinado tempo antes de continuar os próximos passos. Na linha 4 do Código 1, é utilizado o `Thread.sleep(4000)` para aguardar 4 segundos antes de continuar a execução do teste.

```
1 @Test
2 public void sleepyTestExample() {
3   driver.findElement(By.id("elemento1").click()
4   Thread.sleep(4000);
5   driver.findElement(By.id("elemento2").sendKeys("Algum texto aqui");
6 }
```

Código 1: Exemplo - Sleepy Test

2.2. Flaky tests

Os testes são uma parte essencial do desenvolvimento de software. Por meio do desenvolvimento de testes, é possível verificar se alterações no sofware ou refatoração em códigos de produção não prejudicam ou interrompem nenhuma funcionalidade existente. Idealmente, quaisquer novas falhas nos testes indicariam problemas causados pelas alterações mais recentes realizadas, ou seja, espera-se que os resultados dos testes sejam determinísticos, então se um teste não foi modificado é esperado que sempre

passe ou sempre falhe para o mesmo código de teste. No entanto, algumas falhas nos testes podem não ser resultado de alterações realizadas, mas sim do não determinismo nos testes, muitas vezes chamados de *flaky tests* [Luo et al. 2014], [Bell et al. 2018], [Parry et al. 2021] e [Camara et al. 2021]. *Flaky tests* além de serem problemáticos, são comumente encontrados e relatados por muitos profissionais e pesquisadores [Lavers and Peters 2008, Muşlu et al. 2011, Sudarshan 2012, Memon and Cohen 2013, Hao et al. 2013, Zhang et al. 2014, Huo and Clause 2014, Bell and Kaiser 2014, Bell et al. 2015, Gyori et al. 2015, Herzig and Nagappan 2015, Jiang et al. 2017].

Nesse estudo, o termo *flakiness* será frequentemente utilizado para se referir a testes com inconsistência e não determinísticos, ou seja, *flaky tests*.

2.3. Trabalhos Relacionados

Um trabalho inicialmente importante na investigação e estudo da relação entre a refatoração de *test smells* e a correção de *flaky test* foi desenvolvido por Palomba e Zaidman [Palomba and Zaidman 2017]. Entretanto o artigo foi retirado devido a erros encontrados na análise relatada no artigo, resultando em resultados e conclusões imprecisas.

O trabalho original [Palomba and Zaidman 2017] chegou a ser estendido [Palomba and Zaidman 2019], com estudos e avaliações empíricas adicionais sobre a relação entre *test smells* e *flaky test*. A metodologia consistia na extração de um catálogo de *test smells* que lembram o conceito de *flakiness*, para analisar a co-ocorrência entre *test smells* e *flaky test* e se a remoção do *smells* pode ser considerada uma estratégia válida para corrigir o *flakiness*. Porém, assim como o artigo anterior [Palomba and Zaidman 2017], após a reavaliação do experimento, os autores identificaram erros na estratégia de detecção de *flaky tests*, então o artigo também foi retirado.

O artigo [Camara et al. 2021], investiga o uso de uma abordagem alternativa para a predição de *flaky tests*, baseado em *test smells*. Foi realizado um estudo empírico para entender se *test smells* têm um bom desempenho como um classificador para prever *flakiness* em um contexto de projetos cruzados. Como resultado, foi obtido um classificador que apresentou um desempenho superior ao modelo baseado em vocabulário para a previsão em projetos cruzados. Porém, o estudo não analisa o impacto na alteração de *test smells* após a remoção de *flakiness*, que é o principal objetivo do nosso estudo.

3. Metodologia

Neste capítulo, apresentamos a metodologia utilizada para avaliar a relação entre *test smells* e *flaky test*. O estudo foi divido em duas etapas, a quantitativa e a qualitativa. Na etapa quantitativa foi analisada a diferença na quantidade de *test smells* após a remoção de *flakiness* dos testes analisados. Na etapa qualitativa foi realizada a análise individual de cada arquivo de teste que sofreu alteração na quantidade de *test smells* após a remoção do *flaky test*, para identificar o motivo da inconsistência (*flakiness*), qual correção foi realizada, quais as categorias de *test smells* que sofreram alteração e o que motivou a alteração na quantidade de *test smells*.

3.1. Ferramentas

3.1.1. Dataset

Para alcançar o objetivo desse trabalho, primeiramente, foi selecionado o dataset internacional de *flaky tests*, IDoFT [Lam 2020] [Lam et al. 2019], que possui todas as informações relacionadas a *flaky test* necessárias para o estudo. O dataset IDoFT contém 5483 casos de testes *flaky* de 373 projetos Java distintos. As informações utilizadas do dataset para esse estudo foram: o repositório do projeto no qual o *flaky test* foi encontrado, o módulo e nome do teste para localizar o arquivo que possui o *flakiness*, o estado do *flaky test* para identificar se um *Pull Request* (PR) que corrige o *flakiness* já foi aceita, e por fim o link do PR da correção.

3.1.2. Detector de *test smells*

Foi utilizado o TsDetect [Peruma et al. 2019] [Peruma et al. 2020] que é uma ferramenta open source para detecção de *test smells* em testes unitários baseados em JUnit. O TsDetect é capaz de detectar 19 tipos de *test smells* com uma precisão média de 96% e recall médio de 97%, além disso, a ferramenta TsDetect possui uma alta taxa de adoção, sendo utilizada em pelo menos oito estudos [Aljedaani et al. 2021]. A Tabela 1 mostra os tipos de *smells* detectados pela ferramenta e a regras para detecção de cada tipo.

