Trabalho de Conclusão de Curso

Detecção de *smells* em testes automatizados em diferentes linguagens de programação

Gustavo Augusto Calazans Lopes
gacl@ic.ufal.br

Orientadores:

Prof. Dr. Márcio de Medeiros Ribeiro
Prof. Me. Elvys Alves Soares

Maceió, Março de 2023

Gustavo Augusto Calazans Lopes

Detecção de *smells* em testes automatizados em diferentes linguagens de programação

Monografia apresentada como requisito parcial para obtenção do grau de Bacharel em Ciência de Computação do Instituto de Computação da Universidade Federal de Alagoas.

Orientadores:

Prof. Dr. Márcio de Medeiros Ribeiro

Prof. Me. Elvys Alves Soares

Maceió, Março de 2023

Catalogação na Fonte
Universidade Federal de Alagoas
Biblioteca Central
Divisão de Tratamento Técnico
Bibliotecário: Marcelino de Carvalho Freitas Neto – CRB-4 - 1767

L864d Lopes, Gustavo Augusto Calazans.
Detecção de *smells* em testes automatizados em diferentes linguagens de programação / Gustavo Augusto Calazans Lopes. – 2023.
54 f. : il.

Orientador: Márcio de Medeiros Ribeiro.
Coorientador: Elvys Alves Soares.
Monografia (Trabalho de conclusão de curso em Ciência da Computação) – Universidade Federal de Alagoas, Instituto de Computação. Maceió, 2023.

Bibliografia: f. 49-54.

1. *Software* - Testes. 2. Testes Automatizados. 3. *Code smells.* I. Título.

CDU: 004.4

Agradecimentos

A Deus, sem Ele nada seria possível. A Ele a glória.

A minha mãe, que durante toda minha vida se dedicou e esforçou a me criar, prover uma educação de qualidade e todo o incentivo e apoio, apesar de todas as dificuldades.

Ao meu padrinho Antony, que sempre esteve presente. Uma das minhas principais referências e o maior motivador para eu seguir na área de tecnologia.

Ao meus avós, Antônio e Janete, pelos quais possuo profundo carinho e gratidão, que me acolheram em sua casa e cuidaram de mim. Durante todo esse tempo tive a oportunidade de aprender muito e ser quem eu sou hoje. Ao meu tio Tulio e primos, que também me acompanharam nesse processo.

A Luiza, que tive o prazer de conhecer no início do curso, admiro muito sua inteligência, o que motiva a sempre ser melhor.

Aos meus amigos Davi, Guilherme e Pedro, que também foram essenciais para o meu desenvolvimento durante o curso, foram vários anos de muita luta e são amigos que vou levar para a vida. E ao meu grande amigo Philipe, que conheci no ensino fundamental e com quem sempre pude contar.

Aos professores Leandro de Sales, Márcio Ribeiro, Rodrigo Paes e Thiago Cordeiro que serviram como inspiração com seus conhecimentos e dedicação em ensinar.

Gostaria também de agradecer ao Elvys Soares que me orientou com toda paciência, dedicação e suporte durante todo o desenvolvimento do TCC.

E por fim, mas não menos importante, gostaria também de agradecer a todas as pessoas que de certo modo contribuíram na minha formação, das quais posso citar meus tios Alderizo e Dagmar, a dona Eliane, seu Júlio e dona Erci, Aparecida e Sidney.

Resumo

Testes de software são importantes para qualquer produto digital em desenvolvimento ou já em uso. Eles servem como garantia que o sistema está funcionando conforme o esperado e que em eventuais manutenções ou criação de novas funcionalidades, não irão interferir no atual funcionamento do sistema. Os testes automatizados e desenvolvimentos de *scripts* de teste são predominantes na indústria de software. Contudo, assim como códigos de produção, a escrita dos testes automatizados também podem conter problemas no projeto ou na implementação, afetando negativamente a qualidade, os denominados *tests smells*. A identificação desses *smells* por parte dos desenvolvedores não é trivial, o que leva a utilização de ferramentas para isso. Na pesquisa realizada foi visto que, apesar de existirem muitas ferramentas para identificação de smells, não existe uma certa diversidade quanto as linguagens suportadas, além da necessidade de desenvolvimento de novas ferramentas para cada linguagem ou *framework* de testes automatizados. O presente trabalho de conclusão de curso tem por objetivo apresentar uma única ferramenta para identificação de *tests smells* com suporte para diferentes linguagens e *frameworks* de desenvolvimento de testes automatizados. Com isso, intenta-se sanar a necessidade de falta de suporte e retrabalho com a criação de novas ferramentas para um mesmo propósito.

Palavras-chave: Testes de Software; Testes Automatizados; Tests Smells; Ferramenta de teste.

Abstract

Software testing is important for any digital product under development or already in use. It serve as a guarantee that the system is working as expected and that any maintenance or creation of new functionalities will not interfere with the current system operation. Automated tests and test script development are very common in the software industry. However, just like production codes, the writing of automated tests can also contain bad design, negatively affecting quality, as known as test smells. The identification of these smells by developers is not trivial, which leads to the use of tools for this. In the research conducted, it was seen that, although there are many tools for identifying smells, there is not a certain diversity in terms of supported languages, besides to the need to develop new tools for each language or automated test frameworks. The present work aims to present a single tool for identifying test smells with support for different languages and frameworks for developing automated tests. With this, it is intended to remedy the need for lack of support and rework with the creation of new tools for the same goal.

Key-words: Software testing; Automated testing; Tests Smells; Smell detection tools.
Figuras

Lista de Figuras

2.1 Pirâmide de testes [21]. 5
2.2 Pirâmide de testes automatizados [12]. 5
2.3 Diferença no crescimento de um software com e sem testes [33]. 7
2.4 Analogia de testes de integração com peças de quebra-cabeça [54]. 8
2.5 Exemplo de integração com serviços externos [21]. 8
4.1 Exemplo de detecção utilizando a técnica *Information Retrieval* pela ferramenta TEDD [10]. 27
4.2 Representação gráfica de uma AST [1] 29
5.1 Arquitetura da ferramenta. Fonte: autoria própria. 34

Conteúdo

Lista de Figuras **iii**

1 Introdução **1**

2 Fundamentação teórica **3**
2.1 Teste de Software 3
2.2 Testes manuais 3
2.3 Testes Automatizados 4
2.3.1 Testes de unidade 5
2.3.2 Testes de Integração 7
2.3.3 Testes *End-to-End* 8
2.3.4 Testes de Aceitação 9

3 *Tests Smells* **10**
3.1 Desafios da automação de testes 10
3.2 O que são *Tests Smells*? 11
3.3 Tipos de *tests smells* 11
3.3.1 *Assertion Roulette* 12
3.3.2 *Conditional Test* 13
3.3.3 *Unknown Test* 14
3.3.4 *Magic Number* 14
3.3.5 *Duplicate Assert* 15
3.3.6 *Empty Test* 16
3.3.7 *Exception Handling* 16
3.3.8 *Default Test* 18
3.3.9 *Redundant Print* 18
3.3.10 *Sensitive Equality* 19
3.3.11 *Sleepy Test* 19
3.3.12 *Test Run War* 20
3.3.13 *Constructor Initialization* 20
3.4 Impactos negativos dos *tests smells* durante manutenções 21

4 Ferramentas de detecção de *Tests Smells* **22**
4.1 Ferramentas de identificação 22
4.1.1 TsDetect 23
4.1.2 Darts 24
4.1.3 JNose 24
4.2 Características comuns 25

4.3 Técnicas de detecção 25
4.3.1 Métricas 25
4.3.2 Regras/Heurísticas 26
4.3.3 *Information Retrieval* (Recuperação de informação) 26
4.3.4 *Dynamic Tainting* (Rastreamento dinâmico) 27
4.3.5 *Abstract Syntax Tree* (AST) 28

5 Proposta **30**
5.1 Tecnologias Utilizadas 30
5.1.1 Java 31
5.1.2 Reflections 31
5.1.3 JUnit 31
5.1.4 Maven 31
5.1.5 Log4j 32
5.1.6 IntelliJ IDEA 32
5.1.7 GitHub 32
5.1.8 srcML 32
5.2 Arquitetura da ferramenta 34

6 Avaliação **37**
6.1 Métricas 37
6.1.1 Matriz de confusão 37
6.1.2 *Precision* 38
6.1.3 *Recall* 38
6.1.4 *F-Score* 38
6.1.5 *Settings* 39
6.2 Resultados 39
6.2.1 Planejamento 39
6.2.2 Resultados 41
6.2.3 Discussão 43
6.2.4 Ameaças à validade 46
6.2.5 Comparação com o JNose 46

7 Conclusões **48**

Referências bibliográficas **49**

1

Introdução

O ato de realizar testes é tão importante quanto o de escrever o código do produto, pois são eles que garantem a qualidade e o funcionamento esperado do sistema. O desenvolvimento de *scripts* de testes para serem executados de forma automatizada estão cada vez mais presentes na indústria [17][32].

A automação dos testes nos permite evitar muitos problemas durante o ciclo de desenvolvimento, por exemplo, em eventuais modificações de alterar algum caso de uso do sistema, onde o próprio programador consegue descobrir antes de possivelmente inserir um novo *bug* em produção.

Pesquisadores definiram o termo *test smells* para indicar possíveis problemas de implementação que afetam negativamente a qualidade no código de teste como uma analogia aos sintomas transmitidos por código-fonte mal projetado (*code smells*) [60]. Tais sintomas podem levar os testes a apresentarem comportamento errático, por exemplo: i) falha de teste; ii) falsos positivos; e iii) falsos negativos; comprometendo, assim, a qualidade do software devido aos recursos limitados de detecção de tais defeitos [56].

Existem estudos que mostram que a detecção dos *tests smells* por parte dos desenvolvedores não é fácil e podem passar despercebidos [50]. Dessa forma, é necessário a utilização de ferramentas para executar essa tarefa.

Existem algumas ferramentas como, por exemplo, o TsDetect que foi implementado utilizando a linguagem Java e seu foco é a detecção de *tests smells* no *framework* de teste JUnit. O TsDetect possui suporte a detecção de 19 tipos de *tests smells* [49][51].

Há também a ferramenta Darts, um *plug-in* para o Intellij desenvolvido utilizando, também, a linguagem Java e identificação no *framework* de teste JUnit. Possui capacidade de identificação de 3 *tests smells* [35].

No entanto, durante o estudo, foi percebido que a grande maioria das ferramentas existentes são focadas em apenas uma linguagem ou *framework* de teste automatizado. E isso é ruim, pois exige todo um retrabalho para a criação de uma nova ferramenta para se obter esse suporte.

Desta forma, para minimizar esse problema, este trabalho propõe contribuir para se chegar a uma ferramenta capaz de identificar *tests smells* em diferentes *frameworks* de testes e em outras linguagens além de Java, como C e C++; tudo isso em uma só ferramenta, e isso é possível devido ao srcML [4]. Nos próximos capítulos, será demonstrado como a ferramenta foi arquitetada e como foi utilizado o srcML.

A avaliação da ferramenta foi feita utilizando projetos *open-sources* que estão hospedados no GitHub que usem Java e C++, totalizando 6 projetos, sendo 5 deles avaliando para o GoogleTest, com um total de 52 casos de *tests smells*, e 1 projeto para JUnit, com 50 casos. E foi obtido uma média F-score de 93.19%.

Por fim, está monografia está organizada da seguinte forma:

- Capítulo 2: é abordado a fundamentação teórica sobre testes no geral;
- Capítulo 3: apresenta de forma mais detalhada sobre o conceito de *tests smells* e com exemplificações, inclusive, identificadas pela ferramenta criada;
- Capítulo 4: é visto sobre os propósitos de ferramentas de detecção de *tests smells*, estratégias e características comuns;
- Capítulo 5: apresenta a solução implementada, tecnologias utilizadas e como a ferramenta foi arquitetada;
- Capítulo 6: são apresentados os resultados obtidos e análises de ameaças à validade; e
- Capítulo 7: apresenta a conclusão obtida acerca do uso da ferramenta proposta nesta monografia.

2

Fundamentação teórica

Neste capítulo será apresentado e detalhado de forma breve a respeito de Testes de *Software*, Testes Automatizados e alguns dos principais tipos de testes e suas importâncias.

2.1 Teste de Software

Pode-se dizer que teste de software é qualquer atividade destinada a avaliar um atributo ou capacidade de um programa, ou sistema, e determinar se ele atende aos resultados exigidos [30]. Também diz-se que é o processo de execução de um programa ou sistema com a intenção de encontrar erros [42]. O teste de software pode ser conduzido manualmente ou de maneira automatizada, e será detalhado nas próximas seções.

2.2 Testes manuais

No teste manual, um testador humano assume o papel de um usuário final, interagindo e executando o *Software Under Test (SUT)* para verificar seu comportamento e encontrar quaisquer defeitos observáveis [6].

Os grandes problemas de testes manuais é que são demorados, têm alto riscos de falhas, podem não ter uma cobertura muito alta e possuem um alto custo [28]. Por exemplo: imagine ter um produto relativamente grande em produção. Grande parte dos *bugs* são apresentados na introdução de novas funcionalidades, principalmente pelo fator humano; então, como iremos garantir que todas as outras funcionalidades já existentes vão continuar funcionando da mesma forma? Em testes manuais, repetiria todo o processo novamente. Destarte, podemos concluir que o tempo e esforço necessário para se fazer testes manuais são elevados. Ante isto, viu-se a necessidade de serem desenvolvidas ferramentas para automatizar tais processos.

Não existe uma fórmula a se seguir com relação à testes. Para algumas empresas, utilizar testes manuais talvez faça mais sentido do que testes automatizados. Então tudo depende do contexto em que se encontra a empresa.

Testes automatizados são mais caros que os manuais por conta dos custos de configuração (projeto e implementação). E só são utilizados quando há orçamento disponível no projeto, o que não acontece com frequência [28].

2.3 Testes Automatizados

Testes de software automatizados significa a automação de atividades de teste de software geralmente conduzidas por humanos. Em testes automatizados são desenvolvidos códigos de testes usando ferramentas de software conhecidas como ferramentas de teste [57] (ex: JUnit framework). A automação de testes tem uma história de mais de duas décadas e meia, desde cerca de 1990 [7]. A automação pode levar a muitos benefícios, por exemplo, economia de custos e maior qualidade de software [18].

Em um sistema, é muito comum passar por diversos novos desenvolvedores de diferentes senioridades, e existe uma certa curva de aprendizado do produto em que está trabalhando. Então, o objetivo central de ter testes automatizados em um sistema é a garantia de qualidade, redução e prevenção de erros dentro do sistema, redução de riscos e problemas para o futuro.

