Universidade Federal da Bahia
Instituto de Matemática e Estatística

Programa de Pós-Graduação em Ciência da Computação

AVALIAÇÃO EMPÍRICA DA GERAÇÃO AUTOMATIZADA DE TESTES DE SOFTWARE SOB A PERSPECTIVA DE TEST SMELLS

Tássio Guerreiro Antunes Virgínio

DISSERTAÇÃO DE MESTRADO

Salvador/BA
Março de 2020

TÁSSIO GUERREIRO ANTUNES VIRGÍNIO

AVALIAÇÃO EMPÍRICA DA GERAÇÃO AUTOMATIZADA DE TESTES DE SOFTWARE SOB A PERSPECTIVA DE TEST SMELLS

Esta Dissertação de Mestrado foi apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal da Bahia, como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Orientador: Ivan do Carmo Machado

Salvador/BA
Março de 2020

Ficha catalográfica elaborada pela Biblioteca Universitária de Ciências e Tecnologias Prof. Omar Catunda, SIBI – UFBA.

V817 Virgínio, Tássio Guerreiro Antunes

Avaliação empírica da geração automatizada de testes de software sob a perspectiva de Test Smells / Tássio Guerreiro Antunes Virgínio. - Salvador, 2020.

152 f.

Orientadora: Prof. Dr. Ivan do Carmo Machado.

Dissertação (Mestrado) – Universidade Federal da Bahia. Instituto de Matemática e Estatística, 2020.

1. Software. 2. Code smells. 3. Test smells. 4. Automação - Software. 5. Engenharia de Software. I. Machado, Ivan do Carmo. II. Título.

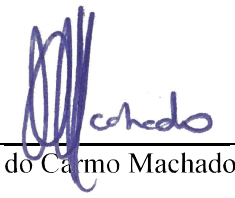
CDU 004.415.538

"*Avaliação empírica da geração automatizada de testes de software sob a perspectiva de Test Smells*"

Tássio Guerreiro Antunes Virgínio

Dissertação apresentada ao Colegiado do Programa de Pós-Graduação em Ciência da Computação na Universidade Federal da Bahia, como requisito parcial para obtenção do Título de Mestre em Ciência da Computação.

Banca Examinadora



Profº. Drº.Ivan do Carmo Machado (Orientador-UFBA)

Profº. Drº. Heitor Augustus Xavier Costa (UFLA)

Profº. Drº. Cláudio Nogueira Sant`Anna (UFBA)

AGRADECIMENTOS

Agradeço a Deus. Agradeço ao meu filho Theo Bernardino Guerreiro Virgínio, que se tornou minha grande fonte de felicidade e força, com a sua presença me impulsionou a realizar esse mestrado. Agradeço a Daniele Bernardino por ter proporcionado ter meu filho sempre perto de mim, o que me deu a força para continuar.

Agradeço aos meus pais Sebastião Virgínio de Oliveira e Neorama Guerreiro Antunes Virgínio por terem corações enormes, e ter me criado e ensinado da melhor forma que existe, pelo exemplo. Aos meus irmãos Tales, Taine, João Cesar, Voltaire, Anderson e meus familiares por acreditarem no meu sucesso, mesmo distantes, sei que sempre torceram por mim.

Aos professores do curso, que me repassaram os seus conhecimentos de forma exemplar, os quais tenho certeza que irei usar na minha vida acadêmica e profissional.

Agradeço ao meu orientador Ivan Machado, o qual tenho como exemplo de pesquisador, agradeço a paciência, ajuda e o tempo disponível no desenvolvimento desse trabalho.

E agradeço aos meus colegas de curso entre os quais tenho um agradecimento especial a Railana Santana, Luana Martins, Sara Mendes Oliveira, Joselito Júnior, Nildo Silva Jr, Denivan Campos, Emmanuel Sávio Silva Freire, Tiago Motta, Mayka Lima e Samia Capistrano. Que estavam tão empolgados e comprometidos quanto eu no decorrer deste mestrado, e que me deram força para que pudesse terminar.

RESUMO

A constante busca pela qualidade sempre está em destaque na área de Engenharia de Software. Dentre as diversas disciplinas dedicadas a essa temática, o teste de software tem se estabelecido como uma das mais importantes, dado sua eficácia na identificação de defeitos, em momento prévio à liberação de sistemas de software para o mercado. O teste de software é atividade-chave para o desenvolvimento de software de qualidade. Entretanto, desenvolver testes é tão ou mais custoso do que desenvolver o código de produção. Uma alternativa para a redução dos custos associados ao teste de software se dá pelo uso intensivo de ferramentas de automação de testes. A proposta dessas ferramentas é reduzir o tempo de produção sem afetar a qualidade do código. Apesar dessa premissa, não é comum encontrar abordagens que incluam uma camada de verificação de qualidade dos testes gerados automaticamente, o que pode reduzir a confiabilidade da eficácia desses testes. Neste cenário, a proposta dessa dissertação é analisar empiricamente massas de dados de teste, sob a perspectiva de *test smells*, no sentido de avaliar a qualidade dos testes produzidos por ferramentas de geração automatizada de testes de software. *Test smells* são más escolhas no design dos testes e tem características sintomáticas e podem acarretar diminuição na qualidade dos sistemas. Considerando os *test smells* em código de teste, o estudo analisa os testes gerados por duas ferramentas amplamente aceitas pela comunidade de testes: Evosuite e Randoop. Um conjunto de vinte e um projetos de software de código aberto, disponíveis na plataforma Github foram considerados no estudo. A análise considerou a dispersão de *test smells* no código de teste desses projetos, bem como a existência de potenciais correlações entre *test smells* e as relações com as métricas estruturais. Como principais resultados, encontramos fortes correlações entre os *test smells* e as métricas de cobertura do código, diferenças significativas entre os dados encontrados nas suítes de testes geradas automaticamente e com os testes pré-existentes nos projetos avaliados.

Palavras-chave: Qualidade de software, code smells, test smells, automação de teste de software.

ABSTRACT

The constant search for quality is always highlighted in the Software Engineering field. Among the various disciplines dedicated to this theme, software testing has been established as one of the most important, given its effectiveness in identifying defects, prior to the release of software systems to the market. Software testing is a key activity for the development of quality software. However, developing tests is just as or more expensive than developing the production code. An alternative for reducing the costs associated with software testing is the intensive use of test automation tools. The purpose of these tools is to reduce production time without affecting the quality of the code. Despite this premise, it is not common to find approaches that include a quality check layer of the automatically generated tests, which can reduce the reliability of the effectiveness of these tests. In this scenario, the purpose of this dissertation is to empirically analyze masses of test data, from the perspective of *test smells*, in order to assess the quality of the tests produced by automated software test generation tools. *Test smells* are poor choices in the design of tests and have symptomatic characteristics and can lead to a decrease in the quality of systems. Considering the *test smells* in test code, the study analyzes the tests generated by two widely accepted tools by the software testing community: Evosuite and Randoop. A set of twenty-one open source software projects, available on the Github platform, were considered in the study. The analysis considered the dispersion of *test smells* in the test code of these projects, as well as the existence of potential correlations between *test smells* and the relationships with structural metrics. As main results, we found strong correlations between the *test smells* and the code coverage metrics, significant differences between the data found in the test suites generated automatically and with the pre-existing tests in the evaluated projects.