Tabela 1: Regras de detecção de *test smells* do tsDetect [Peruma et al. 2020]

Test smells	Regra de detecção
Assertion Roulette	Um método de teste que contém mais de uma declaração de asserção sem explicação/mensagem
Conditional Test Logic	"Um método de teste que contém uma ou mais instruções de controle (e.g., if, switch, for...)
Constructor Initialization	Uma classe de teste que contém uma declaração de construtor
Default Test	Uma classe de teste que é chamada de 'ExampleUnitTest' ou 'ExampleInstrumentedTest'
Duplicate Assert	Um método de teste que contém mais de uma instrução de asserção com os mesmos parâmetros
Eager Test	Um método de teste que contém múltiplas chamadas para vários métodos de produção
Empty Test	Um método de teste que não contém uma única instrução executável
Exception Handling	Um método de teste que contém uma instrução throw ou uma cláusula catch
General Fixture	Todos os campos instanciados no setUp de uma classe não são utilizados por todos os métodos de teste da classe
Ignored Test	Um método ou classe de teste que contém a anotação @Ignore
Lazy Test	Vários métodos de teste chamando o mesmo método de produção
Magic Number Test	Um método de asserção que contém um literal numérico como argumento
Mystery Guest	Um método de teste contendo instâncias de objetos de arquivos e classes de bancos de dados
Redundant Print	Um método de teste que invoca o método print ou println ou printf ou write da classe System
Redundant Assertion	Um método de teste que contém uma declaração de afirmação com os parâmetros esperados e reais iguais
Resource Optimism	Um método de teste que utiliza uma instância de uma classe File sem chamar os métodos do objeto
Sensitive Equality	Um método de teste que invoca o método toString() de um objeto
Sleepy Test	Um método de teste que invoca o método Thread.sleep()
Unknown Test	Um método de teste que não contém uma única instrução de asserção e parâmetro de anotação @Test(expected)

O TsDetect recebe como entrada um arquivo CSV contendo os nomes e a localização dos arquivos de testes a serem analisados, após ser executado, um arquivo CSV é gerado contendo em cada linha o nome da aplicação, a classe de teste analisada, o caminho onde se encontra o arquivo de teste, o número de métodos que o arquivo possui, e a quantidade de *test smells* detectados por categoria.

3.1.3. Score de similaridade

Para comparar a similaridade entre os arquivos de testes antes e depois da correção do *flakiness*, foi utilizado o MOSS [Aiken 1994] que é um sistema automático para determinar

a similaridade de programas. O termo MOSS significa *Measure Of Software Similarity*, e é um sistema amplamente utilizado para detecção de plágio em tarefas de programação [Schleimer et al. 2003]. O objetivo é utilizar o MOSS para comparar os arquivos de testes antes e depois da correção para verificar a similaridade e, caso necessário, filtrar por aqueles que possuem maior similaridade e assim diminuir o ruído dos testes durante o estudo.

3.2. Experimento

Os resultados quantitativos desse estudo foram obtidos por meio de um programa desenvolvido em Python, a Figura 1 mostra o fluxograma geral do programa. O programa é divido em 7 etapas, sendo elas, a etapa de filtro, cópia, tratamento, extração, gerador de entrada, TsDetect e gerador de resultado. O programa recebe como entrada o dataset especificado na Seção 3.1.1 e como resultado, retorna um arquivo CSV contendo as informações dos arquivos de teste que sofreram alteração na quantidade de *test smells*. Mais detalhes sobre os resultado são apresentados na Seção 4.

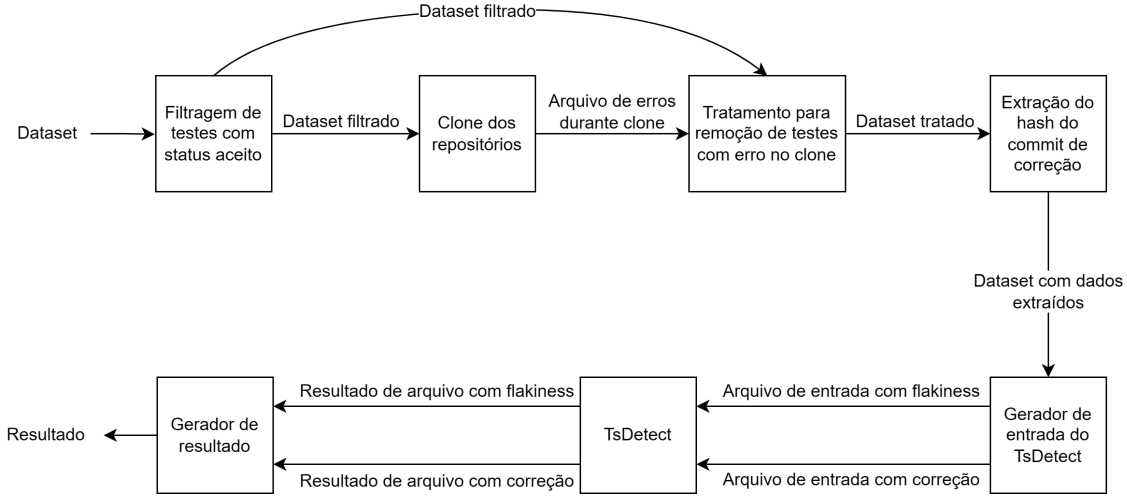


Figura 1: Fluxograma do experimento

3.2.1. Filtro

Nesta etapa inicial, o dataset é filtrado para selecionar apenas os testes que possuem o status "*Accepted*", ou seja, selecionamos apenas os testes no qual o PR para a correção do *flaky test* foi aceito. Dessa forma, a análise é realizada apenas com os testes que tiveram a correção do *flaky test* desenvolvida e aprovada. A etapa de filtro recebe o dataset como entrada e retorna os dados filtrados.

3.2.2. Cópia

Na etapa de cópia, é realizado o clone dos projetos que foram filtrados na etapa anterior. A etapa de cópia além de clonar os projetos, armazena em um arquivo os projetos que tiveram erro durante o clone. Esse arquivo de erro será utilizado na próxima etapa, a etapa de tratamento.

3.2.3. Tratamento

A etapa de tratamento é a unica do fluxo que recebe entrada de duas etapas distintas. Isso ocorre pois, nessa etapa, são removidos todos os testes em que não foi possível realizar o clone do projeto na etapa de cópia. A etapa de tratamento recebe os dados filtrados e o arquivo de erro durante o clone como entrada, e retorna um novo arquivo CSV com os dados tratados.