Os testes automatizados e desenvolvimentos de *scripts* de teste são predominantes na indústria de software [17][32]. Por exemplo, em um livro recente, os engenheiros de teste da Microsoft relataram que "havia mais de um milhão de casos de teste (automatizados) escritos para o Microsoft Office 2007" [47].

Os testes automatizados possuem muitas vantagens sobre os manuais, principalmente na redução considerável da repetitividade e do esforço exigido para se testar um sistema. Consequentemente, há redução de custos, uma vez que podem ser executados em poucos minutos, enquanto os testes manuais poderiam exigir várias horas. Mas também possuem seu ônus: tempo investido em estudos, planejamento, desenvolvimento e uma série de novos desafios. Embora testes automatizados sejam vistos com bons olhos, se implementados de forma incorreta, podem até elevar os custos [24].

Existem diferentes níveis de testes de softwares, e um conceito chave criado por Mike Cohn, em seu livro *Succeeding with Agile* [14], é a pirâmide de testes, que pode ser vista na Figura 2.1. Ela consiste em 3 camadas, e é possível ter uma visão geral das diferentes camadas de teste. Na pirâmide original, suas camadas de baixo para cima são:

- Testes de unidade;
- Testes de Serviço;
- Testes de interface de usuário.

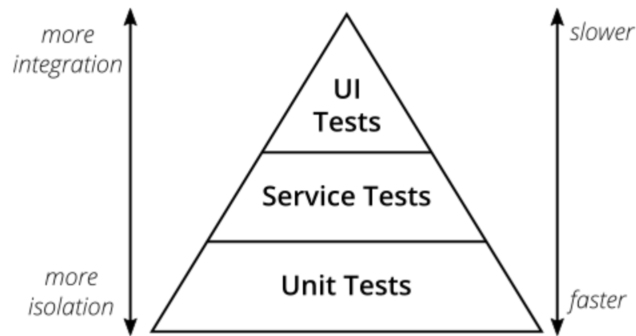


Figura 2.1: Pirâmide de testes [21].

Existem algumas variações da pirâmide de Cohn, por exemplo, a Figura 2.2. Segundo Martin Fowler, de um ponto de vista moderno, a pirâmide de testes de Cohn pode ser vista como excessivamente simplista, mas que devemos nos atentar à essência da pirâmide original em dois pontos [21]:

- Escrever testes com granularidades diferentes;
- Quanto mais alto nível obtiver, menos testes deverá ter.

Ou seja, seguir a forma da pirâmide para criar um conjunto de testes saudável, rápido e sustentável.

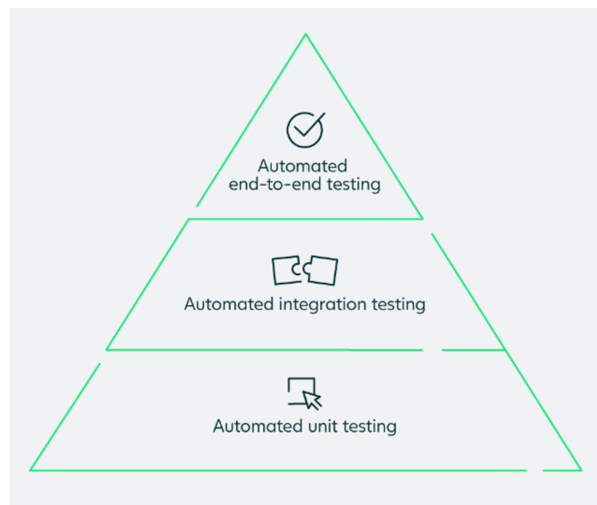


Figura 2.2: Pirâmide de testes automatizados [12].

2.3.1 Testes de unidade

Consertar falhas, desenvolver novas funcionalidades e refatorar partes do sistema faz parte do dia a dia dos desenvolvedores, e em uma eventual necessidade de manutenção, é possível introduzir uma nova falha ou interferir em uma funcionalidade já existente. Com isso, novos

desenvolvedores podem sentir uma certa dificuldade ou insegurança para realizar tarefas. Em uma situação onde possuem testes automatizados, o próprio desenvolvedor pode executar os testes já existentes e verificar se houve alguma situação de falha, e em caso positivo, já ajustar antes do código ir para produção.

Na pirâmide de Cohn, os testes de unidade estão na primeira camada — onde não tem dependências de outros componentes do sistema ou interface. O propósito principal de testes de unidade é isolar cada parte do sistema e provar individualmente que, dada uma certa entrada a uma função, receberá a saída esperada — i.e., devem ser executados totalmente independente de outros testes e quando há falha, deve ser fácil de se identificar o problema.

A definição dada por Roy Osherove [46] é que um teste de unidade é um pedaço de código automatizado que invoca uma unidade de trabalho que está sendo testada e verifica algumas suposições sobre um único resultado dessa unidade. Um teste de unidade é quase sempre escrito usando alguma ferramenta, sendo fácil de escrever e pode ser executado rapidamente — então é desejável, mas não uma regra, evitar acessos a banco de dados ou arquivos externos, por exemplo [46].

Ele é confiável, legível, de fácil manutenção e consistente em seus resultados, desde que o código de produção não seja alterado. Podem existir variações de definições de acordo com o paradigma da linguagem que está sendo usada. Por exemplo, em uma linguagem funcional, uma unidade pode ser uma única função ou método, mas já no paradigma orientado a objetos, uma unidade pode variar de um único método a uma classe inteira [21].

Outro objetivo importante de um teste de unidade, dada por Vladimir Khorikov [33], é a possibilidade do crescimento sustentável do software. Um software pode crescer muito rápido e fica cada vez mais difícil de sustentar esse crescimento ao longo do tempo.

Isso ocorre, pois no início de um projeto, não existem muitas dificuldades bloqueantes, por exemplo, uma má decisão arquitetural. E então, eventualmente, a velocidade de desenvolvimento diminui, e nos piores casos, pode chegar a não conseguir fazer nenhum progresso no projeto de forma geral.

Podemos ver na Figura 2.3 um gráfico exibindo a diferença entre um crescimento de um projeto com e sem testes. O esforço e tempo dedicado a iniciar um projeto já com testes é maior, mas é possível ver que com o tempo tende a ser mais viável na medida que a complexidade do projeto aumenta [33].

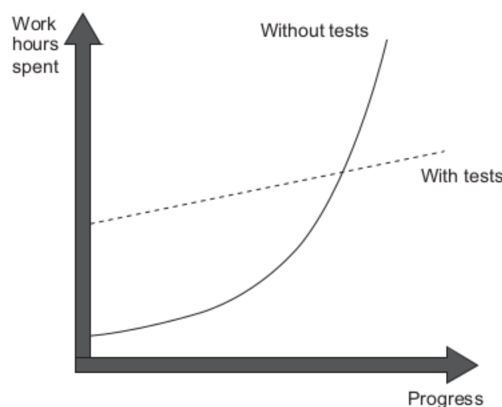


Figura 2.3: Diferença no crescimento de um software com e sem testes [33].

É possível escrever testes para todas as classes do código de produção, e uma boa estruturação para todos os testes, sugerido por Martin Fowler [21], são: i) configurar os dados de testes; ii) chamada do método; e iii) afirmar que os resultados esperados são retomados. E essa estruturação não se limita apenas para testes de unidade, sendo possível utilizar em outros tipos de testes.

Se um código em produção está difícil de criar testes de unidade, tem grandes chances de que seja necessária uma refatoração ou melhorias. Mas o fato de conseguir testar facilmente partes do código não significa que a qualidade está alta. A baixa qualidade de um código pode se manifestar quando está diretamente acoplado com outras partes do código de produção, consequentemente dificultando de testar partes do código separadamente. Como alerta Seguin, "Testes de unidade complexos e bagunçados não adicionam qualquer valor mesmo se o código tenha sido perfeitamente projetado" [55].

E assim como há más práticas no código em produção, da mesma forma há ao se desenvolver testes, o que será mais aprofundado no próximo capítulo.

2.3.2 Testes de Integração

Com os testes de unidade é possível validar se cada componente isoladamente está funcionando da forma correta. Mas mesmo com todos os testes de unidade validados, pode acontecer de, ao se conectar todos esses componentes, causar problemas, tal como serviços externos, banco de dados e até mesmo partes do próprio sistema. E para essa validação é utilizado os testes de integração. O principal objetivo é testar se os módulos desenvolvidos separadamente funcionam juntos como o esperado [22].

Observando a pirâmide de Cohn, é possível ver que os testes de integração estão em um nível acima dos testes de unidade. Portanto, possuem características de serem mais lentos na execução, maior acoplação com o sistema, maior complexidade e mais confiança em que a aplicação está funcionando apropriadamente com todas as partes externas e internas do sistema. Na Figura 2.4 podemos ver, de uma forma análoga em peças de um quebra-cabeça, como funciona

um teste de integração, onde a peça sozinha está representando os testes de unidade e a junção delas, os testes de integração.

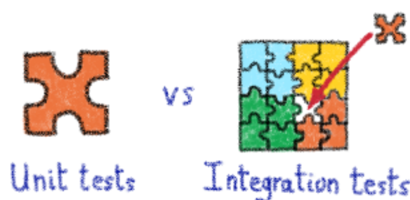


Figura 2.4: Analogia de testes de integração com peças de quebra-cabeça [54].

Em um sistema de compras online, por exemplo, geralmente possui o módulo de catálogo onde a ideia principal é olhar os produtos disponíveis para efetuar uma possível compra. E para se ter uma abordagem mais próxima da realidade, foi criado o carrinho de compras, no qual os clientes podem adicionar vários produtos que desejam comprar para decidirem, ao final, se irão ou não levar os produtos em questão.

Pensando em testes de unidade, serão testados os módulos separadamente, mas sem nenhuma garantia que os dois juntos funcionarão conforme o esperado. É neste ponto, então, que entram os testes de integração, gerando confiança de que funcionam juntos. Esse foi um exemplo de testes de integração de diferentes módulos de um sistema, mas pode haver casos de utilização de serviços desenvolvidos por outras pessoas. Aproveitando o exemplo do sistema de compras, a utilização de um serviço de cálculo de frete.

Figura 2.5: Exemplo de integração com serviços externos [21].

2.3.3 Testes *End-to-End*

A complexidade de uma aplicação pode crescer de acordo com vários fatores, dentre eles integrações e dependências com diversos outros sistemas, internos ou externos. Ciente disso, no contexto de testes, o teste *End-to-End* (de ponta a ponta) determina se várias dependências de um aplicativo estão funcionando e se os componentes entre os sistemas estão se comunicando com precisão. Eles simulam um fluxo de cenário real de um usuário final utilizando a aplicação para garantir que todos os componentes integrados funcionem conforme as expectativas.

É importante salientar alguns benefícios, como: aumento das detecções de *bugs*; aumento da confiabilidade do sistema, expandindo a cobertura de testes; e redução de riscos de falhas,

custos e tempo. Contudo, existem alguns entraves ao se desenvolver testes *End-to-end*. Um dos principais é a instabilidade e inconsistência na execução do teste. Fatores como rede, carregamento de dados ou falhas em outros sistemas, podem afetar os resultados, onde em determinado momento a execução pode falhar ou finalizar com sucesso.

Devido à complexidade de replicação dos cenários de falhas, a depuração e descoberta da causa raiz fica prejudicada, e por consequência, pode levar à improdutividade e até mesmo ao desenvolvedor ignorar o teste, sendo assim diminuindo o valor do mesmo [38]. Um simples exemplo de teste *End-to-end* é:

- Autenticar em sistema de vendas de produtos;
- Escolher um produto;
- Adicionar quantidade desejada;
- Adicionar o produto ao carrinho;
- Informar endereço de entrega;
- Informar método de pagamento;
- Finalizar compra;
- Sair do aplicativo.

2.3.4 Testes de Aceitação

A principal função dos testes de aceitação é a avaliação do sistema do ponto de vista do cliente. Com a verificação realizada pelo cliente, fica evidenciado se o sistema está aceitável para liberação, gerando, assim, a confiança que os componentes estão funcionando corretamente. Com eles é possível: entender se o sistema está fazendo o que o usuário solicitou; expor problemas que testes de unidade podem não mostrar; e ainda fornecem uma definição de como o sistema é [41].

Supondo uma interface que lista produtos em um sistema de produtos, um teste de aceitação pode verificar se a exclusão de um item é refletido corretamente na lista dos produtos [41]. Também é possível verificar se a experiência do usuário com as interfaces estão agradáveis, intuitivas e com fáceis interações.

Assim como os demais testes, o de aceitação também possui seus reveses, tais como a exigência de que o usuário tenha um certo entendimento acerca do produto — e sem uma boa documentação das funcionalidades do sistema, pode atrasar a liberação do mesmo.

3

Tests Smells

No capítulo anterior, foi introduzido sobre testes de software, testes manuais, testes automatizados e seus níveis. Neste capítulo será falado sobre os desafios da automação de testes e detalhado sobre *Tests Smells* e seus tipos com ênfase em testes de unidade.

3.1 Desafios da automação de testes

Apesar dos benefícios citados na Seção 2.3, é preciso relatar sobre alguns problemas comuns que podem ocorrer em testes automatizados, tais como: a garantia de controle de qualidade, iniciar a automação sem planejamentos — o que pode causar a perda de acurácia dos testes —, o custo inicial e até mesmo a colaboração entre times. Para se evitar tais ocorrências, é necessário um bom investimento em treinamentos e um planejamento inicial de quais funcionalidades serão automatizadas e as estratégias necessárias para se ter boas práticas de testes, pois más práticas poderá levar a testes mal organizados, complexos e caóticos que podem não agregar nenhum tipo de valor mesmo que o código em teste seja perfeitamente projetado [55]. Iremos aprofundar mais sobre más práticas em testes de software adiante. Há diversas ferramentas no mercado disponíveis, sejam elas licenciadas ou *open-source*.

A escolha da ferramenta de testes é muito importante, sendo preciso dedicar um bom tempo e fazer análises no que funciona ou não para o seu negócio, os requerimentos e objetivos que se pretende alcançar antes de decidir qual será a ferramenta ideal. Uma má escolha de ferramenta pode resultar em diversos problemas e a automação tenderá a falhar. Um exemplo comum de acontecer é causar uma falsa segurança. Não é porque os testes que foram desenvolvidos estão corretos que não há *bugs* (testes não garantem ausência de *bugs*) ou nenhum tipo de problema. Esses são alguns dos desafios da automação de softwares.