Keywords: Software quality, code smells, test smells, automation of software test.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ACRÔNIMOS

BAT	*Business Acceptance Testing*
CMMI	Capability Maturity Model Integration
JaCoCo	*Java Code Coverage Library*
LOC	Linhas de Código
MPS.BR	Melhoria do Processo de Software Brasileiro
QMT	Quantidade de Métodos de Testes
SQuaRE	Software product Quality Requirements and Evaluation
SUT	*System Under Test*
tsDetect	Test Smells Detector
UAT	*User Acceptance Testing*

Capítulo 1

INTRODUÇÃO

A busca pela qualidade tem sido uma atividade fundamental na Engenharia de Software, desde a sua concepção, há mais de 50 anos. A ideia chave é desenvolver mecanismos para melhorar a qualidade do código, da arquitetura, do desempenho, da interface gráfica etc. (HIRAMA, 2012).

A depender do contexto, qualidade pode ter grande variedade de definições e, no contexto da Engenharia de Software, dependerá muito dos *Stakeholders* envolvidos. Por exemplo, do ponto de vista dos desenvolvedores, qualidade é seguir métodos e padrões de projetos; para os gerentes de projeto, qualidade tem a ver com ficar próximo das estimativas pré-definidas na literatura e na prática (esforço, custo e prazo); do ponto de vista dos usuários, é qualidade o software ser fácil de usar; no caso do cliente, qualidade é o software suprir as necessidades do negócio, prazos e custos; para o gerenciamento de configuração, qualidade é controle eficaz das versões do software (HIRAMA, 2012).

Um dos grandes problemas na qualidade de software diz respeito aos códigos mal elaborados. Apesar das boas intenções, códigos defeituosos continuam a ser o "fantasma" do mercado de software, sendo responsável por até 45% do tempo de inatividade dos sistemas computacionais, o que acarreta sérios prejuízos. Dados apontam que a baixa qualidade dos sistemas de software custaram às empresas americanas cerca de US$ 100 bilhões no ano de 2001, em termos de manutenção e redução da produtividade (RICADEL, 2001).

Em um relatório mais recente, a empresa de teste de software Tricentis analisou 606 falhas de software de 314 empresas para entender melhor o impacto comercial e financeiro das falhas de software. O relatório revelou que essas falhas de software afetaram 3,6 bilhões de pessoas e causaram US$ 1,7 trilhão em perdas financeiras e um total acumulado de 268 anos de inatividade (TRICENTIS, 2019). Em agosto de 2013, a Amazon perdeu US$ 4,8 milhões após uma queda de 40 minutos por causa de uma falha no software, o que resulta em US$ 120.000 por minuto (CISQ, 2019).

Dentre as inúmeras estratégias para melhorar a qualidade dos sistemas de software, a refatoração tem se destacado como uma importante atividade para melhorar a qualidade dos códigos existentes, considerando a longevidade dos sistemas de software, e a inevitável condição de esses acomodarem mudanças (devido à pressão de mercado, mudanças em

legislações, etc.) (FOWLER, 1999). A refatoração foi concebida inicialmente nos círculos do *Smalltalk*, mas tornou-se aplicável (e necessária) para outras linguagens de programação. O termo "*Refactoring*" foi originalmente introduzido por Opdyke (1992) em sua tese de doutorado. O termo foi bastante difundido pelo desenvolvimento de *frameworks*. Em um *framework*, o fluxo de desenvolvimento é evolutivo, ou seja, é pouco provável que o seu desenvolvimento esteja completo em uma primeira tentativa; como consequência, os programadores sabem que o código será lido e modificado mais frequentemente do que escrito (FOWLER, 1999).

Na prática de refatoração, a cada nova funcionalidade ou correção realizada, tem-se a oportunidade de trabalhar o *design* do código até ficar na sua forma mais simples, mesmo que isso implique em alterar um código que esteja em funcionamento (BECK, 2000).

A partir da experiência obtida com a participação em inúmeros projetos de desenvolvimento de software, Fowler (1999) elaborou um guia sobre refatoração de código, que fornece uma lista de problemas recorrentes, como detectá-los, e sugere possíveis soluções. A esses problemas comuns, convencionou-se utilizar o termo *code smells*, uma analogia entre código mal elaborado e odor desagradável, ou seja, é algo que só de "cheirar" pode-se facilmente identificar (FOWLER, 1999).

Para aplicar a refatoração, é necessário uma base importante de apoio, os testes. Fowler (1999) explica que os testes servirão como os validadores do código, pois servem para verificar se foi introduzida alguma falha e se os códigos continuam fazendo o que deveriam fazer. Os testes proveem segurança aos desenvolvedores ao realizar as refatorações.

Deursen et al. (2001) investigaram a relação entre *code smells* e testes, chegando a definição de *test smells*, que são más escolhas no *design* dos testes, tem características sintomáticas e podem acarretar em diminuição na qualidade dos sistemas. O estudo analisou a sua influência na qualidade dos testes de unidade e identificou a existência de onze *test smells*: *Mystery Guest; Resource Optimism; Test Run War; General Fixture; Eager Test; Lazy Test; Assertion Roulette; Indirect Testing; For Testers Only; Sensitive Equality* e *Test Code Duplication*. A Seção 2.3.2 detalha cada um desses *test smells*.

Os *test smells* são objeto de discussões ativas entre profissionais e pesquisadores, e várias diretrizes para lidar com eles são constantemente oferecidas para prevenção, detecção e correção (GAROUSI; KUCUK, 2018). Várias definições para *test smells* foram fornecidas na literatura, por exemplo, Bavota et al. (2012) e Peruma et al. (2019) afirmam que os *test smells* são testes mal planejados e sua presença pode afetar os conjuntos de testes e o código de produção (aspectos de qualidade ou, até mesmo, sua funcionalidade); Rompaey et al. (2007) descrevem que são violações do padrão de teste de quatro fases (configuração, execução, verificação e desmontagem), que resultam em uma diminuição de um ou mais critérios de qualidade do teste; no estudo de Hedayati, Ebrahimzadeh e Sori (2015), *test smells* são descritos como um conjunto de problemas no código de teste; em Garousi e Kucuk (2018), um *test smell* se refere a qualquer sintoma no código de teste que indique um possível problema mais profundo. Como um tipo de *anti-pattern*, os *test smells* são definidos como testes mal projetados e sua presença pode afetar negativamente a qualidade dos conjuntos de testes e, por consequência, o código de produção.

Diante da emergente necessidade de investigar o impacto da presença de *test smells* na qualidade de software, a presente investigação tem como objetivo avaliar a qualidade dos

códigos de teste gerados com suporte ferramental, considerando a difusão dos *test smells*. A utilização dessas ferramentas é essencial para reduzir o esforço e garantir a eficácia dos testes (SOMÉ; CHENG, 2008). Consequentemente, a verificação da qualidade dos códigos de testes gerados por essas ferramentas também deve ser considerada.

