Instituto Federal Catarinense
Curso de Bacharelado em Ciência da Computação
Campus Blumenau

VÍTOR AUGUSTO UENO OTTO

REFATORAÇÃO AUTOMATIZADA DE TESTES UNITÁRIOS: UMA FERRAMENTA PARA CORREÇÃO DO ASSERTION ROULETTE

Blumenau
2024

VÍTOR AUGUSTO UENO OTTO

REFATORAÇÃO AUTOMATIZADA DE TESTES UNITÁRIOS: UMA FERRAMENTA PARA CORREÇÃO DO ASSERTION ROULETTE

Trabalho de Conclusão de Curso submetido ao Curso de Bacharelado em Ciência da Computação do Instituto Federal Catarinense — *Campus* Blumenau para a obtenção do título de Bacharel em Ciência da Computação.
Orientador: Adriano Pizzini

Blumenau
2024

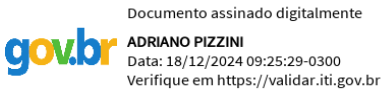
VÍTOR AUGUSTO UENO OTTO

REFATORAÇÃO AUTOMATIZADA DE TESTES UNITÁRIOS: UMA FERRAMENTA PARA CORREÇÃO DO ASSERTION ROULETTE

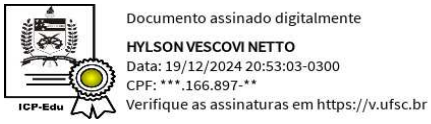
Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de "Bacharel em Ciência da Computação" e aprovado em sua forma final pelo curso de Bacharelado em Ciência da Computação do Instituto Federal Catarinense — Campus Blumenau.

Blumenau, 16 de Dezembro de 2024.

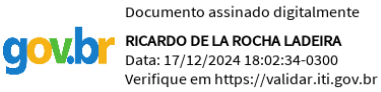
Banca Examinadora:



Prof. Adriano Pizzini, Me.
Orientador – IFC Campus Blumenau



Prof. Hylson Vescovi Netto, Dr.
IFC Campus Blumenau



Prof. Ricardo de la Rocha Ladeira, Me.
IFC Campus Blumenau

Dedico esse trabalho a mim por não desistir.

AGRADECIMENTOS

À minha família pelo apoio e convivência.
Ao Instituto Federal pela jornada transformadora.
À minha irmã por não me deixar desistir.

It has been one of the greatest and most difficult years of my life. I learned everything is temporary. Moments. Feelings. People. Flowers. (KAUR, 2017)

RESUMO

O presente trabalho de conclusão de curso tem como objetivo geral propor uma alternativa de solução para a refatoração de testes de unidade com a presença do *test smell assertion roullete*, má prática presente em códigos de teste de unidade que ocorrem quando as asserções não possuem uma mensagem de descrição que facilita o processo de depuração em caso de erro quando presentes em um método de teste que contenha mais de uma asserção. Após a identificação das principais lacunas da referência mais relacionada ao trabalho, a ferramenta RAIDE, uma ferramenta de prova de conceito para a refatoração de testes de unidade com a presença de *assertion roulette* foi proposta e prototipada. As principais diferenças entre a ferramenta e o RAIDE são a geração de mensagens automáticas baseadas no tipo de asserção, eliminando a necessidade de intervenção manual, e suporte tanto ao JUnit 4 quanto o JUnit 5. Estudos empíricos conduzidos com projetos open source demonstraram que a ferramenta reduz, ainda que não por completo, as ocorrências de assertion roulette e gera mensagens com razoável similaridade em relação às mensagens criadas por LLMs (Modelos de Linguagem de Grande Escala). Embora o trabalho tenha cumprido seus objetivos, foram identificadas limitações, como dificuldades na identificação de asserções em cenários específicos e oportunidades de melhoria no código da ferramenta. Como trabalhos futuros, propõe-se superar essas limitações, conduzir novos experimentos empíricos com desenvolvedores reais e validar a ferramenta em diferentes contextos, comparando sua eficácia com a RAIDE e mensagens geradas por LLMs.

Palavras-chave: Testes Unitários; Assertion Roulette; Refatoração Automática; JUnit; Test Smells.

ABSTRACT

The present undergraduate thesis aims to propose an alternative solution for refactoring unit tests with the presence of the test smell known as assertion roulette. This bad practice in unit test code occurs when assertions lack descriptive messages that facilitate the debugging process in case of failure, especially when multiple assertions are present in a test method. After identifying the main gaps in the most related reference work, the RAIDE tool, a proof-of-concept tool for refactoring unit tests with assertion roulette was proposed and prototyped. The main differences between this tool and RAIDE are the automatic generation of messages based on the type of assertion, eliminating the need for manual intervention, and support for both JUnit 4 and JUnit 5. Empirical studies conducted with open-source projects demonstrated that the tool reduces, though not completely, the occurrences of assertion roulette and generates messages with reasonable similarity to those created by LLMs (Large Language Models). While the work achieved its objectives, limitations were identified, such as difficulties in identifying assertions in specific scenarios and opportunities for improvement in the tool's code. As future work, it is proposed to overcome these limitations, conduct new empirical experiments with real developers, and validate the tool in different contexts, comparing its effectiveness with RAIDE and messages generated by LLMs.

Keywords: Unit tests; Assertion Roulette; Automated refactoring; JUnit; Test Smells.

LISTA DE ILUSTRAÇÕES

LISTA DE QUADROS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

O presente capítulo aborda os conceitos iniciais do trabalho, introduz e justifica o problema apresentado e apresenta os objetivos e metodologia do presente trabalho.

1.1 CONSIDERAÇÕES INICIAIS

A qualidade de software é uma área de importância em ascensão e o conhecimento sobre testes está se tornando essencial para todos os desenvolvedores de software (AMMANN; OFFUTT, 2016). A medida que os softwares se tornam mais complexos, os programadores precisam aprimorar a forma como desenvolvem (ALMEIDA *et al.*, 2015). Nesse sentido, estima-se que mais de 50% dos esforços na criação de software de grande porte estejam relacionados a garantir qualidade do software, entretanto os produtos finais ainda são inaceitavelmente pobres nesse quesito (OSTERWEIL, 1996).

A forma mais comum para se garantir a qualidade de softwares é por meio do uso de testes (AMMANN; OFFUTT, 2016). Dentre as diversas estratégias presentes na hierarquia de testes, o que mais se aproxima do código em seu nível mais basal são os testes de unidade[1], também chamados de testes unitários, responsáveis por garantir o funcionamento dos menores blocos de um software, as unidades, que costumam ser as funções ou métodos de um programa (PRESSMAN; MAXIM, 2021). Segundo Naik e Tripathy (2011) a ideia principal dos testes de unidade é testar as unidades isoladamente e garantir seu funcionamento, além de atrelar essa responsabilidade da qualidade ao programador, pois é de seu interesse reduzir a quantidade de defeitos ao nível de código.

Normalmente, testes de unidade são automatizados através de *frameworks* genéricos que permitem a criação de métodos de teste que avaliam um ou mais pontos das unidades de código. Eles fazem isso por meio de afirmações (*asserts*), funções que comparam os valores obtidos na execução da unidade com um resultado esperado, informando em execução se elas passaram ou reprovaram. Costuma ser possível executar um número grande destes testes em poucos segundos, estimulando a sua execução constante durante o processo de desenvolvimento do software (REZENDE, 2006).

Tão importante quanto a presença de testes unidade é a existência de bons testes de unidade, que podem ser considerados pre-requisitos para um software de Qualidade em si (BOWES *et al.*, 2017). Para Martin (2009) os testes de unidade são as únicas coisas que permitem um código de produção ser flexível, e sem eles, nenhuma das técnicas de refatoração de código limpo em seu livro fazem sentido. Apesar disso, se não forem também bem escritos, podem ser facilmente perdidos. Para Martin (2009), a qualidade de testes, só pode ser garantida pela legibilidade, que é ainda mais importante para os testes do que para o código produção, e que tende a se tornar uma tarefa desafiadora a medida que o *software* é modificado ou cresce.

[1] Por conta do escopo do trabalho, a partir deste ponto, o termo teste e teste de unidade serão usados ndistinguivelmente.

No âmbito de tentar descrever o que são bons ou maus testes de unidade, foram criados catálogos de más práticas ou anti-padrões, chamadas de *test smells* (VAN DEURSEN *et al.*, 2001; BOWES *et al.*, 2017). Estes descrevem erros frequentes na criação e manutenção de testes de unidade, e surgiram como um compilado de alguns engenheiros de software experientes (VAN DEURSEN *et al.*, 2001), quando eles notaram as peculiaridades da refatoração de testes de unidade em relação a códigos convencionais de *software*. Ao todo eles categorizaram onze *test smells* diferentes, bem como a forma específica para refatorá-los.

Dentre os diferentes tipos de *test smells*, um que se destaca é o *assertion roulette*, pois além de simples de notar e tecnicamente fácil de corrigir, é um dos test smells mais comuns em softwares (SANTANA *et al.*, 2020), e um dos que mais dificulta a sua legibilidade e refatoração (ALJEDAANI *et al.*, 2023). Esse *test smell* surge quando existem múltiplas afirmações no caso de teste sem uma explicação, o parâmetro opcional nos *frameworks* xUnit (apresentado na Seção 2.5) que descreve o motivo da falha da asserção. Por consequência, quando aquele caso de teste falhar, pode não ser trivial identificar o motivo da falha, dificultando assim o processo de correção.

1.2 JUSTIFICATIVA

Apesar da existência de trabalhos a respeito dos *test smells* na literatura, eles costumam se restringir a sua classificação e quando apresentam uma ferramenta, normalmente se limitam a identificação desse anti-padrões, e não em sua refatoração (SANTANA *et al.*, 2020).

Nesse aspecto, segundo Aljedaani *et al.* (2023), até o momento em que havia escrito seu trabalho, existiam aproximadamente vinte e duas ferramentas para detecção de *test smells* no meio acadêmico, mas apenas cinco delas também suportavam a refatoração. Para o autor, todos os desenvolvedores se beneficiariam de uma ferramenta que além de apontar para os *test smells*, também pudessem corrigí-los interativamente (ALJEDAANI *et al.*, 2023).

Ainda conforme análises de Aljedaani *et al.* (2023), existe pouco consenso na classificação dos *test smells* das ferramentas e pouca informação a respeito da eficiência e corretude de sua identificação. Apesar disso, os três *test smells* que mais se repetiram entre as ferramentas foram o *general fixture*, *eager test* e o *assertion roulette* (descritos na Seção 2.7).

Em complemento a essa informação, o trabalho de Aljedaani *et al.* (2023) identificou que a presença de códigos de teste com *eager test* e *assertion roulette* impactam negativamente a capacidade de compreensão e refatoração de programadores e que dentre eles, o *assertion roulette* foi o que mais aumentou o tempo necessário para correção de bugs em códigos de produção inseridos aleatoriamente, evidenciando, nas palavras do autor, a necessidade da existência de ferramentas capazes de detectar e eliminar estes *smells* automaticamente.

Não obstante, o estudo de Kim (2020) relatou que durante a evolução de softwares de grande porte, o número de *test smells* cresce numa proporção maior do que eles são removidos, e que o *assertion roulette* foi o segundo *test smell* mais inserido. O autor do estudo estima que

seja durante a adição de novas fucnionalidades que os *test smells* são adicionados com maior frequência, mas que mesmo nas tentativas posteriores de corrigí-los manualmente, o número total de *test smells* tende a ser maior que antes da funcionalidade ser implementada.

Apesar disso, apenas uma das ferramentas citadas por Aljedaani *et al.* (2023) dava suporte para a refatoração do *assertion roulette*: o RAIDE (SANTANA *et al.*, 2020). Trata-se de um *plugin* para a IDE Eclipse com código aberto e licença GNU, capaz de identificar e refatorar testes de unidade Java JUnit tanto com *assertion roulette* quanto com *eager test* através de uma interface amigável para usuário e de forma semi-automática, que segundo a autora, não é comum dentre as ferramentas atuais. No âmbito do *assertion roulette*, a ferramenta é capaz de identificar todos os casos de teste que contam com mais de uma afirmação sem o parâmetro opcional de explicação, inserindo automaticamente um texto padrão que só serve de lembrete ao programador de que ele deve inserir uma descrição naquela asserção, pois não tem significado de explicação (SANTANA *et al.*, 2020).

Entretanto, apesar dos pontos positivos apresentados pela autora, o fato da ferramenta ser exclusiva para a IDE Eclipse pode se mostrar limitante para alguns programadores. Além disso, inserir um texto padrão nas afirmações com *asserion roulette* pode não solucionar um dos principais benefícios que a autora menciona em seu trabalho, o de automatizar e tornar viável a refatoração de códigos em grandes sistemas (SANTANA *et al.*, 2020). Dessa forma, o trabalho da autora apresenta pontos de melhoria em potencial.

Portanto, o presente trabalho propõe uma ferramenta que visa superar as limitações observadas na literatura. Para tal, a ferramenta visa identificar e refatorar testes de unidade com *assertion roulette* interativamente, mas sem se limitar a uma ferramenta de desenvolvimento. Além disso, a ferramenta visa ser capaz de gerar mensagens informativas baseadas no contexto do teste, reduzindo a necessidade de intervenção manual e assim economizando tempo dos desenvolvedores, mesmo em projetos de grande porte.

1.3 OBJETIVOS

O presente trabalho de conclusão de curso tem como objetivo geral propor uma alternativa de solução para a refatoração de testes de unidade com a presença do *test smell* do tipo *assertion roullete* em relação às soluções presentes na literatura. Para tal, propõe-se a criação de uma ferramenta automatizada capaz de gerar explicações para as asserções baseadas no contexto do código em *frameworks* de teste xUnit. Como prova de conceito, a ferramenta será desenvolvida para o *framework* JUnit 4 e 5 da linguagem Java. Como objetivo específico, pretende-se conduzir estudos empíricos que auxiliarão na compreensão da efetividade da ferramenta proposta, que poderá ser validada por meio de experimentos empíricos. Embora não forneçam uma resposta definitiva, eles servirão como um ponto de partida para trabalhos futuros, oferecendo subsídios para o aprimoramento da ferramenta e sua aplicação em outros contextos e *frameworks*.

1.4 METODOLOGIA

A metodologia do trabalho está estruturada para permitir a criação de uma alternativa validação e avaliação de uma ferramenta capaz de identificar e refatorar testes de unidade que apresentem o *test smell* conhecido como *assertion roulette*.