No entanto, se não for implementado adequadamente, o teste automatizado levará a custos e esforços extras e pode até ser menos eficaz do que o teste manual na detecção de falhas [6].

3.2 O que são *Tests Smells*?

Testes de unidade, assim como códigos em produção, estão sujeitos a más práticas de programação, também conhecidos como anti-padrões, defeitos ou "*smells*"[50] — que no contexto de *software* quer dizer que algo "não está cheirando bem". Não necessariamente é um *bug*, já que não impedem a execução dos testes, mas podem afetar negativamente a qualidade dos testes [24].

O conceito de *tests smells* foi introduzido por Van Deursen et al [60], e existe uma vasta definição de *tests smells* na literatura, das quais destacamos: i) de um modo geral, *tests smells* são descritos como um conjunto de problemas no código de teste [29]; ii) são testes mal projetados e sua presença pode afetar negativamente um conjunto de testes e código de produção (aspectos como manutenibilidade ou mesmo sua funcionalidade) [60] [9]; iii) são violações de padrões de quatro fases de teste (*setUp*, exercício, verificação e *tearDown*), resultando numa redução de um ou mais critérios de qualidade do teste [61]; e iv) refere-se a qualquer sintoma no código de teste que indica um problema mais profundo [58].

*J*á os chamados Bugs (falhas) em códigos de teste, são processos que alteram o comportamento esperado do teste [59].

Conforme declarado por Robert Martin, "*Code smells* geralmente não são *bugs*, pois tecnicamente não estão incorretos e não impedem o funcionamento do programa. Ao invés disso, indicam fraquezas de projeção que podem retardar o desenvolvimento, aumentar o risco de *bugs* e falhas no futuro" [37]. Essa mesma definição pode ser aplicada para códigos de testes.

Em consideração às definições acima, podemos ver que os *tests smells* podem ser uma verdadeira "pedra no sapato", causando problemas como baixa manutenibilidade, desempenho, legibilidade, e de mais gastos. E como é o caso para empresas grandes no mercado? Por exemplo, uma postagem no blog oficial de testes do Google é dito que quase 16% dos testes "têm algum nível de instabilidade associados a eles! Este é um número impressionante, significa que mais de 1 em 7 dos testes escritos por nossos engenheiros de classe mundial falham ocasionalmente de uma forma não causada por alterações no código ou nos testes" [40].

Um estudo de 2009 relatou que o custo anual de manutenção manual e evolução de *scripts* de teste foi estimado entre US$ 50 e US$ 120 milhões para uma empresa chamada Accenture. E as ocorrências de *tests smells* tem uma grande contribuição nesses custos [26].

3.3 Tipos de *tests smells*

Nessa seção será apresentado um conjunto dos principais *tests smells* que existem com exemplos em Java e C++. É importante salientar que podem existir *tests smells* específicos para cada ferramenta de teste, por exemplo o JUnit para Java e o GoogleTest para C++.

Além disso, existem catálogos com vários *smells* identificados ao longo do tempo e possíveis formas de correção, na maioria dos casos, com refatorações [60].

3.3.1 *Assertion Roulette*

O *Assertion Roulette* é definido por Van Deursen et al. [60] como uma coleção de asserções inexplicadas em um único método ou função de teste que, no caso de falha de um dos testes, desafia o rastreamento de qual asserção exata que apresentou um problema [60]. Ou seja, como há mais de um teste sem descrição, quando falha, não da para saber por qual razão foi, dessa forma impactando na legibilidade e na manutenibilidade do teste.

Listing 3.1: *Assertion Roulette* [44]

```
1  @MediumTest
2  public void testCloneNonBareRepoFromLocalTestServer() throws Exception {
3     Clone cloneOp = new Clone(false, integrationGitServerURIFor("small-repo.early
          .git"), helper().newFolder());
4     Repository repo = executeAndWaitFor(cloneOp);
5     assertThat(repo, hasGitObject("ba1f63e4430bff267d112b1e8afc1d6294db0ccc"));
6     File readmeFile = new File(repo.getWorkTree(), "README");
7     assertThat(readmeFile, exists());
8     assertThat(readmeFile, ofLength(12));
9  }
```

Listing 3.2: *Assertion Roulette* em C++ [36]

```
10 TEST(activeClass, runNoThread_1)
11 {
12 activeClass::prologueFun<int> pfun = [](int arg)
13   {
14     arg++;
15     return false;
16   };
17
18   activeClass::bodyFun<int,int> bodyfun = [](int arg)
19   {
20     arg++;
21     return arg;
22   };
23
24   activeClass::epilogueFun<int> efun = [](int arg)
25   {
26     arg++;
27     return false;
28   };
29
30   // create the active class object with the functions
31   activeClass::activeClassPtr<int,int> acptr = activeClass::makeActiveClass<int,
        int>(pfun, bodyfun, efun);
32   std::string v = acptr.get()->activeClassVersion();
33
34   // run all the provided functions without generating a new thread; run synch
35   // in this thread
36   auto [prologueResult, bodyResult, epilogueResult, thrData] = acptr.get()->
        activeClass<int,int>::run(5);
37
38   std::thread::id tid = std::this_thread::get_id();
39   ASSERT_EQ(tid, acptr.get()->getThreadId());
40   ASSERT_EQ(6, bodyResult);
41   ASSERT_EQ(5, acptr.get()->getThreadData());
42   ASSERT_EQ(5, thrData);
43 }
```

Em um framework como JUnit, é possível fazer o refatoramento do código, onde nas funções de *assert*, possui um argumento opcional para descrever o erro em caso da asserção falhar [60].

3.3.2 *Conditional Test*

Funções de testes devem ser simples e devem executar todas as instruções que estão escritas. Condições dentro do método de teste vão alterar seu comportamento e a saída esperada, levando, dessa forma, há situações onde o teste falha em detectar defeitos na função de produção, já que as instruções do teste podem ou não serem executadas devido a uma condição. Além disso, pode afetar negativamente a facilidade de compreensão do teste pelos desenvolvedores [50]. Qualquer tipo de instrução de controle dentro de um teste, como *loops*, são considerados como *conditional test smell*.

Listing 3.3: *Conditional Test* [44]

```
1  @Test
2  public void testSpinner() {
3      for (Map.Entry entry : sourcesMap.entrySet()) {
4          String id = entry.getKey();
5          Object resultObject = resultsMap.get(id);
6          if (resultObject instanceof EventsModel) {
7              EventsModel result = (EventsModel) resultObject;
8              if (result.testSpinner.runTest) {
9                  System.out.println("Testing " + id + " (testSpinner)");
10                 //System.out.println(result);
11                 AnswerObject answer = new AnswerObject(entry.getValue(), "", new
                       CookieManager(), "");
12                 EventsScraper scraper = new EventsScraper(RuntimeEnvironment.
                       application, answer);
13                 SpinnerAdapter spinnerAdapter = scraper.spinnerAdapter();
14                 assertEquals(spinnerAdapter.getCount(), result.testSpinner.data.
                       size());
15                 for (int i = 0; i < spinnerAdapter.getCount(); i++) {
16                     assertEquals(spinnerAdapter.getItem(i), result.testSpinner.data
                           .get(i));
17                 }
18             }
19         }
20     }
21 }
```

Listing 3.4: *Conditional Test* em C++ [63]

```
1  TEST(ConfigurationTest, Lookup) {
2      Config cfg {"test"};
3
4      // The variable `x` does not exist in the current configuration.
5      // `LoadYaml` will not create it.
6      const auto node {LoadYamlString("{x: 1}")};
7      cfg.LoadYaml(node);
8      EXPECT_FALSE(cfg.Lookup<int>("x"));
9
10     // Create a variable `x`.
11     cfg.Lookup<int>("x", 0);
```

```
12    const auto x {cfg.Lookup<int>("x")};
13    EXPECT_TRUE(x);
14    if (x) {
15        EXPECT_EQ(x->Value(), 0);
16    }
17
18    // Set `x` from a `YAML` node.
19    cfg.LoadYaml(node);
20    EXPECT_TRUE(x);
21    if (x) {
22        EXPECT_EQ(x->Value(), 1);
23    }
24
25    // The variable type is mismatched.
26    EXPECT_THROW(cfg.Lookup<wchar_t>("x"), std::invalid_argument);
27 }
```

Uma possível refatoração é criar métodos diferentes para cada instrução de controle dentro do teste.

3.3.3 *Unknown Test*

Uma função de asserção descreve uma condição esperada para um método de teste. Analisando a asserção, é possível entender o propósito do teste. No entanto, pode acontecer de um método de teste ser escrito sem nenhuma função de asserção, e em frameworks como o JUnit, o teste resultará em aprovação — caso uma exceção não seja gerada durante a execução [50]. Dessa forma, se a função também não tiver um nome descritivo, dificultará para o desenvolvedor entender o propósito desse teste.

Listing 3.5: *Unknown Test* [44]

```
1 @Test
2 public void hitGetPOICategoriesApi() throws Exception {
3     POICategories poiCategories = apiClient.getPOICategories(16);
4     for (POICategory category : poiCategories) {
5       System.out.println(category.name() + ": " + category);
6     }
7 }
```

Listing 3.6: *Unknown Test* em C++ [48]

```
1 TEST(Std, NoOutput)
2 {
3     std::cout << "cout\n";
4     std::cerr << "cerr\n";
5 }
```

3.3.4 *Magic Number*

Esse *smell* ocorre quando um método de teste contém literais numéricos inexplicados e não documentados como parâmetros. Esses valores não indicam suficientemente o propósito ou

significado do número, dificultando, dessa forma, a compreensão do código [50].

Listing 3.7: *Magic Number* [44]

```
@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));
}
```

Listing 3.8: *Magic Number* em C++ [34]

```
TEST(AngleStructs, DegreesToRadiansAndBack)
{
   Degrees d1(45.0f);

   Radians r(d1);

   Degrees d2(r);

   EXPECT_FLOAT_EQ(d2, 45.0f);
}
```

A refatoração desse *smell*, é a substituição por constantes ou variáveis, fornecendo um nome descritivo para o valor.

3.3.5 *Duplicate Assert*

O *duplicate assert smell* acontece quando um método de teste testa a mesma condição várias vezes no mesmo método. E as possíveis explicações para a ocorrência são: i) várias condições para testar um único método; ii) quando desenvolvedores estão realizando depuração no código; e iii) copiar e colar partes do código acidentalmente [50].

Listing 3.9: *Duplicate Assert* [44]

```
@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("Frutzbux");
   assertEquals("Frutzbux 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);
```

```
21      System.out.println("Dot test - passed");
22
23      valid = XmlSanitizer.isValid("Fritz-box");
24      assertEquals("Minus is valid", true, valid);
25      System.out.println("Minus test - passed");
26
27      valid = XmlSanitizer.isValid("Fritz-box");
28      assertEquals("Minus is valid", true, valid);
29      System.out.println("Minus test - passed");
30  }
```

Listing 3.10: *Duplicate Assert* em C++ [43]

```
1  TEST_F(VariableTests, GlobalVarTest) {
2      while (global_var_.update());
3      ASSERT_EQ("7", getline(*output_));
4      ASSERT_EQ("7", getline(*output_));
5      output_->clear();
6  }
```

A refatoração recomendada para esse *smell* é que, em casos de um teste realmente precisar testar a mesma condição, mas usando valores diferentes, deve-se criar um novo método para ele.

3.3.6 *Empty Test*

Ocorre quando um método de teste não possui nenhum tipo de instrução. São métodos que possivelmente foram criados para fins de depuração sem serem excluídos ou contém instruções comentadas.

Listing 3.11: *Empty Test* [44]

```
1  public void testCredGetFullSampleV1() throws Throwable{
2  //       ScrapedCredentials credentials = innerCredTest(FULL_SAMPLE_v1);
3  //       assertEquals("p4ssw0rd", credentials.pass);
4  //       assertEquals("user@example.com",credentials.user);
5  }
```

Métodos de testes vazios pode ser considerados problemáticos e mais perigosos do que não ter um caso de teste, pois ferramentas como JUnit indicará que o teste foi aprovado mesmo se não houver instruções presentes no corpo do método.

Dessa forma, se os desenvolvedores introduzirem alterações no código de produção de maneira que apresente um *bug*, não serão notificados sobre os resultados, pois a ferramenta de teste relatará o teste como aprovado [50].

3.3.7 *Exception Handling*

Esse *smell* ocorre quando a aprovação ou reprovação de um método de teste depende explicitamente do método de produção gerar uma exceção e o desenvolvedor manipula manualmente o resultado do teste [50].

Listing 3.12: *Exception Handling* [44]

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

Listing 3.13: *Exception Handling* em C++ [43]

```
TEST_F(GridTest, NullTest) {
    try {
      Grid test("nonexistent_text_file.txt");
      FAIL() << "File read error not thrown";
    } catch (Grid::GridFileNotFoundException& ex) {
      ASSERT_STREQ("File nonexistent_text_file.txt not found", ex.what());
    } catch (...) {
      FAIL() << "Random exception occured while reading file";
    }
}
```

A refatoração sugerida por Peruma et al. é de, ao invés de escrever um código personalizado ou lançar a exceção manualmente, utilizar os recursos do framework de teste para fazer o tratamento da exceção [50].

3.3.8 *Default Test*

Esse *smell* pode não ocorrer em todas as ferramentas utilizadas para desenvolver testes automatizados. Algumas ferramentas, como o Android Studio, geram classes de testes padrões quando um projeto é criado. Essas classes de teste servem como exemplos para desenvolvedores escreverem testes de unidade e devem ser removidas ou renomeadas [50].

Listing 3.14: *Default Test* [44]

```
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");
      .....
   }
   .....
}
```

Esses arquivos no projeto podem fazer com que os desenvolvedores comecem a adicionar métodos de teste, tornando a classe *default* um contêiner de casos de testes. Tal ação viola as boas práticas de teste. Também podem surgir problemas quando as classes precisassem ser renomeadas no futuro [50].

3.3.9 *Redundant Print*

Funções de exibir mensagens específicas na tela — conhecidas como funções da saída — são redundantes em testes de unidade, pois os testes de unidade são executados como parte de um script automatizado. Além disso, podem consumir recursos de computação ou aumentar o tempo de execução se o desenvolvedor chamar um método intensivo ou longo como parâmetro da função de saída [50].