Palomba et al. (2016) realizaram um estudo empírico sobre a difusão dos *test smells* em testes gerados automaticamente utilizando a ferramenta Evosuite[1]. Os seus resultados mostraram uma alta difusão de *test smells*, frequentemente com co-ocorrência de diferentes tipos, e uma forte correlação com características estruturais. Tal estudo baseou-se na investigação de Bavota et al. (2015), que descobriu alta difusão de *test smells* em sistemas de software de código aberto e comerciais e forneceu evidências que os *test smells* têm forte impacto negativo na compreensão do código de teste.

A presente investigação estende os estudos de Bavota et al. (2015) e Palomba et al. (2016) no sentido de ampliar a compreensão da influência dos *test smells* nos códigos de teste, gerados pelas ferramentas Evosuite e Randoop[2], assim como os códigos de teste pré-existentes nos projetos de software. Estes estudos utilizaram a linguagem Java e do *framework* de testes JUnit[3]. Com o propósito descrito de dar continuidade aos estudos, seguiu-se a mesma estrutura, utilizando a mesma linguagem e o mesmo *framework*. As ferramentas Evosuite e o Randoop foram escolhidas por serem dois dos geradores de testes na linguagem Java mais referenciados na literatura e que geram códigos de teste utilizando o JUnit (RUEDA et al., 2016).

1.1 PROBLEMA

Foi identificado somente um estudo na literatura sobre a influência dos *test smells* na qualidade dos testes gerados de forma automática (PALOMBA et al., 2016). Além disso, houve pequena abrangência desse estudo em relação às ferramentas existentes. Como citado, no estudo de Palomba et al. (2016) somente foi utilizada a EvoSuite. Outro ponto de interesse é a falta de evidências sobre as características dos métodos de geração de testes utilizados nas ferramentas, no que diz respeito à difusão de *test smells* nos códigos de teste.

A seguinte questão de pesquisa direciona a presente investigação:

De que forma os geradores de teste influenciam a qualidade dos testes de unidade na perspectiva dos *test smells*?

A pergunta de pesquisa principal foi refinada em três perguntas específicas, a saber:

RQ1 Existe diferença (estatisticamente significativa) na qualidade do código de teste gerado pelas ferramentas Randoop e Evosuite?
Essa pergunta tem como objetivo realizar uma análise sobre o impacto das abordagens distintas da Evosuite e da Randoop para gerar os códigos de teste na perspectiva dos *test smells* encontrados.

[1] <http://www.evosuite.org/>
[2] <https://randoop.github.io/randoop/>
[3] <https://junit.org/junit5/>

RQ2 Existe diferença (estatisticamente significativa) na qualidade do código de teste gerado pela ferramenta Randoop e os códigos de testes pré-existentes?
Essa pergunta tem como objetivo realizar uma análise sobre o impacto da abordagem distinta da Randoop em comparação com a codificação existente dos testes de unidade, na perspectiva dos *test smells*.

RQ3 Existe diferença (estatisticamente significativa) na qualidade do código de teste gerado pela ferramenta Evosuite e os códigos de testes pré-existentes?
Essa pergunta tem como objetivo realizar uma análise sobre o impacto da abordagem distinta da Evosuite em comparação com a codificação existente dos testes de unidade, na perspectiva dos *test smells*.

1.2 OBJETIVOS

O objetivo geral deste trabalho é **analisar a qualidade dos testes de unidade gerados de forma automatizada na perspectiva dos *test smells***. Os seguintes objetivos específicos foram definidos para esta investigação:

O1 Desenvolver suporte ferramental para auxiliar na detecção de *test smells*;

O2 Avaliar empiricamente o impacto dos *test smells* na qualidade dos códigos de teste existentes nos projetos de software;

O3 Avaliar empiricamente o impacto dos *test smells* na qualidade dos códigos de teste gerados pela ferramenta Randoop;

O4 Avaliar empiricamente o impacto dos *test smells* na qualidade dos códigos de teste gerados pela ferramenta Evosuite;

O5 Analisar as co-ocorrências dos *test smells* para cada suite de teste.

1.3 METODOLOGIA

A presente investigação tem caráter exploratório, cujo propósito é avaliar a qualidade dos códigos de testes gerados de forma automática e pré-existentes, a partir da seleção de projetos *open source* disponíveis no repositório GitHub[4], e que atendam aos seguintes critérios: projetos Maven, desenvolvidos na linguagem Java, e que tenham classes de testes que utilizem a *framework* JUnit. Foram utilizados vinte e um projetos selecionados de forma aleatória os quais atendem a esses critérios. Em seguida, foram gerados os testes de unidade a partir das ferramentas EvoSuite e Randoop, com base nos códigos fontes dos projetos selecionados.

A partir dos códigos de testes gerados foi executada a ferramenta de detecção de *test smells*, **JNose Test**[5] que foi proposta e desenvolvida neste estudo. A ferramenta proveu

[4] <https://github.com>
[5] <https://github.com/tassiovirginio/jnose_test>

suporte à geração automatizada de métricas de qualidade, dentre elas, dados quantitativos de *test smells* encontrados. A partir dos dados coletados, verifica-se a qualidade dos testes de unidade com base nos *test smells*. Em contrapartida, a qualidade das ferramentas de geração de testes também é avaliada.

Este estudo contempla a execução das seguintes fases:

Fase 1 - Fundamentação teórica: Trata dos conhecimentos pertinentes ao tema proposto com a realização de uma revisão de literatura com o objetivo de compreensão dos conceitos relacionados;

Fase 2 - Desenvolvimento do ferramental de apoio: Trata do desenvolvimento da ferramenta **JNose Test**. Nessa fase, o objetivo foi realizar o levantamento dos requisitos, definir a arquitetura, implementar, testar e tornar disponível *releases* estáveis dessa ferramenta;

Fase 3 - Coleta dos dados: Os dados foram coletados utilizando a ferramenta **JNose Test**, com a execução nos pacotes de testes existentes e nos pacotes de testes gerados pelas ferramentas Evosuite e Randoop;

Fase 4 - Avaliação empírica: A partir dos dados obtidos na fase anterior, realizou-se uma análise da qualidade dos códigos na perspectiva dos *test smells* para cada pacote de teste, e pode-se verificar suas características e uma avaliação entre os mesmos.

1.4 CONTRIBUIÇÕES ESPERADAS

A presente investigação busca contribuir com a área de Engenharia de Software no que diz respeito a:

- Desenvolver a ferramenta JNose Test para detecção de *test smells* e outras métricas de forma automatizada na linguagem Java;
- Identificar a distribuição dos *test smells* em códigos de teste existentes nos 21 projetos selecionados e nos códigos gerados com as ferramentas Evosuite e Randoop;
- Prover análise sobre os tipos de *test smells* característicos em cada pacote de teste, com a identificação dos que se destacam em todos os pacotes de testes;
- Prover compreensão baseada em evidências acerca das diferenças entre os códigos de testes gerados e os existentes na perspectiva dos *test smells*;

1.5 ESTRUTURA DO TRABALHO

Os demais capítulos desta dissertação estão organizados da seguinte forma:

- **Capítulo 2 - Referencial teórico:** apresenta uma revisão bibliográfica sobre os conceitos fundamentais para a compreensão desta investigação: qualidade de software, testes de software, refatoração, *code smells* e *test smells*;
- **Capítulo 3 - JNose Test:** descreve a ferramenta para detecção de *test smells*, especificando a linguagem utilizada, as bibliotecas, sua arquitetura e seu funcionamento;

- **Capítulo 4 - Estudo Experimental:** descreve o protocolo do estudo experimental, considerando o processo de levantamento de dados, os projetos candidatos, e detalhes sobre a execução do JNose Test, análise e discussão dos resultados;
- **Capítulo 5 - Considerações finais:** apresenta os trabalhos relacionados, síntese das contribuições e perspectivas dos trabalhos futuros com base nos achados deste estudo.