3.2.4. Extrator

O extrator recebe os dados tratados como entrada, e retorna um novo arquivo CSV com os dados extraídos. O extrator é responsável por extrair do link de PR que está presente no dataset, o hash do commit de correção, o SHA. Isso é necessário pois no dataset tem apenas a informação do commit de quando o *flaky test* foi identificado, porém não tem o commit após a correção do *flaky test*. Para obter essa informação, foi desenvolvido um *web scraper* em Python, para extrair o SHA do commit de correção do *flaky test*. Analisando os *Pull Requests* (PRs) de correção, foi possível observar que alguns eram finalizados com o status de "Merged" no qual o commit é mesclado a branch ao qual o PR foi aberto, e outros eram finalizados com status de "Closed" em que o PR era fechado em um commit específico. Embora a estrutura das páginas sejam semelhantes para ambos os status, o conteúdo e localização do commit de correção são distintos, então o extrator foi desenvolvido para lidar com as duas situações de PRs e obter o SHA correspondente ao commit de correção como mostra o Código 2.

```
response = requests.get(url)
if response.status_code == 200:
    soup = BeautifulSoup(response.text, "lxml")
    #checks if PR state is Merged
    if soup.select('span[class="State State--merged"]'):
        elements = soup.select('div[class="TimelineItem-body"] > a > code[class^="Link--primary text-bold"]')
        commit_sha = elements[0].text
    #checks if PR state is Closed
    elif soup.select('span[class="State State--closed"]'):
        elements = soup.select('div[class="TimelineItem-body"] > code > a[data-hovercard-type^="commit"]')
        commit_sha = elements[0].text
```

Código 2: Extrator do hash de commit

Do total de 1097 PRs de entrada para realizar a extração, 770 eram duplicados, ou seja o mesmo link de PR estava presente em dois ou mais testes e 327 eram únicos. Foi extraído 1082 hashs, sendo 764 duplicados e 318 únicos. O extrator obteve 9 falhas na extração. Ao verificar os nove PRs que tiveram falha, foi constatado que um PR estava com status "*Open*", que não é considerado para esse estudo. Os demais PRs possuíam uma forma específica, como por exemplo o PR é fechado e a correção realizada em um commit referenciando o PR. Como esses casos não possuiam um padrão e a quantidade deles são pequenas em relação ao total, eles foram desconsiderados, o que resultou em uma taxa de sucesso de 97.2% na extração.

3.2.5. Gerador de entrada

Nessa etapa é gerado o arquivo de entrada da ferramenta de detecção de *test smells*, o TsDetect. A Figura 2 mostra o fluxograma do gerador de entrada.

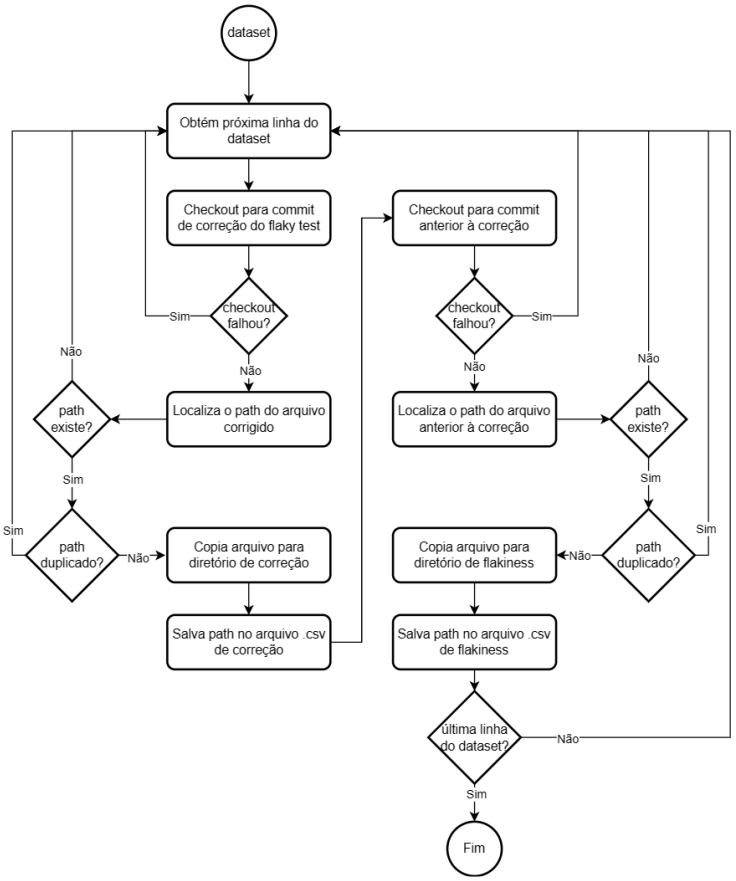


Figura 2: Fluxograma do gerador de entrada

O gerador de entrada recebe os dados extraídos e produz como saída dois arquivos CSV no formato esperado pelo TsDetect, sendo um arquivo CSV referente aos testes com *flakiness*, e o outro arquivo CSV referente aos testes corrigidos. Após receber os dados extraídos, o gerador percorre cada linha do arquivo e, para cada linha, realiza o checkout para o commit de correção extraído na etapa anterior. Caso o checkout seja bem sucedido, localiza o caminho do arquivo de teste corrigido. Ao encontrar o caminho do arquivo de teste e verificar que é a primeira vez que localiza o arquivo, uma cópia do arquivo é feita para um diretório onde são armazenados os arquivos com correção. Em seguida, o novo caminho do arquivo é adicionado a uma lista para posteriormente ser salvo no arquivo CSV de saída. Na sequência, é realizado o checkout para o commit anterior a correção do *flakiness*, ou seja, o commit que ainda possui o *flaky test*. Caso o checkout seja realizado com sucesso, é feita a busca pelo arquivo de teste antes da correção. Ao encontrar o caminho do arquivo e verificar que ele não foi localizado anteriomente, uma cópia do arquivo é realizada para o diretório de testes com *flakiness*. Finalmente, o novo caminho do arquivo é adicionado a uma lista para posteriormente ser salvo no arquivo CSV.

3.2.6. TsDetect

Nessa etapa, a ferramenta TsDetect é executada para obter o resultado com a quantidade de *test smells* para cada arquivo de teste. O TsDetect analisa os dois arquivos produzidos na etapa anterior de forma separada, e retorna como saída dois arquivos CSV com o resultado da análise. A Tabela 2 mostra o exemplo de saída do TsDetect para três arquivos de testes. Para facilitar a visualização a tabela foi transposta e alguns dados foram removidos, como por exemplo o caminho de localização do arquivo de teste.