Listing 3.15: *Redundant Print* [44]

```
@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);
}
```

3.3.10 *Sensitive Equality*

Igualdade sensível ocorre quando uma asserção contém uma verificação de igualdade por meio do uso do método toString(). Os métodos de teste verificam os objetos invocando o método padrão toString() de sua instância e comparando a saída com uma cadeia de caracteres específica. Alterações na implementação de toString() podem resultar em falha, pois podem depender de detalhes usados na comparação, como vírgulas, espaços e etc [60].

Listing 3.16: *Sensitive Equality* [44]

```
1  @Test
2  public void test1() throws UnknownHostException {
3      String peersPacket = "F8 4E 11 F8 4B C5 36 81 " +
4          "CC 0A 29 82 76 5F B8 40 D8 D6 0C 25 80 FA 79 5C " +
5          "FC 03 13 EF DE BA 86 9D 21 94 E7 9E 7C B2 B5 22 " +
6          "F7 82 FF A0 39 2C BB AB 8D 1B AC 30 12 08 B1 37 " +
7          "E0 DE 49 98 33 4F 3B CF 73 FA 11 7E F2 13 F8 74 " +
8          "17 08 9F EA F8 4C 21 B0";
9      byte[] payload = Hex.decode(peersPacket);
10     byte[] ip = decodeIP4Bytes(payload, 5);
11     assertEquals(InetAddress.getByAddress(ip).toString(), ("/54.204.10.41"));
12 }
```

A refatoração sugerida é a implementação de um método personalizado dentro do objeto para fazer essa comparação [60].

3.3.11 *Sleepy Test*

Esse *smell* pode ser introduzido por desenvolvedores quando precisam pausar a execução de instruções de teste por um determinado período — por exemplo, para simulação de um evento externo — e depois continuar a execução.

Listing 3.17: *Sleepy Test* [44]

```
1  public void testEdictExternSearch() throws Exception {
2      final Intent i = new Intent(getInstrumentation().getContext(), ResultActivity
           .class);
3      i.setAction(ResultActivity.EDICT_ACTION_INTERCEPT);
4      i.putExtra(ResultActivity.EDICT_INTENTKEY_KANJIS, "");
5     tester.startActivity(i);
6     assertTrue(tester.getText(R.id.textSelectedDictionary).contains("Default"));
7     final ListView lv = getActivity().getListView();
8     assertEquals(1, lv.getCount());
9     DictEntry entry = (DictEntry) lv.getItemAtPosition(0);
10    assertEquals("Searching", entry.english);
11    Thread.sleep(500);
12    final Intent i2 = getStartedActivityIntent();
13    final List result = (List) i2.getSerializableExtra(ResultActivity.
          INTENTKEY_RESULT_LIST);
14    entry = result.get(0);
15    assertEquals("(adj-na,n,adj-no) blank space/vacuum/space/null (NUL)/(P)",
          entry.english);
16    assertEquals("    ", entry.getJapanese());
17    assertEquals("          ", entry.reading);
18    assertEquals(1, result.size());
19 }
```

Causar explicitamente encadeamentos utilizando métodos de *sleep*, pode levar a resultados inesperados. Devido a vários tipos de ambientes e configurações, o tempo de processamento de uma tarefa pode diferir nas execuções [50].

3.3.12 *Test Run War*

Essas "guerras"ocorrem quando os testes funcionam bem, desde que somente uma pessoa esteja testando, mas falham quando outros programadores os executam. Provavelmente acontece por interferências de recursos, por exemplo, a alocação de arquivos temporários que podem ser usados por outros, ou recursos do sistema [60].

Listing 3.18: *Test Run War*

```
1  @Test
2  public void testEmptyList() {
3      System.setProperty("test.path.property", "");
4      List<Path> result = env.getPaths("test.path.property").collect(Collectors.
          toList());
5      MatcherAssert.assertThat(result, Matchers.hasSize(0));
6  }
```

O exemplo acima foi retirado do projeto Cryptomator[1]. A refatoração sugerida por Van Deursen et al. é tornar o recurso único, que consiste em criar identificadores únicos para todos os recursos alocados por um caso de teste [60].

3.3.13 *Constructor Initialization*

Idealmente, classes de testes não devem ter construtores. Inicialização dos objetos devem estar no método *setUp* [50]. O método *setUp* é executado uma vez antes de cada método de teste definido dentro da classe, com propósito de preparar tudo que será necessário utilizar nos testes.

Existe também o método *tearDown*, responsável por fazer a limpeza final, por exemplo, apagar os dados salvos no banco de dados. Também é executado apenas uma vez e após a execução de cada método de teste. Desenvolvedores que desconhecem a finalidade desses métodos, ativariam esse *smell* definindo construtores para a classe [50].

Listing 3.19: *Constructor Initialization* [44]

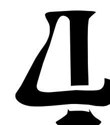
```
1  public class TagEncodingTest extends BrambleTestCase {
2     private final CryptoComponent crypto;
3     private final SecretKey tagKey;
4     private final long streamNumber = 1234567890;
5
6     public TagEncodingTest() {
7        crypto = new CryptoComponentImpl(new TestSecureRandomProvider());
```

[1]https://git.io/JLWby

```
8       tagKey = TestUtils.getSecretKey();
9     }
10
11    @Test
12    public void testKeyAffectsTag() throws Exception {
13       Set set = new HashSet<>();
14       for (int i = 0; i < 100; i++) {
15          byte[] tag = new byte[TAG_LENGTH];
16          SecretKey tagKey = TestUtils.getSecretKey();
17          crypto.encodeTag(tag, tagKey, PROTOCOL_VERSION, streamNumber);
18          assertTrue(set.add(new Bytes(tag)));
19       }
20    }
21  ...
22 }
```

3.4 Impactos negativos dos *tests smells* durante manutenções

Bavota et. al. [9] conduziu um estudo em seu artigo *"An Empirical Analysis of the Distribution of Unit Test Smellsand Their Impact on Software Maintenance"* e provou os impactos negativos dos *smells* em códigos de teste na manutenibilidade e legibilidade. Nesse estudo, o experimento foi realizado em 20 mestrandos, selecionando classes que continham pelo menos um *smell* e utilizando dois sistemas do mercado. E para obter uma versão de cada classe sem *smells*, os códigos foram manualmente refatorados seguindo as orientações fornecidas pelo Van Deursen et al [60].



Ferramentas de detecção de *Tests Smells*

No capítulo anterior, foi visto o que são os *tests smells* e que eles podem apresentar problemas, assim como o código de produção, afetando negativamente o produto. Neste capítulo, será abordado o tema das ferramentas de identificação de smells, o propósito da utilização, as estratégias e características comuns de detecção e implementação dessas ferramentas que motiva a proposta do presente trabalho.

4.1 Ferramentas de identificação

Ferramentas de identificação de *smells* é de suma importância, visto que más práticas podem passar despercebidos pelos desenvolvedores [50] — existe um estudo especificamente sobre ferramentas de detecção de *tests smells*.

Grande parte das ferramentas possui suporte nas linguagens de programação Java, Scala, C++ e Smaltalk. Dentre elas, a mais popular para suporte em *tests smells* é Java, suportando 39 *smells* no total; em segundo lugar está Smalltalk, com 28 tipos de *smells*. Na Tabela 4.1, será visto as ferramentas mais populares que existem, com informações como: linguagem de implementação, linguagens que a ferramenta analisa, frameworks de testes suportados e técnicas de detecção. Algumas das ferramentas citadas chegam a analisar duas ou mais estruturas de teste, como a TestQ, TestHound e TestEvoHound [5]. Em sequência, será detalhado sobre algumas das ferramentas.

Tabela 4.1: Características de ferramentas de detecção de *tests smells*[5].

Ferramenta	Implementação	Linguagem analisada	*Test Framework* suportado	Técnica de detecção
DARTS	Java	Java	JUnit	Informational Retrieval
DrTest	Smalltalk	Pharo	SUnit	Rule, Dynamic Tainting
DTDetector	Java	Java	JUnit	Dynamic Tainting
EletricTest	Java	Java	JUnit	Dynamic Tainting
JNose Test	Java	Java	JUnit	Rule
OraclePolish	Java	Java	JUnit	Dynamic Tainting
PolDet	Java	Java	JUnit	Dynamic Tainting
PraDet	Java	Java	JUnit	Dynamic Tainting
RAIDE	Java	Java	JUnit	Rule
RTj	Java	Java	JUnit	Rule, Dynamic Tainting
SoCRATES	Scala	Scala	ScalaTest	Rule
Taste	Desconhecido	Java	JUnit	Information Retrievel
TeReDetect	Java	Java	JUnit	Metrics, Dynamic Tainting
TestEvoHound	Java	Java	JUnit, TestNG	Metrics
TestHound	Java	Java	JUnit, TestNG	Metrics
TestLint	Smalltalk	Smalltalk	SUnit	Rule, Dynamic Tainting
TestQ	Python	C++, Java	CppUnit, JUnit, Qtest	Metrics
TRex	Java	Java	TTCN-3	Rule
TsDetect	Java	Java	JUnit	Rule
Unnamed	Desconhecido	Java	JUnit	Rule

4.1.1 TsDetect

O TsDetect é uma ferramenta *open-source*, com suporte para 19 dos *test smells* mais comuns. A ferramenta usa como entrada o código-fonte do projeto de software. Primeiro separa o conjunto de arquivos de testes de unidade dos arquivos-fonte de produção e em seguida gera suas Árvores de Sintaxe Abstrata (ASTs) para procurar quaisquer padrões predefinidos de *tests smells* sintaticamente, usando regras de detecção. Após, é gerado como saída um arquivo contendo todas as violações detectadas [49][51].

O TsDetect foi projetado para ser fácil de estender, ou seja, os desenvolvedores podem

facilmente calibrar as regras predefinidas e adicionar suas próprias regras personalizadas, se necessário. Além disso, embora detecte atualmente 19 *tests smells*, ele é projetado com um alto nível de flexibilidade para incorporar facilmente novos tipos de *smells* [49][51].

Para a avaliação do TsDetect, foi realizado uma análise qualitativa em um conjunto de 65 arquivos de testes de unidade que contêm instâncias de vários tipos de *smells*. A ferramenta em análise é capaz de detectar corretamente *tests smells* com uma precisão variando de 85% a 100%, *recall score* de 90% a 100%, com média *F-score* de 96,5% [49][51].

4.1.2 Darts

Darts (*Detection And Refactoring of Test Smells*) é um *plug-in* para o Intellij. O Intellij é uma IDE (Integrated development environment), ou Ambiente de Desenvolvimento Integrado, para desenvolvimento de código na linguagem Java ou Kotlin. O Darts foi implementado utilizando Java 8 e funciona ao nível de *commit* — *commit* é um *snapshot* do código em um determinado tempo para controle de versão. A implementação desse *plug-in* seguiu uma filosofia *just-in-time*, permitindo aos desenvolvedores descobrirem e refatorarem os códigos de testes assim que um novo *commit* é executado [35].

A ferramenta é *open-source* e está hospedado no Github, sendo capaz de identificar três *tests smells*, sendo eles: I) *Eager Test*; II) *General Fixture* e III) *Lack of Cohesion of Test Methods* [35].

4.1.3 JNose

JNose é uma ferramenta *web open-source* desenvolvida para detectar *tests smells*, sendo uma extensão do TsDetect e com suporte a detecção de 21 *tests smells*. A principal proposta do JNose é a detectação de cobertura de código e métricas de evolução de software, e um conjunto de *tests smells* em todas as versões de software [62]. É focada em: i) identificar possíveis falhas de projeto de teste; ii) analisar a evolução da qualidade do projeto de software; e iii) reduzir o esforço para realizar a garantia de qualidade de um conjunto de testes [62].

O JNose implementa uma estrutura para detectar os *tests smells* e contém classes para suportar uma análise estática do código de teste por uma AST gerada pelo JavaParser.[1] Foram definidos níveis de granularidades para detectar a localização dos *smells*, sendo eles: i) linha, *tests smells* que ocorrem em uma linha específica; ii) bloco, *tests smells* que ocorrem em um nível de bloco de instrução, por exemplo, try/catch e declarações condicionais; iii) método, *tests smells* que ocorrem no nível de método e iv) classe, *tests smells* que ocorrem em um nível de classe de teste. Além de possuir a escolha de análise por classe de teste, *test smell*, arquivo de teste, e evolução que faz a análise pelas versões [62].

[1] Online. Disponível em `https://javaparser.org/`

4.2 Características comuns

Do ponto de vista da detecção, a maioria das ferramentas depende da análise estática do código-fonte — análise que não depende da execução do código —, mas outras ferramentas funcionam por meio da análise dinâmica — análise que depende da execução do código. Essas estratégias de detecção podem ser agrupadas em 4 categorias: i) Métricas; ii) Regras/Heurísticas; iii) Recuperação de informação; e iv) Rastreamento dinâmico.

Pela Tabela 4.1, é possível notar que a maioria das ferramentas baseadas em análise estática para identificar os *smells*, optam por utilizar estratégias de detecção baseada em Regras. Grande parte dessas ferramentas analisa código-fonte Java, sendo mais específico, o *framework* JUnit. Apesar de muitas linguagens hoje em dia estarem aumentando sua popularidade, como C#, JavaScript e Python, não existem muitas ferramentas fornecendo esse suporte.

4.3 Técnicas de detecção

4.3.1 Métricas

O uso de métricas para detecção de *tests smells* é uma das técnicas mais comuns. Van Rompaey et al. [61] propôs uma técnica baseada em métrica que calcula várias métricas estruturais, como a quantidade de chamadas de código de produção feitas por um caso de teste, e combina em regras de detecção para destacar a probabilidade de o teste conter algum *smell* [52]. Então, os *smells* são medidos por meio de seu impacto nas medições estruturais e semânticas e é considerado *smell* caso ultrapasse o valor predefinido como limite. As métricas e valores de limite podem ser combinadas em uma árvore baseada em regras para se tomar decisões, e normalmente nessa técnica de detecção é utilizado a AST para converter o código-fonte a ser analisado [5].

Um exemplo simples de uma métrica é para o *Empty Test*, em que ocorre caso possui códigos comentados ou o método de teste estar vazio. A ferramenta TestQ [11] que utiliza técnicas de métricas para detecção de testes, define um sinal (*flag*) como 0 e se atinge essa métrica o *smell* é detectado.