Capítulo

2

REFERENCIAL TEÓRICO

Este capítulo apresenta a fundamentação teórica deste trabalho, com discussões sobre os conceitos pertinentes a esta dissertação de mestrado. A Seção 2.1 aborda o assunto de qualidade de software. A Seção 2.2 aborda a qualidade na perspectiva de teste de software. A Seção 2.3 aborda a qualidade dos códigos a partir dos *code smells* e *test smells*. A Seção 2.4 apresenta as ferramentas de geração de teste.

2.1 QUALIDADE DE SOFTWARE

Software é considerado um bem fundamental para sociedade, por causa da sua integração à diversas atividades do cotidiano (HIRAMA, 2012). Contudo, de acordo com Bartié (2002), os software passaram a ganhar complexidade, e como consequência direta, surgiram problemas envolvendo a sua qualidade.

O termo qualidade pode assumir diversos significados que dependem de fatores e de definições pré-estabelecidas. A ISO 9000, por exemplo, define qualidade como o grau em que um conjunto de características inerentes preenchem os requisitos (SCHMAUCH, 1995). Outra definição do termo qualidade é apresentado por Crosby (1979) e Gryna (2001), que a definem como "conformidade com requisitos" e "adequação para uso", respectivamente.

Guerra e Colombo (2009) descrevem que o software é um produto complexo, dependendo em boa parte da interpretação das necessidades do usuário, necessitando converter os requisitos dos usuários nos do produto para o seu desenvolvimento; dessa forma, necessita de técnicas de verificação e validação para todo o ciclo de desenvolvimento do produto e técnicas de avaliação do produto intermediário e final, garantindo que o produto é correto, seguro, entre outros atributos que asseguram a sua qualidade.

Qualidade de software constitui uma área cuja demanda está crescendo significativamente. Os usuários exigem cada vez mais eficiência, eficácia, dentre outras características de qualidade consideradas importantes para um produto como o software (GUERRA; COLOMBO, 2009). Contudo, verificar a qualidade de um software não é trivial e muito

menos sem importância. Deixar de se preocupar com qualidade ou não dar a devida atenção à mesma é considerado imprudente.

Em 2012 um dos maiores fabricantes norte-americanos de software para o mercado de ações lutou para se manter à tona depois que um *bug* provocou uma perda de US$ 440 milhões em apenas 30 minutos. As ações da empresa perderam 75% em valor de mercado em apenas dois dias após o software defeituoso inundar o mercado com operações não-intencionais. Os algoritmos de negociação do fabricante começaram a empurrar negócios erráticos em cerca de 150 ações diferentes (EHA, 2012). Outro caso de prejuízo causado pela qualidade de software ocorreu entre os anos de 2017 e 2018, onde foram perdidos mais de 500 milhões em criptomoedas, motivados por *bugs* em códigos (SEDGWICK, 2018).

De acordo com Bartié (2002), mais de 30% dos projetos são cancelados antes da sua conclusão, e aproximadamente 70% desses projetos são entregues com falhas nas funcionalidades previstas. Além disso, o custo médio dos projetos de software costuma exceder mais de 180% do que o previsto, e os prazos ultrapassam em 200% a definição inicial. De acordo com especialistas, bastam três ou quatro defeitos a cada 1.000 Linhas de Código (LOC) para que um programa execute de forma inadequada.

Apesar de os modelos aplicados na garantia da qualidade de software atuarem, em especial, no processo de desenvolvimento, o principal objetivo é garantir um produto final que satisfaça às expectativas do cliente, dentro daquilo que foi acordado inicialmente. No entanto, é impossível obter um software de qualidade com processos de desenvolvimento frágeis e deficientes. Portanto, não é possível estabelecer um processo de garantia da qualidade que não enfoque simultaneamente o produto tecnológico e o processo de desenvolvimento desse software.

De modo geral, na Engenharia de Software o termo qualidade pode ser compreendido em duas dimensões fundamentais: a qualidade do processo e a qualidade do produto. De acordo com Bartié (2002), para se atingir a qualidade do produto de software é necessário contemplar a qualidade do processo de software, pois são dimensões complementares e que não podem ser analisadas de forma separada. As seções 2.1.1 e 2.1.2 discutem essas duas dimensões.

2.1.1 Qualidade do processo de software

Processo de software consiste em um conjunto de atividades, métodos, práticas e transformações que a equipe de desenvolvimento emprega para construir e manter software e seus produtos associados. Especificação de requisitos, gerência de configuração, desenvolvimento de software são exemplos de processos que podem ser formalizados e documentados para desenvolver um produto (BARTIé, 2002).

Na década de 1990 até os dias atuais, houve uma grande preocupação com a modelagem e a melhoria do processo para a produção de software de qualidade, dentro do prazo e com orçamentos confiáveis. Algumas normas e modelos desenvolvidos na busca pela qualidade de processo de software são:

- ISO/IEC/IEEE 90003:2018[1]
- ISO/IEC 12207:2017[2]
- ISO/IEC 33001:2015[3]
- Capability Maturity Model Integration (CMMI)[4]
- Melhoria do Processo de Software Brasileiro (MPS.BR)[5]

Apesar de os modelos aplicados na garantia da qualidade de software atuarem, em especial, no processo, é importante garantir um produto final que satisfaça às expectativas do cliente, dentro daquilo que foi acordado inicialmente. Para estabelecer um processo de garantia da qualidade robusto, é necessário evidenciar simultaneamente o produto tecnológico e o processo de desenvolvimento desse software. Embora seja compreendido a relevância das duas dimensões de qualidade, essa pesquisa tem como finalidade explorar particularidade da qualidade do produto de software.

2.1.2 Qualidade do produto de software

Guerra e Colombo (2009) definem qualidade do produto de software como a conformidade a requisitos funcionais declarados explicitamente, padrões de desenvolvimento claramente documentados e as características implícitas que são esperadas de todo software desenvolvido profissionalmente. Por requisitos explícitos, pode-se entender requisitos do usuário ou requisitos funcionais; por sua vez, requisitos implícitos, ou requisitos não-funcionais, relacionam-se com os atributos de qualidade da aplicação, por exemplo, com o desempenho de execução do sistema.

A qualidade do produto de software tem o propósito de garantir a qualidade do produto tecnológico gerado durante o ciclo de desenvolvimento. Todas as atividades que tenham por objetivo "estressar" telas e funcionalidades de um sistema informatizado podem ser categorizadas na dimensão da garantia da qualidade do produto tecnológico (BARTIé, 2002).