Tabela 2: Resultado da análise do TsDetect

	Activiti/Activiti	Adyen/adyen-java-api-library	alibaba/fastjson
TestClass	RetryInterceptorTest.java	CheckoutTest.java	Issue1584.java
NumberOfMethods	3	131	1
Assertion Roulette	1	99	1
Conditional Test Logic	0	0	0
Constructor Initialization	0	0	0
Default Test	0	0	0
EmptyTest	0	0	0
Exception Catching Throwing	0	109	1
General Fixture	0	0	0
Mystery Guest	0	0	0
Print Statement	0	0	0
Redundant Assertion	0	0	0
Sensitive Equality	0	9	0
Verbose Test	0	0	0
Sleepy Test	0	0	0
Duplicate Assert	0	0	1
Unknown Test	0	0	0
IgnoredTest	0	0	0
Resource Optimism	0	0	0
Magic Number Test	1	117	1
Dependent Test	0	0	0

3.2.7. Gerador de resultado

O gerador de resultado percorre simultaneamente os dois arquivos de saída do TsDetect, o arquivo com os resultados antes da correção do *flakiness* e o arquivo de resultados após a correção. Para cada linha dos arquivos, é calculada a diferença na quantidade de *test smells* por categoria, o total de redução e aumento de *test smells*, e o score de similaridade entre os arquivos antes e depois da correção utilizando a ferramenta MOSS. Além disso, é realizado o somatório da quantidade total de redução e de aumento para cada categoria de *test smells*.

Para obter o score de similaridade, os arquivos de teste antes e após a correção do *flakiness*, são enviados para a ferramenta MOSS que retorna uma URL contendo o resultado da comparação. No gerador de resultado também foi desenvolvido um *web scraper* para obter o score a partir da URL retornada. O Código 3 foi desenvolvido para realizar a extração do score, no qual a variável `test_score` armazena o score do arquivo de teste sem correção em relação ao arquivo de teste com correção, enquanto a variável `fix_score`, armazena o score do arquivo de teste com correção em relação ao arquivo

de teste sem correção. O score é em porcentagem, que é a porcentagem do código em um arquivo considerada correspondente ao código do outro arquivo. Para esse estudo vamos utilizar apenas o `fix_score`, pois estamos interessados em analisar o impacto de *test smells* na correção de *flaky test*.

```
response = requests.get(url)
if response.status_code == 200:
    soup = BeautifulSoup(response.text, "lxml")
    elements = soup.select('tr > td > a')
    length = len(elements)
    if length > 0:
        test_score = extractScore(elements[0].text)
    if length > 1:
        fix_score = extractScore(elements[1].text)

    return (test_score, fix_score)
```

Código 3: Extrator do score de similaridade

O gerador de resultado é a ultima etapa do programa desenvolvido, ele retorna o resultado em um arquivo CSV. Os resultados são descritos na Seção 4.1.

3.3. Análise qualitativa

A partir dos resultados obtidos no experimento realizado, foi possível fazer uma análise detalhada da relação entre *test smells* e *flaky test*. A análise tem como objetivo responder os seguintes questionamentos:

- Quais categorias de *test smells* sofreram alteração após a correção do *flakiness*?
- Qual o motivo do teste possuir *flakiness*?
- Qual correção foi realizada para remover o *flakiness*?
- O que ocasionou a alteração na quantidade de *smells*?
- A correção do *test smells* remove o *flakiness* do teste?

Para responder aos questionamentos levantados, primeiramente, todos os arquivos de teste que sofreram alteração na quantidade de *test smells* foram selecionados. A partir dessa seleção, os arquivos foram classificados de acordo com o tipo de *test smells* que foi alterado. Para cada arquivo classificado, foram verificadas as informações presentes no PR. Também foi realizada a comparação de mudanças (*diff*) entre os arquivos de testes antes e depois da correção para determinar o motivo do *flakiness*, qual a correção realizada para remover o *flakiness* e identificar a causa da alteração na quantidade de *test smells*. Com o auxílio da ferramenta TsDetect, foi realizada uma validação na causa identificada para a alteração na quantidade de *test smells*. E por fim, foi definido o impacto da correção do *test smells* na remoção do *flakiness*. Os resultados da análise qualitativa estão detalhados na Seção 4.2.

4. Resultados

Nessa seção são detalhados os resultados obtidos do experimento e da análise qualitativa.

4.1. Resultado quantitativo

No experimento, do total de 472 arquivos de teste analisados pela ferramenta TsDetect (vide Seção 3.2.6), 38 arquivos sofreram alterações na quantidade de *test smells*. Desses

38 arquivos, 8 arquivos tiveram aumento e redução de *test smells* simultaneamente para categorias distintas. A Figura 3 exibe o gráfico da quantidade total de *test smells* por categoria. É possível perceber que a categoria *Assertion Roulette* foi a que teve o valor mais destoante em relação às demais categorias, sendo 52 alterações no total, das quais 43 são reduções e 9 são aumentos.

