UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Mateus Brandão Damasceno Góes

Monografia em Sistemas de Informação II

IDENTIFICAÇÃO E PREVENÇÃO DE TEST SMELLS:
Uma Abordagem para a Qualidade de Software

Orientador: Eduardo Figueiredo

Belo Horizonte
2024

1. INTRODUÇÃO

Na literatura os *test smells* possuem diversas definições como por exemplo: em termos gerais, os *test smells* são descritos como um conjunto de problemas no código de teste (Hedayati et al., 2015); *test smells* são testes mal projetados, e sua presença pode afetar negativamente conjuntos de testes e código de produção - aspectos como a manutenibilidade ou até mesmo sua funcionalidade - (Van Deursen et al., 2001) (Bavota et al.,2012); e, *test smells* são violações do padrão de teste de quatro fases (configuração, execução, verificação e limpeza), resultando em uma redução em um ou mais critérios de qualidade do teste (Van Rompaey et al., 2007). Inspirados na definição de *code smells*, podemos simplificar a definição de *test smells* da seguinte maneira: Um *test smell* refere-se a qualquer sintoma no código de teste que indica um problema mais profundo (Tufano et al., 2015). Eles podem resultar em testes inadequados, dificultando a identificação de erros e a manutenção do software. Portanto, a identificação e prevenção de *test smells* são questões de grande relevância no campo da engenharia de software.

Este trabalho tem como objetivo abordar esse problema, desenvolvendo uma abordagem para a identificação e prevenção de *test smells* em testes de software, visando melhorar a qualidade do software. Ao longo do trabalho, serão identificados e categorizados os principais tipos de *test smells* encontrados em testes de software, citadas diretrizes e boas práticas para a prevenção desses problemas durante o processo de desenvolvimento de testes e avaliada a eficácia da abordagem proposta por meio de estudos de caso. O resultado final do trabalho será uma visão geral de casos onde o problema ocorre e o estudo das melhores recomendações práticas para desenvolvedores com o objetivo de melhorar a qualidade dos testes e, consequentemente, do software como um todo.

Na fase inicial, a pesquisa buscou uma compreensão profunda dos *test smells*, incluindo uma avaliação das ferramentas existentes e a exploração de métodos de coleta de dados por meio da revisão de artigos e experimentação prática, incluindo técnicas de mineração de dados.

A segunda etapa do trabalho aplicou esse conhecimento em diversos contextos, coletando e analisando dados para criar uma base robusta de informações. Essa

abordagem permite conclusões fundamentadas sobre estratégias eficazes para lidar com o desafio dos *test smells*. A análise dos resultados será conduzida de forma criteriosa, buscando insights que possam informar futuras práticas de desenvolvimento de testes.

Essa abordagem abrangente busca prevenir e corrigir problemas específicos de *test smells*, mas também busca contribuir para a evolução contínua das práticas de garantia de qualidade de software. Ao explorar a fundo as razões da introdução de *test smells* e ao propor soluções concretas, este trabalho procura oferecer uma contribuição para o avanço da qualidade do software na comunidade de engenharia de software.

2. REFERENCIAL TEÓRICO

2.1 TEST SMELLS

Nesta seção, será explorado o conceito, classificação e importância dos *test smells* no contexto de testes de software. Os *test smells* representam sinais indicativos de problemas potenciais nos testes, comprometendo a eficácia do processo de garantia de qualidade e, consequentemente, a qualidade do software final. O conceito de *test smells* envolve indícios de práticas inadequadas nos testes de software, manifestando-se como padrões reconhecíveis que apontam para possíveis falhas e defeitos no processo de desenvolvimento e execução de testes (Meszaros, 2007). Identificar esses sinais é importante para a detecção precoce de problemas, permitindo correções antes que impactem a integridade do software.

A diversidade de *test smells* exige uma classificação sistemática para uma compreensão mais profunda. Tipos comuns incluem duplicação de lógica de teste e testes longos demais. Cada tipo será abordado com detalhes, fornecendo exemplos práticos para ilustrar a natureza desses problemas. A relevância dos *test smells* na engenharia de software reside na capacidade de sinalizar áreas propensas a erros e dificuldades de manutenção. Ignorar ou negligenciar esses sinais pode resultar em testes inadequados, prejudicando a identificação de falhas e comprometendo a qualidade do software entregue.

Na literatura, a catalogação de *test smells* teve início com o artigo *Refactoring Test Code* (Van Deursen et al., 2001), que introduziu um catálogo inicial composto por 11 *test smells*, além de propor padrões de refatoração para corrigi-los. Atualmente, a

pesquisa nesse domínio evoluiu, e um estudo posterior (Garousi et al., 2018) já identificou aproximadamente 196 tipos de *smells*. A seguir, serão apresentados os principais tipos de *test smells* considerados mais comuns na literatura:

2.1.1 Assertion Roulette

Assertion Roulette é um padrão de *test smell* que foi caracterizado inicialmente por ter um número de asserções que não tem explicação (Van Deursen et al., 2001). Mais recentemente esse padrão foi descrito como "Casos de teste que contém múltiplas asserções não documentadas" (Garousi and Kuçuk, 2018). Esse *test smell* pode ser caracterizado também pelo seu sintoma: "É difícil determinar qual dos *asserts* dentro do mesmo método de teste causou a falha no teste." (Meszaros, 2007, p.224)