A ISO/IEC 25010:2011 [6] define um modelo de qualidade de produto detalhado, sendo uma das normas Software product Quality Requirements and Evaluation (SQuaRE). É utilizado uma estrutura hierárquica de características de qualidade, o qual é descrito o que se espera de um produto de software. Nessa norma são definidos os conceitos de qualidade em uso e qualidade do produto.

A SQuaRE constitui uma estrutura para garantir que todas as características de qualidade sejam consideradas, sendo dividida em três modelos de qualidade: modelo de

[1] <https://www.iso.org/standard/74348.html>
[2] <https://www.iso.org/standard/63712.html>
[3] <https://www.iso.org/standard/54175.html>
[4] <https://cmmiinstitute.com>
[5] <https://softex.br/mpsbr/>
[6] <https://www.iso.org/standard/35733.html>

qualidade em uso, modelo de qualidade do produto e o modelo de qualidade de dados presente na ISO/IEC 25012 (ISO, 2008b)[7].

De acordo com a ISO/IEC 25010 (2011), a qualidade do produto é categorizada em características e subcaracterísticas, como é mostrado na Figura 2.1.

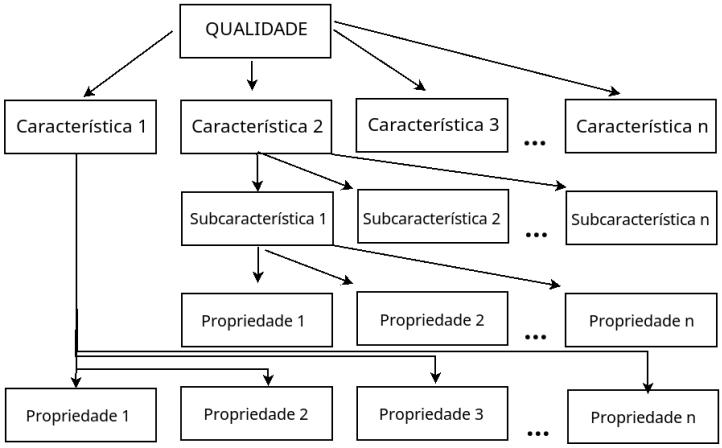


Figura 2.1 Estrutura do modelo de qualidade. ISO/IEC 25010:2011

O modelo de qualidade em uso define cinco características: efetividade, eficiência, satisfação, ausência de risco e cobertura de contexto. A qualidade em uso caracteriza o impacto que o produto tem sobre o usuário.

O modelo de qualidade do produto caracteriza as propriedades do produto em oito características: adequabilidade funcional, eficiência do desempenho, compatibilidade, usabilidade, confiabilidade, segurança, portabilidade e manutenibilidade. Essa características estão descritas a seguir:

- **Adequabilidade Funcional** - Representa o grau em que um produto ou sistema forneceu as funções que atendem às necessidades anteriormente declaradas e implícitas quando usadas em condições especificas. Sub-características: completude funcional, correção funcional, adequação funcional.

- **Eficiência de Desempenho** - Representa o desempenho em relação ao número de recursos usados nas condições definidas. Sub-características: comportamento temporal, utilização de recursos, capacidade.

- **Compatibilidade** - Grau de troca de informações e/ou execução das suas funções entre os produtos, sistemas ou componentes, enquanto compartilham do mesmo ambiente de software e hardware. Sub-características: co-existência, interoperabilidade.

- **Usabilidade** - Grau de uso dos usuários de um produto ou sistema para alcançar metas especificas com eficácia, eficiência e satisfação em um contexto especifico de

[7] <https://www.iso.org/standard/35736.html>

uso. Sub-características: reconhecimento de adequabilidade, aprendizagem, operacionalidade, proteção contra erros do usuário, estética da interface do usuário, acessibilidade.

- **Confiabilidade** - Grau que durante um período de tempo um produto, sistema ou componente executa funções específicas sob condições específicas. Sub-características: maturidade, disponibilidade, tolerância à falhas, recuperabilidade.

- **Segurança** - Grau para o qual um sistema ou produto protege dados e informações para que pessoas/produtos/sistemas tenham o nível de acesso e autorização dos dados que sejam autorização. Sub-características: confidencialidade, integridade, não-repúdio, autenticidade, prestação de contas.

- **Manutenibilidade** - Grau de eficácia e eficiência com o qual um produto/sistema pode ser melhorado, corrigido e adaptável as mudanças nos requisitos e no ambiente. A norma NBR ISO/IEC 9126-1[8] descreve: "Capacidade do produto de software de ser modificado. As modificações podem incluir correções, melhorias ou adaptações do software devido a mudanças no ambiente e nos seus requisitos ou especificações funcionais."

- **Portabilidade** - Grau de eficiência/eficácia que um sistema pode ser transferido de ambiente (hardware, software, ambiente operacional). Sub-características: adaptabilidade, capacidade de ser instalado (instalabilidade), capacidade para substituir (substitubilidade).

As características definidas pelos modelos de qualidade de uso e qualidade de produto são relevantes para todos os produtos de software e sistemas de computador. As características e sub-características fornecem uma terminologia consistente para especificar, medir e avaliar a qualidade do sistema de software. Elas também fornecem um conjunto de características de qualidade com relação às quais os requisitos de qualidade declarados podem ser comparados.

Embora o escopo do modelo de qualidade do produto seja o software e os sistemas de computador, muitas das características também são relevantes para sistemas e serviços mais amplos. A ISO/IEC 25012:2008[9] contém um modelo para qualidade de dados que é complementar ao modelo ISO/IEC 25010:2011[10]. O escopo dos modelos exclui propriedades puramente funcionais, mas inclui adequação funcional.

O escopo de aplicação dos modelos de qualidade inclui suporte à especificação e avaliação de software e sistemas de computação intensivos de software de diferentes perspectivas por aqueles associados com sua aquisição, requisitos, desenvolvimento, uso, avaliação, suporte, manutenção, controle e garantia de qualidade e auditoria.

Os modelos podem, por exemplo, ser usados por desenvolvedores, adquirentes, equipe de controle e garantia de qualidade e avaliadores independentes, particularmente aqueles

[8] <https://aplicacoes.mds.gov.br/sagirmps/simulacao/sum_executivo/pdf/fichatecnica_21.pdf>
[9] <https://www.iso.org/standard/35736.html>
[10] <https://www.iso.org/standard/35733.html>

responsáveis por especificar e avaliar a qualidade do produto de software. De acordo com a ISO/IEC 25010:2011[11], as atividades durante o desenvolvimento do produto que podem se beneficiar do uso dos modelos de qualidade incluem:

- identificação de requisitos de software e sistema;
- validar a abrangência de uma definição de requisitos;
- identificar objetivos de design de software e sistema;
- identificar objetivos de teste de software e sistema;
- identificar critérios de controle de qualidade como parte da garantia da qualidade;
- identificar critérios de aceitação para um produto de software e / ou sistema de computador com uso intensivo de software;
- estabelecimento de medidas de características de qualidade em apoio a essas atividades.

Esta seção introduziu aspectos inerentes à qualidade de software, em particular no que diz respeito às duas preocupações de qualidade que, embora distintas, possuem similar importância: qualidade do processo e qualidade do produto, realizando uma ênfase na qualidade do produto e suas oito características, com o advento dos *code smells* e *test smells* que descreveremos mais adiante nesta seção. Uma das etapas para alcançar a qualidade do software é a realização de testes. A próxima seção se dedicará a apresentar conceitos fundamentais sobre a área de testes de software.