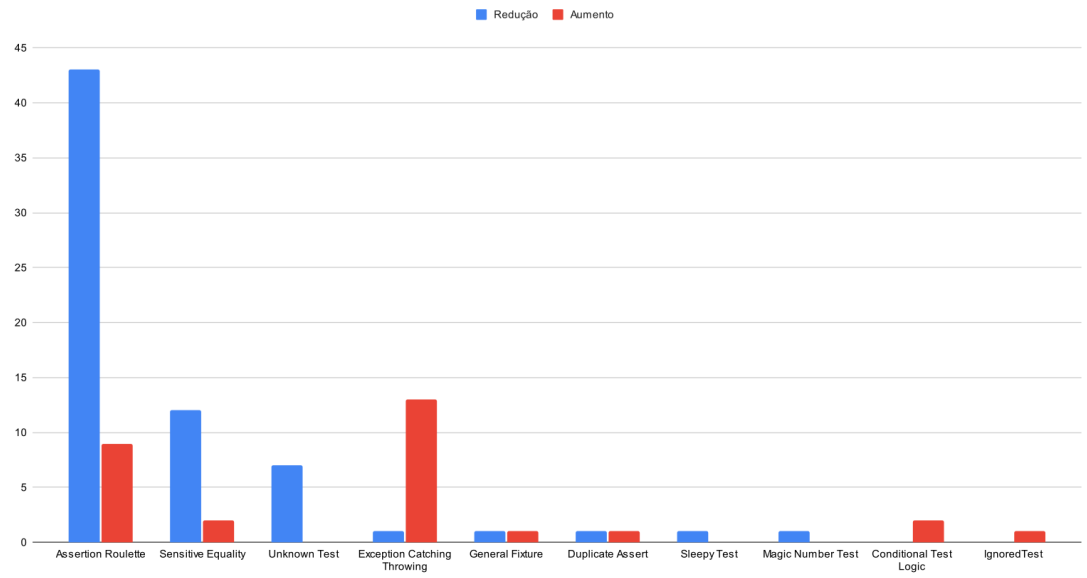


Figura 3: Gráfico da quantidade de *test smells* por categoria

No geral a redução de *test smells* foi mais frequente que a quantidade de aumentos. Do total de 10 categorias que sofreram alterações, três tiveram apenas reduções, que são as categorias *Unknown Test*, *Sleepy Test* e *Magic Number Test*. Duas categorias tiveram apenas aumento de *test smells*, que são as categorias *Conditional Test Logic* e *Ignored Test*. As demais categorias tiveram tanto aumento quanto redução, porém a categoria *Exception Catching Throwing* foi a única a ter a quantidade de aumentos maior que a quantidade de reduções.

Observando a similaridade dos arquivos após a correção do *flaky test*, nota-se poucas modificações dos arquivos. Ou seja, os arquivos de correção são similares aos arquivos antes da correção, o que indica menos alterações no código e, consequentemente, a alteração na quantidade de *test smells* foi ocasionada devido a correção do *flakiness*. Os scores de similaridade foram obtidos no gerador de resultado, detalhado na Seção 3.2.7. Do total de 38 arquivos que sofreram alteração na quantidade de *test smells*, o score de similaridade médio foi de 84%, o mínimo foi de 49% e o máximo de 99%. Pelo histograma da Figura 4, é possível verificar que a maioria dos arquivos tiveram um score acima de 73%, sendo apenas 5 arquivos com score inferior a 73%.

A Figura 5 mostra os gráficos da quantidade de *test smells* por categoria, porém dessa vez, separado por projeto. Foram selecionados os quatro projetos com maior quantidade de arquivos de testes que sofreram alteração na quantidade de *smells* após a correção do *flakiness*. Nota-se que cada projeto possui caracteristicas únicas. No caso do projeto `alibaba/fastjson` da Figura 5a, temos reduções nas categorias

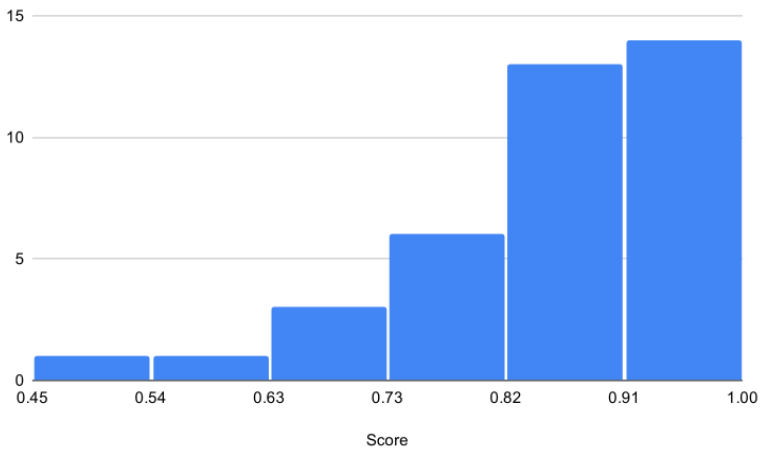
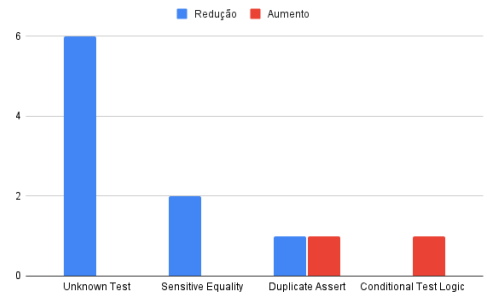
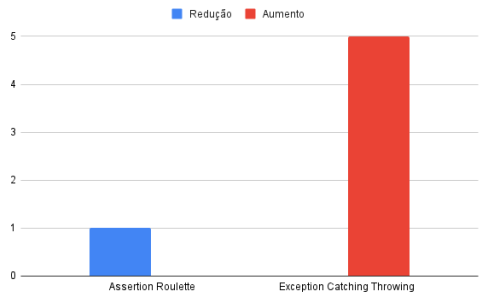


Figura 4: Histograma do score de similaridade

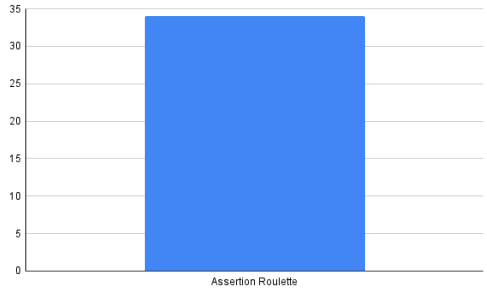
Unknown Test e *Sensitive Equality*, enquanto a categoria *Duplicate Assert* sofreu tanto aumento quanto redução. Já a categoria *Conditional Test Logic* apenas aumentou a quantidade de *test smells*. No projeto `apache/pulsar`, da Figura 5b, ocorreu redução na categoria *Assertion Roulette* e aumento de *smells* na categoria *Exception Catching Throwing*. Por fim, os projetos `adyen/adyen-java-api-library` e `apache/camel-spring-boot`, mostrados nas Figuras 5c e 5d respectivamente, sofreram alterações apenas na quantidade de *smells* da categorias *Assertion Roulette*, sendo 34 reduções no projeto `adyen/adyen-java-api-library` e 3 aumentos no projeto `apache/camel-spring-boot`.