Esse problema pode ter diversas causas para sua ocorrência, uma delas é o caso em que o método de teste tenta verificar muitas funcionalidades de uma vez, o chamado *Eager Test*, que será explicado em tópicos posteriores. Abaixo mostro um exemplo de Assertion Roulette:

```
public void testGetFieldTypes() throws Throwable {
    MockPartial mock = new MockPartial();
    DateTimeFieldType[] vals = mock.getFieldTypes();
    assertEquals("Add Assertion Explanation here", 2, vals.length);
    assertEquals(DateTimeFieldType.year(), vals[0]);
    assertEquals(DateTimeFieldType.monthOfYear(), vals[1]);
}
```

Figura 1. Exemplo de Assertion Roulette

Na imagem acima, podemos perceber que o caso de teste possui diversas asserções, porém os dois métodos ao final do código - nas chamadas de *assertEquals* - não possuem qualquer mensagem ou documentação para explicar a sua existência. Isso pode gerar um grande problema porque se o teste falha, se torna difícil dizer qual asserção realmente falhou. Em um caso onde o método é mais complexo, achar a causa do problema pode ser difícil e consumir muito tempo.

Uma possível maneira de se corrigir esse problema é dividir o teste em casos de um único *assert* (*Single-Condition Tests*), para isso, pode-se usar a refatoração de Extração de Método (Fowler, 1999, p.110) para criar métodos de teste separados, cada um contendo sua lógica de teste específica

2.1.2 Conditional Test Logic

Conditional Test Logic é um tipo de *test smell* que possui estruturas de controle presentes no método, foi descrito por Meszaros (2007) como um teste que contém código que pode ou não ser executado. Esse tipo de *test smell* é fácil de ser identificado pois se o teste possui alguma estrutura de controle já é o suficiente para levantar suspeitas.

O grande problema desse padrão é que a complexidade do código fica muito mais elevada, já que o comportamento do teste passa a ser imprevisível. A imprevisibilidade associada a esse tipo de *test smell* pode resultar em testes que falham de maneira inconsistente ou que não cobrem adequadamente os diferentes cenários. Além disso, a complexidade aumentada do código de teste pode dificultar a identificação e correção de falhas, contribuindo para um processo de desenvolvimento mais propenso a erros. Assim, é recomendável evitar estruturas de controle em testes sempre que possível e buscar abordagens mais claras e determinísticas na escrita do código.

Um exemplo de Conditonal Test Logic em um código de produção real pode ser visto no repositório do aplicativo de gestão estudantil da universidade de Darmstadt na Alemanha, demonstrado abaixo:

```
@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));
                }
            }
         }
    }
}
```

Figura 2. Exemplo de Conditional Test Logic

Como podemos perceber, o código se torna extremamente confuso ao adicionar as estruturas, não se sabe exatamente qual comportamento está sendo testado e qual parte do código será executada. Uma maneira de corrigir isso seria realizar a Extração de Método assim como foi estabelecida na solução do Assertion Roulette, dessa forma, cada parte do código que pode ser executada deve ter seu próprio caso de teste.

2.1.3 General Fixture

General Fixture é um *smell* definido por Van Deursen (2001) que contextualiza no contexto do JUnit, que permite a criação de um método chamado setUp que executa sempre antes da chamada dos casos de testes. O problema surge quando o setUp, ou fixture, passa a ser generalizante e diferentes testes acessam apenas partes da fixture.

Esse *smell* gera diferentes tipos de complicação, a primeira e mais clara é a dificuldade de entendimento, já que, ao ler a fixture não será certo qual parte dela será realmente utilizada no método de teste que está sendo analisado. Outro problema é que os testes podem passar a executar mais lentamente, uma vez que, é feito trabalho desnecessário sempre que o método é executado.

Uma maneira de resolver esse problema seria aplicando a refatoração de Inline Method (Fowler, 1999, p.117), onde as linhas de código e comportamento criado na fixture que são aplicados em apenas alguns métodos devem ser inseridas apenas nesses métodos respectivos. Van Deursen sugere também que o Inline Method deve ser aplicado criando um método separado se for necessário inserir o código em múltiplos testes, para evitar a duplicação de código.

2.1.4 Eager Test

Eager Test é um tipo de *smell*, que pode ser considerado também como uma causa para outros tipos de test *smell* também, assim como visto no caso do *Assertion Roulette*, onde esses dois problemas possuem uma alta correlação entre si (Peruma, 2019).

Esse tipo de *smell* é caracterizado como um método de teste que checa diversos métodos do objeto a ser testado. Em termos gerais, pode ser considerado um teste que verifica muitas funcionalidades em um método de teste singular (Meszaros, 2007, p.187).

Além de servir de causa para outros problemas no teste, esse *smell* isoladamente deixa o método de teste difícil de ler e entender, já que não se sabe qual é o comportamento específico que a função procura testar. Um exemplo claro disso foi encontrado no código de produção no repositório do Github do aplicativo chamado GPSLogger, como demonstrado abaixo:

```
@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"));
}
```

Figura 3. Exemplo de Eager Test

Neste método de teste, o desenvolvedor chama vários métodos da classe NmeaSentence. Testar vários métodos da classe de produção em um único método de teste causa confusão sobre o que exatamente o método de teste está avaliando.

2.1.5 Lazy Test

Lazy Test é um tipo de *test smell* que ocorre quando vários métodos de teste verificam o mesmo método utilizando a mesma fixture, ou setUp (Van Deursen, 2001). Outros autores já consideram esse *smell* sempre que dois ou mais métodos de teste invocam o mesmo método do objeto de produção que está sendo testado - não necessariamente utilizando a fixture (Peruma et al., 2019).