2.2 TESTE DE SOFTWARE

Testar um software é uma estratégia eficiente para identificar defeitos ainda no processo de desenvolvimento e evitar que os usuários finais se deparem com tais defeitos. A finalidade do teste é definida em Hirama (2012) como um processo em que os aspectos estruturais, lógicos e sistêmicos do software são verificados com o objetivo de identificar defeitos. Teste é um dos pilares da qualidade de software, e de acordo com Guerra e Colombo (2009) "teste de software é a atividade de executar um software com o objetivo de encontrar diferenças entre os resultados obtidos e os resultados esperados.". Em outras palavras, o teste busca por defeitos no software e por aspectos que não estão corretos com a descrição da documentação prescrita.

De acordo com Sommerville (2011), de muitas formas, o teste é um processo individualista, e o número de tipos diferentes de testes varia tanto quanto as diferentes estratégias de desenvolvimento. Por muitos anos, nossas únicas defesas contra os erros de programação eram um projeto cuidadoso e a inteligência do programador. Estamos agora em uma era na qual as modernas técnicas de projeto e revisões técnicas nos ajudam a reduzir a

[11] <https://www.iso.org/standard/35733.html>

quantidade de erros iniciais inerentes ao código. Da mesma forma, diferentes métodos de teste estão começando a se agrupar em várias abordagens e filosofias distintas.

De acordo com Sommerville (2011), o processo de teste tem dois objetivos distintos:

- Demonstrar ao desenvolvedor e ao cliente que o software atende a seus requisitos;
- Descobrir situações em que o software se comporta de maneira incorreta, indesejável ou de forma diferente das especificações.

2.2.1 Abordagens de Testes

Existe duas abordagens para testes: a abordagem funcional ("black box" ou "caixa preta") e a abordagem estrutural ("White box" ou "caixa branca") (SOMMERVILLE, 2011), conforme descrição a seguir:

- Funcional (Caixa Preta): não importa o código interno do programa ou a tecnologia utilizada, o testador visualiza o software como uma caixa preta. O objetivo é considerar somente os dados de entrada e observar os dados de saída gerados, verificando se estão de acordo com o esperado.
- Estrutural (Caixa Branca): nesta abordagem o testador está interessado no que acontece dentro do código, avalia as funcionalidades internas dos componentes, com base no código fonte, com foco nas estruturas de controle interno e dados, o testador valida os caminhos lógicos possíveis.

2.2.2 Níveis dos Testes

Normalmente a execução dos teste de software são realizados durante o ciclo de vida do desenvolvimento do software em diferentes estágios. Quatro níveis são conhecidas de acordo com o objetivo principal do teste (GRAHAM et al., 2008; MYERS; SANDLER, 2004):

Teste de Unidade. Os testes de unidade priorizam as menores unidades dos projetos de software, que podem ser componentes ou módulos do sistema. Neste tipo de teste, é usado como guia a descrição do projeto a nível de componente, e os caminhos de controle são testados para descobrir erros dentro dos limites do módulo. A complexidade relativa dos testes e os erros que revelam são limitados pelo escopo do teste de unidade. Esse teste evidencia o fluxo da lógica interna e os dados dentro nos limites do componente. Pode ser realizado em paralelo para diversos componentes (GRAHAM et al., 2008).

Naik e Tripathy (2018) descrevem o fluxo dos testes de unidade da seguinte forma: os programadores testam unidades de software individuais, como procedimentos, funções, métodos ou classes, isoladamente. Depois de garantir que as unidades individuais funcionem de forma satisfatória, os módulos são montados para construir subsistemas maiores, seguindo as técnicas de teste de integração.

Teste de Integração. As interfaces entre os componentes e as unidades que foram testadas individualmente no estágio anterior são testadas de forma integrada. Para realizar a integração, os componentes são adicionados um a um e, após cada passo, um teste é necessário (teste incremental). Com a vantagem de antecipar a detecção de defeitos e corrigi-los mais rapidamente na etapa de teste, facilitando a identificação dos erros. A desvantagem é o custo elevado para testar cada componente adicionado, já que vai ser necessário um novo teste. Há duas formas de se realizar o teste de integração: Bottom-up e Top-down. Na primeira os testes começam na parte mais básica, como os testes de unidade, até o nível mais alto, como os testes de interface. Na segunda os testes são realizados de forma inversa (GRAHAM et al., 2008).

De acordo com Naik e Tripathy (2018), o teste de integração é realizado em conjunto por desenvolvedores de software e engenheiros de teste de integração.

Teste de Validação/Aceitação. O usuário é o responsável pela realização deste teste. O propósito é determinar se o sistema está de acordo com os requisitos do usuário, estando relacionado com a validação do mesmo. Os testes de aceitação podem ser realizados de duas formas: testes alfa e testes beta. Os primeiros são realizados nas instalações do desenvolvedor, que observa o funcionamento do sistema a partir da utilização dos usuários, anotando os problemas identificados. Por sua vez os testes beta são realizados no ambiente real de trabalho do usuário, que instala o sistema e testa, sem a presença do desenvolvedor, o qual registra os problemas encontrados para o envio aos desenvolvedores para correção (GRAHAM et al., 2008).

Naik e Tripathy (2018) descrevem os testes de aceitação quando o cliente realiza sua própria série de testes. O objetivo do teste de aceitação é medir a qualidade do produto, em vez de procurar os defeitos, o que é objetivo do teste do sistema. Uma noção fundamental no teste de aceitação é a expectativa do cliente em relação ao sistema. Durante o teste de aceitação, o cliente deve ter desenvolvido seus critérios de aceitação com base em suas próprias expectativas do sistema. Naik e Tripathy (2018) descrevem dois tipos de testes de aceitação, conforme explicado a seguir:

- *Business Acceptance Testing* (BAT): É realizado dentro da organização de desenvolvimento do fornecedor. A ideia de ter um BAT é garantir que o sistema acabará por passar no teste de aceitação do usuário. É um ensaio da *User Acceptance Testing* (UAT) nas instalações do fornecedor.

- UAT: O teste de aceitação do usuário é conduzido pelo cliente, para garantir que o sistema satisfaz os critérios de aceitação contratual, satisfazendo as necessidades do usuário;

Teste de Sistema. Testes de sistema conta com uma infra-estrutura mais complexa de hardware (se aproximando do ambiente de produção). Devem concentrar-se mais nos aspectos de performance, segurança, disponibilidade, instalação, recuperação, entre outros. Nesse nível de testes, determinados aspectos das funcionalidades que não puderam ser testados deverão integrar o planejamento dos testes de sistema, como testes de integra-

ções com sistemas externos (sem simulação), testes com dispositivos físicos (hardware), utilitários externos e operacionalização do ambiente tecnológico (BARTIé, 2002).

Naik e Tripathy (2018) afirmam que o teste do sistema é uma fase crítica em um processo de desenvolvimento de software devido à necessidade de cumprir um cronograma apertado perto da data de entrega, para descobrir a maioria das falhas e para verificar se as correções estão funcionando e não resultaram em novas falhas. O teste do sistema compreende várias atividades distintas: criar um plano de teste, projetar um conjunto de testes, preparar ambientes de teste, executar os testes seguindo uma estratégia clara e monitorar o processo de execução do teste.