A metodologia do trabalho está estruturada a fim de atender aos objetivos do trabalho. A pesquisa realizada é aplicada, com abordagem quantitativa e qualitativa, com objetivo explicativo e descritivo. Os procedimentos adotados foram o de pesquisa em trabalhos relacionados, prototipagem de uma ferramenta refatoradora visando suprir as lacunas presentes na literatura e a realização de estudos empíricos.

A fundamentação inicial para a formulação da ferramenta baseou-se em pesquisa na literatura e em trabalhos relacionados com foco na detecção e refatoração do *test smells assertion roulette* em testes de unidade. As lacunas identificadas no principal trabalho relacionado, Santana *et al.* (2020), foram então levados em consideração para a proposição de uma ferramenta prova de conceito.

Quanto ao desenvolvimento da ferramenta, utilizou-se a biblioteca JavaParser para análise e manipulação de código Java. As funcionalidades principais da ferramenta consistem em propor uma maneira de refatorar testes de unidade com presença de *assertion roulette* por meio da geração de mensagens de descrição de asserções.

Após a criação da ferramenta proposta por este trabalho, um primeiro estudo empírico foi realizado em projetos de código aberto para validar os principais aspectos da ferramenta e levantar possíveis pontos de melhoria. Para tal, foram obtidos dados estatísticos a respeito do antes e depois da aplicação da refatoração das asserções nos projetos de código aberto, que foram então discutidos.

Por fim, um segundo estudo empírico foi realizado com o objetivo de identificar a similaridade as mensagens de descrição de asserções geradas pela ferramenta desenvolvida neste trabalho com aquelas criadas pelas LLMs ChatGPT e Gemini por meio do algoritmo de similaridade Jaro. Esses resultados podem servir de base em validações futuras realizadas por meio de experimentos empíricos.

2 FUNDAMENTAÇÃO TEÓRICA

A fundamentação teórica apresentada neste capítulo aborda os conceitos essenciais que sustentam este trabalho. Inicialmente, discute-se a importância da qualidade de software no desenvolvimento e manutenção de sistemas confiáveis, seguida por uma introdução aos testes de software como principal método de avaliação e validação durante o desenvolvimento. Em seguida, são apresentados os diferentes tipos de teste de software, com ênfase nos testes de unidade, detalhando sua estrutura, objetivos e aplicação prática. Após isso explora-se o *framework* xUnit, destacando suas funcionalidades e diferenças entre as versões JUnit 4 e JUnit 5, bem como o conceito de *test smells*, com foco especial no *Assertion Roulette*, um problema central abordado ao longo deste trabalho. Por fim, o tema da refatoração de código é apresentado.

2.1 QUALIDADE DE SOFTWARE

Atualmente, os softwares estão presentes nas mais variadas aplicações, desde as mais convencionais como a internet, aplicativos de celular, aplicações financeiras e software embarcado em eletrodomésticos, até aplicações mais críticas e complexas como software de aviões, sistemas de tráfego aéreo ou sistemas de naves espaciais (AMMANN; OFFUTT, 2016). Nesse sentido, de acordo com Homès (2024), o impacto da falha do software que usamos no dia a dia pode ser devastador.

Mais do que isso, em meio à competição, qualidade se tornou um fator crucial de sucesso, com empresas reconhecendo sua dependência em software e investindo cada vez mais em habilidades de desenvolvimento (SPILLNER; LINZ, 2021). Gradualmente, a qualidade de software está se tornando essencial em todos os negócios e um conhecimento necessário por todos seus desenvolvedores (AMMANN; OFFUTT, 2016).

2.2 TESTE DE SOFTWARE

Apesar dos diversos fatores incluídos na garantia de qualidade de software, os testes são a forma primária de avaliação de software durante desenvolvimento (AMMANN; OFFUTT, 2016). Segundo Agarwal, Gupta e Tayal (2009), testes de são o conjunto de atividades que podem ser planejadas previamente e executadas sistematicamente, o que também se aplica ao software, que faz uso de modelos (do inglês *template*) e casos de teste, que por sua vez podem ser vistos como pares de entradas de um software, ou parte dele, e sua saída esperada (NAIK; TRIPATHY, 2011). Em outra definição, segundo Myers (2006), teste de software se trata de executar um programa com a intenção de encontrar erros.

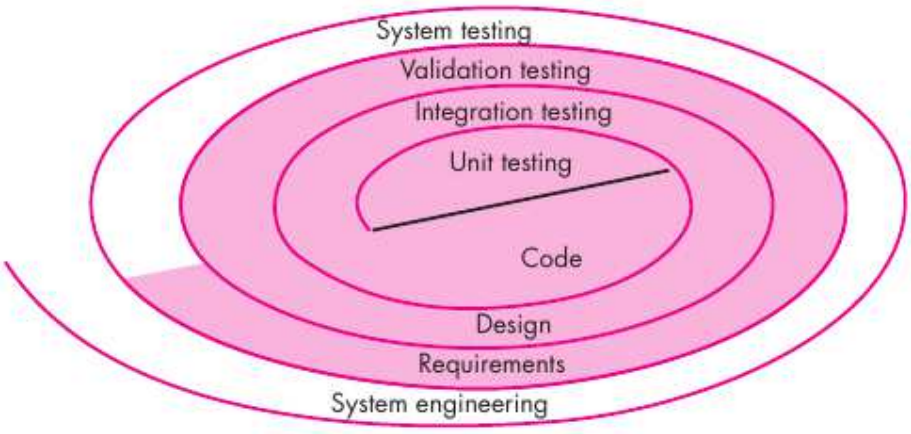
De acordo com o autor, encontrar erros pode parecer algo destrutivo quando o objetivo final é aumentar a qualidade de software, mas agrega muito mais valor a ele, posto que erros encontrados podem ser corrigidos. Ammann e Offutt (2016) e Sommerville (2011) contribuem para essa linha de raciocínio ao mencionar que é impossível tornar um software completamente

à prova de falhas ou garantir que ele vai se comportar de acordo em qualquer situação, mas nem por isso deve-se ignorar a importância dos testes.

2.3 TIPOS DE TESTE DE SOFTWARE

Os testes podem ser classificados de acordo com o seu nível de abstração, partindo de níveis mais baixos e próximos do código-fonte, até níveis mais altos que se preocupam com o funcionamento completo do software (PRESSMAN; MAXIM, 2021). A Figura 1 ilustra os níveis de abstração em espiral, com os níveis mais altos de abstração representados mais externamente.

Figura 1 – Níveis de abstração de testes de software



Fonte: (PRESSMAN; MAXIM, 2021)

De acordo com Naik e Tripathy (2011), os quatro tipos mais comuns de teste de software são os testes de unidade, que testam o funcionamento de funções, métodos ou classes isoladamente; testes de integração, que testam os componentes e a interação entre eles, com foco no design e construção da arquitetura do software; teste de sistema, que testam o software como um todo e englobam sua funcionalidade, segurança, robustes, carga, entre outros; e por fim os testes de aceitação, que são realizados pelo cliente e não focam em identificar erros, mas sim em validar se suas expectativas foram atendidas.

Além desses, a literatura apresenta muitos outros tipos de teste e categorizações. Um exemplo são os testes de validação, em que o funcionamento do teste é comparado com os requisitos do software para checar se eles são atendidos (PRESSMAN; MAXIM, 2021), ou os testes de regressão, executados sempre que o software é modificado, a fim de garantir que as alterações de software não inseriram novas falhas nas porções não tocadas (NAIK; TRIPATHY, 2011).

2.4 TESTE DE UNIDADE

Segundo Sommerville (2011), os testes de unidade são o tipo de teste com menor granularidade, o que significa que ele cobre o sistema em seu nível mais basal, uma unidade. Para (AMMANN; OFFUTT, 2016), uma unidade se trata de uma função, procedimento, subrotina ou método, a depender da nomenclatura, conforme a linguagem ou paradigma de programação.

Dessa forma, um teste de unidade avalia o nível mais baixo de um sistema, a implementação de seus menores blocos "chamáveis", por consequência, costuma ser de responsabilidade de programadores que implementaram a unidade e não de testadores, pois eles têm mais familiaridade com o seu funcionamento interno (NAIK; TRIPATHY, 2011). Apesar disso, testadores podem ter um papel fundamental em encontrar casos de teste que podem ter sido ignorados pelo desenvolvedor.

Para Naik e Tripathy (2011), o conceito principal dos testes de unidade é o de levar qualidade ao nível mais basal de uma organização e empoderar o programador, que se torna responsável pela qualidade do software, de forma que é de seu interesse tomar ações preventivas para reduzir defeitos.

Em termos de escopo, (PRESSMAN; MAXIM, 2021) ressalta que os testes de unidade devem focar no processamento interno da unidade, seus fluxos paralelos, estruturas de dados e afins, isoladamente de outras unidades. Ele também enfatiza a importância de testar as condições de contorno (do inglês, *boundary conditions*) para garantir o funcionamento correto sobre todo o domínio, incluindo seus limites. Naik e Tripathy (2011) mencionam que a importância de testar as unidades isoladamente é que, além de tirar a dependência com outros módulos, é preciso garantir que elas estejam funcionando satisfatoriamente antes que possam ser usadas por outras unidades ou testadas de forma integrada.

Normalmente, testes de unidade são automatizados através de *frameworks* genéricos que permitem a criação de métodos de teste que avaliam um ou mais pontos das unidades de código. Eles fazem isso por meio de *asserts* (também chamados de asserções ou afirmações)[2], funções que comparam os valores obtidos na execução da unidade com um resultado esperado, informando em tempo de execução se elas passaram ou reprovaram. Costuma ser possível executar um número grande destes testes em poucos segundos, estimulando a sua execução constante durante o processo de desenvolvimento do software (SOMMERVILLE, 2011).

2.4.1 Partes de um Teste de Unidade

Um teste de unidade geralmente é composto por três partes principais (DELPLANQUE *et al.*, 2019) (ALÉGROTH; GONZALEZ-HUERTA, 2017):

1. **Configuração:** parte que corresponde a preparação do ambiente para a execução do teste, que pode incluir a criação de objetos, inicialização de variáveis e preparação de dados.

[2] A partir deste ponto, será utilizado o termo asserção para se referir a este conceito.

2. **Estímulo:** trata-se da ação executada no componente que está sendo testada, como a chamada de um método que se deseja testar.
3. **Asserções**: parte do teste que verifica se o comportamento do componente sob teste corresponde ao resultado esperado. Elas comparam o resultado da execução do estímulo com um valor ou estado pré-determinado, utilizando métodos como *assertEquals*, *assertTrue* ou *assertFalse*.

A Figura 2 apresenta um método de teste de unidade com cada uma das partes comentadas. No código, o método *testAddition* (linha dois), é um teste de unidade, o que é indicado pela anotação *@test* (linha um), responsável por verificar a funcionalidade de adição da classe *Calculator*. A linha quatro corresponde à etapa de configuração, pois é criado um objeto *minhaCalculadora*, necessário para a execução dos passos subsequentes do teste. Em seguida ocorre o estímulo (linha sete), onde o método *add* é chamado, e o resultado é armazenado na variável *resultado*. Por fim, a linha dez realiza a etapa de asserção, onde o valor de *resultado*, obtido em tempo de execução, é comparado com o valor esperado, '5', usando o método *assertEquals*. Caso ambos os valores forem iguais, o teste passa, do contrário, ele falha.

Figura 2 – Exemplo de teste de unidade com partes comentadas

```
1     @Test
2     void testAddition() {
3        // configuração
4        Calculator minhaCalculadora = new Calculator();
5
6        // estímulo
7        int resultado = minhaCalculadora.add(2, 3);
8
9        // asserção
10       assertEquals(5, resultado);
11    }
```

Fonte: (Autoria própria, 2024)

2.5 FRAMEWORK XUNIT

O xUnit é uma família de *frameworks* de teste de unidade automatizado com características em comum e dos quais praticamente todas as linguagens de programação mais populares tem uma implementação (MESZAROS, 2007). Segundo Meszaros (2007), é vantajoso que os testes de unidade sejam escritos na mesma linguagem que o código que ela pretende testar, posto que isso facilita as chamadas necessárias e permite uma curva de aprendizagem menor.

A família xUnit surgiu a partir do *framework* Junit, responsável por popularizar a metodologia *TDD* (Test Driven Development, Desenvolvimento Orientado a Testes), (NAIK; TRIPATHY, 2011). Este *framework* foi criado por Kent Beck e Erich Gamma, autores que contribuíram com trabalhos a respeito de padrões de projeto e outros tópicos da programação orientada a objetos (GAMMA, 2009).

As características dos *frameworks* xUnit são a de permitir criação de testes através de métodos em classes de teste, especificar o resultado esperado de um teste através da chamada de métodos de *assert*, agregar um conjunto de testes em uma *suíte* que pode ser chamada múltiplas vezes em uma única operação e fornecer relatórios com os detalhes da execução dos testes de unidade (MESZAROS, 2007).

2.5.1 Diferenças entre versões do JUnit

O JUnit 5 é uma versão mais recente em comparação ao JUnit 4, introduzindo alterações na estrutura do framework e no uso de anotações e métodos de teste **??**. Uma diferença específica entre essas versões é a posição do parâmetro opcional de descrição nas asserções. No JUnit 4, o parâmetro de descrição era apresentado como o primeiro da lista de argumentos das asserções. Já no JUnit 5, essa posição foi alterada, e o parâmetro de descrição passou a ser sempre o último da lista de argumentos. Além disso, o JUnit 5 trouxe melhorias gerais de flexibilidade e organização, mas sem alterar a lógica essencial das asserções abordadas no presente estudo.

2.6 LEGIBILIDADE

No contexto das linguagens de programação, legibilidade é, segundo Sebesta (2018), um critério que descreve a facilidade com a qual um programa pode ser lido e compreendido. O autor menciona que a legibilidade está diretamente relacionada com a facilidade de manutenção de softwares e, portanto, deve ser considerado parte de seu domínio.

Para Martin (2009), a legibilidade é o fator mais importante em testes de unidade e, em muitos contextos, ainda mais importante do que a legibilidade dos códigos de produção. De acordo com ele, um teste deve ser capaz de dizer muito com um conjunto mínimo de expressões.

2.7 TEST SMELLS

Test smells[3] são anti-padrões ou más práticas presentes em testes de unidade que surgiram como análogos aos *code smells* (VAN DEURSEN *et al.*, 2001). Trata-se de indícios de problemas nos códigos de teste que afetam sua qualidade e legibilidade, tornando-os mais difíceis de entender, manter e evoluir.