Isso é um problema porque se os testes usam do mesmo método de produção nos seus testes, eles têm comportamento similar e, portanto, gera um acoplamento entre os casos de teste diferentes. Esse acoplamento gera um sério problema se o método do objeto testado for alterado, visto que, vários testes diferentes devem ser modificados para se adequarem às novas mudanças.

```
@Test
public void testDecrypt() throws Exception {
                ...
    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);
}
```

Figura 4. Exemplo de Lazy Test

A imagem acima representa um exemplo de Lazy Test, já que o método *Cryptographer.decrypt* é chamado em dois casos de teste diferentes. Uma maneira de corrigir esse problema seria agrupar a lógica dos diferentes casos em um único caso que representa o teste do comportamento do método.

2.1.6 Mystery Guest

O *test smell* chamado de Mystery Guest ocorre quando um teste usa de recursos externos como arquivos, banco de dados. Isso é um problema porque o código em si passa a não ter informações suficientes para entender a funcionalidade testada, dificultando o uso do teste como documentação (Van Deursen, 2001).

Esse padrão, além da dificuldade de entendimento, também adiciona a questão

da dependência dos recursos externos, onde um recurso se for editado, ou corrompido, o teste passa a falhar sendo que não houve alteração do sistema que está sendo testado (SUT). Um outro problema que esse *test smell* trás é o overhead necessário para utilizar esses recursos necessários, podendo deixar os outros testes ou até mesmo o sistema com uma lentidão maior.

Maneiras de corrigir esse problema variam desde a criação de um Mock, ou seja, uma simulação do recurso externo utilizado dentro do método de teste ou a alocação do recurso especificamente dentro do método e liberando o que foi alocado ao finalizar o teste.

3 Jnose

Para realizar a detecção e a análise dos dados que serão incluídos na próxima etapa do trabalho, será utilizada a ferramenta *Jnose* apresentada no artigo *On the influence of Test Smells on Test Coverage* (Virginio et al., 2019).

O programa é uma ferramenta *open-source* que funciona para a detecção de 19 tipos de test *smell* diferentes em códigos da linguagem Java que usam como base a biblioteca JUnit, para a criação de seus testes de unidade. Alcançando uma pontuação de precisão entre 85% e 100% e uma pontuação de recall variando de 90% a 100% (Peruma et al, 2020).

A arquitetura do Jnose foi criada como uma extensão da ferramenta *TsDetect* com sua própria interface de usuário e com o uso da inserção direta de repositórios do github, disponibilizada como um arquivo .jar autônomo. Esta abordagem é interessante porque não tem a necessidade de um ambiente de desenvolvimento integrado (IDE) específico para os códigos de produção que serão executados.

Para que o *Jnose* funcione é preciso passar um link para um repositório do github que se quer testar ou o caminho para seu diretório no computador na aba *By ClassTest*, o programa faz o *clone* do repositório e começa a etapa de análise usando o *tsDetect*, ao final, gera-se um arquivo .csv contendo todos os tipos de *test smells* encontrados para cada classe de testes presente no repositório.

A partir da leitura dos arquivos, o programa gera uma Árvore Sintática Abstrata (AST) utilizando a biblioteca JavaParser para realizar a análise dos diferentes *test*

smells que foram compactados em diferentes módulos do código fonte do projeto, o que facilita a possibilidade da criação de novos meios de detecção de diferentes *test smells* utilizando o código da ferramenta inicial.

Na figura abaixo apresento um diagrama simplificado (Peruma et al, 2020) da arquitetura do código do Jnose:

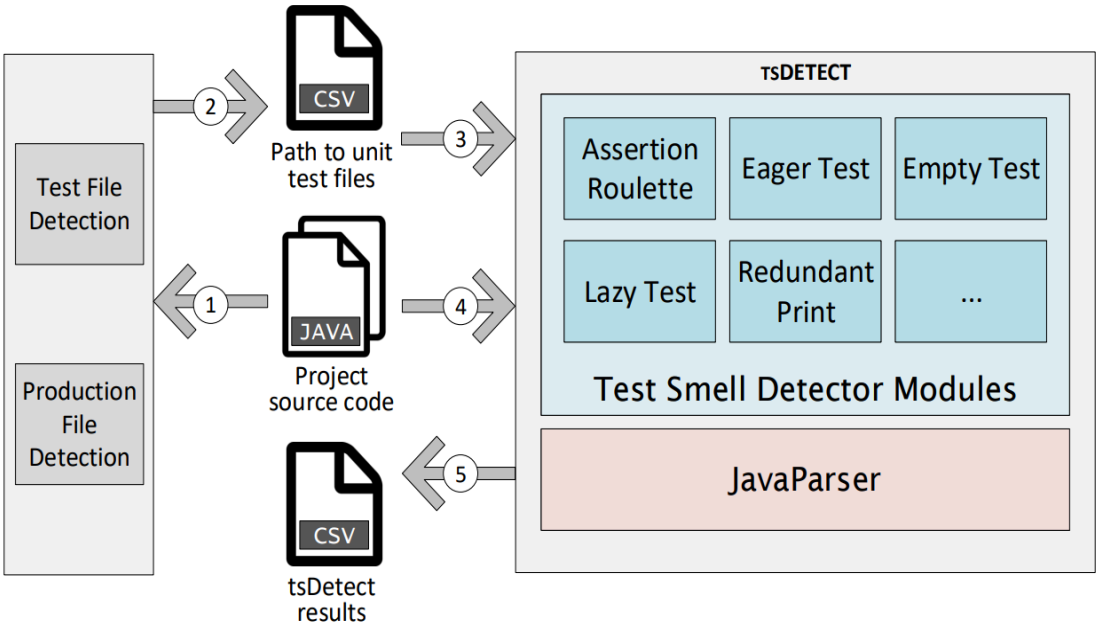

Figura 5. Arquitetura em alto nível do Jnose

Como podemos ver acima, em 1 e 2, ocorre a identificação dos arquivos do teste e da classe que está sendo testada. Em 3 e 4, a ferramenta detecta os diferentes tipos de *test smell* e em 5, gera o arquivo de output com o resultado da análise feita. Esse arquivo contém uma lista de *test smells* presentes em cada classe e a quantidade deles que foram detectados separados por classe. Dessa forma, podemos extrair esses dados para programas de banco de dados relacionais e utilizá-los para coletar informações dos códigos apontados para análise.

Na próxima etapa, foram escolhidas diversas bases de código no github para entender melhor o problema completo dos *test smells* em códigos de produção e, a partir desses dados, serão propostas abordagens e sugestões para a resolução dos problemas, bem como uma análise da proporção da quantidade e dos tipos de *test smells* por projeto definido.

A escolha dos repositórios foi realizada com alguns critérios determinados: com

um grande número de contribuidores, que seja consolidado e usado em produção, com pelo menos mais de 500 estrelas. Além disso, como descrito, os repositórios precisam conter testes de unidade automatizados utilizando o framework JUnit. Esses critérios de escolha podem ser demonstrados no diagrama a seguir:

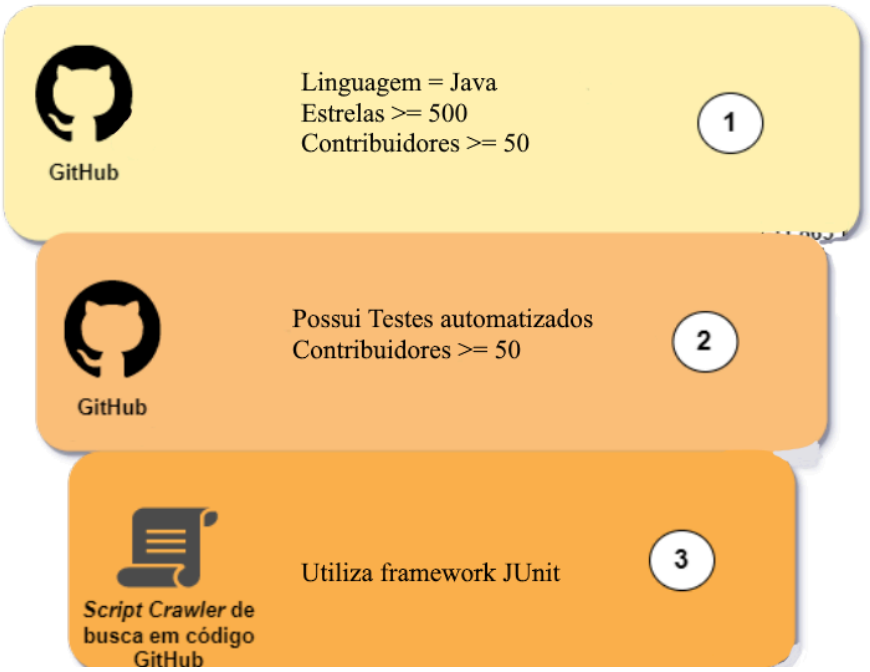

Figura 6. Critérios de escolha dos repositórios no Github

4 Análise de repositórios

A partir dos critérios de escolha dos repositórios apresentados acima, foram determinadas 10 das principais bases de código em Java para que as análises sejam realizadas. As escolhas foram feitas também de forma a abrangir diferentes funções das aplicações. A partir da escolha, foi utilizado o Jnose para obter dados relacionados aos *test smells* presentes em cada um dos repositórios. Os repositórios escolhidos foram:

- **Commons-lang**: Biblioteca de utilitários que complementa a linguagem Java com métodos adicionais para manipulação de strings, números, datas, etc.
- **Jfreechart**: Biblioteca Java para criação de gráficos e visualizações de dados.
- **Elasticsearch**: Motor de busca e análise distribuído, baseado em RESTful, utilizado para pesquisa em texto completo e análise de grandes volumes de dados.

- **HikariCP**: Pool de conexões JDBC de alta performance para Java.
- **Cryptomator**: Aplicativo de software livre para criptografia de arquivos e pastas, proporcionando segurança para dados armazenados em serviços de nuvem.
- **Libgdx**: Framework de desenvolvimento de jogos em Java, oferecendo suporte para múltiplas plataformas.
- **Stirling-PDF**: Biblioteca Java para manipulação e criação de arquivos PDF.
- **Redisson**: Biblioteca de cliente Redis para Java que oferece fácil integração com Redis e suporte para estruturas de dados distribuídas e utilitários de alta performance.
- **Jenkins**: Ferramenta de integração contínua e entrega contínua, escrita em Java, para automatização de processos de desenvolvimento.
- **Arthas**: Ferramenta de diagnóstico e monitoramento em tempo real para aplicativos Java, desenvolvida para ajudar a solucionar problemas em ambientes de produção