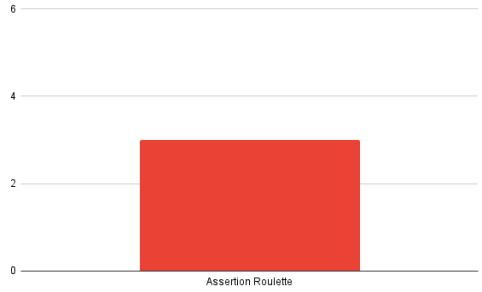
(a) alibaba/fastjson



(b) apache/pulsar



(c) adyen/adyen-java-api-library



(d) apache/camel-spring-boot

Figura 5: Gráficos por projeto da quantidade de *test smells* por categoria

4.2. Resultado qualitativo

O resultado qualitativo foi obtido através da análise individual de cada arquivo de teste que sofreu alteração. Para cada arquivo de teste, foram respondido os questionamentos levantados na Seção 3.3, a fim de obter uma conclusão sobre a relação entre *test smells* e *flaky test*. Além disso, validar se os resultados quantitativos obtidos previamente estão relacionados com a correção do *flaky test*.

```
558     @Test
559     public void TestSepaDirectDebitDetailsSerialization() throws JsonProcessingException {
560         String expectedJson = "{\"amount\":{\"value\":1000,\"currency\":\"USD\"},\"merchantAccount\"
561
562         SepaDirectDebitDetails sepaDirectDebitDetails = new SepaDirectDebitDetails();
563         sepaDirectDebitDetails.setOwnerName("A. Schneider");
564         sepaDirectDebitDetails.setIban("DE87123456781234567890");
565         PaymentsRequest paymentsRequest = createPaymentsCheckoutRequest();
566         paymentsRequest.setPaymentMethod(sepaDirectDebitDetails);
567
568         String gson = GSON.toJson(paymentsRequest);
569-        assertEquals(expectedJson, gson);
570
571         String jackson = OBJECT_MAPPER.writeValueAsString(paymentsRequest);
572-        assertEquals(expectedJson, jackson);
573     }
```

(a) Código com *flakiness*

```
559     @Test
560     public void TestSepaDirectDebitDetailsSerialization() throws JsonProcessingException {
561         String expectedJson = "{\"amount\":{\"value\":1000,\"currency\":\"USD\"},\"merchantAccount\"
562
563         SepaDirectDebitDetails sepaDirectDebitDetails = new SepaDirectDebitDetails();
564         sepaDirectDebitDetails.setOwnerName("A. Schneider");
565         sepaDirectDebitDetails.setIban("DE87123456781234567890");
566         PaymentsRequest paymentsRequest = createPaymentsCheckoutRequest();
567         paymentsRequest.setPaymentMethod(sepaDirectDebitDetails);
568
569         String gson = GSON.toJson(paymentsRequest);
570+        assertJsonStringEquals(expectedJson, gson);
571
572         String jackson = OBJECT_MAPPER.writeValueAsString(paymentsRequest);
573+        assertJsonStringEquals(expectedJson, jackson);
574     }
      .
      .
      .
2401+   private void assertJsonStringEquals(String firstInput, String secondInput) {
2402+       JsonParser parser = new JsonParser();
2403+       assertEquals(parser.parse(firstInput), parser.parse(secondInput));
2404+   }
```

(b) Código corrigido

Figura 6: Trecho de código do arquivo de teste CheckoutTest.java

A Figura 6 mostra um trecho do código do arquivo de teste `CheckoutTest.java` do projeto `Adyen/adyen-java-api-library` antes e depois da correção do *flakiness*. Com base na análise quantitativa, o arquivo `CheckoutTest.java` teve 33 reduções na quantidade de *smells* da categoria *Assertion Roulette*. Analisando as alterações no código, foi constatado que o motivo do *flakiness* foi devido comparações diretas entre strings formatadas em JSON que podem ocasionalmente falhar em mudanças de ambiente, pois as ordens dos elementos em JSON não são preservadas. Essas comparações são mostradas nas linhas 569 e 572 da

Figura 6a. Para *asserts* que incluem comparação bruta de strings no formato JSON foi implementada a função auxiliar `assertJsonStringEquals()`, mostrada na linha 2401 da Figura 6b. A função `assertJsonStringEquals()` faz comparações diretas após analisar strings de entrada com o método `JsonParser()` do Gson, permitindo a permutação de ordens de elementos em JSONs e assim corrigindo o *flakiness* do teste. *Assertion Roulette* ocorre quando um teste contém mais de um *assert* sem mensagem explicativa. Como alguns testes foram refatorados para utilizar a função auxiliar `assertJsonStringEquals()`, assim como mostrado nas linhas 570 e 573 da Figura 6b, a ferramenta TsDetect detectou que a quantidade de *Assertion Roulette* diminuiu, porém a quantidade continua a mesma. No caso do arquivo de teste `CheckoutTest.java`, a redução de *test smells* não influencia na correção do *flaky test*. Isso se dá porque a ferramenta não detectou o *smell* devido a chamada do *assert* no novo método `assertJsonStringEquals()`, o que pode ser visto na linha 2403 da Figura 6b. Porém, o teste continua fazendo o *assert* de maneira duplicada e sem mensagem explicativa, a diferença é que agora o *assert* está dentro de um método separado fazendo com que a ferramenta TsDetect não detecte o *smell*. O desenvolvedor poderia fazer o parsing do Json dentro do próprio teste e chamar o *assert*, o que também resolveria o *flakiness*, porém não reduziria o *test smells*.

```
30    public void testFromAnnotation()
31    {
32        JsonIncludeProperties.Value v = JsonIncludeProperties.Value.from(Bogus.class.getAnnotation(annotationClass:JsonIncludeProperties.class));
33        assertNotNull(v);
34        Set<String> included = v.getIncluded();
35        assertEquals(2, v.getIncluded().size());
36        assertEquals(_set(...args:"foo", "bar"), included);
37-       assertEquals("JsonIncludeProperties.Value(included=[bar, foo])", v.toString());
38        assertEquals(v, JsonIncludeProperties.Value.from(Bogus.class.getAnnotation(annotationClass:JsonIncludeProperties.class)));
39    }
```