Dentre os diferentes classificação de *test smell* presentes na literatura (ALJEDAANI *et al.*, 2021), destacam-se os seguintes exemplos:

1. **Assertion Roulette (AR)**[4], que ocorre quando um método de teste possui múltiplas asserções sem mensagens de descrição, parâmetro opcional presente nos *frameworks* xUnit. A ausência da descrição em asserções dificulta a identificação da causa da

[3] Daqui em diante, o termo *test smell* e *smell* serão usados indistinguivelmente.
[4] Daqui em diante, o termo *assertion roulette* e AR serão usados indistinguivelmente.

falha do teste (SANTANA *et al.*, 2020). É considerado um dos *test smells* mais comuns e impactantes (ALJEDAANI *et al.*, 2023).

2. **Duplicate Assert (DA)**, que ocorre quando um método de teste possui a mesma afirmação repetidas vezes (SANTANA *et al.*, 2020).
3. **General Fixture (GF)**, que ocorre quando o método de teste não utiliza tudo que foi definido na configuração do teste (ALJEDAANI *et al.*, 2021).
4. **Eaget Test (ET)**, que ocorre quando um método de teste verifica múltiplas funcionalidades do código de produção invocando diversos métodos. Torna-se difícil entender o real propósito do teste e aumenta o acoplamento entre o código de teste e o código de produção, impactando negativamente a manutenção (ALJEDAANI *et al.*, 2023).
5. **Test Code Duplication (CD)**, quando há a presença de duplicação de código em testes (VAN DEURSEN *et al.*, 2001).

Dentre os *test smells* apresentados, o *Assertion Roulette* é o que apresenta maior relevância para o presente trabalho. A Figura 3 ilustra um exemplo de um teste de unidade com a presença deste *smell*. No código, observa-se que nenhuma das asserções (linha três à linha doze) apresenta uma string contendo uma explicação como parâmetro opcional (que corresponderia a um terceiro parâmetro nos métodos de *assertEquals*). Ressalta-se que, em caso de falha de uma destas asserções, o motivo exato da falha pode ser claro para o programador.

Figura 3 – Exemplo de teste de unidade com a presença do *test smell assertion roulette*

```
1     @Test
2     void calculadoraTest() {
3        assertEquals(5, calculator.add(2, 3));
4        assertEquals(-8, calculator.add(-4, -4));
5        assertEquals(10, calculator.multiply(2, 5));
6        assertEquals(calculator.multiply(-2, -2), 4);
7        assertEquals(calculator.add(2, 2), calculator.add(3, 6));
8        assertEquals(2, calculator.subtract(5, 3));
9        assertEquals(3, calculator.subtract(8, 5));
10       assertEquals(1, calculator.subtract(10, 9));
11       assertEquals(2, calculator.divide(6, 3));
12       assertEquals(1, calculator.divide(5, 0)));
13    }
```

Fonte: (Autoria própria, 2024)

2.8 REFATORAÇÃO DE CÓDIGO

Refatoração é uma abordagem para alteração do *design* de um código sem alterar seu comportamento (MESZAROS, 2007). Os principais benefícios da refatoração é o de tornar os códigos mais legíveis e assim tornar mais fácil o processo de identificação de bugs e de desenvolvimento (BAQAIS; ALSHAYEB, 2020). Apesar disso, o processo de refatoração precisa

seguir uma série de passos que se não forem executados corretamente podem resultar em refatorações incorretas ou problemas piores (BAQAIS; ALSHAYEB, 2020). Por conta disso, é comum na literatura a presença de ferramentas automatizadas para a realização dessa refatoração e costumam-se dividir entre algoritmos semi-automatizadas, que permitem a revisão do código antes da refatoração ou totalmente automatizada que aplica a refatoração sem questionar (BAQAIS; ALSHAYEB, 2020). No âmbito dos testes de unidade, a refatoração por meio de ferramentas automatizadas é ainda mais relevante, pois não há testes de unidade para os próprios testes de unidade, ou seja, não há garantias de que algo não foi quebrado (MESZAROS, 2007).

3 TRABALHOS RELACIONADOS

No contexto da refatoração do *assertion roulette*, um dos *test smells* mais mencionados na literatura e que mais impactam negativamente o processo de depuração em caso de erros em códigos de produção (ALJEDAANI *et al.*, 2023), destaca-se o trabalho de Santana *et al.* (2020), que apresenta a ferramenta RAIDE, capaz de reduzir o tempo total necessário para a refatoração de testes unitários com a presença desse *test smell* (SANTANA *et al.*, 2022). Não obstante, até o momento da pesquisa de Aljedaani *et al.* (2021), a RAIDE era a única ferramenta que realizava a refatoração do *assertion roulette*. Por sua relevância e por ser um trabalho recente, Santana *et al.* (2020) constitui a referência central para o presente trabalho.

A ferramenta RAIDE (SANTANA *et al.*, 2020) é um *plugin* desenvolvido para a IDE Eclipse, que visa identificar e refatorar de forma semi-automatizada *test smells* presentes em testes de unidade Java com JUnit 4. De forma mais específica, a ferramenta suporta os *test smells* de *assertion roulette* e *duplicate assert*. Para a refatoração do *assertion roulette*, a ferramenta adiciona um texto padrão de marcador que serve de lembrete para o programador adicionar uma descrição explicativa à asserção. Já para o *duplicate assert*, ao identificar a presença de duas ou mais asserções com os mesmos parâmetros em um método de teste, a RAIDE cria novos métodos para separar as asserções duplicadas, preservando o método original como um comentário para referência do desenvolvedor.

Apesar disso, RAIDE apresenta algumas limitações, destacando-se as relacionadas à refatoração do *assertion roulette*, por conta do escopo do trabalho atual. Primeiramente, o uso da ferramenta está restrito ao JUnit na versão 4, limitando sua aplicabilidade a projetos que utilizam versões mais recentes do *framework*, como o JUnit 5. Além disso, a RAIDE opera exclusivamente na IDE Eclipse, restringindo sua adoção por desenvolvedores que utilizam outras IDEs ou editores de código. Por fim, o processo de refatoração não é totalmente automatizado, uma vez que o programador é responsável por inserir manualmente as descrições das asserções no lugar do texto de marcador.

Dessa forma, a ferramenta proposta por este trabalho visa preencher essas lacunas, ampliando as funcionalidades apresentadas na ferramenta RAIDE. Diferentemente do RAIDE, a ferramenta pretende oferecer suporte tanto ao JUnit 4 quanto ao JUnit 5, aumentando sua aplicabilidade a projetos mais recentes. Além disso, a proposta prevê sua execução via linha de comando, o que elimina a dependência de uma única IDE, proporcionando maior flexibilidade no ambiente de desenvolvimento. Por fim, a ferramenta visa realizar a refatoração do *assertion roulette* de forma totalmente automática, por meio da geração de mensagens de descrição baseadas no tipo de asserção e nos parâmetros associados, eliminando a necessidade de intervenção manual por parte do programador.

4 APRESENTAÇÃO DA FERRAMENTA PROPOSTA

O presente capítulo descreve a ferramenta de prova de conceito proposta pelo trabalho, apresenta sua arquitetura e funcionamento, a proposta das mensagens de asserção, limitações da ferramenta, exemplo de funcionamento e os conceitos da ciência da computação aplicados para sua contrução.

4.1 DESCRIÇÃO DA FERRAMENTA

A ferramenta proposta neste trabalho é projetada para identificar e refatorar o *test smell* do tipo *assertion roulette* em arquivos de teste de unidade que utilizam o JUnit nas versões 4 e 5. O objetivo principal da ferramenta é adicionar descrições às asserções, visando melhorar a legibilidade e o entendimento dos testes, especialmente em cenários de falha.

A escolha por esse framework e esta linguagem se deve ao fato de grande parte das ferramentas presentes na literatura de *test smells* também utilizarem dela (ALJEDAANI *et al.*, 2023). Além disso, enquanto uma prova de conceito, a escolha de um *framework* xUnit torna a lógica da implementação possivelmente portável para outros *frameworks* xUnit presentes em outras linguagens.

Nesse sentido, a ferramenta desenvolvida oferece suporte a todos os métodos de asserção disponíveis nas versões 4 e 5 do JUnit (JUNIT, 2024a) (JUNIT, 2024b), conforme detalhado no Quadro 1. O quadro também inclui a quantidade de parâmetros obrigatórios de cada método de asserção, ilustrando a variação entre eles. Por exemplo, métodos como *assertEquals* e *assertArrayEquals* exigem dois parâmetros obrigatórios para realizar as comparações esperadas. Já o método *assertFalse* requer apenas um parâmetro, que é a expressão booleana a ser avaliada. Em contraste, o método *fail* não possui parâmetros obrigatórios, permitindo o uso direto para forçar uma falha nos testes de unidade.

A ferramenta parte do princípio de que a falta do parâmetro de descrição em testes de unidade com mais de uma asserção é prejudicial para a legibilidade e manutenibilidade do código de teste (VAN DEURSEN *et al.*, 2001) (ALJEDAANI *et al.*, 2023) (SANTANA *et al.*, 2020). Em contrapartida, a presença do parâmetro, além de melhorar a legibilidade da afirmação em caso de falha, pode servir como documentação.

Em termos do funcionamento, a ferramenta pode ser executada pela linha de comando ou diretamente via IDE pela execução do arquivo principal do projeto, e está preparada para receber como argumento o caminho para um diretório ou projeto contendo testes de unidade JUnit.

4.2 ARQUITETURA

A ferramenta foi construída com Java 11 e faz uso da biblioteca Java Parser em suas funcionalidades de análise sintática dos arquivos de teste Java. Java Parser consiste em anali-

Quadro 1 – Lista de métodos de asserções suportados pela ferramenta

Método de Asserção	Versão do JUnit	Parâmetros Obrigatórios
assertArrayEquals	JUnit 4, JUnit 5	2
assertEquals	JUnit 4, JUnit 5	2
assertFalse	JUnit 4, JUnit 5	1
assertNotEquals	JUnit 4, JUnit 5	2
assertNotNull	JUnit 4, JUnit 5	1
assertNotSame	JUnit 4, JUnit 5	2
assertNull	JUnit 4, JUnit 5	1
assertSame	JUnit 4, JUnit 5	2
assertThat	JUnit 4	2
assertThrows	JUnit 4, JUnit 5	2
assertTrue	JUnit 4, JUnit 5	1
fail	JUnit 4, JUnit 5	0
assertAll	JUnit 5	1
assertDoesNotThrow	JUnit 5	1
assertIterableEquals	JUnit 5	2
assertLinesMatch	JUnit 5	2
assertInstanceOf	JUnit 5	2
assertThrowsExactly	JUnit 5	2
assertTimeout	JUnit 5	2
assertTimeoutPreemptively	JUnit 5	2

Fonte: autoria própria, 2024.

sador sintático que permite a manipulação de código Java como uma AST (*Abstract Syntax Tree*, Árvore Sintática Abstrata) (SMITH; VAN BRUGGEN; TOMASSETTI, 2017). A AST fornece uma representação hierárquica do código-fonte e disponibiliza métodos para a análise e manipulação dele de forma conveniente.

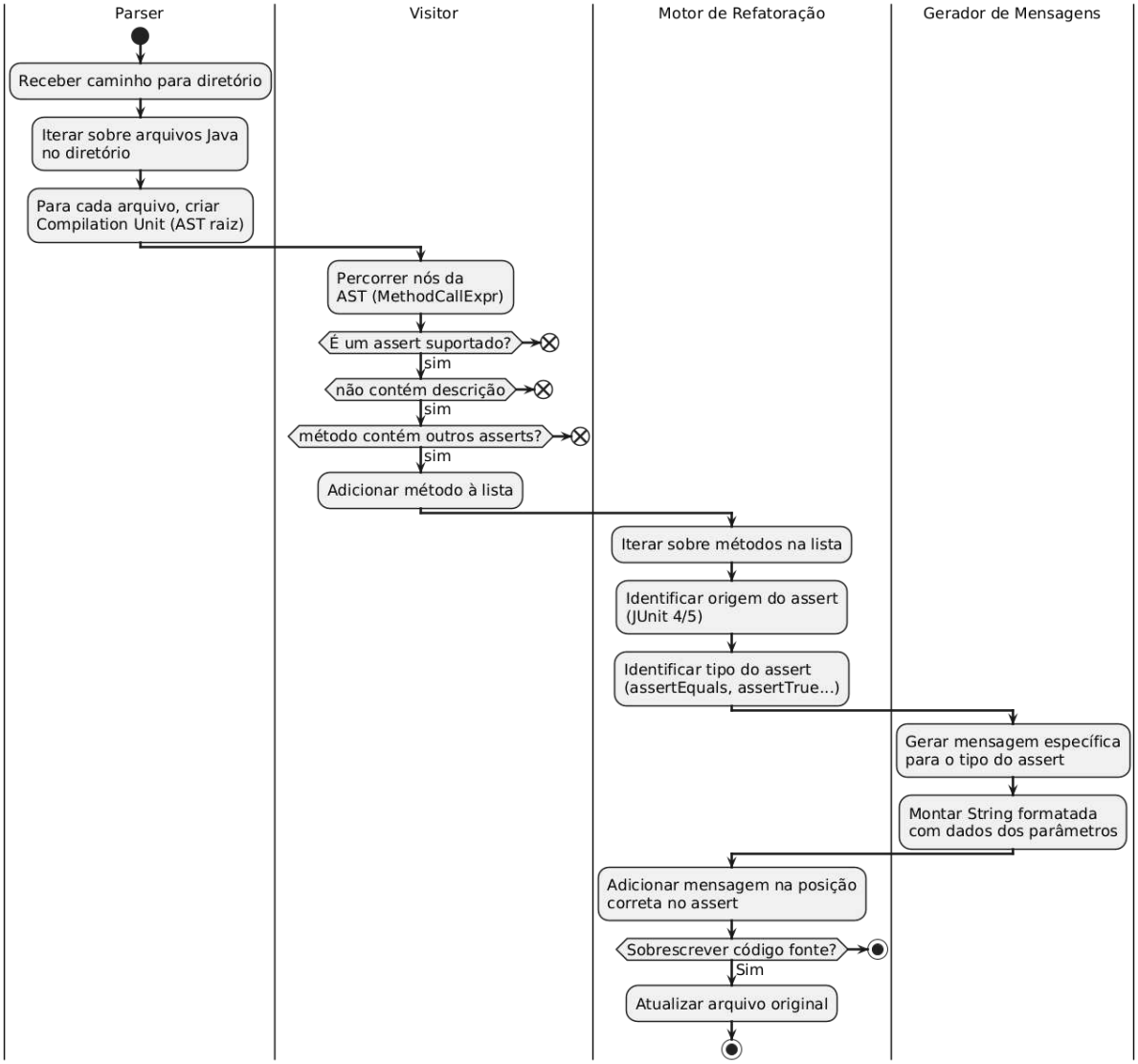
A ferramenta é composta por quatro componentes principais: *parser* ou analisador sintático, o *visitor*, o motor de refatoração e o gerador de mensagens contextuais. O código-fonte da ferramenta pode ser acessado no repositório do Github[5] disponível em https://github.com/vitorueno/tcc. Os passos executados pela ferramenta no processo de análise e refatoração das asserções estão descritos no diagrama de atividade da Figura 4, que está separado conforme o componente.