2.3 REFATORAÇÃO

Refatoração é uma técnica disciplinada para reestruturar o corpo do código existente, alterando sua estrutura interna sem alterar seu comportamento externo. Seu âmago é uma série de pequenas alterações de comportamentos internos, preservando as transformações que ocorrem externamente. Cada transformação (chamada de "refatoração") faz pouco, mas uma sequência dessas transformações pode produzir uma reestruturação significativa. Como cada refatoração é pequena, é menos provável que dê errado. O sistema é mantido em funcionamento após cada refatoração, reduzindo as chances de um sistema ser seriamente quebrado durante a reestruturação. A oportunidade de realizar a refatoração no código é chamado de code smell (FOWLER, 1999).

2.3.1 Code Smell

O termo code smell foi descrito pela primeira vez por Kent Beck enquanto ajudava Martin Fowler na obra *Refactoring* (FOWLER, 1999). Fowler (1999), em conjunto com Kent Beck, levantou o problema de quando fazer a refatoração do código, chegando a uma analogia com *bad smell*, os quais são códigos que podem gerar possíveis problemas no futuro.

Um trabalho empírico recente mostrou como os *bad smells* acompanham o código que tem uma probabilidade maior de precisar ser alterado para corrigir uma falha (KHOMH et al., 2011).

Esses problemas são exacerbados quando o código-fonte contém combinações de diferentes *bad smells* (ABBES et al., 2011). Os *smells* podem apresentar-se no código de várias formas. A seguir é apresentada a classificação definida por Fowler (1999):

- ***Duplicated Code***: É o *code smell* mais comum. O *smell* acontece quando se tem replicação de um determinado código em um lugar ou mais. O melhor meio de refatorar é extrair o código para um local que possa ser acessado pelas partes que irão utiliza-lo;

- ***Long Method***: Os programas orientados a objeto que vivem por mais tempo e com qualidade são aqueles com métodos curtos. Métodos que acumulam tarefas, dificultando sua compreensão são grandes candidatos a refatoração;

- ***Large Class***: Uma classe que faz muita coisa, com muitas variáveis de instância, é uma ótima candidata a refatoração. Por exemplo, uma classe com 10 métodos grandes pode se tornar uma classe com 10 métodos menores, 5 métodos auxiliares. A possibilidade de códigos repetidos com classes grande é maior;

- ***Long Parameter List***: Métodos com muitos paramentros de entrada são difíceis de entender. A ideia é passar somente as informações absolutamente necessárias ao método para o seu funcionamento;

- ***Divergent Change***: Uma alteração divergente é quando uma classe é frequentemente alterada de diferentes maneiras por diferentes razões. Esse *smell* indica a necessidade de particionar as regras de negócio da classe em uma nova classe ou mais;

- ***Shotgun Surgery***: De certa forma o oposto do *smell* anterior, a "cirurgia com rifle" é quando é necessário alterar várias classes para realizar uma mudança. Quando as alterações são difíceis de encontrar, acaba sendo fácil não realizar alguma das alterações necessárias;

- ***Feature Envy***: É um método que está interessado mais em dados de outras classes do que na sua própria classe;

- ***Data Clumps***: Se temos um conjuntos de dados e sempre aparecem juntos, e de certa forma acabam sendo dependentes uns dos outros, essa é uma boa possibilidade de realizar uma refatoração, e talvez criar uma classe com esses dados;

- ***Primitive Obsession***: A maioria dos ambientes de programação tem dois tipos de dados: os tipos registros e os tipos primitivos. Os primitivos são os blocos de construção dos tipos registros. A ideia aqui é utilizar as possibilidades de criar seus próprios tipos registros, especializando os dados em classes apropriadas para tal. E como base, utilizar os tipos primitivos para estruturar seus registros;

- ***Switch Statements***: O problema principal do *switch* são os códigos duplicados, normalmente os comandos *switch*(case) ficam espalhados pelo código, e quando é necessário realizar alguma alteração, como por exemplo uma nova opção de um *enum*, teremos que alterar em todos os *switch* que precisem desse novo fluxo. Uma solução é a utilização de polimorfismo para substituir o *switch*;

- ***Parallel Inheritance Hierarchies***: Cada vez que for necessário criar uma subclasse de uma classe, tem que criar uma subclasse de outra. Nesse caso as classes são de certa forma dependentes. É um caso especial de *Shotgun Surgery*;

- ***Lazy Class***: É uma classe que não está realizando muitas tarefas, de modo que não necessariamente deveria ser uma classe, e seus métodos poderiam estar melhor alocados em outros locais, não havendo sua necessidade;

- ***Speculative Generality***: Quando um código é desenvolvido especulando sua necessidade em um futuro próximo (ou não). Normalmente é de fácil detecção, pois somente as classes de testes o utilizam. Esse *smell* dificulta a legibilidade do código, já que o mesmo não está sendo utilizado em produção. Sua refatoração é simples, basta apagar, tanto o código, quando sua classe de teste, já que a mesma não será necessária;

- ***Temporary Field***: Por padrão, os campos em um objeto tem que ter significado e objetivo. Assim, quando há um campo e não sabemos o porquê de sua existência, trata-se de um *smell*. Campos temporários dificultam o entendimento da classe e a lógica por trás dele. Quando o mesmo ocorre é um candidato a refatoração;

- ***Message Chains***: Quando temos uma cadeia de requisições (métodos `getThis`) em cadeia, um objeto chama outro objeto que chama outro objeto e assim por diante. Se for necessário alterar um método pai, todas as requisições filhas terão que ser alteradas também, além de dificultar a manutenção e legibilidade das regras de negócio vinculadas a essa cadeia;

- ***Middle Man***: Quando uma classe tem em sua maioria métodos que redirecionam as requisições para métodos de outras classes, transformando a classe atual como uma fachada. Essa delegação dificulta o entendimento do código, obrigando o desenvolvedor navegar nas classes para entender o seu funcionamento;

- ***Inappropriate Intimacy***: Classes intimamente ligadas, como por exemplo classes filhas com acesso a propriedades das classes pais. Esse tipo de *smell* diminui a coesão das classes e o controle das suas propriedades;

- ***Alternative Classes with Different Interfaces***: Métodos que fazem a mesma coisa, mas com assinatura diferente. Esse *smell* pode ocorrer na mesma classe ou em classes distintas;

- ***Incomplete Library Class***: A reutilização é muitas vezes citada como motivo da orientação a objeto. O problema desse *smell* é que muitas vezes é ruim, e normalmente impossível, modificar uma biblioteca de classes para realizar algo que gostaríamos que ela fizesse;

- ***Data Class***: São classes que tem atributos e métodos de acesso e alteração desses atributos, os famosos *getters* e *setters*. Para Fowler (1999), "classes de dados são como crianças. Elas são boas como ponto de partida, mas para participarem como métodos maduros, precisam adquirir um pouco de responsabilidade", ou seja, devemos adicionar ao seu corpo métodos que são da sua regra de negócio;

- ***Refused Bequest***: Se a subclasse não utiliza os métodos que a super classe tem, provavelmente a super classe tem métodos que não deveria ter. Esses métodos podem ser utilizados somente por uma subclasse das várias que herdam da super classe. Dessa forma, a refatoração é necessária de modo que esses métodos sejam

inseridos na subclasse. As superclasses devem conter somente os métodos que são comuns a todas as subclasses;

- ***Comments***: Comentários não são ruins ao código. Com tudo, utilizar comentários para explicar um trecho de código é um *smell*, pois ó código deveria ser entendível, sem a necessidade de um comentário.