Listing 4.1: Exemplo *Assertion Roulette e Duplicate tests smells* detectados pelo TestQ [11]

```
1  void PathTest::testParseVMS1() {
2      Path p;
3      p.parse(     ,Path::PATHVMS);
4      assert(p.isRelative());
5      assert(!p.isAbsolute());
6      assert(p.depth()==0);
7      assert(p.isDirectory());
8      assert(!p.isFile());
9      assert(p.toString(Path::PATHVMS)==  );
10
11     p.parse(  []   ,Path::PATHVMS);
12     assert(p.isRelative());
```

```
13      assert(!p.isAbsolute());
14      assert(p.depth()==0);
15      assert(p.isDirectory());
16      assert(!p.isFile());
17      assert(p.toString(Path::PATHVMS) ==  );
18      ...
19  }
```

4.3.2 Regras/Heurísticas

É como um complemento para a técnica de métricas aumentando com padrões que podem ser encontrados no código-fonte. Então, o *smell* é detectado quando a entrada corresponde a um conjunto predefinido de limites com a existência de alguns padrões. Por exemplo, o *smell Assertion Roulette*, é detectado pela definição de uma heurística que verifica se um método de teste contém várias declarações de asserção sem uma mensagem de explicação como parâmetro para cada método de asserção [5].

A seguir, um exemplo de código detectado pela ferramenta SoCRATES[16], que utiliza a técnica de regras para detecção. Foi implementada e analisa *tests smells* na linguagem Scala e possui suporte ao *framework* de teste Scalameta[2]:

Listing 4.2: Exemplo *General Fixture test smell* [16]

```
1   class RecipeTestSuite extends FlatSpec {
2
3     trait RecipeFixture {
4       val ingredients1 = List(Ingredient("Cookie", 100), Ingredient("Milk", 200))
5       val ingredients2 =List(Ingredient("Eggs", 100),Ingredient("Bacon", 200))
6       val cookiesAndMilk = Recipe("CookiesandMilk",ingredients1)
7       val baconAndEggs = Recipe("Eggs", ingredients2)
8     }
9
10    "The recipe" should "have two ingredients (Eggs, Bacon)" in
11      new RecipeFixture {
12        val r = baconAndEggs.names.equals(List("Eggs", "Bacon"))
13        assert(r == true, "...")
14      }
15
16    "The recipe" should "have two ingredients" in
17      new RecipeFixture {
18        assert(cookiesAndMilk.ingredients.size == 2, "...")
19      }
20  }
```

4.3.3 *Information Retrieval* (Recuperação de informação)

Nessa técnica, as principais etapas incluem extrair informações/conteúdo do código de teste e normalizá-lo. No processo de extração, são apenas considerados os recursos necessários para identificar os *tests smells* do conteúdo textual de cada classe de teste. Essas características

[2] https://scalameta.org/

são normalizadas por meio de várias etapas de pré-processamento de texto, e o resultado é a aplicação de algoritmos de aprendizagem de máquina para extrair recursos textuais que podem identificar os *smells* [5].

A ferramenta TEDD *(Test Dependency Detector)* é uma ferramenta que utiliza da técnica de *Information Retrieval* para detectar dependências de testes para casos de teste da Web E2E com base na análise de strings e processamento de linguagem natural (NLP) [10]. Na Figura 4.1, é possível ver um exemplo de detecção pela ferramenta TEDD de dependências de testes, em que um teste depende dos dados do outro.

```
1   @Test
2   public void addUserTest() {
3    driver.findElement(By.id("login")).sendKeys("admin");
4    driver.findElement(By.id("password")).sendKeys("admin");
5    driver.findElement(By.xpath("//button")).click();
6    driver.findElement(By.linkText("Platform administration")).click();
7    driver.findElement(By.linkText("Create user")).click();
8    driver.findElement(By.id("lastname")).sendKeys("Name001");
9    driver.findElement(By.id("firstname")).sendKeys("Firstname001");
10   driver.findElement(By.id("username")).sendKeys("user001");
11   driver.findElement(By.id("password")).sendKeys("password001");
12   driver.findElement(By.id("password_conf")).sendKeys("password001");
13   assertEquals("The new user has been created", driver.findElement(By.↩
         xpath("//*[@id='claroBody']")).getText());
14   driver.findElement(By.id("logout")).click();
15  }
16  @Test
17  public void searchUserTest() {
18   driver.findElement(By.id("login")).sendKeys("admin");
19   driver.findElement(By.id("password")).sendKeys("admin");
20   driver.findElement(By.xpath("//button")).click();
21   driver.findElement(By.linkText("Platform administration")).click();
22   driver.findElement(By.id("search_user")).sendKeys("user001");
23   driver.findElement(By.cssSelector("input[type='submit']")).click();
24   assertEquals("Name001" ,driver.findElement(By.id("L0")).getText());
25   assertEquals("Firstname001",driver.findElement(By.xpath("//td[3]")).↩
         getText());
26   driver.findElement(By.id("logout")).click();
27  }
```

Figura 4.1: Exemplo de detecção utilizando a técnica *Information Retrieval* pela ferramenta TEDD [10].

4.3.4 *Dynamic Tainting* (Rastreamento dinâmico)

Essa técnica monitora o código-fonte enquanto é executado — utilizando-se da análise dinâmica —, permitindo a análise dos dados reais com base nas informações de tempo de execução. Funciona em duas etapas: i) executar o código-fonte junto a uma predefinição de valor/marca, ou seja, a entrada do usuário; ii) encontrar o motivo de quais execuções são afetadas por esse valor/marca [5].

A seguir, um exemplo de detecção de dependência de testes pela ferramenta PraDeT [23] que utiliza técnicas de *Dynamic Tainting*. Neste exemplo, um teste depende do outro, pois leem e escrevem no mesmo arquivo estático salvo na variável *data*.

Listing 4.3: Exemplo de testes com dependências de dados [23]

```
public class DataSourceTest {
  public static DataSource data;

  @Test
  public void testSetField() {
    String short_name = "Repository";
    RepoKind repo_kind = RepoKind.HG;
    data = new DataSource(short_name, "path", repo_kind);
    assertEquals(short_name, data.getShortName());
    assertEquals(repo_kind, data.getKind());
  }

  @Test
    public void testSetCloneString() {
    assertTrue(data.getCloneString().equals("path"));
    data.setCloneString("path_2");
    assertTrue(data.getCloneString().equals("path_2"));
  }

  @Test
  public void testToString() {
    String short_name = "short_name";
    RepoKind kind = RepoKind.HG;
    String cloneString = "clone_string";
    data.setShortName(short_name);
    data.setKind(kind);
    data.setCloneString(cloneString);
    assertTrue(data.toString().equals(short_name + "_" + kind + "_" + cloneString
        ));
  }
}
```

4.3.5 *Abstract Syntax Tree* (AST)

Abstract Syntax Tree ou árvore de estrutura sintática, é uma estrutura de dados para representação da sintaxe de código-fonte de uma linguagem de programação em forma hierárquica de árvore [3]. Com uma determinada AST é possível reproduzir código funcionalmente idêntico ao código-fonte que o gerou originariamente [1]. Cada nó da árvore denota uma instrução que ocorre no código-fonte. Durante a conversão do código-fonte para AST, apenas detalhes estruturais e relacionados ao conteúdo do código são preservados, e quaisquer detalhes adicionais são descartados [2].

Na Figura 4.2, é possível ver um exemplo de AST para o código: :if a = b then return "equal"else return a + "not equal to"+ b [1].

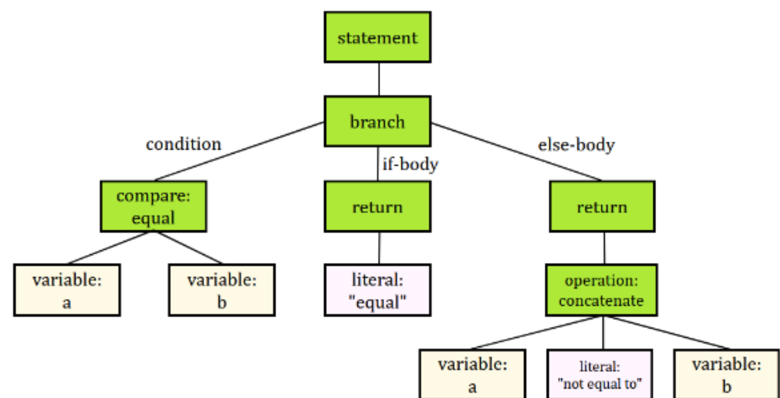


Figura 4.2: Representação gráfica de uma AST [1]

As informações vitais para o propósito da AST são: i) tipos de variáveis e a localização de cada declaração; ii) ordem e definição das instruções executáveis — que devem ser bem definidas; iii) componente esquerdo e direito de operações binárias devem ser armazenados e identificados corretamente; e iv) identificadores e seus valores atribuídos [2].

Em linguagens de comunicação, a estrutura sintática pode variar de língua para língua. Por exemplo, a estrutura sintática de uma frase em português pode ser muito diferente de uma em inglês. Em linguagens de programação não é diferente. Exemplificando: em C, a declaração de uma função normalmente começa com o tipo de retorno, seguido do nome da função, já em Python começa com a palavra-chave *def* e o nome da função.

Essa estrutura é de suma importância no uso da criação de ferramentas de identificação de *smells*, onde é necessário processar o conteúdo dos arquivos de testes para encontrá-los.

5

Proposta

No capítulo anterior foram apresentadas ferramentas de detecções de *tests smells*, sua importância e características comuns entre elas. Foi constatado que grande parte das ferramentas são específicas para JUnit, sendo que poucas possuem suporte para outras linguagens e até mesmo outros frameworks de teste. Notando essa carência, a proposta da presente monografia é contribuir para se chegar a uma ferramenta de detecção de *tests smells* que possui suporte a outras linguagens além de Java e outros frameworks de testes. Visa-se, assim, que a ferramenta aqui desenvolvida evite o trabalho de se criar novas ferramentas para cada novo framework que for lançado, de modo que necessite apenas ajustes na forma de detecção.

5.1 Tecnologias Utilizadas

Para o desenvolvimento da ferramenta, foi utilizado a linguagem de programação Java 8, Maven como gerenciador de dependências e bibliotecas como: i) JUnit4 para testes automatizados; ii) Log4j para observabilidade; e iii) Reflections. Para a escrita do código-fonte foi utilizado a IDE IntelliJ, e para versionamento o Git.[1] O código está hospedado no Github, com uma branch para a ferramenta funcionar para códigos C++ e outra branch para códigos Java.

A principal ferramenta utilizada foi o srcML, que será detalhada e demonstrada a importância da utilização na ferramenta para identificação dos *tests smells*.

Atualmente, a ferramenta possui suporte para as linguagens Java e C++ e os frameworks de teste JUnit e GoogleTest[2], mas com possibilidades de expansão para outros frameworks como Mockito[3], além de linguagens como C# e C.

[1] https://git-scm.com/
[2] http://google.github.io/googletest/
[3] https://site.mockito.org/

5.1.1 Java

Para o desenvolvimento da ferramenta foi utilizado a linguagem de programação Java 8. Java é uma linguagem de programação multiplataforma e plataforma de computação lançada pela Sun Microsystems em 1995 [45].

É uma linguagem orientada a objetos e centrada em rede, que pode ser usada como uma plataforma em si. É uma linguagem de programação rápida, segura e confiável para codificar tudo, desde aplicações móveis a software empresarial, até aplicações de *big data* [8].

5.1.2 Reflections

Reflections permite que um programa Java em execução examine ou "introspeccione"sobre si e manipule as propriedades internas do programa [39]. É possível pegar informações das classes, construtores e métodos, também permitindo a execução dos métodos da classe.

Ao arquitetar a ferramenta, a utilização de *reflections* abriu possibilidades para manter a ferramenta padronizada e de fácil expansão. Será detalhado adiante na Seção 5.2.

5.1.3 JUnit

Conforme informação certificada encontrada no *Frequently Asked Questions* (FAQ), o JUnit é um simples *framework open-source* para escrever e executar testes repetíveis. É uma instância da arquitetura xUnit para estruturas de teste de unidade. Os recursos do JUnit incluem: i) Asserções para testar os resultados esperados; ii) Dispositivos de teste para compartilhar dados de teste comuns; e iii) Executores de teste para executar testes. JUnit foi originalmente escrito por Erich Gamma e Kent Beck [13].

Na ferramenta foi utilizado o JUnit para criar casos de testes para cada *smell* suportado e testes de unidade para as funções desenvolvidas.

5.1.4 Maven

Segundo o site oficial, o Apache Maven é uma ferramenta de gerenciamento e compreensão de projetos de software. Com base no conceito de *project object model (POM)* — arquivo declarativo utilizado para controlar toda a estrutura do projeto —, o Maven pode gerenciar a compilação (*build*), gerenciamento de dependências, criação de relatórios e a documentação de um projeto a partir de uma informação central [19]. Há também possibilidades de usos de *plug-ins* para utilizar várias outras ferramentas de desenvolvimento para geração de relatórios ou processos de *build*.

Foi feito o uso do Maven para gerenciamento de dependências como JUnit, Log4j e Reflections. O *plug-in* Apache Maven Compiler [20] foi utilizado para compilar todos os códigos-fontes java.

5.1.5 Log4j

Condizente com as definições obtidas em site oficial, o Apache Log4j é uma estrutura de registro altamente escalável, robusta e versátil. Essa API simplifica a gravação do código de registro (*logs*) em um aplicativo, mas permite a flexibilidade de controlar a atividade de *logs* utilizando um arquivo de configuração externo. Também nos permite publicar informações de *logs* na granularidade desejada — por exemplo, com propósitos de depuração ou informativos —, dependendo do detalhe das informações de *logs* adequadas a cada aplicação [27].

A utilização de API's de logs é de suma importante para localizar erros, mal funcionamentos no sistema e manter uma linha do tempo do que foi executado.

5.1.6 IntelliJ IDEA

IntelliJ IDEA é um ambiente de desenvolvimento integrado (IDE) para linguagens JVM desenvolvido pela JetBrains. Foi projetado para maximizar a produtividade do desenvolvedor. A ferramenta faz as tarefas rotineiras e repetitivas, fornecendo conclusão de código inteligente, análise estática de código e refatorações, e permite que você se concentre no lado positivo do desenvolvimento de software, tornando-o não apenas produtivo, mas também uma experiência agradável [31].

A escolha da IDE da JetBrains para o desenvolvimento da ferramenta foi devido aos grandes benefícios citados anteriormente, a sua versão não paga possui diversas funcionalidades que ajudam a aumentar a produtividade.

5.1.7 GitHub

O GitHub é uma plataforma de hospedagem de código para controle de versão e uma grande rede social para desenvolvedores, permitindo a colaboração de outros desenvolvedores de qualquer parte do mundo possibilitando a visualização do seu código e submissão de mudanças ou sugestões de melhorias [25].