De forma geral, o *parser* realiza a análise sintática dos códigos Java por meio da biblioteca Java Parser. Dessa forma, o analisador recebe como entrada o caminho para um diretório contendo arquivos Java e para cada um, cria uma *Compilation Unit*, raiz da AST, que representa o módulo como um todo. A partir dela, é possível atrelar *visitors*, um componente baseado no padrão de projeto *visitor* que permite a navegação por tipos de nós específicos da árvore de forma facilitada (SMITH; VAN BRUGGEN; TOMASSETTI, 2017).

Na ferramenta, o *visitor* percorre cada chamada de método (*MethodCallExpr*) a fim de

[5] https://github.com/

Figura 4 – Diagrama de atividades demonstrando funcionamento da ferramenta por componente



Fonte: (autoria própria, 2024)

identificar se são asserções e obter informações contextuais, como a quantidade de parâmetros, se possui ou não uma descrição e se além dele há outras asserções no mesmo método de teste. Com essas informações é possível realizar o levantamento a respeito da presença ou não do *test smell assertion roulette*. Os métodos que se enquadram na categoria de asserções suportados pela ferramenta e com a presença *assertion roulette* são então adicionados em uma lista.

Essa lista é percorrida pelo motor de refatoração, que aplica regras de transformação com base na origem da asserção (JUnit 4, JUnit 5) e no tipo de asserção (*assertEquals*, *assertTrue*...), posto que a quantidade de parâmetros varia conforme o tipo. Asserções como o *fail* não recebem nenhum parâmetro obrigatório, enquanto o *assertTrue* recebe um e o *assertEquals* recebe dois, apenas para exemplificar. Após a identificação do tipo, o gerador de mensagens é chamado.

O gerador de mensagens realiza a criação das *strings* de mensagem de descrição das asserções inserindo os dados de seus parâmetros na mensagem através da montagem de *strings*

formatadas com auxílio dos recursos do Java Parser. Cada tipo de asserção recebe uma mensagem específica, conforme descrito na Seção 4.3.

Por fim, o motor de refatoração finaliza a refatoração através da adição da mensagem gerada na posição correta da asserção, conforme a versão do JUnit, posto que no JUnit 4 o parâmetro de descrição é o primeiro parâmetro das asserções, enquanto no JUnit 5 é o último. Finalmente, o *Compilation Unit* é atualizado e seu conteúdo sobrescreve opcionalmente o código-fonte de origem.

Embora tenha sido apresentada uma divisão clara das responsabilidades da ferramenta, ressalta-se que o código-fonte não segue rigidamente essa separação. O foco principal durante o desenvolvimento foi garantir a funcionalidade e a implementação de recursos essenciais, o que resultou em uma estrutura de código que, embora funcional, ainda não reflete totalmente uma organização ideal e o seguimento de boas práticas e padrões de projeto. Devido a essa abordagem priorizada pela funcionalidade, o código está pendente de refatorações para garantir uma separação de responsabilidades mais clara e facilitar a manutenção e evolução futura da ferramenta.

4.3 PROPOSTA DE MENSAGENS

A proposta de mensagens foi definida com base no tipo de asserção utilizado e nos dados fornecidos como entrada. Os critério para a criação da proposta inicial de mensagens de asserção foram definidos visando clareza, pois as mensagens devem ser diretas, contextualização, pois devem conter informações relevantes para depuração e padronização das mensagens, para garantir uniformidade de experiência. A lista contento a proposta de mensagens da ferramenta está descrita na Tabela 2.

Nota-se que a maioria das mensagens segue o formato "era esperado x, mas o valor real foi y". Esse formato de mensagem destaca as discrepâncias num caso de depuração e mantém uma consistência para facilitar a interpretação das asserções, tudo isso em uma *string* curta. Apesar disso, no caso do *assertAll* e do *fail*, por limitações técnicas e de tempo, optou-se pela criação de uma mensagem fixa. Idealmente, uma mensagem para o *fail* deveria representar o inverso do que o teste de unidade deveria fazer, enquanto para o *assertAll*, deveria englobar todas as asserções contidas nele. Em ambos os casos, grandes reestruturações precisariam ser realizadas nos componentes da ferramenta, portanto optou-se pelo uso de uma mensagem fixa na versão de prova de conceito da ferramenta.

Ressalta-se, no entanto, que essa proposta de mensagens ainda está pendente de validação com usuários reais. A realização de avaliações empíricos com desenvolvedores seria fundamental para avaliar a eficácia das mensagens geradas e identificar possíveis melhorias. O *feedback* obtido nesse processo poderia servir como base para ajustes na ferramenta, garantindo que as mensagens atendam às necessidades práticas dos usuários, sejam intuitivas, e realmente contribuam para uma depuração mais eficiente. Além disso, esses testes poderiam revelar opor-

tunidades de refinamento, como a inclusão de informações adicionais ou a personalização das mensagens para diferentes contextos de uso.

Quadro 2 – Proposta de mensagem da ferramenta para cada método de asserção.

Método de Asserção	Formato de mensagem
assertEquals	Era esperado valores iguais, mas <%s> é diferente de <%s>.
assertArrayEquals	Era esperado arrays iguais, mas <%s> é diferente de <%s>.
assertFalse	Era esperado falso, mas <%s> é verdadeiro.
assertNotEquals	Era esperado valores diferentes, mas <%s> é igual a <%s>.
assertNotNull	Era esperado não nulo, mas <%s> é nulo.
assertNotSame	Era esperado objetos diferentes, mas <%s> é igual a <%s>.
assertNull	Era esperado nulo, mas <%s> não é nulo.
assertSame	Era esperado objetos iguais, mas <%s> é diferente de <%s>.
assertThat	Era esperado que <%s> satisfizesse a condição <%s>, mas não a satisfez.
assertThrows	Era esperado que a exceção <%s> fosse lançada, mas não foi.
assertTrue	Era esperado verdadeiro, mas <%s> é falso.
fail	A execução falhou por uma causa deliberada.
assertAll	Era esperado que todos os testes deste grupo passassem, mas pelo menos um falhou.
assertDoesNotThrow	Era esperado que nenhuma exceção fosse lançada, mas <%s> foi lançada.
assertIterableEquals	Era esperado que as listas fossem iguais, mas <%s> não é igual a <%s>.
assertLinesMatch	Era esperado que as linhas combinassem, mas <%s> não corresponde a <%s>.
assertInstanceOf	Era esperado que <%s> fosse uma instância de <%s>, mas não é.
assertThrowsExactly	Era esperado que a exceção <%s> fosse lançada, mas não foi.
assertTimeout	Era esperado que o tempo de execução <%s> não fosse ultrapassado, mas foi.

Fonte: autoria própria, 2024.

4.4 LIMITAÇÕES

Apesar de atingir o objetivo de identificar e corrigir o *test smell* de assertion roulette em testes de unidade JUnit 4 e JUnit 5 (Conforme apresentado no Capítulo 5), a ferramenta apresenta algumas limitações que podem impactar seu desempenho e aplicabilidade em certos cenários. Tais limitações estão descritas a seguir.

1. A ferramenta é restrita a linguagem Java com testes em JUnit 4 e JUnit 5, portanto *frameworks* alternativos como AssertJ ou Hamcrest não são suportados e, em casos bem específicos, podem trazer comportamentos inesperados como falsos positivos na etapa de análise ou refatorações indevidas que podem gerar erros de compilação. Apesar disso, a ferramenta valida a origem das asserções e na grande maioria das vezes ignora asserções de origens indeterminadas ou que não sigam o padrão do JUnit. O suporte a outros *frameworks*, bem como a implementação da ferramenta para outras linguagens é uma possibilidade que segue aberto.
2. Mesmo em asserções padrões do JUnit, a identificação da origem das asserções pode falhar quando por algum motivo as importações não seguem o formato padrão do JUnit 4 ou JUnit 5. Em alguns casos importações não estáticos podem não ser interpretados incorretamente, pois os métodos apresentam prefixos desconsiderados pela ferramenta. Isso pôde ser observado em alguns projetos, como o *ClassGraph*, *SQLite JDBC* e *SpringDoc OpenAI*, em que não foi possível efetuar nenhuma refatoração (conforme apresentado no Capítulo 5).

3. Asserções personalizadas ou que fogem da lista de asserções suportados são desconsiderados pela ferramenta. Não obstante, a refatoração não ocorre quando as asserções estão dentro de estruturas de controle.
4. A ferramenta gera mensagens apenas em português, independentemente da linguagem usada na documentação do projeto.
5. Ainda que a ferramenta cumpra seu objetivo principal, o código-fonte apresenta oportunidades de melhoria no que diz respeito à organização, clareza e aderência de padrões de projeto. Durante o desenvolvimento o foco foi a priorização das funcionalidades e a viabilidade da ferramenta como prova de conceito. A criação de uma nova versão do código ou uma refatoração seriam necessárias para permitir uma melhor manutenibilidade do código.
6. As mensagens geradas pela ferramenta levam apenas em consideração o tipo de asserção e insere os dados específicos dos parâmetros da asserção. Dessa forma, para algumas situações as mensagens podem ser genéricas e não captam bem o contexto no qual a asserção está inserido. Para alguns tipos de asserção, como o *assertAll* no qual um número arbitrário de parâmetros podem ser passados, ou o *fail*, em que todo o contexto do método de teste de unidade deve ser considerado, a tarefa de gerar uma mensagem é mais complexa e optou-se pela criação de uma mensagem genérica. De qualquer forma, a ferramenta parte do princípio de que mesmo uma mensagem genérica é melhor do que a ausência de uma, tendo em vista que ela pode facilitar o processo de depuração em caso de falha de um teste de unidade.
7. A ferramenta está pendente de validações com usuários reais, portanto experimentos precisam ser conduzidos para que pontos como a validade das mensagens geradas e a usabilidade da ferramenta sejam avaliados.

Dessa forma, as limitações representam pontos de possível melhoria do projeto e poderão servir de base na elaboração dos trabalhos futuros. Em especial, os problemas relacionados à identificação da origem das asserções, blocos de controle, refatoração do código-fonte e realização de novos testes empíricos serão as prioridades.

4.5 EXEMPLO DE FUNCIONAMENTO

Para demonstrar o funcionamento da ferramenta, foi criado um projeto fictício contendo testes de unidade em JUnit. Esse projeto tem como objetivo simular alguns cenários comuns presentes em projetos reais e nos quais a ferramenta deveria suportar a refatoração. A aplicação se trata de uma classe simples de calculadora que efetua operações de soma, subtração, multiplicação e divisão. Assim, foram criados testes de unidade para essa calculadora. O repositório do projeto fictício está disponível no repositório do Github[6] https://github.com/vitorueno/

[6] https://docs.github.com/en/get-started/start-your-journey/about-github-and-git

fakeProject/tree/main/src.

A Figura 5 apresenta a classe de teste de unidade antes da refatoração. O método de teste de unidade *testCalculatorOperations* avalia o funcionamento de operações de adição, subtração, multiplicação e divisão da classe de calculadora. No teste, são usados as asserções *assertEquals*, que avalia se o valor real obtido é igual ao valor esperado (linhas 7, 10, 16), *assertTrue* (linhas 8 e 14) e *assertFalse* (linha 11), que avaliam se uma expressão lógica é verdadeira ou falsa, respectivamente, *assertNotEquals* (linha 13), que verifica se os dois valores são diferentes e o *assertThrows* (linha 17), que verifica se uma exceção específica é lançada.

Observa-se que de acordo com as definições apresentadas na Seção 2.7, trata-se de um teste de unidade com a presença do *test smell assertion roulette*, pois existe mais de uma asserção no método de teste de unidade e não há a presença do parâmetro de descrição. Não obstante, o método também pode ser caracterizado como *eager test*, pois verifica múltiplas funcionalidades do código de produção (todas as quatro operações da calculadora) em um único método de teste de unidade. Ressalta-se, no entanto, que a presença de *eager test* não influencia no funcionamento da ferramenta.

Figura 5 – Código de exemplo antes da refatoração

```
1 class CalculatorTest {
2
3     private final Calculator calculator = new Calculator();
4
5     @Test
6     void testCalculatorOperations() {
7         assertEquals(5, calculator.add(2, 3));
8         assertTrue(calculator.add(1, -1) == 0);
9
10        assertEquals(-1, calculator.subtract(2, 3));
11        assertFalse(calculator.subtract(5, 3) < 0);
12
13        assertNotEquals(0, calculator.multiply(2, 3));
14        assertTrue(calculator.multiply(-1, 2) < 0);
15
16        assertEquals(2, calculator.divide(6, 3));
17        assertThrows(ArithmeticException.class, () -> calculator.divide
      (5, 0));
18    }
19 }
```

Fonte: (Autoria própria, 2024)

Observando o código após a refatoração (Figura 6), nota-se que cada asserção recebeu o parâmetro opcional de descrição textual, neste caso como último parâmetro das asserções (evidenciado que este é um projeto em Junit 5). Nota-se que nos casos em que a mensagem gerada inclui os dados dos parâmetros da asserção, uma variável é criada para armazenar os resultados. Isso ocorre para evitar uma dupla execução do método testado, que poderia modificar indevidamente do estado de seu objeto. Essas variáveis temporárias podem ser vistas no código de exemplo precedendo os *assertEquals* (linha 8, linha 16 e linha 32) e os *assertNotEquals*

(linha 24). Já para as demais asserções, a mensagem não inclui a chamada de métodos, portanto não há a criação da variável (linhas 12, 20 e 36).