4.1 Dados dos repositórios

Nome	Data inicial	Linhas de código	Estrelas	Contribuidores	Issues abertas	Métodos de teste
Commons-lang	21/05/2009	181.786	2.694	213	0	2.300
Jfreechart	01/02/2016	283.002	1.177	26	109	2.399
Elasticsearch	08/02/2010	109.823	68.745	1957	3770	8.589
HikariCP	08/10/2013	19.733	19.702	139	491	91
Cryptomator	02/02/2014	27.231	11.074	56	252	77
Libgdx	10/08/2012	235.854	23.026	564	238	108
Stirling-PDF	27/01/2023	20.528	35.081	141	208	53
Redisson	11/01/2014	274.818	23.043	160	349	2.584
Jenkins	22/11/2010	312.717	22.790	787	0	3.850
Arthas	29/08/2018	73.461	35.164	199	316	216

Tabela 1. Dados dos repositórios escolhidos

Os dados acima foram obtidos através da API do Github, as linhas de código contabilizam os arquivos que possuem a extensão .java no projeto. Os repositórios de commons-lang e Jenkins não permitem a abertura de issues, por isso o valor é 0 para ambos.

A partir das informações obtidas acima, podemos ver que para a análise dos dados foram escolhidos repositórios importantes para o desenvolvimento em Java já que possuem um número elevado de estrelas e contribuidores. Além disso, eles englobam diversos tipos de uso diferentes, como bibliotecas, aplicações, frameworks, etc.

Após a escolha e análise inicial, foi feito o uso do programa *Jnose* em cada um dos repositórios para determinar os tipos e a quantidade de *test smells* presentes. Os 10 principais *test smells* determinados pelos relatórios gerados pelo Jnose foram escolhidos. Os critérios para a detecção de cada um dos tipos de *test smells* foram descritos no artigo *On the test smells detection: an empirical study on the JNose Test accuracy* (Virgínio et al., 2021) e estão descritos na tabela abaixo:

Nome	Regra de detecção
Assertion Roulette	Uma linha com instruções de asserção sem o parâmetro de explicação/mensagem
Conditional Test Logic	Um bloco de código com instruções condicionais
General Fixture	Uma linha com um campo instanciado dentro do método setUp() que não é utilizado por todos os métodos de teste
Eager Test	Uma linha que contém uma chamada para outro método de produção
Lazy Test	Uma linha de método que chama o mesmo método de produção que outro método de teste
Mystery Guest	Um método que acessa instâncias de objetos de classes de arquivos e bancos de dados
Sensitive Equality	Um método que contém uma asserção que invoca o método toString() de um objeto
Magic Number Test	Uma linha com método de asserção que contém um literal numérico como argumento
Verbose Test	Um método com mais de 30 linhas, contando instruções não executáveis e anotações
Unknown Test	Um método que usa a anotação @Test, mas não contém instruções de asserção

Tabela 2. Regras para a detecção de *test smells*

A quantidade de *test smells* encontrada para cada um dos repositórios foi determinada pelo programa e estão descritas abaixo

Test Smell	Commons-lang	JFree Chart	ElasticSearch	HikariCP	Cryptomator	Libgdx	Stirling-PDF	Redisson	Jenkins	Arthas	Total
Assertion Roulette	9173	9119	9014	123	101	319	91	5084	4973	542	38.539
Conditional Test Logic	383	8	1315	13	0	61	0	506	332	26	2.644
General Fixture	38	1	650	0	0	0	1	105	46	2	843
Eager Test	668	1223	1079	14	11	59	1	871	594	45	4.565
Lazy Test	464	822	848	26	23	47	6	507	788	58	3.589
Mystery Guest	4	0	79	0	1	0	0	3	209	13	309
Sensitive Equality	319	5	1315	1	7	10	0	51	310	5	2.023
Magic Number Test	2831	2738	1879	70	33	275	17	916	1163	66	9.988
Verbose Test	163	198	787	5	1	14	0	115	185	13	1.481
Unknown Test	85	63	405	10	5	0	0	215	177	11	971
Total	14.128	14.177	17.371	262	182	785	116	8.373	8.777	781	

Tabela 3. Quantidade de instâncias de *test smells* presentes nos repositórios por tipo

Com os dados obtidos, podemos extrair alguns fatos interessantes sobre *test smells* em geral. Primeiramente, podemos notar que existe uma grande quantidade do tipo *Assertion Roulette.* Isso ocorre porque cada *assert* a mais sem explicação contido dentro de cada teste é contabilizado como um *test smell*. Durante as pesquisas pelos repositórios, notou-se que grande parte dos testes continham 2 ou mais *asserts* sem explicação por método testado. Um possível motivo para isso se deve a tentativa dos desenvolvedores de utilizar um método de teste para múltiplas instâncias de certo comportamento, isso agiliza o processo de escrever o teste, porém deixa o teste mais difícil de ser compreendido.

Outro destaque é a grande presença do *Conditional Test Logic*. Esse é um *test smell* que normalmente deixa o código de teste confuso devido a presença de estruturas

lógicas. Esse é um *test smell* que pode ser facilmente evitado e, mesmo assim, apresenta um grande número nos grandes projetos como os analisados, reforçando a necessidade de adotar melhores práticas na escrita de testes para garantir a manutenção e legibilidade dos testes ao longo do tempo.