Para desenvolver a ferramenta, foi necessário utilizar o GitHub para fazer o controle de versão e hospedagem do código-fonte de forma que ficasse assegurado sua integralidade em eventual falha de *hardware*. Como a plataforma fornece códigos de outros desenvolvedores, foi utilizada para a busca de repositórios com finalidade de validação.

5.1.8 srcML

O srcML é um programa de linha de comando que faz a conversão do código-fonte para uma representação estruturada com o XML, onde as tags de marcação do XML são usadas para identificar elementos da sintaxe abstrata da linguagem [4]. Vários recursos tornam o srcML particularmente útil para evolução e manutenção. A sua filosofia principal é ter uma visão do

código centrada no programador, em vez de centrada no compilador. Primeiro, a conversão do código-fonte para srcML é sem perdas. Ou seja, nenhuma formatação, comentários ou código real é perdido. Há uma equivalência de ida e volta do código-fonte para srcML e vice-versa. Além disso, macros, modelos e instruções de pré-processador são marcados. Ou seja, o pré-processador não é executado (ou não precisa ser executado) antes da conversão para srcML. Isso também implica que o código sem bibliotecas incluídas ou fragmentos de código ausentes, pode ser convertido normalmente para srcML. Por fim, a conversão para srcML é extremamente eficiente, rodando mais rápido que um compilador [4].

A seguir, um exemplo de um código em C++, retirado do próprio site do srcML, para demonstração:

Listing 5.1: Exemplo programa C++ [4]

```
1  #include "rotate.h"
2  // rotate three values
3  void rotate(int& n1, int& n2, int& n3)
4  {
5    // copy original values
6    int tn1 = n1, tn2 = n2, tn3 = n3;
7    // move
8    n1 = tn3;
9    n2 = tn1;
10   n3 = tn2;
11 }
```

Listing 5.2: Exemplo da versão srcML correspondente [4]

```
1  <cpp:include>#<cpp:directive>include</cpp:directive> <cpp:file>"rotate.h"</
      cpp:file></cpp:include>
2  <comment type="line">// rotate three values</comment>
3  <function><type><name>void</name></type> <name>rotate</name><parameter_list>(<
      parameter><decl><type><name>int</name>
4  <modifier>&amp;</modifier></type>
5  <name>n1</name></decl></parameter>, <parameter><decl><type><name>int</name><
      modifier>&amp;</modifier></type> <name>n2</name></decl></parameter>, <
      parameter><decl><type><name>int</name><modifier>&amp;</modifier></type> <name
      >n3</name></decl></parameter>)</parameter_list>
6  <block>{<block_content>
7    <comment type="line">// copy original values</comment>
8    <decl_stmt><decl><type><name>int</name></type> <name>tn1</name> <init>= <expr><
        name>n1</name></expr></init></decl>, <decl><type ref="prev"/><name>tn2</name
        > <init>= <expr><name>n2</name></expr></init></decl>, <decl><type ref="prev"
        /><name>tn3</name> <init>= <expr><name>n3</name></expr></init></decl>;</
        decl_stmt>
9    <comment type="line">// move</comment>
10   <expr_stmt><expr><name>n1</name> <operator>=</operator> <name>tn3</name></expr>
        ;</expr_stmt>
11   <expr_stmt><expr><name>n2</name> <operator>=</operator> <name>tn1</name></expr>
        ;</expr_stmt>
12   <expr_stmt><expr><name>n3</name> <operator>=</operator> <name>tn2</name></expr>
        ;</expr_stmt>
13 </block_content>}</block></function>
```

E a partir dessa estruturação em XML do código-fonte, a mesma será utilizada para fazer a identificação dos *tests smells*.

5.2 Arquitetura da ferramenta

Dado o cenário de uso determinado pelo srcML, utilizamos a detecção baseada em regras para implementar a ferramenta, e foi projetada para se utilizar um recurso do Java chamado *Reflections*. Ao ser executado, o programa realizará uma varredura em todas as classes de teste do sistema a ser analisado, executando primeiramente o *srcML*, e através da AST gerada, será identificada cada função de teste existente.

Um dos principais motivos da utilização do *Reflections* na ferramenta, foi para a padronização de como é implementada a detecção dos smells. Foi desenvolvida uma classe abstrata "SmellMatcher"que contém duas funções: i) função "match", função na qual será implementada para detectar o *smell*; ii) função "write", responsável por escrever os resultados no arquivo final. Cada nova classe de *smell* a ser implementada ou existente, herdará da classe "SmellMatcher"para implementar a função "match"de acordo com suas particularidades para detecção.

Com isso, é utilizado *Reflections* para iterar sobre todas as classes que estão herdando a classe abstrata "SmellMatcher"e invocado a função "match", de forma que a execute para todas as funções de testes coletadas anteriormente. Essa arquitetura utilizada facilita na manutenção e implementação para novos *smells*.

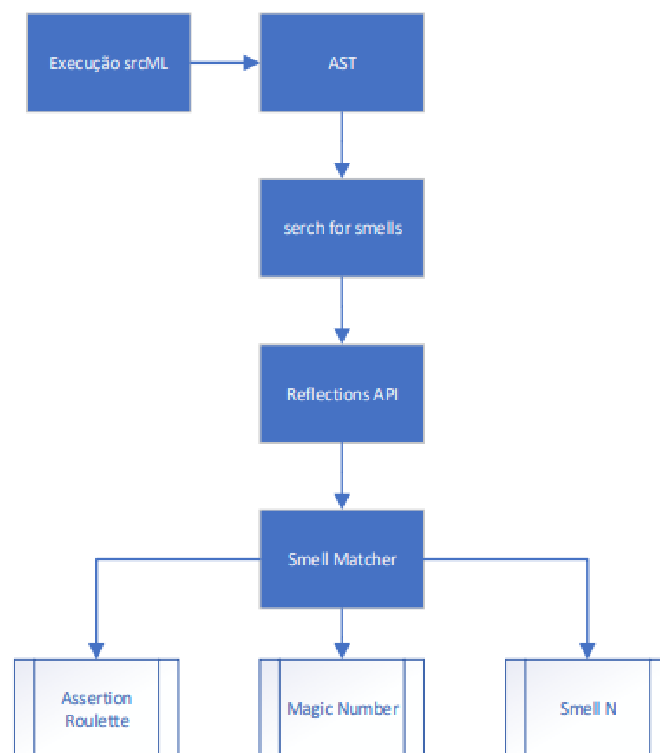


Figura 5.1: Arquitetura da ferramenta. Fonte: autoria própria.

A seguir, um exemplo da classe de detecção para o *smell Unknown Test* com a classe "UnknownTestMatcher"para C++ e Java. Podemos perceber que a implementação é idêntica, mudando apenas a sintaxe do *framework*, nesse caso o GoogleTest e o JUnit. Sendo possível dessa

forma, a centralização de ambas sintaxes de asserções de cada *framework* sem a necessidade de duplicação de código.

Listing 5.3: Exemplo UnknownTestMatcher.java C++

```
public class UnknownTestMatcher extends SmellMatcher {
    @Override
    protected void match(TestClass testClass) {
        for (TestMethod testMethod : testClass.getTestMethods()) {
            Node root = testMethod.getMethodDeclaration();
            boolean hasUnknownTestSmell = matchUnknownTest(root);
            if (hasUnknownTestSmell) {
                write(testMethod.getTestFilePath(), "Unknown Test", testMethod.
                    getMethodName(), "[]");
            }
        }
    }

    @Override
    public void write(String filePath, String testSmell, String name, String
        lines) {
        OutputWriter.getInstance().write(filePath, testSmell, name, lines);
        Logger.getLogger(UnknownTestMatcher.class.getName()).info("Found unknown
            test in method \"" + name + "\" in lines " + lines);
    }

    private boolean matchUnknownTest(Node root) {
        int assertionCount = 0;
        DocumentTraversal traversal = (DocumentTraversal) root.getOwnerDocument();
        TreeWalker iterator = traversal.createTreeWalker(root, NodeFilter.SHOW_ALL
            , null, false);
        Node node = null;
        while ((node = iterator.nextNode()) != null) {
            String textContent = node.getTextContent().trim();
            if (node.getNodeName().equals("expr") && (textContent.startsWith("
                EXPECT_") || textContent.startsWith("FAIL") || textContent.
                startsWith("ASSERT_"))) {
                assertionCount += 1;
            }
        }
        return assertionCount == 0;
    }
}
```

Listing 5.4: Exemplo UnknownTestMatcher.java Java

```
public class UnknownTestMatcher extends SmellMatcher {

  @Override
  protected void match(TestClass testClass) {
    for (TestMethod testMethod : testClass.getTestMethods()) {
      Node root = testMethod.getMethodDeclaration();
      boolean hasUnknownTestSmell = matchUnknownTest(root);
      if (hasUnknownTestSmell) {
        write(testMethod.getTestFilePath(), "Unknown Test", testMethod.
            getMethodName(), "[]");
      }
    }
  }

  @Override
  public void write(String filePath, String testSmell, String name, String lines)
```

```
        {
16    OutputWriter.getInstance().write(filePath, testSmell, name, lines);
17    Logger.getLogger(UnknownTestMatcher.class.getName()).info("Found assertion
        roulette in method \"" + name + "\" in lines " + lines);
18  }
19
20  private boolean matchUnknownTest(Node root) {
21    int assertionCount = 0;
22    DocumentTraversal traversal = (DocumentTraversal) root.getOwnerDocument();
23    TreeWalker iterator = traversal.createTreeWalker(root, NodeFilter.SHOW_ALL,
        null, false);
24    Node node = null;
25    while ((node = iterator.nextNode()) != null) {
26      String textContent = node.getTextContent().trim();
27      if (node.getNodeName().equals("expr") && textContent.toLowerCase().
          startsWith("assert") || textContent.toLowerCase().startsWith("fail")) {
28        assertionCount += 1;
29      }
30    }
31    return assertionCount == 0;
32  }
33 }
```

6

Avaliação

O objetivo desta avaliação é realizar um estudo exploratório para ver se a ferramenta consegue detectar um *test smell* corretamente. Para avaliar a ferramenta proposta neste trabalho, iremos utilizar as métricas de avaliação de desempenho: i) precisão; ii) *recall* e iii) *F-Score*.

6.1 Métricas

6.1.1 Matriz de confusão

Matriz de confusão é uma tabela que permite a visualização dos erros e acertos de um modelo, para medir o desempenho. A matriz com apenas uma dimensão mostrará apenas os valores reais, ou seja, quanto o modelo acertou e quanto errou. Já com duas dimensões, é registrado também quantas vezes o modelo resultou em verdadeiro, mas na verdade era falso, e quantas vezes resultou em falso, mas era verdadeiro. A matriz pode ter várias dimensões, aumentando a complexidade e tendo disponibilidade de outras métricas de desempenho [53]. No caso, para a ferramenta proposta, foi utilizado a matriz de duas dimensões, e a partir dessa matriz, derivando algumas métricas como a precisão, *recall* e *F-Score*.

Tabela 6.1: Exemplo matriz de confusão

		Valores classificados	
		Positivo	Negativo
Valores reais	Positivo	Verdadeiro positivo (VP)	Falso negativo (FN)
	Negativo	Falso positivo (FP)	Verdadeiro negativo (VN)

- VP: Verdadeiro positivo, é quanto o algoritmo classificou como positivo e é realmente positivo;

- FN: Falso negativo, é quanto o algoritmo classificou como falso mas o resultado é positivo.
- FP: Falso positivo, é quanto o algoritmo classificou como positivo mas o resultado é falso.
- VN: Verdadeiro negativo, é quanto o algoritmo classificou como negativo e é realmente negativo;

6.1.2 *Precision*

Com a precisão é calculado do total de positivos a quantidade de verdadeiros positivos. Então, é incluído também a quantidade de classificações rotuladas como falsos positivos. Uma alta precisão indica que poucos negativos foram classificados incorretamente como positivos [15][53].

Fórmula da precisão:

$$P = \frac{Verdadeiropositivo(VP)}{Verdadeiropositivo(VP) + Falsopositivo(FP)}$$

6.1.3 *Recall*

O *recall* calcula quanto dos resultados positivos o algoritmo rotulou como positivo. Ou seja, é o total de verdadeiros positivos dividido pela soma de verdadeiros positivos e falsos negativos (algoritmo classificou como negativo mas era positivo). Um alto *recall* significa que muito poucos positivos são classificados como negativos [15][53].

Fórmula *recall*:

$$R = \frac{Verdadeiropositivo(VP)}{Verdadeiropositivo(VP) + Falsonegativo(FN)}$$

6.1.4 *F-Score*

O F-score — também chamado de *F-measure* — é definido como a média ponderada da precisão e do *recall*, dependendo do peso (β) da função de ponderação. O *F1-score* significa a média harmônica entre precisão e *recall*. A pontuação *F1-Score* pode ter índices diferentes, dando pesos diferentes para precisão e o *recall* [15][53].

A seguir a fórmula *F-Score*, onde P indica o valor da precisão e R do *recall*:

$$F_\beta = (1+\beta^2) * \frac{P * R}{\beta^2 + P + R}$$

Com β = 1, temos o *F1-Score*:

$$F_1 = 2 * \frac{P * R}{P + R}$$

6.1.5 *Settings*

Foram selecionados alguns projetos *open-sources* que estão hospedados no GitHub que usem Java e C++ para validar a ferramenta. Foram considerados os projetos abaixo, selecionados aleatoriamente, filtrando pelo *frameworks* de testes GoogleTest e JUnit, através do GitHub. Foi feito o *download* de cada projeto e executado a ferramenta para iniciar o processo de avaliação. Manualmente, foram selecionados 7 ou mais casos identificados pela ferramenta para cada *test smell* e verificado a qual classificação pertencia.