Figura 6 – Código de exemplo após a refatoração

```
1 class CalculatorTest {
2
3     private final Calculator calculator = new Calculator();
4
5     @Test
6     void testCalculatorOperations() {
7         String result0 = String.valueOf(calculator.add(2, 3));
8         assertEquals(5, calculator.add(2, 3),
9             "Era esperado valores iguais, mas  5 é diferente de " +
10            result0 + " <calculator.add(2, 3)>");
11
12        assertTrue(calculator.add(1, -1) == 0,
13            "Era esperado verdadeiro, mas o valor obtido é falso");
14
15        String result1 = String.valueOf(calculator.subtract(2, 3));
16        assertEquals(-1, calculator.subtract(2, 3),
17            "Era esperado valores iguais, mas  -1 é diferente de " +
18            result1 + " <calculator.subtract(2, 3)>");
19
20        assertFalse(calculator.subtract(5, 3) < 0,
21            "Era esperado falso, mas o valor é verdadeiro");
22
23        String result2 = String.valueOf(calculator.multiply(2, 3));
24        assertNotEquals(0, calculator.multiply(2, 3),
25            "Era esperado valores diferentes, mas  0 é igual a " +
26            result2 + " <calculator.multiply(2, 3)>");
27
28        assertTrue(calculator.multiply(-1, 2) < 0,
29            "Era esperado verdadeiro, mas o valor obtido é falso");
30
31        String result3 = String.valueOf(calculator.divide(6, 3));
32        assertEquals(2, calculator.divide(6, 3),
33            "Era esperado valores iguais, mas  2 é diferente de " +
34            result3 + " <calculator.divide(6, 3)>");
35
36        assertThrows(ArithmeticException.class,
37            () -> calculator.divide(5, 0),
38            "Era esperado que a exceção  ArithmeticException.class fosse
39            lançada, mas não foi");
40    }
41 }
```

Fonte: (Autoria própria, 2024)

4.6 CONCEITOS DE CIÊNCIA DA COMPUTAÇÃO APLICADOS

Para a criação desta ferramenta, foram aplicados conceitos de Ciência da Computação vistos ao longo do curso e que serviram como base para a tomada de decisões durante a criação da ferramenta proposta por este trabalho:

- **Engenharia de Software:** area principal que fundamentou o desenvolvimento da ferramenta, posto que a área de testes de unidade está inserida nele (SOMMERVILLE, 2011).
- **Programação Orientada a objetos**: A ferramenta foi implementada em Java, uma linguagem orientada a objetos. Assim, conceitos como encapsulamento, abstração e herança foram aplicados na estruturação do código (SEBESTA, 2018).
- **Programação Lógica e Funcional:** Embora o paradigma principal utilizado na implementação seja orientado a objetos, elementos de programação funcional também foram empregados. Funções de alta ordem, como lambdas e streams, foram utilizadas em diversas partes do código para facilitar a manipulação de coleções e o processamento de dados (SEBESTA, 2018).
- **Padrões de Projeto:** O desenvolvimento da ferramenta também envolveu a aplicação de conceitos de padrões de projeto. Por exemplo, o padrão *Visitor* foi empregado para navegar pela árvore sintática abstrata (AST) gerada pelo JavaParser (GAMMA, 2009).
- **Compiladores:** A ferramenta utiliza conceitos de compiladores para realizar a análise e refatoração de código-fonte aplicados através da utilização da biblioteca JavaParser permite transformar códigos Java em árvores sintáticas abstratas (ASTs), que são uma representação hierárquica do código (SMITH; VAN BRUGGEN; TOMASSETTI, 2017). Com base nessa representação, a ferramenta identifica chamadas de métodos de asserção e aplica transformações para inserir mensagens descritivas. A compreensão de estruturas de código, como declarações de métodos e listas de argumentos, foi essencial para o sucesso da implementação.

5 ESTUDO EMPÍRICO EM PROJETOS CÓDIGO ABERTO

Um estudo empírico foi conduzido com o intuito de explorar alguns aspectos de projetos Java que utilizam o *framework* JUnit nas versões 4 e 5, assim como uma avaliação preliminar do impacto da aplicação de uma ferramenta em sua versão inicial para refatoração desses testes. Neste estudo observou-se a frequência de utilização de mensagens de asserção com e sem o parâmetro opcional de descrição, além de analisar a ocorrência do *test smell assertion roulette* nas asserções, tanto no estado original dos projetos quanto após a refatoração realizada com o uso da ferramenta.

Para tal, foram considerados 35 projetos Java de código aberto que fazem uso de testes de unidade Junit. O critério para a seleção dos projetos foi o de serem de código aberto, utilizarem Junit nas versões 4 ou 5, possuírem uma quantidade razoável de testes de unidade e serem populares e maduros. A lista dos projetos e a versão considerada está descrita no Quadro 3.

Para a realização do estudo empírico, foram considerados apenas as asserções suportadas pela ferramenta e os mesmos critérios para a classificação realizado pelo componente *visitor* da ferramenta (Seção 4.2), isto é, para a identificação de asserções com e sem a mensagem de descrição de asserção e que contenham ou não o *test smell assertion roulette*. Para isso, executou-se a ferramenta duas vezes, uma com a refatoração desligada, para obter as informações dos projetos de código aberto em seu estado original e outra com a refatoração ligada, a fim de apresentar os dados após a refatoração. Os resultados obtidos foram apresentados de forma tabular e discutidos. Os dados brutos do estudo empírico podem ser obtidos parcialmente no Apêndice A do presente documento ou em sua totalidade no repositório do Github do projeto em (https://github.com/vitorueno/tcc/tree/main/estudo_empirico_reducao_assertion_roulette). O resultado desse estudo empírico se encontra na Seção 5.1.

Outra informação obtida através da execução da ferramenta para a condução deste estudo empírico foi a de mensagens de descrição das asserções que já possuíam este parâmetro nos projetos em seu estado original (sem a aplicação da refatoração). Dessa forma, uma observação foi conduzida com o intuito de identificar possíveis categorias para os tipos de mensagens mais comuns. Não foi adotado um critério rígido para a classificação; no entanto, buscou-se avaliar se as mensagens apresentavam características como generalidade (isto é, pouco contexto sobre o propósito da asserção e repetitividade entre diferentes asserções), se incluíam ou não os dados envolvidos na asserção, e se exigiam conhecimento prévio do código de produção para serem compreendidas (com base na compreensão do autor). Os dados utilizados como base para essa observação se encontram em https://github.com/vitorueno/tcc/tree/main/experimento_descricoes. Os resultados e discussão dessa observação estão descritos na Seção 5.2.

Quadro 3 – Lista de projetos de código aberto considerados para o estudo empírico e suas versões.

#	**Projeto**	**Versão**
1	Avro	1.a11.1
2	ClassGraph	4.8.155
3	Cucumber JVM	7.11.1
4	Disruptor	4.0.0.RC1
5	Dropwizard	2.1.4
6	Fastjson2	2.0.24
7	GraphHopper	6.2
8	Jasypt Spring Boot	3.0.5
9	JavaParser	3.25.1
10	JDA	5.0.0-beta.5
11	JetCache	2.7.3
12	JFreeChart	1.5.3
13	Jodd	5.3.0
14	Jsoup	1.15.3
15	Liquibase	4.19.1
16	Logback	1.4.5
17	OpenPDF	1.3.30
18	OptaPlanner	9.35.0.Beta2
19	POI-TL	1.12.1
20	Recaf	2.21.13
21	Redisson	3.19.3
22	RipMe	1.7.95
23	RoaringBitmap	0.9.39
24	RSocket Java	1.1.3
25	Simple Binary Encoding	1.27.0
26	Simplify	1.3.0
27	Spring Batch	5.0.1
28	Spring Cloud Netflix	4.0.0
29	Spring Data JPA	3.0.3
30	SpringDoc OpenAPI	2.0.2
31	SQLite JDBC	3.41.0.0
32	TableSaw	0.43.1
33	Thymeleaf	3.1.1.RELEASE
34	Unirest Java	3.14.2
35	WireMock	3.0.0-beta.7

Fonte: Elaboração própria, 2024.

5.1 RESULTADOS E DISCUSSÃO

O resultado do estudo empírico do antes e depois da aplicação da ferramenta proposta por este trabalho pode ser observado na Tabela 1. A tabela está organizada como se segue: os dados estão divididos em quatro colunas principais:

1. Asserções sem descrição: asserções que não apresentam o parâmetro de descrição de asserção

2. Asserções com descrição: asserções que apresentam o parâmetro de descrição de asserção.
3. Assertion roulette: asserções classificadas como *assertion roulette*, isto é, sem parâmetro opcional de descrição e contidos em métodos de teste de unidade que contém outras asserções.
4. Total asserções: número total de asserções.

Para cada uma dessas categorias, exceto para o número total de asserções, os resultados se subdividem em:

- I: dados obtidos no projeto puro, isto é, antes da aplicação da refatoração por meio da ferramenta proposta por este trabalho.
- II: dados obtidos depois da aplicação da ferramenta proposta pelo trabalho.

Por fim, os dados também se encontram subdivididos entre o valor absoluto obtido e o percentual em relação à coluna Total de asserções. Não obstante, cada linha da tabela corresponde a um projeto analisado, enquanto a última linha consolida os resultados gerais de todos os projetos, funcionando como um totalizador.

5.1.1 Antes da refatoração

Ao todo, antes da aplicação da ferramenta, das 111.967 asserções (100%), 105.774 (93,10%) não possuem descrição, 6.126 (6,83%) possuem e 90.409 (74,19%) apresentam *assertion roulette*.

Nota-se que projetos como o Poi, Dropwizard, Spring Data JPA e Rsocket apresentam uma porcentagem próxima de 100% das asserções sem descrição. Isso pode indicar que o uso de descrições de asserções não é uma prática comum nestes projetos ou que eles adotam formas de documentação de testes de unidade fora do código. Por outro lado, os dois projetos que mais tiveram asserções com descrição foram o SpringDoc OpenAPI e o Cucumber.

Nota-se que 5 projetos não possuem nenhuma asserção com descrição e que dos 35, 14 apresentam menos de 1% de asserções com descrição. Na verdade, apenas 7 projetos apresentam mais do que 10% de asserções com descrição, com o OpenPDF apresentando o maior percentual (31.67%) e o GrassHopper o maior número absoluto (1.848).

Nesse sentido, a mediana das colunas podem facilitar essa interpretação. Para as asserções sem descrição ela fica em 97.55%, para os com descrição, 2.15% e, para o *assertion roulette*, 75.26%. Dessa forma, observa-se que mais da metade dos projetos apresentam pelo menos 97.5% das asserções sem descrição e 75.26% desses com *assertion roulette*.

Além disso, o número de asserções sem descrição é maior que os com descrição para todos os projetos, o que já era de se esperar, tendo em vista que a ausência deste parametro não é necessariamente considerada uma má prática em todas as situações. Testes de unidade com apenas uma asserção não precisam do parâmetro de descrição, pois teoricamente apresentam informações de depuração suficiente apenas pelo contexto.

Entretanto, observa-se que todos os projetos apresentam um número maior de asserções com *assertion roulette* do que com descrição, o que indica que a presença de múltiplas asserções por caso de teste é uma prática comum e que, mesmo nas situações em que o parâmetro opcional não é fornecido.

Não obstante, nenhum projeto esteve livre de *assertion roulette*, corroborando para a proposição de que este é o *test smell* mais comum em projetos com códigos de teste (SANTANA *et al.*, 2020).

5.1.2 Depois da refatoração

Observa-se que o número de asserções total permaneceu o mesmo (111.967), entretanto, o número de asserções sem descrição caiu de 105.774 (93,10%) para apenas 45.127 (40,30%), enquanto o número de asserções com descrição foi de 6.126 (6,83%) para 66.797 (59,70%). Por fim, os casos de *assertion roulette* em asserções caíram de 90.409 (74,19%) para apenas 38,58%.

Observa-se que a depender do projeto, a ferramenta se mostrou mais ou menos efetiva. Projetos como JSoup, Jodd, JFreeChart e Poi TL apresentaram um aumento significativo no número de asserções com descrição, que passaram a ficar acima dos 95%. Por outro lado, projetos como o SQLite JDBC, Spring Data JPA, Spring Cloud Netflix e Dropwizard apresentaram um percentual de asserções com descrição próximos de 0%. O mesmo comportamento inconstante pode ser observado na coluna de assertion roulette, com 11 projetos (31,42% do total) que reduziram as ocorrências para abaixo dos 10%, como o Unirest Java, Tablesaw, Spring Batch, Simplify e Poi Tl, enquanto 7 projetos (20% do total) apresentam um percentual de *assertion roulette* acima dos 80%.

Isso pode ter ocorrido por conta das limitações descritas na Seção 4.4, em especial pela presença de estruturas de controle nas asserções ou por importações fora dos padrões esperados pela ferramenta. Não obstante, como uma ferramenta prova de conceito, a ferramenta se mostrou satisfatória por apresentar uma redução razoável nas asserções sem descrição e nas ocorrências de *assertion roulette*, mesmo em ambientes de projetos reais. De qualquer forma, os resultados ainda evidenciam que há margem para aperfeiçoamento na ferramenta, tal como a redução das ocorrências de *assertion roulette* para algo próximo de zero por cento, e a apresentação de resultados mais constantes independentemente do projeto.

5.2 CLASSIFICAÇÃO DE MENSAGENS DE ASSERÇÃO

A observação das mensagens de descrição de asserção presente nos projetos de código aberto analisados serviram de base para a criação de uma classificação. Dessa forma, a presente seção visa classificar os padrões identificados e pontuar exemplos de cada. A seguir, será apresentado cada uma das classificações observadas:

1. **Mensagem com descrição detalhada e com dados**: mensagem que apresenta uma descrição da causa da falha da asserção e apresenta dados utilizados na asserção.