(a) Código com *flakiness*

```
32    public void testFromAnnotation()
33    {
34        JsonIncludeProperties.Value v = JsonIncludeProperties.Value.from(Bogus.class.getAnnotation(annotationClass:JsonIncludeProperties.class));
35        assertNotNull(v);
36        Set<String> included = v.getIncluded();
37        assertEquals(2, v.getIncluded().size());
38        assertEquals(_set(...args:"foo", "bar"), included);
39+       String tmp = v.toString();
40+       boolean test1 = tmp.equals(anObject:"JsonIncludeProperties.Value(included=[foo, bar])");
41+       boolean test2 = tmp.equals(anObject:"JsonIncludeProperties.Value(included=[bar, foo])");
42+       assertTrue(test1 || test2);
43        assertEquals(v, JsonIncludeProperties.Value.from(Bogus.class.getAnnotation(annotationClass:JsonIncludeProperties.class)));
44    }
```

(b) Código corrigido

Figura 7: Trecho de código do arquivo de teste JsonIncludePropertiesTest.java

Já na Figura 7, é mostrado um trecho de código do arquivo de teste `JsonIncludePropertiesTest.java` do projeto `FasterXML/jackson-annotation` que teve uma redução de *smells* na categoria *Sensitive Equality*. O *flakiness* ocorre pois a variável `included` é um set. Portanto, a ordem de `foo` e `bar` não é fixa. Ao verificar a igualdade com a string fixa na linha 37 da Figura 7a o teste pode falhar ocasionalmente. Para corrigir o *flakiness*, foi adicionado um *assert* que compara o objeto com todas as variações na ordem de `foo` e `bar`, assim independentemente da ordem dos elementos o teste irá passar. O *smells* foi reduzido pois a categoria *Sensitive Equality* ocorre quando um teste utiliza o método `toString()` de um objeto, então na correção foi removido o `toString()` de dentro

do *assert* e atribuida a variável `tmp` (linha 39 da Figura 7b), ocasionando a não detecção do *smells* pela ferramenta e diminuindo um *smell* da categoria. Nesse caso, a correção do *test smells* pode ocasionar a remoção do *flakiness*, pois como o desenvolvedor é obrigado a remover o `toString()` do *assert*, a refatoração tende a remover a comparação com strings fixas. Nota-se que o método `toString()` ainda é utilizado, porém por não estar dentro do *assert* a ferramenta TsDetect não identificou o *smells*, uma melhoria é necessária no TsDetect para identificar esses casos, mas mesmo com isso a correção desse tipo de *smells* é capaz de remover o *flakiness* do código pois força o desenvolvedor a utilizar outros tipos de comparação no lugar de comparar strings fixas.

Assim como a análise realizada nos arquivos `CheckoutTest.java` e `JsonIncludePropertiesTest.java` mostradas nessa seção, foi realizada a análise para os demais arquivos de teste que sofreram alteração na quantidade de *smells*. Todas as informações das análises foram disponibilizadas em um documento no repositório do projeto que pode ser acessado através do link: https://github.com/thalissonmt/test-smells-and-flaky-test/tree/main

5. Conclusão

Neste trabalho, foi realizado um estudo da relação entre *test smells* e *flaky test* para verificar se a correção de *test smells* ocasiona a remoção de *flakiness* dos testes, a partir dos resultados quantitativos obtidos, análisar cada teste que sofreu alteração na quantidade de *smells* para identificar a causa e correção do *flakiness*, como também, a causa da alteração na quantidade de *smells* e sua relação com a correção do *flakiness*. As principais conclusões foram:

- Foi observado que em todos os testes que sofreram alteração na quantidade de *smells* da categoria *Sensitive Equality* a inconsistência foi devido a comparação entre strings fixas, que leva em consideração a ordem dos elementos. Além disso, a correção dos *smells* força o desenvolvedor a remover a utilização do método `toString()`, fazendo-o desenvolver outra solução no lugar de comparar as strings, o que faz com que o *flakiness* seja removido do teste.
- Todos os casos de *flakiness* da categoria *Assertion Roulette* foram devido a comparação com strings fixas, o que torna o teste não determinístico caso a ordem dos elementos de uma das strings seja diferente. Todas as correções, embora com implementações distintas, seguiram a ideia de comparar todos os casos possíveis para eliminar a dependência de ordem ou converter as strings em objetos e compará-los. A depender da abordagem escolhida pelo desenvolvedor para corrigir o *flakiness*, a quantidade de *smells* da categoria *Assertion Roulette* reduz ou aumenta, além disso, o *smells* pode ser corrigido adicionando uma mensagem explicativa nos *asserts*, o que corrige o *smells* mas não remove o *flakiness*, o que torna o argumento mais sólido de que as correções desse tipo de *smells* não impacta na correção do *flakiness* presente no teste.
- Alguns dos valores obtidos no resultado quantitativo se mostraram incorretos devido a falha na identificação do *smells* pela ferramenta TsDetect. Na análise qualitativa todos esses casos foram evidênciados.

5.1. Trabalhos futuros

No futuro, focaremos em identificar padrões de correções de *flaky tests* baseados nas *root causes*. Planejamos aprimorar a ferramenta *TsDetect* para reconhecer uma variedade mais

ampla de *asserts*, incluindo aqueles que não são exclusivos do JUnit, e detecção de *asserts* em funções auxiliares usadas nos testes, aprimorando a identificação de *smells* da categoria *Assertion Roulette*. Além disso, melhoraremos a detecção do método `toString()`, uma vez que, no momento, somente identifica o uso direto dentro de *asserts*, omitindo casos em que o resultado é atribuído a variáveis externas. Adicionalmente, investigaremos mais profundamente a relação entre a qualidade do código de teste e sua propensão a bugs.

Agradecimentos

Primeiramente, agradeço a Deus pela minha vida e por me dar força e conhecimento durante toda a gradução para chegar até aqui.

Ao professor Leopoldo e ao coorientador Rodrigo, por toda orientação e disponibilidade durante o desenvolvimento desse estudo.

A Andressa, minha companheira e amor da minha vida, por acreditar em mim, sempre me apoiar em todos os momentos, me encorajar diante das dificuldades, se alegrar com minhas conquistas e por todo amor e carinho que sempre recebi.

Aos meus pais, por nunca terem medido esforços para proporcionar, para mim e para o meu irmão, um ensino de qualidade. Também agradeço pela educação que me deram e por serem exemplos de dedicação e perseverança na minha vida.