Sistemas analisados pela ferramenta			
Projeto	*GitHub stars*	Implementação	*Test Frameworks* utilizados
vscode-catch2-test-adapter	149	TypeScript, C++, JavaScript	GoogleTest, GUnit, DocTest
IRCIS	105	C++	GoogleTest
Physically-based-deferred-shading	11	C++	GoogleTest
Echo-Web-Server	57	C++	GoogleTest
active-class	2	C++	GoogleTest
janusgraph	4.767	Java	JUnit4

Tabela 6.2: Sistemas utilizados para avaliação

6.2 Resultados

6.2.1 Planejamento

Para a avaliação da ferramenta, no GoogleTest, foram utilizados no total 5 projetos para identificação de *smells*, sendo eles: IRCIS [43], Physically-based-deferred-shading [34], vscode-catch2-test-adapter [48], active-class [36] e Echo-Web-Server [63]. Já no caso da avaliação para o Java, foi utilizado o projeto JanusGraph.[1]

[1]Online. Disponível em https://github.com/JanusGraph/janusgraph

Assertion Roulette		
Total de detecções analisadas: 7		
	Positivo	Negativo
Verdadeiro	7	0
Falso	0	0

Conditional Test		
Total de detecções analisadas: 7		
	Positivo	Negativo
Verdadeiro	7	0
Falso	0	0

Duplicate Assert		
Total de detecções analisadas: 7		
	Positivo	Negativo
Verdadeiro	7	0
Falso	0	0

Empty Test		
Total de detecções analisadas: 8		
	Positivo	Negativo
Verdadeiro	4	0
Falso	4	0

Exception Handling		
Total de detecções analisadas: 7		
	Positivo	Negativo
Verdadeiro	7	0
Falso	0	0

Magic Number		
Total de detecções analisadas: 7		
	Positivo	Negativo
Verdadeiro	7	0
Falso	0	0

Unknown Test		
Total de detecções analisadas: 7		
	Positivo	Negativo
Verdadeiro	3	0
Falso	4	0

Resultados	
Total VP	Total VN
42	0
Total FP	Total FN
8	0

Tabela 6.3: Resultados Java. Fonte: dados da pesquisa.

Assertion Roulette		
Total de detecções analisadas: 15		
	Positivo	Negativo
Verdadeiro	13	0
Falso	2	0

Conditional Test		
Total de detecções analisadas: 7		
	Positivo	Negativo
Verdadeiro	7	0
Falso	0	0

Duplicate Assert		
Total de detecções analisadas: 7		
	Positivo	Negativo
Verdadeiro	7	0
Falso	0	0

Empty Test		
Não detectado		
	Positivo	Negativo
Verdadeiro	0	0
Falso	0	0

Exception Handling		
Total de detecções analisadas: 6		
	Positivo	Negativo
Verdadeiro	6	0
Falso	0	0

Magic Number		
Total de detecções analisadas: 16		
	Positivo	Negativo
Verdadeiro	13	0
Falso	3	0

Unknown Test		
Total de detecções analisadas: 1		
	Positivo	Negativo
Verdadeiro	1	0
Falso	0	0

Resultados	
Total VP	Total VN
47	0
Total FP	Total FN
5	0

Tabela 6.4: Resultados C++. Fonte: dados da pesquisa.

A planilha com os resultados está disponível pelo *link*:

https://docs.google.com/spreadsheets/d/1BEodj3_7qp8JbBmJ4yIiCC0-pna69S0_eX5iYNLb1p8/edit#gid=0

6.2.2 Resultados

A ferramenta foi capaz de detectar corretamente *tests smells* nos projetos informados com uma precisão de 87.25%, *recall score* de 100% e média F-score de 93.19%.

Resultado Geral			
Total VP	Total FP	Total FN	Total VN
89	13	0	0
Precisão	Recall	F-Score	
0.8725490196	1	0.9319371728	

Tabela 6.5: Resultados. Fonte: dados da pesquisa.

Houve casos em que não foi possível fazer a análise devido ao srcML não traduzir corretamente algumas funções, por exemplo, no projeto Echo-Web-Server [63], mais precisamente no arquivo *block_deque_text.cpp*, existe a função:

Listing 6.1: block_deque_text.cpp [63]

```
TEST_F(BlockDequeTest, MultiThreadPushPop) {
    std::latch done {4};

    const auto push {[&done, this]() {
        deq1_.PushBack(0);
        done.count_down();
    }};

    const auto pop {[&done, this]() {
        deq1_.Pop(std::chrono::milliseconds {5000});
        done.count_down();
    }};

    EXPECT_EQ(deq1_.Size(), 2);
    std::thread {push}.detach();
    std::thread {pop}.detach();
    std::thread {pop}.detach();
    std::thread {pop}.detach();
    done.wait();
    EXPECT_TRUE(deq1_.Empty());
}
```

Listing 6.2: Tradução feita pelo srcML

```
<macro><name>TEST_F</name><argument_list>(<argument>BlockDequeTest</argument>, <
    argument>MultiThreadPushPop</argument>)</argument_list></macro><block>{
<block_content>
<decl_stmt><decl><type><name><name>std</name><operator>::</operator>
<name>latch</name>
</name></type><name>done</name><argument_list>{<argument><expr><literal type="
    number">4</literal></expr></argument>}</argument_list></decl>;</decl_stmt>
<decl_stmt>
<decl><type><specifier>const</specifier><specifier>auto</specifier></type><name>
    push</name>
<argument_list>{<argument><expr><lambda><capture>[<argument><modifier>&amp;</
    modifier>
<name>done</name></argument>,
<argument><name>this</name></argument>]</capture><parameter_list>()</
    parameter_list><block>{<block_content><expr_stmt><expr><call><name><name>
    deq1_</name><operator>.</operator><name>PushBack
</name></name><argument_list>(<argument><expr><literal type="number">0</literal><
    /expr></argument>)</argument_list></call></expr>;</expr_stmt><expr_stmt><expr
    ><call><name><name>done</name><operator>.</operator><name>count_down</name></
    name><argument_list>()</argument_list></call></expr>;</expr_stmt><expr_stmt/>
    }</block_content>}</block></lambda></expr></argument>;

    <argument><expr><specifier>const</specifier><name>auto</name><call><name>pop<
        /name><argument_list>{<argument><expr><lambda><capture>[<argument><
        modifier>&amp;</modifier><name>done</name></argument>, <argument><name>
        this</name></argument>]</capture><parameter_list>()</parameter_list><block
        >{<block_content><expr_stmt><expr><call><name><name>deq1_</name><operator>
        .</operator><name>Pop</name></name><argument_list>(<argument><expr><call><
        name><name>std</name><operator>::</operator><name>chrono</name><operator>
        ::</operator><name>milliseconds</name></name><argument_list>{<argument><
        expr><literal type="number">5000</literal></expr></argument>}</
```

```
    argument_list></call></expr></argument>)</argument_list></call></expr>;</
    expr_stmt><expr_stmt><expr><call><name><name>done</name><operator>.</
    operator><name>count_down</name></name><argument_list>()</argument_list></
    call></expr>;</expr_stmt><expr_stmt/>}</block_content>}</block></lambda></
    expr></argument>;

  <argument><expr><call><name>EXPECT_EQ</name><argument_list>(<argument><expr><
    call><name><name>deq1_</name><operator>.</operator><name>Size</name></name
    ><argument_list>()</argument_list></call></expr></argument>, <argument><
    expr><literal type="number">2</literal></expr></argument>)</argument_list>
    </call></expr></argument>;
  <argument><expr><call><name><name>std</name><operator>::</operator><name>
    thread</name></name><argument_list>{<argument><expr><name>push</name></
    expr></argument>}</argument_list></call><operator>.</operator><call><name>
    detach</name><argument_list>()</argument_list></call></expr></argument>;
  <argument><expr><call><name><name>std</name><operator>::</operator><name>
    thread</name></name><argument_list>{<argument><expr><name>pop</name></expr
    ></argument>}</argument_list></call><operator>.</operator><call><name>
    detach</name><argument_list>()</argument_list></call></expr></argument>;
  <argument><expr><call><name><name>std</name><operator>::</operator><name>
    thread</name></name><argument_list>{<argument><expr><name>pop</name></expr
    ></argument>}</argument_list></call><operator>.</operator><call><name>
    detach</name><argument_list>()</argument_list></call></expr></argument>;
  <argument><expr><call><name><name>std</name><operator>::</operator><name>
    thread</name></name><argument_list>{<argument><expr><name>pop</name></expr
    ></argument>}</argument_list></call><operator>.</operator><call><name>
    detach</name><argument_list>()</argument_list></call></expr></argument>;
  <argument><expr><call><name><name>done</name><operator>.</operator><name>wait
    </name></name><argument_list>()</argument_list></call></expr></argument>;
  <argument><expr><call><name>EXPECT_TRUE</name><argument_list>(<argument><expr
    ><call><name><name>deq1_</name><operator>.</operator><name>Empty</name></
    name><argument_list>()</argument_list></call></expr></argument>)</
    argument_list></call></expr></argument>;
}</argument_list></call>
```

O problema com essa função é que o scrML não fez a tradução corretamente, onde algumas *tags* possui a abertura mas não há fechamento.

6.2.3 Discussão

Para aferir se um resultado é positivo, é necessário observar as métricas. Acima, nas seções 6.1.2, 6.1.3 e 6.1.4, foram apresentadas as métricas aplicadas no presente trabalho e é importante salientar que o oráculo para validação se os resultados foram corretos foi o próprio autor, onde, manualmente foram analisadas funções apontadas com *smell* verificando se realmente existia. Desta forma, tendo por base a métrica Precisão, para ser considerada alta, deve indicar que poucos negativos foram classificados incorretamente como positivo, ou seja, o quanto a ferramenta classificou como positivo os realmente positivos, apontando um baixo índice de falsos positivos. Na amostra utilizada para validação da ferramenta, indicou uma precisão de 87.25%, implicando que poucos negativos foram classificados como positivo.

Para a métrica *Recall*, quanto maior o percentual, melhor é a classificação de acertos, pois calcula quanto dos resultados positivos o algoritmo classificou como positivo. A ferramenta obteve 100% na amostra feita, indicando que não houve positivos classificados incorretamente

como negativos.

Por fim, a métrica *F-score*, por se tratar de uma média harmônica, é uma forma de visualização única da métrica de precisão e *recall*, dando a mesma importância para ambas as métricas. O F-score alto nesse contexto, indica haver um alto equilíbrio entre a precisão e o *recall*. E a ferramenta obteve um f-score de 93.19%.

Portanto, os resultados obtidos pela análise da ferramenta nessa amostragem, indica que possui uma alta taxa de precisão e *recall*, o que a torna confiável para uso.

A ferramenta atualmente está pecando mais em falsos negativos, por exemplo, para o *smell Empty Test*, quando se está utilizando a anotação *Ignored* ou *Disabled* no JUnit, significa que esse teste não será executado, mas a ferramenta irá identificar o *smell* incorretamente, então para melhorar as detecções é necessário avaliar mais casos de uso. Já no caso *Unknown test*, apesar de não ter nenhuma asserção, existe uma chamada de função que contém todas as asserções necessárias, mas a ferramenta não irá verifica-la, levando a um falso positivo. A seguir, é possível ver exemplos dos *smells Empty Test e Unknown Test* detectado pela ferramenta erroneamente no projeto JanusGraph.[2]

Listing 6.3: InMemoryGraphTest.java

```
1  @Override @Test @Disabled
2  public void testFixedGraphConfig() {}
```

Listing 6.4: JanusGraphPartitionGraphTest.java

```
1  @Test
2  public void testPartitionSpreadFlushBatch() {
3      testPartitionSpread(true,true);
4  }
5
6  private void testPartitionSpread(boolean flush, boolean batchCommit) {
7      Object[] options = {option(GraphDatabaseConfiguration.IDS_FLUSH), flush};
8      clopen(options);
9
10     int[] groupDegrees = {10,15,10,17,10,4,7,20,11};
11     int numVertices = setupGroupClusters(groupDegrees,batchCommit?CommitMode.
           BATCH:CommitMode.PER_VERTEX);
12
13     IntSet partitionIds = new IntHashSet(numVertices); //to track the "spread" of
            partition ids
14     for (int i=0;i<groupDegrees.length;i++) {
15         JanusGraphVertex g = getOnlyVertex(tx.query().has("groupid","group"+i));
16         assertCount(groupDegrees[i],g.edges(Direction.OUT,"contain"));
17         assertCount(groupDegrees[i],g.edges(Direction.IN,"member"));
18         assertCount(groupDegrees[i],g.query().direction(Direction.OUT).edges());
19         assertCount(groupDegrees[i],g.query().direction(Direction.IN).edges());
20         assertCount(groupDegrees[i]*2,g.query().edges());
21         for (JanusGraphVertex o : g.query().direction(Direction.IN).labels("member
               ").vertices()) {
22             int pid = getPartitionID(o);
23             partitionIds.add(pid);
24             assertEquals(g, getOnlyElement(o.query().direction(Direction.OUT).
                   labels("member").vertices()));
```

[2]Online. Disponível em https://github.com/JanusGraph/janusgraph

```
25            VertexList vertexList = o.query().direction(Direction.IN).labels("
                 contain").vertexIds();
26            assertEquals(1,vertexList.size());
27            assertEquals(pid,idManager.getPartitionId(vertexList.getID(0)));
28            assertEquals(g,vertexList.get(0));
29         }
30      }
31      if (flush || !batchCommit) { //In these cases we would expect significant
           spread across partitions
32         assertTrue(partitionIds.size()>numPartitions/2); //This is a probabilistic
                test that might fail
33      } else {
34         assertEquals(1,partitionIds.size()); //No spread in this case
35      }
36   }
```

No exemplo a seguir, no projeto Physically-based-deferred-shading [34] (C++), acontece algo similar a casos do *Unknown test* no JUnit. Houve um falso positivo na detecção da ferramenta, onde identificou um *Magic Number smell* dentro de uma chamada de função que não é uma asserção.

Listing 6.5: Mat4_Tests.cpp [34]

```
1
2   TEST(Operations, MatMultVec)
3   {
4      using namespace TestUtils;
5
6      float const EXPECTED[16] {24.2f, 33.0f, 44.0f, 4.4f};
7
8      // Scale by [2, 3, 4] then translate by [5, 6, 7]
9      const Mat4 mat1 = Mat4::translate({5, 6, 7}) * Mat4::scale({2, 3, 4});
10
11     const Vec4 v1 {1.1f, 2.2f, 3.3f, 4.4f};
12
13     Vec4 v2 = mat1 * v1;
14
15     EXPECT_VEC4_EQ_FLOAT_ARRAY_ROW(v2, EXPECTED, 0);
16  }
```

Embora tenha suas falhas, a ferramenta demonstra resultados significantes. A forma de identificação utilizando o srcML, possibilita o uso de uma única ferramenta para múltiplas linguagens. Levando-se em consideração que a proposta do trabalho é a identificação automatizada de *smells* através da ferramenta para diferentes linguagens de programação, os resultados obtidos apontam um alto índice de acerto, tanto para linguagem Java, quanto para C++, por exemplo.

Quanto mais tentamos generalizar a ferramenta devemos nos atentar quanto a casos específicos de cada *framework* de teste para cada linguagem, pode haver casos em que uma certa situação seja diferente em um outro *framework* ou linguagem e gerar algum tipo de detecção incorreta ou até mesmo a não identificação, ocasionando um decremento da precisão. Seria necessário maiores análises em mais projetos para chegar a tal conclusão.