Tabela 1 – Estudo empírico em projetos de código aberto antes e após a refatoração com a ferramenta, com valores absolutos e percentuais

Projeto	Asserções sem descrição				Asserções com descrição				Assertion roulette				Total asserções
	I		II		I		II		I		II		
	Abs	%	Abs	%	Abs	%	Abs	%	Abs	%	Abs	%	
JDA	379	97,68	343	88,40	8	2,06	44	11,34	375	96,65	339	87,37	388
OpenPDF	244	67,22	202	55,65	119	32,78	161	44,35	180	49,59	148	40,77	363
Recaf	515	87,44	77	13,07	73	12,39	511	86,76	471	79,97	69	11,71	589
RoaringBitmap	6.507	98,19	1.712	25,76	120	1,81	4.935	74,24	5.821	87,84	1.616	24,31	6.627
Avro	2.144	81,89	964	36,82	474	18,11	1.654	63,18	1.806	68,98	829	31,67	2.618
Classgraph	565	96,91	565	96,91	18	3,09	18	3,09	472	80,96	472	80,96	583
Cucumber	1.993	92,06	1.381	63,79	130	6,00	742	34,27	1.253	57,88	933	43,09	2.165
Disruptor	278	95,86	134	46,21	12	4,14	156	53,79	151	52,07	90	31,03	290
Dropwizard	4.314	99,95	4.288	99,35	2	0,05	28	0,65	3.385	78,43	3.369	78,06	4.316
Fastjson2	20.278	99,44	5.184	25,42	114	0,56	15.208	74,58	17.520	85,92	4.650	22,80	20.392
Graphhopper	4.341	69,73	386	6,20	1.883	30,25	5.841	93,79	4.152	66,70	375	6,02	6.225
Jasypt	62	91,18	5	7,35	6	8,82	63	92,65	20	29,41	4	5,88	68
JavaParser	5.981	97,13	805	13,07	158	2,57	5.335	86,62	4.521	73,42	459	7,45	6.158
Jetcache	1.041	99,24	706	67,30	8	0,76	343	32,70	982	93,61	691	65,87	1.049
JFreeChart	8.235	86,95	215	2,27	1.236	13,05	9.256	97,73	7.568	79,91	207	2,19	9.471
Jodd	4.336	96,72	204	4,55	147	3,28	4.279	95,45	4.055	90,45	186	4,15	4.483
Jsoup	3.225	99,08	78	2,40	30	0,92	3.177	97,60	2.848	87,50	68	2,09	3.255
Liquibase	1.269	86,86	244	16,70	192	13,14	1.217	83,30	1.056	72,28	182	12,46	1.461
Logback	1.705	92,92	647	35,26	130	7,08	1.188	64,74	1.269	69,16	536	29,21	1.835
Optaplanner	5.177	99,29	5.107	97,95	34	0,65	104	1,99	4.742	90,95	4.674	89,64	5.214
Poi TL	424	100,00	16	3,77	0	0,00	408	96,23	380	89,62	14	3,30	424
Redisson	7.503	98,87	7.224	95,19	86	1,13	365	4,81	6.591	86,85	6.328	83,38	7.589
Ripme	118	88,06	112	83,58	16	11,94	22	16,42	49	36,57	44	32,84	134
Rsocket	1.540	99,87	1.421	92,15	2	0,13	121	7,85	1.366	88,59	1.254	81,32	1.542
Simple Binary Encoding	914	97,55	734	78,34	23	2,45	203	21,66	842	89,86	711	75,88	937
Simplify	423	95,27	43	9,68	21	4,73	401	90,32	323	72,75	34	7,66	444
Spring Batch	6.600	92,62	665	9,33	526	7,38	6.461	90,67	5.346	75,02	606	8,50	7.126
Spring Cloud Netflix	342	99,42	342	99,42	2	0,58	2	0,58	236	68,60	236	68,60	344
Spring Data JPA	1.657	99,94	1.652	99,64	1	0,06	6	0,36	1.063	64,11	1.060	63,93	1.658
Springdoc OpenAPI	117	71,78	99	60,74	46	28,22	64	39,26	59	36,20	52	31,90	163
Sqlite JDBC	1.887	99,79	1.887	99,79	4	0,21	4	0,21	1.837	97,14	1.837	97,14	1.891
Tablesaw	2.574	88,79	178	6,14	325	11,21	2.721	93,86	2.170	74,85	161	5,55	2.899
Thymeleaf	5.261	99,73	5.100	96,68	14	0,27	175	3,32	4.904	92,97	4.818	91,34	5.275
Unirest Java	988	94,73	81	7,77	55	5,27	962	92,23	706	67,69	61	5,85	1.043
Wiremock	2.837	96,23	2.326	78,90	111	3,77	622	21,10	1.890	64,11	1.604	54,41	2.948
Total/Média	**105.774**	**93,10**	**45.127**	**49,30**	**6.126**	**6,83**	**66.797**	**50,62**	**90.409**	**74,19**	**38.717**	**39,67**	**111.967**

Legenda: I - Antes da refatoração com a ferramenta; II - Depois da aplicação da refatoração com a ferramenta.
Fonte: (autoria própria, 2024).

Mensagens nesse formato são os que mais se assemelham as mensagens criadas pela ferramenta.

a) String.format("The key %s of %s is not in the default bundle", key, lang)
b) "Expected 5000 +/- 100 but was "+ diff

2. **Mensagem contextualizadas**: diferem da classificação anterior pois não apresentam necessariamente os dados utilizados na asserção, mas sim o contexto em que a asserção ocorreu, como a função testada ou um fluxo específico em que ele ocorreu. A criação e compreensão desse tipo de mensagem parece exigir mais conhecimento específico do código de produção e do contexto envolvido. Supõe-se que em alguns casos esse tipo de mensagem pode não auxiliar no processo de *debug*, principalmente quando não há conhecimento do código testado.

a) "context must return a map with one element when resolve an object which has one field with constraint annotations"
b) "PdfReader fails to report the correct number of pages"
c) "Should have contained a '&xa0;' or a ' '."
d) "Only consume [start, end) ranges!"

e) "Sanity check - aiming to test non array backed branch"

3. **Mensagem de localização**: mensagens que informam onde o erro ocorreu, como a linha da asserção ou o ponto no código testado, mas não informam muitas outras informações.
    a) "Error on iteration "+ i
    b) "mismatch at "+ i
    c) "fail "+ i
4. **Mensagem genérica ou pouco contextualizada**: mensagens que não contextualizam bem a causa do erro nem informa a localização ou os dados envolvidos. Muitos projetos apresentaram mensagens genéricas que se repetem em mais de uma asserção. Esse se mostrou ser o tipo mais comum de mensagem adotada pelos projetos analisados. Supõe-se que para alguns casos essas mensagens são suficientes, mas que dependendo do caso de teste a depuração pode ser afetada em caso de falha de uma das afirmações pela ausência de motivos específicos da falha.
    a) "at least one failure should have occured"
    b) "Search text not found"
    c) "Expected element"
    d) "Exceeded maximum expected time."
    e) "Incorrect transformed value"

No caso da ferramenta proposta por esse trabalho, as mensagens geradas parecem se enquadrar mais na classificação 1, pois se tratam de mensagens simples que não envolvem o contexto específico envolvido no teste de unidade, mas apresenta a causa da falha da asserção com os dados esperados e obtidos.

6 COMPARATIVO LLM

O presente capítulo descreve um estudo empírico que teve como objetivo comparar a similaridade das mensagens de descrição de asserções geradas pela ferramenta proposta por este trabalho e LLMs (*Large Language models*, grandes modelos de linguagem).

6.1 MATERIAIS E MÉTODOS

Para realizar o estudo empírico, optou-se pelo uso do ChatGPT e o Gemini, devida à sua relevância (MINAEE *et al.*, 2024), apesar de terem sido feito tentativas com LLama [7], que se mostrou muito lento por ser executado localmente. De forma mais específica, os modelos utilizados foram o gpt-3.5-turbo e o gemini-1.5-flash. O estudo empírico foi conduzido no dia 06 de dezembro de 2024. Ressalta-se que os resultados seriam diferentes caso tivessem sido realizados em outra data.

Quanto aos dados, foram utilizados como base asserções dos projetos de código aberto listados no Quadro 3 Desse grupo, foram desconsiderados os projetos Classgraph, Sqlite Jdbc e Spring Cloud Netflix, uma vez que a refatoração pela ferramenta se mostrou problemática para eles por conta das limitações descritas na Seção 4.4, principalmente por conta de importações de arquivos fora do padrão esperado pela ferramenta. Não obstante, devido a limitações do número de requisições máximas permitidas por dia pelas APIs do ChatGPT e do Gemini, optou-se por selecionar apenas as cem primeiras asserções de cada projeto. Os dados brutos de entrada usadas no estudo empírico, que correspondem as asserções dos projeto de código aberto e às mensagens geradas pela ferramenta proposta por este trabalho para cada uma dessas asserções, estão disponíveis nas URLs https://github.com/vitorueno/tcc/tree/main/estudo_empirico_llm e https://github.com/vitorueno/tcc/tree/main/estudo_empirico_refactor, respectivamente. Por limitação de espaço, os dados brutos não foram incluídos no presente trabalho.

Para interagir com as LLMs utilizaram-se práticas comuns de elaboração de *prompts* utilizadas na literatura (YUAN *et al.*, 2024) (DONG *et al.*, 2024). A prática consiste montar o *prompt* em duas partes, uma que contextualiza a LLM quanto ao cenário e ao papel que ela deve assumir (*role ou contexto*), e outra que instrui ela quanto ao que deve fazer de forma clara e as expectativas da tarefa (instrução) (YUAN *et al.*, 2024). Para a condução do estudo empírico, foram utilizados dois *prompts*, um que não especifica o formato esperado para as entradas e saídas, e outro que o especifica. Os dois prompts considerados pelo estudo empírico estão apresentados no Quadro 4. Nota-se que a parte de contextualização é comum aos dois *prompts*, alterando-se apenas as instruções. Conforme mencionado, o *prompt* 1 não apresenta o formato de entradas e saídas esperadas, mas o segundo sim. Ressalta-se que na execução do estudo empírico as entradas reais foram concatenadas a mensagem.

Por fim, para a obtenção da similaridade entre as mensagens geradas pela ferramenta

[7] https://www.llama.com/docs/overview

Quadro 4 – Contexto e instruções para construção de prompts.

Contexto: Você é um desenvolvedor senior especialista em testes de unidade Junit com muitos anos de experiência e recebe a missão de criar mensagens de descrição de erro de asserts, o parâmetro opcional de asserts Junit, para corrigir o test smell de assertion roulette.	
Instrução do Prompt 1	**Instrução do Prompt 2**
Para cada assert abaixo, gere o parâmetro opcional de mensagem de descrição de erro de asserts Junit. Retorne apenas a mensagem gerada em português, sem estar dentro do método de assert. Também não dê explicações, exemplos, contexto ou códigos, apenas o parâmetro gerado.	Para cada assert abaixo gere uma mensagem apropriada que o descreva. Como resposta retorne apenas a mensagem em português conforme o exemplo: exemplo de entrada: assertEquals(5, calculator.add(2, 3)); assertEquals(7, calculator.add(4, 2)); assertEquals(10, calculator.add(5, 5)); exemplo de saída: eram esperados valores iguais, mas 5 é diferente de calculator.add(2, 3) eram esperados valores iguais, mas 7 é diferente de calculator.add(4, 2) eram esperados valores iguais, mas 10 é diferente de calculator.add(5, 5) entradas reais:

O prompt é composto pelo contexto em conjunto com a instrução.
Fonte: Elaboração própria.

com as geradas pelas LLMs, utilizou-se do algoritmo Jaro de similaridade de strings por sua capacidade de gerar pontuações de similaridade com qualidade (GONDIM, 2006). O resultado do algoritmo Jaro é uma numeração entre 0 e 1 que indica o quão similar duas *strings* são, com 0 indicando que as *strings* não são nem um pouco similares e 1 indicando serem idênticas. Optou-se pelo uso de uma implementação pronta do algoritmo, disponível através da biblioteca python jaro-winkler [8].

Os códigos utilizados para a condução do estudo empírico estão disponíveis em https://github.com/vitorueno/tcc/tree/main/comparacaoLLM. O código que realiza o estudo empírico foi escrito em python e realiza todo o processo de leitura das asserção dos projetos de código aberto, geração das mensagens com LLM, escrita dos resultados em arquivos e comparação destes resultados com os obtidos pela ferramenta proposta por esse trabalho. Ao todo foram realizadas apenas duas execuções para cada um dos dois *prompts*, por conta dos limites diários de utilização das LLMs e pelo fato de que idealmente todas as execuções deveriam ocorrer num mesmo dia. Por fim, os dados de resultado do estudo empírico podem ser observados em https://github.com/vitorueno/tcc/tree/main/comparacaoLLM/mensagem1 e https://github.com/vitorueno/tcc/tree/main/comparacaoLLM/mensagem2. Por limitação de espaço, estes não foram incluídos no presente trabalho, mas estão resumidos nas tabelas da Seção 6.2.

6.2 RESULTADOS

Os resultados das comparações estão divididos conforme o *prompt* utilizado. O resultado do primeiro *prompt*, cuja instrução não contém exemplos de saída (Quadro 4, instrução 1), pode ser visto na Tabela 2. O resultado do segundo *prompt*, que contém exemplos de saída esperada (Quadro 4, instrução 2), pode ser visto na Tabela 3.