A partir dos dados da Tabela 1 e da Tabela 3, podemos construir diversos gráficos para buscar possíveis correlações entre os metadados dos repositórios e os dados obtidos sobre a quantidade de *test smells* presentes

4.2 Análise de dados

Para ilustrar os diferentes tipos de *test smells* nos diferentes projetos, foi criado um gráfico de distribuição abaixo com os projetos com maior quantidade de *smells*. A partir desse gráfico, nota-se a discrepância entre os principais tipos de *test smells,* como o *Assertion Roulette* e *Magic Number Test*, comparado aos outros *smells.* Nota-se também que os repositórios grandes e com muito casos de teste possuem alta incidência de diferentes tipos de *test smells*, como é o caso do Commons-lang e ElasticSearch

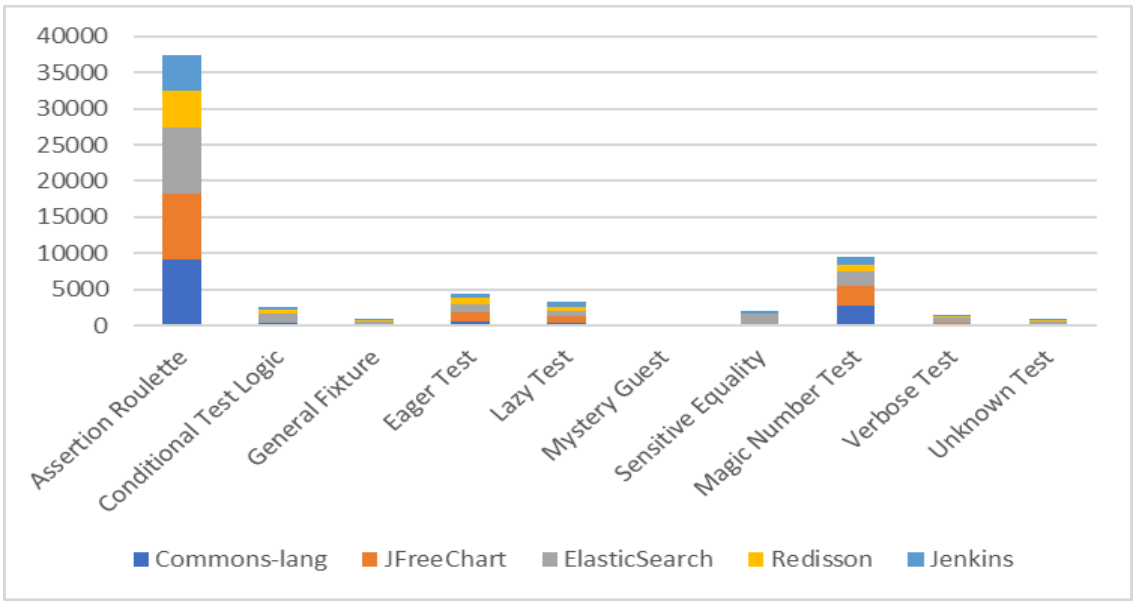

Figura 7. Distribuição de tipos de *test smells* nos sistemas

Uma análise interessante para ser feita é a relação entre a quantidade de linha de códigos dos repositórios comparado com a quantidade de *test smells* presentes.

Espera-se que exista uma certa correlação, visto que com menos linhas de código, menos testes serão criados, e com menos testes a possibilidade de existirem erros neles é menor. Porém, na teoria essa correlação não é necessária, visto que um repositório com grande quantidade de testes pode se atentar para que os testes não apresentem nenhum *smell* em sua composição.

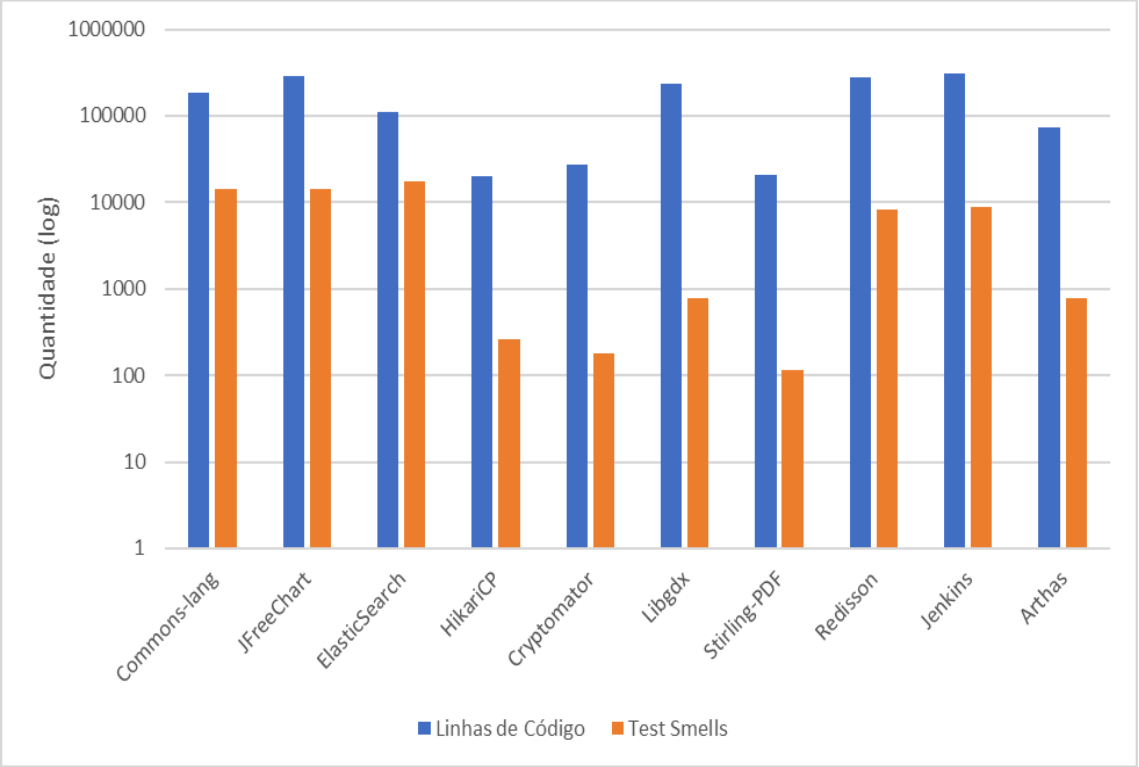

Figura 8. Gráfico de agrupamento de Linhas de código e *test smells* por repositório