Desta forma, vê-se que o objetivo lançado foi alcançado, uma vez que se prova, através de

números, a validade da proposta dessa ferramenta.

6.2.4 Ameaças à validade

Para a implementação de detecções em projetos que utilizam o JUnit, a ferramenta não possui suporte para todas as versões. Por exemplo, a versão 5, em que há novas anotações e uma série de outras mudanças. O mesmo se aplica para *frameworks* de testes para C++.

Além disso, possui as limitações do srcML que não cobre versões mais recentes do Java, C++ e as outras linguagens que possuem suporte, e ainda podem existir outros *bugs* relacionados a geração da AST, conforme citado na Seção 6.2.2, em que não foi realizado a tradução corretamente em algumas funções.

E como o oráculo para validação dos resultados e do comparativo entre a ferramenta e o JNose foi o próprio autor, pode haver a possibilidade de alguma validação incorreta ter passado despercebido.

Por fim, é possível que a ferramenta não tenha considerado outros casos de usos de detecções não percebidos durante a implementação.

6.2.5 Comparação com o JNose

Para fazer um comparativo entre a ferramenta desenvolvida e o JNose, foram analisadas 28 classes do projeto JanusGraph, sendo a maioria métodos de testes da mesma seleção utilizada para validação da ferramenta, totalizando em 55 análises, e o resultado dentro da amostragem selecionada foi:

Resultado		
	Ferramenta	JNose
Precision	100,00%	93,68%
Recall	83,56%	81,16%

Houve casos em que a ferramenta desenvolvida não fez a detecção, resultando assim em falsos negativos:

Classe	Método	Smell
CQLResultSetKeyIteratorTest	testUneven	Assertion Roulette
AbstractConfiguredGraphFactoryTest	shouldBeAbleToRemoveBogusConfiguration	Exception Handling
JanusGraphIndexTest	testConditionalIndexing	Duplicate Assert

Já o JNose, houve casos de falsos negativos, como por exemplo:

Classe	Método	Smell
DistributedStoreManagerTest	testGetLocalKeyPartition	Magic Number
MultiWriteKeyColumnValueStoreTest	deletionsAppliedBeforeAdditions	Conditional Test
JanusGraphIndexTest	testDateIndexing	Magic Number
QueryTest	testComplexConditions	Magic Number

Mas também houve casos de falsos positivos:

Classe	Método	Smell
KeyColumnValueStoreTest	testGetKeysColumnSlicesOnLowerTriangular	Conditional Test
KCVSCacheTest	testSmallCache	Conditional Test
JanusGraphIndexTest	testConditionalIndexing	Conditional Test

7

Conclusões

A importância de se testar softwares está atrelada ao fato de se saber se o sistema está atendendo aos resultados exigidos e se comporta como esperado. Um bom software facilita a interação do usuário, tornando mais simples a utilização de um determinado sistema.

Nem sempre é possível aos desenvolvedores testar devidamente as funcionalidades de um sistema e garantir a qualidade. É por esta razão que são realizados os chamados testes manuais. Conforme apresentado na Seção 2.2, os testes manuais requerem a repetição de tarefas simples que podem demandar muito tempo de execução, algo que, se automatizados, podem levar minutos e reduzir esforços e custos. Com isso, vimos que vale a pena utilizar testes automatizados.

Apesar da proposta agradar aos olhos, não é fácil manter testes automatizados de boa qualidade. Más práticas em desenvolvimento de testes automatizados podem ser indicativos de possíveis problemas no projeto ou na implementação — chamados *tests smells*. Em tal cenário, os custos e esforços mais se elevariam que reduziriam. Contudo, foi vista a dificuldade para desenvolvedores identificarem os *tests smells*, de forma a ser necessário a utilização de ferramentas para identificações.

Conforme demonstrado no presente trabalho, as ferramentas de identificação de *smells* existentes não abrangem grande parte das linguagens de desenvolvimento, razão pela qual se faz necessário o desenvolvimento da ferramenta apresentada, com fins de suprir essa carência.

Com os testes, foi possível analisar que a ferramenta obteve bons resultados na identificação de *smells* tanto para o *framework* GoogleTest, quanto para o JUnit.

É uma ferramenta promissora, com grande potencial de expansão para multi-linguagens e ferramentas de testes. Os códigos para as detecções ainda estão separados por *branchs*, então, como implementação futura, seria interessante juntar os projetos. O srcML também tem suporte para C# e C, o que seria de grande valia desenvolver as detecções para essas linguagens também. E é possível ir mais além, como implementações de algoritmos de aprendizado de máquina, de modo que possa refatorar da melhor forma o código de teste.

Referências bibliográficas

[1] Abstract syntax tree. URL https://www.codecademy.com/resources/docs/general/abstract-syntax-tree.

[2] Abstract syntax tree. URL https://deepsource.io/glossary/ast/.

[3] Abstract syntax tree (ast) in java. URL https://www.geeksforgeeks.org/abstract-syntax-tree-ast-in-java/.

[4] What is srcml? URL https://www.srcml.org/about.html.

[5] Wajdi Aljedaani, Anthony Peruma, Ahmed Aljohani, Mazen Alotaibi, Mohamed Wiem Mkaouer, Ali Ouni, Christian D Newman, Abdullatif Ghallab, and Stephanie Ludi. Test smell detection tools: A systematic mapping study. *Evaluation and Assessment in Software Engineering*, pages 170–180, 2021.

[6] Yasaman Amannejad, Vahid Garousi, Rob Irving, and Zahra Sahaf. A search-based approach for cost-effective software test automation decision support and an industrial case study. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, pages 302–311. IEEE, 2014.

[7] Kent C Archie, Owen R Fonorow, Mary C McGould, Robert E McLear III, Edward C Read, Edwin M Schaefer III, Suzanne E Schwab, and Dennis Wodarz. Test automation system, June 4 1991. US Patent 5,021,997.

[8] Aws. O que é java? URL https://aws.amazon.com/pt/what-is/java.

[9] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and David Binkley. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *2012 28th IEEE international conference on software maintenance (ICSM)*, pages 56–65. IEEE, 2012.

[10] Matteo Biagiola, Andrea Stocco, Ali Mesbah, Filippo Ricca, and Paolo Tonella. Web test dependency detection. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 154–164, 2019.

[11] Manuel Breugelmans and Bart Van Rompaey. Testq: Exploring structural and maintenance characteristics of unit test suites. In *WASDeTT-1: 1st International Workshop on Advanced Software Development Tools and Techniques*. Citeseer, 2008.

[12] Lucia Cavero-Baptista. What is a test automation pyramid? URL https://www.leapwork.com/blog/what-is-a-test-automation-pyramid.

[13] Mike Clark. Junit. URL https://junit.org/junit4/faq.html.

[14] Mike Cohn. *Succeeding with agile: software development using Scrum*. Pearson Education, 2010.

[15] Hercules Dalianis. *Evaluation Metrics and Evaluation*, pages 45–53. Springer International Publishing, Cham, 2018. ISBN 978-3-319-78503-5. **DOI** 10.1007/978-3-319-78503-5$_6$. URL https://doi.org/10.1007/978-3-319-78503-5_6.

[16] Jonas De Bleser, Dario Di Nucci, and Coen De Roover. Socrates: Scala radar for test smells. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Scala*, pages 22–26, 2019.

[17] Elfriede Dustin, Jeff Rashka, and John Paul. *Automated software testing: Introduction, management, and performance: Introduction, management, and performance*. Addison-Wesley Professional, 1999.

[18] J Feldstein. How to recruit, motivate, and energize superior test, dec. 2014. *Last accessed: http://www. youtube. com/watch.*

[19] The Apache Software Foundation. Apache maven, . URL https://maven.apache.org/what-is-maven.html.

[20] The Apache Software Foundation. Apache maven compiler, . URL https://maven.apache.org/plugins/maven-compiler-plugin/index.html.

[21] Martin Fowler. The practical test pyramid, Feb 2018. URL https://martinfowler.com/articles/practical-test-pyramid.html.

[22] Martin Fowler. Integration test, Jan 2018. URL https://martinfowler.com/bliki/IntegrationTest.html.

[23] Alessio Gambi, Jonathan Bell, and Andreas Zeller. Practical test dependency detection. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–11. IEEE, 2018.

[24] Vahid Garousi and Barış Küçük. Smells in software test code: A survey of knowledge in industry and academia. *Journal of systems and software*, 138:52–81, 2018.

[25] GitHub. Github. URL https://docs.github.com/en/get-started/quickstart/hello-world.

[26] Mark Grechanik, Qing Xie, and Chen Fu. Maintaining and evolving gui-directed test scripts. In *2009 IEEE 31st International Conference on Software Engineering*, pages 408–418. IEEE, 2009.

[27] Samudra Gupta. *Understanding Apache log4j*, pages 121–175. Apress, Berkeley, CA, 2003. ISBN 978-1-4302-0765-8. **DOI** 10.1007/978-1-4302-0765-8_5. URL https://doi.org/10.1007/978-1-4302-0765-8_5.

[28] Benedikt Hauptmann. *Reducing System Testing Effort by Focusing on Commonalities in Test Procedures*. PhD thesis, Technische Universität München, Germany, Jul 2016.

[29] Akram Hedayati, Maryam Ebrahimzadeh, and Amir Abbaszadeh Sori. Investigating into automated test patterns in erratic tests by considering complex objects. *International Journal of Information Technology and Computer Science (IJITCS)*, 7(3):54, 2015.

[30] Bill Hetzel. *The complete guide to software testing*. QED Information Sciences, Inc., 1988.

[31] JetBrains. Intellij idea. URL https://www.jetbrains.com/help/idea/discover-intellij-idea.html.

[32] Katja Karhu, Tiina Repo, Ossi Taipale, and Kari Smolander. Empirical observations on software testing automation. In *2009 International Conference on Software Testing Verification and Validation*, pages 201–209. IEEE, 2009.

[33] Vladimir Khorikov. *Unit Testing Principles, Practices, and Patterns*. Simon and Schuster, 2020.

[34] Jonas Kristoffersen. Physically based deferred shading. URL https://github.com/jonaskris/Physically-based-deferred-shading.

[35] Stefano Lambiase, Andrea Cupito, Fabiano Pecorelli, Andrea De Lucia, and Fabio Palomba. Just-in-time test smell detection and refactoring: The darts project. In *Proceedings of the 28th international conference on program comprehension*, pages 441–445, 2020.

[36] Massimo Marino. active class. URL https://github.com/massimo-marino/active-class.

[37] Robert C Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[38] Dennis Martinez. Top challenges of automated end-to-end testing, Oct 2021. URL https://www.telerik.com/blogs/top-challenges-automated-end-to-end-testing.

[39] Glen McCluskey. Using java reflection, Jan 1998. URL https://www.oracle.com/technical-resources/articles/java/javareflection.html.

[40] John Micco. Flaky tests at google and how we mitigate them. *Online] https://testing. googleblog. com/2016/05/flaky-tests-at-google-and-how-we. html*, 2016.

[41] Roy Miller and Christopher T Collins. Acceptance testing. *Proc. XPUniverse*, 238, 2001.

[42] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[43] Arjun Nair. Ircis. URL https://github.com/batman-nair/IRCIS.

[44] Students of the Department of Software Engineering. Software unit test smells. URL https://testsmells.org/pages/testsmellexamples.html.

[45] Oracle. What is java technology and why do i need it? URL https://www.java.com/en/download/help/whatis_java.html.

[46] Roy Osherove. *The Art of Unit Testing: with examples in C*. Simon and Schuster, 2013.

[47] Alan Page, Ken Johnston, and Bj Rollison. *How we test software at Microsoft*. Microsoft Press, 2008.

[48] Mate Pek. vscode catch2 test adapter. URL https://github.com/matepek/vscode-catch2-test-adapter.

[49] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. 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, 2019. IBM Corp.

[50] Anthony Peruma, Khalid Saeed Almalki, Christian D Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. On the distribution of test smells in open source android applications: An exploratory study. 2019.

[51] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. 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, 2020. Association for Computing Machinery. ISBN 9781450370431. **DOI** 10.1145/3368089.3417921. URL https://doi.org/10.1145/3368089.3417921.

[52] Valeria Pontillo, Dario Amoroso d'Aragona, Fabiano Pecorelli, Dario Di Nucci, Filomena Ferrucci, and Fabio Palomba. Machine learning-based test smell detection. *arXiv preprint arXiv:2208.07574*, 2022.

[53] Chat Room. Confusion matrix. *Mach. Learn*, 6:27, 2019.

[54] Thibaut Rousseau and Gwendal Leclerc. Declarative integration tests in a microservice environment, Aug 2020. URL https://blog.ovhcloud.com/declarative-integration-tests-in-a-microservice-environment/.

[55] K. Seguin. Unit testing: Do repeat yourself,. http://codebetter.com/karlseguin/2009/09/12/unit-testing-do-repeat-yourself/, 2009. Last accessed: April 2017.

[56] Elvys Soares, Marcio Ribeiro, Rohit Gheyi, Guilherme Amaral, and André Santos. Refactoring test smells with junit 5: Why should developers keep up-to-date? *IEEE Transactions on Software Engineering*, 49(3):1152–1170, 2022.

[57] Ossi Taipale, Jussi Kasurinen, Katja Karhu, and Kari Smolander. Trade-off between automated and manual software testing. *International Journal of System Assurance Engineering and Management*, 2(2):114–125, 2011.

[58] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and why your code starts to smell bad. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 403–414, 2015. **DOI** 10.1109/ICSE.2015.59.

[59] Arash Vahabzadeh, Amin Milani Fard, and Ali Mesbah. An empirical study of bugs in test code. In *2015 IEEE international conference on software maintenance and evolution (ICSME)*, pages 101–110. IEEE, 2015.

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

[61] Bart Van Rompaey, Bart Du Bois, Serge Demeyer, and Matthias Rieger. On the detection of test smells: A metrics-based approach for general fixture and eager test. *IEEE Transactions on software engineering*, 33(12):800–817, 2007.

[62] Tássio Virgínio, Luana Martins, Railana Santana, Adriana Cruz, Larissa Rocha, Heitor Costa, and Ivan Machado. On the test smells detection: an empirical study on the jnose test accuracy. *Journal of Software Engineering Research and Development*, 9:8–1, 2021.

[63] Chen Zhenshuo. Echo web server. URL https://github.com/Zhuagenborn/Echo-Web-Server.