Com relação ao primeiro *prompt*, nota-se uma média geral de similaridade de 0,63 para o ChatGPT e 0,65 para o Gemini, quando comparados às mensagens geradas pela ferramenta

[8] https://pypi.org/project/jaro-winkler/

Tabela 2 – Similaridade das mensagens de asserção geradas por LLM em relação às geradas por ferramenta (primeiro *prompt*)

Projeto	**ChatGPT**		**Gemini**		**Média**	
	I	**II**	**I**	**II**	**ChatGPT**	**Gemini**
Disruptor	0.68	0.67	0.66	0.66	0.67	0.66
Logback	0.63	0.64	0.69	0.69	0.64	0.69
Cucumber JVM	0.66	0.69	0.66	0.63	0.67	0.64
Jetcache	0.53	0.63	0.61	0.68	0.58	0.65
FastJson2	0.62	0.68	0.59	0.60	0.65	0.60
Jasypt Spring Boot	0.63	0.69	0.72	0.71	0.66	0.71
Tablesaw	0.60	0.61	0.62	0.62	0.60	0.62
Liquibase	0.66	0.66	0.68	0.69	0.66	0.68
Redisson	0.58	0.55	0.62	0.58	0.56	0.60
RoaringBitmap	0.64	0.60	0.64	0.67	0.62	0.65
Unirest Java	0.72	0.64	0.67	0.64	0.68	0.65
Jsoup	0.67	0.66	0.63	0.65	0.67	0.64
Javaparser	0.56	0.57	0.59	0.64	0.56	0.61
Avro	0.66	0.58	0.65	0.59	0.62	0.62
Wiremock	0.66	0.63	0.57	0.62	0.64	0.59
Spring Data Jpa	0.64	0.72	0.62	0.62	0.68	0.62
OpenPDF	0.58	0.59	0.65	0.66	0.58	0.66
Spring Batch	0.63	0.72	0.60	0.60	0.68	0.60
Thymeleaf	0.59	0.65	0.65	0.63	0.62	0.64
Rsocket Java	0.65	0.65	0.71	0.65	0.65	0.68
Optaplanner	0.66	0.59	0.74	0.66	0.62	0.70
Jodd	0.52	0.66	0.70	0.64	0.59	0.67
Simple Binary Encoding	0.61	0.73	0.60	0.66	0.67	0.63
JDA	0.67	0.60	0.74	0.61	0.64	0.67
Dropwizard	0.63	0.65	0.64	0.62	0.64	0.63
Jfreechart	0.62	0.51	0.66	0.64	0.57	0.65
Graphhopper	0.67	0.61	0.65	0.67	0.64	0.66
Poi-tl	0.58	0.63	0.62	0.68	0.60	0.65
Ripme	0.66	0.58	0.65	0.65	0.62	0.65
Springdoc-openapi	0.62	0.64	0.63	0.61	0.63	0.62
Recaf	0.58	0.62	0.67	0.67	0.60	0.67
Simplify	0.66	0.63	0.65	0.64	0.64	0.65
Média	0.63	0.63	0.65	0.64	0.63	0.65

I e II representam as duas execuções de cada LLM, enquanto as médias foram calculadas a partir delas.
Fonte: (autoria própria, 2024)

desenvolvida neste trabalho (Tabela 2). Isso representa uma similaridade acima de 60% para ambos os casos, ainda que o Gemini se mostrou mais consistente do que o ChatGPT, posto que para seis projetos o ChatGPT apresentou similaridades abaixo de 60%, enquanto o Gemini apresentou apenas um. Entretanto, de forma geral, para ambas os resultados de similaridade apresentaram pouca variação.

Ainda assim, as diferenças das *strings* podem estar justamente em pontos que dificultam a compreensão da intenção das asserções, como os dados envolvidos na afirmação. Não obstante, os resultados gerados por LLMs não são tão previsíveis quanto os gerados pela ferramenta, podendo estar sujeitos a erros de compilação mais frequentes, tal como ocorre nos testes de

unidade gerados pela ferramenta de Yuan *et al.* (2024).

De forma mais específica, nota-se que projetos como o Unirest e o Spring Data JPA apresentaram similaridades maiores para o ChatGPT, enquanto o Gemini apresentou as maiores similaridades para os projetos Jasypt e Optaplanner, que estão na casa dos 70% similar. No entanto, nota-se que similaridade para os projetos Jodd e Redisson com ChatGPT e Wiremock no Gemini ficaram abaixo dos 60%. Isso pode sugerir que ou as LLMs, ou a ferramenta apresentam dificuldades para gerar mensagens a depender do projeto.

Com relação ao segundo *prompt* 3, as médias de similaridade foram de 0,71 para o ChatGPT e 0,74 para o Gemini. Novamente o Gemini apresentou resultados de similaridade maiores do que o ChatGPT, além de mais constantes, com apenas um projeto abaixo dos 70%, em média, contra 7 do ChatGPT.

Percebe-se também que a média das taxas de similaridade foram maiores do que as obtidas com o primeiro *prompt* para a maioria dos projetos, ainda que não para todos. Esse resultado pode indicar que o segundo *prompt* foi mais claro do que o primeiro, talvez em decorrência do fato do segundo *prompt* descrever o formato das entradas e saídas esperadas. Nesse sentido, apresentar exemplos de saída é considerado uma boa prática com o uso de LLMs (YUAN *et al.*, 2024).

6.2.1 Considerações sobre os resultados

Para ambos os *prompts*, as taxas de similaridade das *strings* se mostraram acima dos 60%, mas o segundo prompt apresentou níveis de similaridade maiores, acima dos 70%, por conter em sua instrução informações a respeito dos formatos de entradas e saídas esperadas.

Apesar disso, pouco pode-se concluir com certeza a respeito da efetividade das mensagens geradas pela ferramenta proposta por este trabalho sem que experimentos empíricos sejam conduzidos. Pode-se apenas supor possíveis implicações dessa relação. Por exemplo, caso seja possível afirmar que as mensagens de descrição geradas pelas LLMs são satisfatórias, seria possível afirmar que as mensagens geradas pela ferramenta também seriam ao menos razoavelmente satisfatórias.

Pode-se afirmar, no entanto, que a ferramenta apresenta características positivas por não ser paga e por não possuir limites diários de execução e por criar mensagens constantes, que independem de fatores externos.

Por fim, evidencia-se a necessidade da realização de novos testes e de experimentos empíricos. Nesse sentido, experimentos poderiam verificar se as mensagens geradas pela ferramenta são relevantes para usuários e se de fato eles são preferíveis aos resultados obtidos por LLMs.

Tabela 3 – Similaridade das mensagens de asserção geradas por LLM em relação às geradas pela ferramenta (segundo prompt)

Projeto	**ChatGPT**		**Gemini**		**Média**	
	I	**II**	**I**	**II**	**ChatGPT**	**Gemini**
Disruptor	0.76	0.76	0.75	0.75	0.76	0.75
Logback	0.52	0.64	0.75	0.75	0.58	0.75
Cucumber JVM	0.61	0.63	0.67	0.74	0.62	0.71
Jetcache	0.72	0.76	0.69	0.71	0.74	0.70
FastJson2	0.73	0.73	0.76	0.72	0.73	0.74
Jasypt Spring Boot	0.73	0.67	0.75	0.74	0.70	0.75
Tablesaw	0.67	0.73	0.77	0.77	0.70	0.77
Liquibase	0.79	0.79	0.70	0.73	0.79	0.71
Redisson	0.71	0.74	0.73	0.73	0.72	0.73
RoaringBitmap	0.74	0.75	0.73	0.74	0.75	0.73
Unirest Java	0.76	0.75	0.77	0.77	0.76	0.77
Jsoup	0.72	0.57	0.66	0.71	0.65	0.68
Javaparser	0.70	0.72	0.73	0.70	0.71	0.71
Avro	0.75	0.69	0.66	0.75	0.72	0.71
Wiremock	0.76	0.76	0.72	0.69	0.76	0.70
Spring Data	0.70	0.74	0.71	0.73	0.72	0.72
OpenPDF	0.68	0.73	0.73	0.72	0.70	0.72
Spring Batch	0.74	0.74	0.75	0.72	0.74	0.73
Thymeleaf	0.75	0.74	0.75	0.76	0.74	0.76
Rsocket Java	0.74	0.70	0.77	0.77	0.72	0.77
Optaplanner	0.67	0.75	0.77	0.77	0.71	0.77
Jodd	0.76	0.74	0.77	0.76	0.75	0.76
Simple Binary Encoding	0.78	0.78	0.72	0.73	0.78	0.72
JDA	0.66	0.68	0.75	0.75	0.67	0.75
Dropwizard	0.75	0.73	0.74	0.75	0.74	0.75
Jfreechart	0.70	0.63	0.74	0.74	0.67	0.74
Graphhopper	0.74	0.66	0.73	0.75	0.70	0.74
Poi-tl	0.77	0.76	0.77	0.72	0.76	0.74
Ripme	0.66	0.72	0.77	0.68	0.69	0.72
Springdoc Openapi	0.75	0.71	0.72	0.68	0.73	0.70
Recaf	0.77	0.76	0.71	0.73	0.76	0.72
Simplify	0.50	0.74	0.78	0.77	0.62	0.77
Média	0.71	0.72	0.73	0.74	0.71	0.74

I e II representam as duas execuções de cada LLM, enquanto as médias são calculadas a partir delas.
Fonte: (autoria própria, 2024)

7 CONCLUSÃO

O presente trabalho cumpriu o objetivo de propor uma alternativa de solução para a refatoração de testes de unidade com a presença do *test smell* do tipo *assertion roullete* em relação às soluções presentes na literatura. Para tal, foram destacados as principais lacunas do trabalho de Santana *et al.* (2020), a principal referência deste estudo, devido à sua relevância e por ser recente e, visando superar essas lacunas, foi criado uma ferramenta como prova de conceito em Java e Java Parser.

As principais diferenças da ferramenta proposta por este trabalho em relação ao RAIDE estão no fato de que a presente ferramenta não é restrita ao JUnit 4 nem à IDE Eclipse, pois suporta também o JUnit 5 e pode ser executável pela linha de comando. Não obstante, a ferramenta gera mensagens automáticas com base no tipo da asserção, ao invés de gerar uma mensagem fixa que deve ser substituída pelo programador.

Apesar disso, algumas limitações foram identificadas, como problemas de identificação caso as importações dos arquivos de teste não sigam os padrões esperados pela ferramenta ou caso as asserções estejam presentes em uma estrutura de controle. Ressalta-se também que o código da ferramenta apresenta oportunidades de melhoria no que diz respeito à organização, clareza e aderência de padrões de projeto.

Não obstante, o objetivo específico do trabalho, que consistia na realização de estudos empíricos para auxiliar na compreensão da efetividade da ferramenta, também foi cumprido. Nos estudos empíricos conduzidos, pôde-se concluir que é comum a presença do *test smell assertion roulette* nos projetos de código aberto selecionados e que a ferramenta é capaz de reduzir, ainda que não totalmente, essas ocorrências. Por fim, também pôde-se concluir que as mensagens geradas pela ferramenta são razoavelmente similares as geradas por LLMs, mas pouco se pode concluir sobre isso sem que sejam realizados experimentos.

Como trabalhos futuros, pretende-se superar as limitações identificadas, principalmente as que resultam em problemas de identificação. Também pretende-se realizar experimentos empíricos para validar a eficácia e usabilidade da ferramenta em diferentes contextos e com desenvolvedores reais. Os experimentos podem validar a efetividade em relação ao trabalho da Santana *et al.* (2020), bem como em relação a mensagens geradas por LLMS.

REFERÊNCIAS

AGARWAL, BB; GUPTA, Maheśa; TAYAL, SP. **Software engineering and testing**. [*S.l.*]: Jones & Bartlett Publishers, 2009.

ALÉGROTH, Emil; GONZALEZ-HUERTA, Javier**. Towards a mapping of software technical debt onto testware**. *In*: IEEE**. 2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)**. [*S.l.*: *s.n.*], 2017. p. 404–411.

ALJEDAANI, Wajdi; MKAOUER, Mohamed Wiem; PERUMA, Anthony; LUDI, Stephanie**. Do the Test Smells Assertion Roulette and Eager Test Impact Students' Troubleshooting and Debugging Capabilities?** *In*: IEEE**. 2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)**. [*S.l.*: *s.n.*], 2023. p. 29–39.

ALJEDAANI, Wajdi; PERUMA, Anthony; ALJOHANI, Ahmed; ALOTAIBI, Mazen; MKAOUER, Mohamed Wiem; OUNI, Ali; NEWMAN, Christian D.; GHALLAB, Abdullatif; LUDI, Stephanie. **Test Smell Detection Tools: A Systematic Mapping Study**. [*S.l.*: *s.n.*], 2021. arXiv: 2104.14640 [cs.SE]. Disponível em: https://arxiv.org/abs/2104.14640.

ALMEIDA, Diogo; CAMPOS, José Creissac; SARAIVA, João; SILVA, João Carlos**. Towards a catalog of usability smells**. *In***: Proceedings of the 30th Annual ACM Symposium on Applied Computing**. [*S.l.*: *s.n.*], 2015. p. 175–181.

AMMANN, Paul; OFFUTT, Jeff. **Introduction to software testing**. [*S.l.*]: Cambridge University Press, 2016.

BAQAIS, Abdulrahman Ahmed Bobakr; ALSHAYEB, Mohammad. Automatic software refactoring: a systematic literature review. **Software Quality Journal**, Springer, v. 28, n. 2, p. 459–502, 2020.

BOWES, David; HALL, Tracy; PETRIC, Jean; SHIPPEY, Thomas; TURHAN, Burak**. How good are my tests?** *In*: IEEE**. 2017 IEEE/ACM 8th Workshop on Emerging Trends in Software Metrics (WETSoM)**. [*S.l.*: *s.n.*], 2017. p. 9–14.

DELPLANQUE, Julien; DUCASSE, Stéphane; POLITO, Guillermo; BLACK, Andrew P; ETIEN, Anne**. Rotten green tests**. *In*: IEEE**. 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)**. [*S.l.*: *s.n.*], 2019. p. 500–511.

DONG, Yihong; JIANG, Xue; JIN, Zhi; LI, Ge. **Self-collaboration Code Generation via ChatGPT**. [*S.l.*: *s.n.*], 2024. arXiv: 2304.07590 [cs.SE]. Disponível em: https://arxiv.org/abs/2304.07590.

GAMMA, Erich. **Padrões de projetos: soluções reutilizáveis**. [*S.l.*]: Bookman editora, 2009.

GONDIM, F. Algoritmo de comparação de strings para integração de esquemas de dados. **Trabalho de Conclusão de Curso (Graduação)**, 2006.

HOMÈS, Bernard. **Fundamentals of software testing**. [*S.l.*]: John Wiley & Sons, 2024.

JUNIT. **Class Assertions - Junit 4**. [*S.l.*: *s.n.*], 2024. https://junit.org/junit4/javadoc/4.12/org/junit/Assert.html. Acessado em: 18 nov. 2024.

JUNIT. **Class Assertions - Junit 5**. [*S.l.*: *s.n.*], 2024. https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/Assertions.html. Acessado em: 18 nov. 2024.

KAUR, Rupi. **The Sun and Her Flowers**. [*S.l.*]: Andrews McMeel Publishing, 2017. ISBN 9781449486792.

KIM, Dong Jae**. An empirical study on the evolution of test smell**. *In***: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings**. [*S.l.*: *s.n.*], 2020. p. 149–151.

MARTIN, Robert C. **Clean code: a handbook of agile software craftsmanship**. [*S.l.*]: Pearson Education, 2009.

MESZAROS, Gerard. **xUnit test patterns: Refactoring test code**. [*S.l.*]: Pearson Education, 2007.