Como pode ser visto acima, existe uma certa correlação entre a quantidade de linhas de código e a quantidade de *test smells* presentes. Porém, existem alguns casos, como HikariCP, Cryptomator e Libgdx, onde a quantidade de linhas de código é bem maior que a quantidade de *test smells* presentes. Isso indica que nem sempre a quantidade total de linhas de código implicará em maior incidência de *test smells*.

Para buscar possíveis correlações entre as métricas apresentadas e a quantidade de *test smells*, foi feito uma análise usando a biblioteca *pandas* em python, para criar um mapa de calor que apresenta as possíveis correlações e está descrito abaixo:

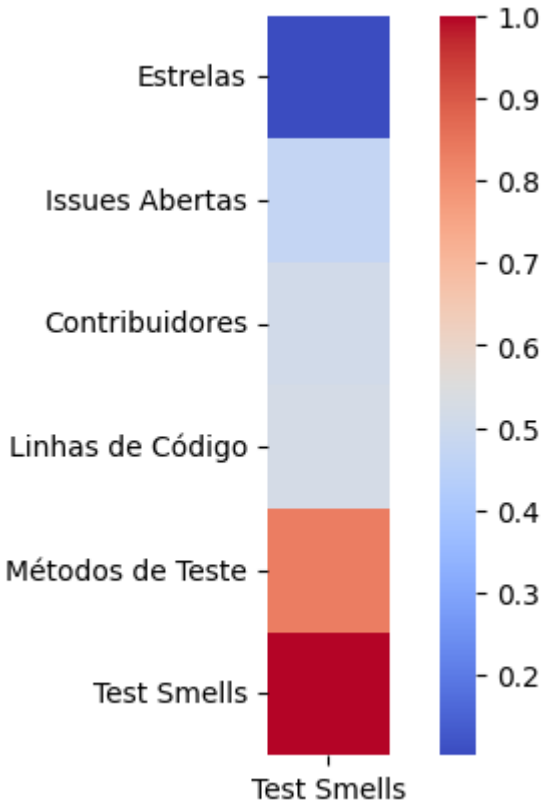

Figura 9. Mapa de calor de correlações entre *test smells* e métricas dos repositórios

A imagem acima mostra que a principal correlação com a quantidade de *test smells* é a quantidade de métodos de teste presentes no código. Ou seja, a incidência de *test smells* em repositórios públicos como os analisados depende fortemente da quantidade de métodos inseridos. Isso ocorre porque os desenvolvedores tendem a gastar grande parte de seu tempo com desenvolvimento e apenas uma pequena parcela com o desenvolvimento de testes. Em uma pesquisa feita por Straubinger e Fraser no artigo *A Survey on What Developers Think About Testing* (2023), foi analisado que cerca de 44% dos participantes alocaram menos de 20% do seu tempo para desenvolvimento de testes e 44% dos desenvolvedores escrevem apenas testes simples para bater as metas de cobertura com o mínimo de esforço. Esses são fatores que corroboram para o surgimento de *test smells* já que os desenvolvedores com pouco tempo e atenção tendem a gerar mais erros.

A quantidade de contribuidores, estrelas e número de *issues* abertas não parecem

ter correlação direta com a presença de *test smells*. Isso ocorre porque os contribuidores usualmente tendem a seguir o mesmo padrão de código do repositório, incluindo os testes, o que gera testes similares entre os contribuidores, seja testes bons ou testes com *smells*. Já o número de issues normalmente está atrelado a qualidade do código ou a quantidade de ferramentas ou funções que o repositório oferece.

Por fim, os dados descritos acima demonstram um grande problema presente atualmente nos softwares devido a sua grande quantidade de *test smells*. A grande quantidade desses *smells* tem um forte impacto na identificação e avaliação da corretude de testes e podem levar a erros em produção e em uma quantidade de prejuízo monetário relacionado ao tempo gasto pelo desenvolvedor para corrigir os problemas e refatorar o código e testes (Bavota et al., 2014)

5. Conclusão

O estudo acima, abordou o problema dos *test smells* ressaltando sua grande importância na identificação precoce de problemas nos testes de software. A diversidade de padrões ao longo do tempo destaca a complexidade desses sinais de práticas inadequadas. Além disso, através da ferramenta Jnose, traçou-se uma base interessante para realizar a coleta e extração de dados em diversos repositórios reais, para tentar aplicar o conhecimento adquirido em um contexto diverso, além da academia e literatura.

Os resultados obtidos na análise do código nos repositórios demonstra um cenário preocupante devido a alta quantidade de *test smells* presentes nos projetos analisados reforçando a necessidade de se aplicar boas práticas nos testes e dos desenvolvedores dedicarem mais tempo a escrita dos testes, também é importante aplicar as práticas de refatoração específica para cada um dos *smells* como foi descrito na etapa inicial desta monografia.