A Thiago, meu irmão, por todo companherismo e pelas chamadas de atenção durante os momentos de distrações.

Aos integrantes do Nomu, especialmente Rafael e Gabriel, por estarem presentes nos melhores momentos da graduação.

Ao grupo de imigrantes, Luis e Marcos, pelos trabalhos em grupo que sempre deram certo e pelas madrugadas de depuração de códigos e conversas sem sentido.

Por fim, mas não menos importante, a minha família e amigos, por toda confiança e suporte que me proporcionam.

Referências

[Aiken 1994] Aiken, A. (1994). MOSS: A System for Detecting Software Similarity.

[Aljedaani et al. 2021] Aljedaani, W., Peruma, A., Aljohani, A., Alotaibi, M., Mkaouer, M. W., Ouni, A., Newman, C. D., Ghallab, A., and Ludi, S. (2021). Test smell detection tools: A systematic mapping study. *Evaluation and Assessment in Software Engineering*, pages 170–180.

[Bell and Kaiser 2014] Bell, J. and Kaiser, G. (2014). Unit test virtualization with vmvm. In *Proceedings of the 36th International Conference on Software Engineering*, pages 550–561.

[Bell et al. 2015] Bell, J., Kaiser, G., Melski, E., and Dattatreya, M. (2015). Efficient dependency detection for safe java test acceleration. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, pages 770–781.

[Bell et al. 2018] Bell, J., Legunsen, O., Hilton, M., Eloussi, L., Yung, T., and Marinov, D. (2018). Deflaker: Automatically detecting flaky tests. In *Proceedings of the 40th international conference on software engineering*, pages 433–444.

[Camara et al. 2021] Camara, B., Silva, M., Endo, A., and Vergilio, S. (2021). On the use of test smells for prediction of flaky tests. In *Proceedings of the 6th Brazilian Symposium on Systematic and Automated Software Testing*, SAST '21, page 46–54, New York, NY, USA. Association for Computing Machinery.

[Gyori et al. 2015] Gyori, A., Shi, A., Hariri, F., and Marinov, D. (2015). Reliable testing: Detecting state-polluting tests to prevent test dependency. In *Proceedings of the 2015 international symposium on software testing and analysis*, pages 223–233.

[Hao et al. 2013] Hao, D., Lan, T., Zhang, H., Guo, C., and Zhang, L. (2013). Is this a bug or an obsolete test? In *ECOOP 2013–Object-Oriented Programming: 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings 27*, pages 602–628. Springer.

[Herzig and Nagappan 2015] Herzig, K. and Nagappan, N. (2015). Empirically detecting false test alarms using association rules. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 39–48. IEEE.

[Huo and Clause 2014] Huo, C. and Clause, J. (2014). Improving oracle quality by detecting brittle assertions and unused inputs in tests. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 621–631.

[Jiang et al. 2017] Jiang, H., Li, X., Yang, Z., and Xuan, J. (2017). What causes my test alarm? automatic cause analysis for test alarms in system and integration testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 712–723. IEEE.

[Lam 2020] Lam, W. (2020). International Dataset of Flaky Tests (IDoFT).

[Lam et al. 2019] Lam, W., Oei, R., Shi, A., Marinov, D., and Xie, T. (2019). iDFlakies: A framework for detecting and partially classifying flaky tests. In *ICST 2019: 12th IEEE International Conference on Software Testing, Verification and Validation*, pages 312–322, Xi'an, China.

[Lavers and Peters 2008] Lavers, T. and Peters, L. (2008). *Swing Extreme Testing*. Packt Pub.

[Luo et al. 2014] Luo, Q., Hariri, F., Eloussi, L., and Marinov, D. (2014). An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, pages 643–653.

[Memon and Cohen 2013] Memon, A. M. and Cohen, M. B. (2013). Automated testing of gui applications: models, tools, and controlling flakiness. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 1479–1480. IEEE.

[Mens et al. 2008] Mens, T., Demeyer, S., Moonen, L., van Deursen, A., Zaidman, A., and Bruntink, M. (2008). On the interplay between software testing and evolution and its effect on program comprehension. *Software evolution*, pages 173–202.

[Meszaros 2007] Meszaros, G. (2007). *xUnit test patterns: Refactoring test code*. Pearson Education.

[Muşlu et al. 2011] Muşlu, K., Soran, B., and Wuttke, J. (2011). Finding bugs by isolating unit tests. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 496–499.

[Palomba and Zaidman 2017] Palomba, F. and Zaidman, A. (2017). Notice of retraction: Does refactoring of test smells induce fixing flaky tests? In *2017 IEEE international conference on software maintenance and evolution (ICSME)*, pages 1–12. IEEE.

[Palomba and Zaidman 2019] Palomba, F. and Zaidman, A. (2019). Retracted article: The smell of fear: on the relation between test smells and flaky tests. *Empirical Software Engineering*, 24:2907–2946.

[Parry et al. 2021] Parry, O., Kapfhammer, G. M., Hilton, M., and McMinn, P. (2021). A survey of flaky tests. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 31(1):1–74.

[Peruma et al. 2019] Peruma, A., Almalki, K., Newman, C. D., Mkaouer, M. W., Ouni, A., and Palomba, F. (2019). On the distribution of test smells in open source android applications: An exploratory study. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, CASCON '19, page 193–202, USA. IBM Corp.

[Peruma et al. 2020] Peruma, A., Almalki, K., Newman, C. D., Mkaouer, M. W., Ouni, A., and Palomba, F. (2020). 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*, ESEC/FSE 2020, page 1650–1654, New York, NY, USA. Association for Computing Machinery.

[Schleimer et al. 2003] Schleimer, S., Wilkerson, D. S., and Aiken, A. (2003). Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 76–85.

[Sudarshan 2012] Sudarshan, P. (2012). No more flaky tests on the go team. *Accessed: Aug*, 6:2019.

[Van Deursen et al. 2001] Van Deursen, A., Moonen, L., Van Den Bergh, A., and Kok, G. (2001). Refactoring test code. In *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*, pages 92–95. Citeseer.

[Zhang et al. 2014] Zhang, S., Jalali, D., Wuttke, J., Muşlu, K., Lam, W., Ernst, M. D., and Notkin, D. (2014). Empirically revisiting the test independence assumption. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 385–396.