MINAEE, Shervin; MIKOLOV, Tomas; NIKZAD, Narjes; CHENAGHLU, Meysam; SOCHER, Richard; AMATRIAIN, Xavier; GAO, Jianfeng. Large language models: A survey. **arXiv preprint arXiv:2402.06196**, 2024.

MYERS, Glenford J. **The art of software testing**. [*S.l.*]: John Wiley & Sons, 2006.

NAIK, Kshirasagar; TRIPATHY, Priyadarshi. **Software testing and quality assurance: theory and practice**. [*S.l.*]: John Wiley & Sons, 2011.

OSTERWEIL, Leon. Strategic directions in software quality. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 28, n. 4, p. 738–750, 1996.

PRESSMAN, Roger S; MAXIM, Bruce R. **Engenharia de software-9**. [*S.l.*]: McGraw Hill Brasil, 2021.

REZENDE, Denis Alcides. **Engenharia de software e sistemas de informação**. [*S.l.*]: Brasport, 2006.

SANTANA, Railana; MARTINS, Luana; SOARES, Larissa; VIRGÍNIO, Tássio; CRUZ, Adriana; COSTA, Heitor; MACHADO, Ivan**. RAIDE: a tool for Assertion Roulette and Duplicate Assert identification and refactoring**. *In*: p. 374–379.

SANTANA, Railana; MARTINS, Luana; VIRGÍNIO, Tássio; SOARES, Larissa; COSTA, Heitor; MACHADO, Ivan**. Refactoring Assertion Roulette and Duplicate Assert test smells: a controlled experiment**. *In***: Anais do XXV Congresso Ibero-Americano em Engenharia de Software**. Córdoba: SBC, 2022. p. 263–277.

SEBESTA, Robert W. **Conceitos de Linguagens de Programação-11**. [*S.l.*]: Bookman Editora, 2018.

SMITH, Nicholas; VAN BRUGGEN, Danny; TOMASSETTI, Federico. Javaparser: visited. **Leanpub, oct. de**, v. 10, p. 29–40, 2017.

SOMMERVILLE, Ian. Software engineering (ed.) **America: Pearson Education Inc**, 2011.

SPILLNER, Andreas; LINZ, Tilo. **Software Testing Foundations: A Study Guide for the Certified Tester Exam-Foundation Level-ISTQB® Compliant**. [*S.l.*]: dpunkt. verlag, 2021.

VAN DEURSEN, Arie; MOONEN, Leon; VAN DEN BERGH, Alex; KOK, Gerard**. Refactoring test code**. *In*: CITESEER**. Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)**. [*S.l.*: *s.n.*], 2001. p. 92–95.

YUAN, Zhiqiang; LIU, Mingwei; DING, Shiji; WANG, Kaixin; CHEN, Yixuan; PENG, Xin; LOU, Yiling. Evaluating and improving chatgpt for unit test generation. **Proceedings of the ACM on Software Engineering**, ACM New York, NY, USA, v. 1, FSE, p. 1703–1726, 2024.

Apêndices

APÊNDICE A – Dados do estudo empírico em projetos de código aberto

avro-release-1.11.1

Antes do refactor:

- Total asserções: 2618
- Asserções sem descrição: 2144
- Asserções com descrição: 474
- Assertion Roulette: 1806

Após o refactor:

- Total asserções: 2618
- Asserções sem descrição: 964
- Asserções com descrição: 1654
- Assertion Roulette: 829

classgraph-classgraph-4.8.155

Antes do refactor:

- Total asserções: 583
- Asserções sem descrição: 565
- Asserções com descrição: 18
- Assertion Roulette: 472

Após o refactor:

- Total asserções: 583
- Asserções sem descrição: 565
- Asserções com descrição: 18
- Assertion Roulette: 472

cucumber-jvm-7.11.1

Antes do refactor:

- Total asserções: 2165
- Asserções sem descrição: 1993
- Asserções com descrição: 130
- Assertion Roulette: 1253

Após o refactor:

- Total asserções: 2165
- Asserções sem descrição: 1381
- Asserções com descrição: 742
- Assertion Roulette: 933

disruptor-4.0.0.RC1

Antes do refactor:

- Total asserções: 290
- Asserções sem descrição: 278
- Asserções com descrição: 12
- Assertion Roulette: 151

Após o refactor:

- Total asserções: 290
- Asserções sem descrição: 134
- Asserções com descrição: 156
- Assertion Roulette: 90

dropwizard-2.1.4

Antes do refactor:

- Total asserções: 4316
- Asserções sem descrição: 4314
- Asserções com descrição: 2
- Assertion Roulette: 3385

Após o refactor:

- Total asserções: 4316
- Asserções sem descrição: 4288
- Asserções com descrição: 28
- Assertion Roulette: 3369

fastjson2-2.0.24

Antes do refactor:

- Total asserções: 20392
- Asserções sem descrição: 20278
- Asserções com descrição: 114
- Assertion Roulette: 17520

Após o refactor:

- Total asserções: 20392
- Asserções sem descrição: 5184
- Asserções com descrição: 15208
- Assertion Roulette: 4650

graphhopper-6.2

Antes do refactor:

- Total asserções: 6225
- Asserções sem descrição: 4341
- Asserções com descrição: 1883
- Assertion Roulette: 4152

Após o refactor:

- Total asserções: 6228
- Asserções sem descrição: 386
- Asserções com descrição: 5841
- Assertion Roulette: 375

jasypt-spring-boot-jasypt-spring-boot-parent-3.0.5

Antes do refactor:

- Total asserções: 68
- Asserções sem descrição: 62
- Asserções com descrição: 6
- Assertion Roulette: 20

Após o refactor:

- Total asserções: 68
- Asserções sem descrição: 5
- Asserções com descrição: 63
- Assertion Roulette: 4

javaparser-javaparser-parent-3.25.1

Antes do refactor:

- Total asserções: 6158
- Asserções sem descrição: 5981
- Asserções com descrição: 158
- Assertion Roulette: 4521

Após o refactor:

- Total asserções: 6158
- Asserções sem descrição: 805
- Asserções com descrição: 5335
- Assertion Roulette: 459

JDA-5.0.0-beta.5

Antes do refactor:

- Total asserções: 388

- Asserções sem descrição: 379
- Asserções com descrição: 8
- Assertion Roulette: 375

Após o refactor:

- Total asserções: 388
- Asserções sem descrição: 343
- Asserções com descrição: 44
- Assertion Roulette: 339

jetcache-2.7.3

Antes do refactor:

- Total asserções: 1049
- Asserções sem descrição: 1041
- Asserções com descrição: 8
- Assertion Roulette: 982

Após o refactor:

- Total asserções: 1049
- Asserções sem descrição: 706
- Asserções com descrição: 343
- Assertion Roulette: 691

jfreechart-1.5.3

Antes do refactor:

- Total asserções: 9471
- Asserções sem descrição: 8235
- Asserções com descrição: 1236
- Assertion Roulette: 7568

Após o refactor:

- Total asserções: 9471
- Asserções sem descrição: 215
- Asserções com descrição: 9256
- Assertion Roulette: 207

jodd-5.3.0

Antes do refactor:

- Total asserções: 4483
- Asserções sem descrição: 4336

- Asserções com descrição: 147
- Assertion Roulette: 4055

Após o refactor:

- Total asserções: 4483
- Asserções sem descrição: 204
- Asserções com descrição: 4279
- Assertion Roulette: 186

jsoup-jsoup-1.15.3

Antes do refactor:

- Total asserções: 3255
- Asserções sem descrição: 3225
- Asserções com descrição: 30
- Assertion Roulette: 2848

Após o refactor:

- Total asserções: 3255
- Asserções sem descrição: 78
- Asserções com descrição: 3177
- Assertion Roulette: 68

liquibase-4.19.1

Antes do refactor:

- Total asserções: 1461
- Asserções sem descrição: 1269
- Asserções com descrição: 192
- Assertion Roulette: 1056

Após o refactor:

- Total asserções: 1461
- Asserções sem descrição: 244
- Asserções com descrição: 1217
- Assertion Roulette: 182

logback-v_1.4.5

Antes do refactor:

- Total asserções: 1835
- Asserções sem descrição: 1705
- Asserções com descrição: 130

- Assertion Roulette: 1269

Após o refactor:

- Total asserções: 1835
- Asserções sem descrição: 647
- Asserções com descrição: 1188
- Assertion Roulette: 536

OpenPDF-1.3.30

Antes do refactor:

- Total asserções: 363
- Asserções sem descrição: 244
- Asserções com descrição: 119
- Assertion Roulette: 180

Após o refactor:

- Total asserções: 363
- Asserções sem descrição: 202
- Asserções com descrição: 161
- Assertion Roulette: 148

optaplanner-9.35.0.Beta2

Antes do refactor:

- Total asserções: 5214
- Asserções sem descrição: 5177
- Asserções com descrição: 34
- Assertion Roulette: 4742

Após o refactor:

- Total asserções: 5214
- Asserções sem descrição: 5107
- Asserções com descrição: 104
- Assertion Roulette: 4674

poi-tl-1.12.1

Antes do refactor:

- Total asserções: 424
- Asserções sem descrição: 424
- Asserções com descrição: 0
- Assertion Roulette: 380

Após o refactor:

- Total asserções: 424
- Asserções sem descrição: 16
- Asserções com descrição: 408
- Assertion Roulette: 14

Recaf-2.21.13

Antes do refactor:

- Total asserções: 589
- Asserções sem descrição: 515
- Asserções com descrição: 73
- Assertion Roulette: 471

Após o refactor:

- Total asserções: 589
- Asserções sem descrição: 77
- Asserções com descrição: 511
- Assertion Roulette: 69

redisson-redisson-3.19.3

Antes do refactor:

- Total asserções: 7589
- Asserções sem descrição: 7503
- Asserções com descrição: 86
- Assertion Roulette: 6591

Após o refactor:

- Total asserções: 7589
- Asserções sem descrição: 7224
- Asserções com descrição: 365
- Assertion Roulette: 6328

ripme-1.7.95

Antes do refactor:

- Total asserções: 134
- Asserções sem descrição: 118
- Asserções com descrição: 16
- Assertion Roulette: 49

Após o refactor:

- Total asserções: 134
- Asserções sem descrição: 112
- Asserções com descrição: 22
- Assertion Roulette: 44

RoaringBitmap-0.9.39

Antes do refactor:

- Total asserções: 6627
- Asserções sem descrição: 6507
- Asserções com descrição: 120
- Assertion Roulette: 5821

Após o refactor:

- Total asserções: 6627
- Asserções sem descrição: 1712
- Asserções com descrição: 4935
- Assertion Roulette: 1616

rsocket-java-1.1.3

Antes do refactor:

- Total asserções: 1542
- Asserções sem descrição: 1540
- Asserções com descrição: 2
- Assertion Roulette: 1366

Após o refactor:

- Total asserções: 1542
- Asserções sem descrição: 1421
- Asserções com descrição: 121
- Assertion Roulette: 1254

simple-binary-encoding-1.27.0

Antes do refactor:

- Total asserções: 937
- Asserções sem descrição: 914
- Asserções com descrição: 23
- Assertion Roulette: 842

Após o refactor:

- Total asserções: 937

- Asserções sem descrição: 734
- Asserções com descrição: 203
- Assertion Roulette: 711

simplify-1.3.0

Antes do refactor:

- Total asserções: 444
- Asserções sem descrição: 423
- Asserções com descrição: 21
- Assertion Roulette: 323

Após o refactor:

- Total asserções: 444
- Asserções sem descrição: 43
- Asserções com descrição: 401
- Assertion Roulette: 34

spring-batch-5.0.1

Antes do refactor:

- Total asserções: 7126
- Asserções sem descrição: 6600
- Asserções com descrição: 526
- Assertion Roulette: 5346

Após o refactor:

- Total asserções: 7126
- Asserções sem descrição: 665
- Asserções com descrição: 6461
- Assertion Roulette: 606

spring-cloud-netflix-4.0.0

Antes do refactor:

- Total asserções: 344
- Asserções sem descrição: 342
- Asserções com descrição: 2
- Assertion Roulette: 236

Após o refactor:

- Total asserções: 344
- Asserções sem descrição: 342

- Asserções com descrição: 2
- Assertion Roulette: 236

spring-data-jpa-3.0.3

Antes do refactor:

- Total asserções: 1658
- Asserções sem descrição: 1657
- Asserções com descrição: 1
- Assertion Roulette: 1063

Após o refactor:

- Total asserções: 1658
- Asserções sem descrição: 1652
- Asserções com descrição: 6
- Assertion Roulette: 1060

springdoc-openapi-2.0.2

Antes do refactor:

- Total asserções: 163
- Asserções sem descrição: 117
- Asserções com descrição: 46
- Assertion Roulette: 59

Após o refactor:

- Total asserções: 163
- Asserções sem descrição: 99
- Asserções com descrição: 64
- Assertion Roulette: 52

sqlite-jdbc-3.41.0.0

Antes do refactor:

- Total asserções: 1891
- Asserções sem descrição: 1887
- Asserções com descrição: 4
- Assertion Roulette: 1837

Após o refactor:

- Total asserções: 1891
- Asserções sem descrição: 1887
- Asserções com descrição: 4

- Assertion Roulette: 1837

tablesaw-0.43.1

Antes do refactor:

- Total asserções: 2899
- Asserções sem descrição: 2574
- Asserções com descrição: 325
- Assertion Roulette: 2170

Após o refactor:

- Total asserções: 2899
- Asserções sem descrição: 178
- Asserções com descrição: 2721
- Assertion Roulette: 161

thymeleaf-thymeleaf-3.1.1.RELEASE

Antes do refactor:

- Total asserções: 5275
- Asserções sem descrição: 5261
- Asserções com descrição: 14
- Assertion Roulette: 4904

Após o refactor:

- Total asserções: 5275
- Asserções sem descrição: 5100
- Asserções com descrição: 175
- Assertion Roulette: 4818

unirest-java-3.14.2

Antes do refactor:

- Total asserções: 1043
- Asserções sem descrição: 988
- Asserções com descrição: 55
- Assertion Roulette: 706

Após o refactor:

- Total asserções: 1043
- Asserções sem descrição: 81
- Asserções com descrição: 962
- Assertion Roulette: 61

wiremock-3.0.0-beta-7

Antes do refactor:

- Total asserções: 2948
- Asserções sem descrição: 2837
- Asserções com descrição: 111
- Assertion Roulette: 1890

Após o refactor:

- Total asserções: 2948
- Asserções sem descrição: 2326
- Asserções com descrição: 622
- Assertion Roulette: 1604