A metodologia procurou estabelecer uma base sólida de conhecimento, buscando obter insights valiosos sobre como os *test smells* afetam projetos de software reais. Espera-se que esses resultados contribuam para aprimorar as práticas de teste, impulsionando o desenvolvimento de software mais robusto e de alta qualidade.

6. Metodologia

A metodologia seguida para este estudo envolveu diversas etapas, visando obter

informações relevantes e uma compreensão profunda do tema. Inicialmente, foram lidos os artigos recomendados pelo orientador, com foco na identificação dos pontos-chave e aspectos importantes. De forma paralela, foram feitas consultas online para esclarecer conceitos mais densos.

Após a fase inicial de pesquisa, foi feito à revisão dos pontos-chave identificados, destacando as informações relevantes. Estas foram compiladas de forma resumida, resultando em um guia de consulta para a segunda parte da monografia. A metodologia adotada combinou a leitura de fontes bibliográficas, consultas online e a síntese das informações, proporcionando uma base sólida para conduzir o estudo e redigir a monografia.

Por fim, foi realizada a parte prática onde foi feita a busca de projetos de código aberto no Github, e o uso da ferramenta Jnose em cima das bases de códigos escolhidas. A partir desses dados gerou-se as visualizações em gráficos e análise dos dados relacionados aos *test smells* de grandes projetos.

7. Trabalhos Relacionados

O principal trabalho que introduziu o conceito dos *test smells* e serviu de base inicial de pesquisa foi o artigo intitulado *Refactoring Test Code* de *Van Deursen et al.* (2001) que trouxe a tona os principais conceitos e maneiras de refatoração que são utilizadas até hoje no contexto da Engenharia de Software.

Outros trabalhos importantes para a análise e para o uso das ferramentas explicitadas foram *An empirical study into the relationship between class features and test smells* (Tahir et al., 2016), que introduziu o contexto da análise de *test smells* no contexto de software de produção no mercado e, por fim, *Smells in software test code: A survey of knowledge in industry and academia* (Garousi and Kuçuk, 2018) que trouxe à tona a revisão da literatura no contexto de testes de software e classificou e catalogou diversos tipos de *test smells*.

REFERÊNCIAS BIBLIOGRÁFICAS

Hedayati, A.; Ebrahimzadeh, M., Sori, A. (2015) "Investigating into Automated Test Patterns in Erratic Tests by Considering Complex Objects," International Journal on Information Technology and Computer Science.

Van Deursen, A., Moonen L., Van den Bergh, A., Kok G. (2001) "Refactoring Test Code," International Conference on extreme programming and flexible processes in software engineering.

Bavota, G., Abdallah, Q., Oliveto, R., Lucia, A., Binkley, D. (2012) "An empirical analysis of the distribution of unit test smells and their impact on software maintenance", IEEE International Conference on Software Maintenance.

Van Rompaey, B.; Du Bois, B.; Demeyer, S.; Rieger, M. (2007) "On The Detection of Test Smells: A Metrics-Based Approach for General Fixture and Eager Test," IEEE Transactions on Software Engineering.

Tufano, M.; Palomba, F.; Bavota, G.; Di Penta, M.; Oliveto, R.; De Lucia, A.; Poshyvanyk, D. (2016) "An empirical investigation into the nature of test smells," IEEE/ACM International Conference on Automated Software Engineering.

Spadini, D.; Palomba, F.; Zaidman, A.; Bruntink, M.; Bacchelli, A. (2018) "On the Relation of Test Smells to Software Code Quality," IEEE International Conference on Software Maintenance and Evolution (ICSME).

Peruma, A., Almalki, K., Newman, C. D., Mkaouer, M. W., Ouni, A., Palomba, F. (2020) "TsDetect: An Open Source Test Smells Detection Tool," ESEC/FSE.

Meszaros, G. (2007) "xUnit Test Patterns: Refactoring Test Code." Boston, Estados Unidos: Pearson Education.

Fowler, M; Beck, K.; Brant, J.; Opdyke, W.; Roberts, D. (1999) “Refactoring: Improving the Design of Existing Code.” Boston, Estados Unidos: Addison-Wesley Longman Publishing Co.

Peruma, A.; Almalki, K.; Newman, C.; Mkaouer, M.; Ouni, A. (2019) "On the Distribution of Test Smells in Open Source Android Applications: An Exploratory Study."

Tahir, A.; Counsell, S.; Macdonell, S. A. (2016) "An empirical study into the relationship between class features and test smells," Asia-Pacific Software Engineering Conference (APSEC).

Virgínio, T.; Santana, R.; Martins, L.; Soares, L.; Costa, H.; Machado, I. (2019) “On the influence of Test Smells on Test Coverage”

Virgínio, T.; Martins, L.; Santana, R.; Cruz, A.; Rocha, L.; Costa, H.; Machado, I. (2021) “On the test smells detection: an empirical study on the JNose Test accuracy”

Straubinger, P.; Fraser, G. (2023) “A Survey on What Developers Think About Testing”

Bavota, G.; Qusef, A.; Oliveto, R.; Lucia, A.; Binkley, D. (2014) “Are test smells really harmful? An empirical study”

Garousi, V.; Kuçuk, B.; Felderer, M. (2018) "What we know about smells in software test code."

Garousi, V.; Kuçuk, B.; Felderer, M. (2018) "Smells in software test code: A survey of knowledge in industry and academia." The Journal of Systems and Software