Fowler (1999) deixa o modo de detectar esses *code smells* de forma subjetiva, mesmo os exemplos sendo escritos em Java, as descrições apresentadas por ele pode ser utilizadas em outras linguagens. Há diversos estudos sobre *code smells*, abrangendo e até expandindo o quantitativo de *code smells* (SHARMA; SPINELLIS, 2018). No que diz respeito a ferramental de apoio, existe suporte à detecção automática de *code smells* no código-fonte, e.g., (FOKAEFS; TSANTALIS; CHATZIGEORGIOU, 2007), que visa facilitar a vida dos programadores e tende a melhorar o código gerado, e por consequência a qualidade do sistema.

Para esse estudo vamos nos aprofundar em alguns tipos específicos de *code smells*, os que podem ser encontrados em códigos de testes, os *test smells*.

2.3.2 Test Smells

De acordo com Deursen et al. (2001), os *code smells* não se limitam ao código-fonte de produção, mas também podem prejudicar os códigos de testes. Neste estudo foi levantado um conjunto distinto de *bad smells* de teste ou simplesmente *test smells*. Da mesma forma que os *code smells*, os *test smells* são sintomáticos e podem diminuir a qualidade dos sistemas (GAROUSI; KUCUK, 2018). Segundo Deursen et al. (2001), são tipos de *test smells*:

- ***Mystery Guest***: Quando um teste se utiliza de recursos externos, como por exemplo um arquivo necessário a sua execução, esse recurso externo não torna o teste autônomo. Consequentemente, não há informações suficientes para entender a funcionalidade testada, dificultando a utilização desse teste como documentação. Além disso, esses recursos externos podem ser compartilhados. E o seu uso introduz dependências ocultas: se alguma força altera ou exclui esse recurso, os testes começam a falhar. O Anexo A.5 ilustra através de exemplo, esse test smell;

- ***Resource Optimism***: O código de teste que faz suposições otimistas sobre a existência ou ausência de um determinado recurso externo, e o estado deste recursos externos (como diretórios particulares ou tabelas de banco de dados) pode causar um comportamento não determinístico nos resultados do teste. A situação em que os testes rodam bem em um momento e falham em outro não é uma situação que o teste deve ocorrer. O Anexo A.6 ilustra através de exemplo, esse test smell;

- ***Test Run War***: Este teste smell surge quando os testes rodam bem, contanto que você seja o único a testar, mas falhe quando mais programadores os executam. Isso é provavelmente causado por interferência de recursos: alguns testes no seu conjunto alocam recursos, como arquivos temporários que também são usados por outros;

- ***General Fixture***: No framework JUnit, um programador pode escrever um método *setUp* que será executado antes de cada método de teste para criar um ambiente para os testes serem executados. O test smell se torna evidente quando o ambiente *setUp* é muito geral e os testes necessitam apenas de parte dessa configuração. Os métodos *setUp* começam a ficar grandes e seu entendimento é reduzido, e com o acréscimo de funções, começa a ficar lento. O perigo de ter teste que leve muito tempo para ser concluído, interferindo com o processo de desenvolvimento, incentiva os programadores a não execução dos testes. O Anexo A.3 ilustra através de exemplo, esse test smell;

- ***Eager Test***: Quando um método de teste verifica vários métodos do objeto a ser testado, é difícil ler e entender e, portanto, é mais difícil usá-lo como documentação. Além disso, torna os testes mais dependentes uns dos outros e mais difíceis de manter. O Anexo A.2 ilustra através de exemplo, esse test smell;

- ***Lazy Test***: Esse test smell ocorre quando vários métodos de teste verificam o mesmo método usando o mesmo equipamento (mas, por exemplo, verificam os valores de diferentes variáveis de instância). Esses testes geralmente só têm significado quando são considerados juntos. O Anexo A.4 ilustra através de exemplo, esse test smell;

- ***Assertion Roulette***: Este test smell ocorre quando o método de teste tem uma série de *assertions* sem uma descrição. Se um *assertion* falhar, não se sabe qual deles gerou a falha e seu motivo. O Anexo A.1 ilustra através de exemplo, esse test smell;

- ***Indirect Testing***: Uma classe de teste deve testar sua contraparte no código de produção. O test smell ocorre quando uma classe de teste contém métodos que realmente realizam testes em outros objetos (por exemplo, porque há referências a eles na classe a ser testada). O fato de esse Smell surgir também indica que pode haver problemas com a ocultação de dados no código de produção.

- ***For Testers Only***: Quando uma classe de produção contém métodos que são usados apenas por métodos de teste, esses métodos não são necessários e podem ser removidos, ou são necessários apenas para configurar uma estrutura para o teste. Criar uma subclasse separada ajuda a lidar com esse problema;

- ***Sensitive Equality***: É rápido e fácil escrever verificações de igualdade usando *string*. Uma maneira típica é calcular um resultado real, mapeá-lo para uma string, que é então comparada a uma string literal que representa o valor esperado. Tais testes, no entanto, podem depender de muitos detalhes irrelevantes, como vírgulas, citações, espaços, etc. Sempre que uma *string* é alterada, os testes começam a falhar. A solução é substituir as verificações de igualdade que utiliza *string* por verificações reais de igualdade;

- ***Test Code Duplication***: O código de teste pode conter duplicação indesejada. Em particular, as partes que configuram auxiliares de teste são suscetíveis a esse

problema. O caso mais comum de código de teste será a duplicação de código na mesma classe de teste. Para duplicação nas classes de teste, pode ser útil espelhar a hierarquia de classes do código de produção na hierarquia de classes de teste. Um caso especial de duplicação de código é a implicação do teste: os testes A e B cobrem o mesmo código de produção, e A falha se e somente se B falhar. Um exemplo típico ocorre quando o código de produção é refatorado: antes dessa refatoração, A e B cobriam códigos diferentes, mas depois eles lidam com o mesmo código e não é mais necessário manter os dois testes.

São apresentados novos test smells no estudo de Meszaros, Smith e Andrea (2003), os quais apresentamos a seguir:

- ***Can't See the Forest for the Trees***: Possui muito código de teste, o qual dificulta entender o quê o teste está verificando. Os testes não agem como uma especificação porque demoram muito para serem entendidos;

- ***Complex Undo Logic***: O código complexo de desmontagem do teste aumenta a probabilidade de deixar o ambiente de teste corrompido por não ser limpo corretamente. Isso resulta em "vazamentos de dados" que, posteriormente, podem fazer com que esse ou outros testes falhem sem motivo aparente;

- ***Complex Test Code/Verbose Test***: Excesso de código de teste ou Lógica de Teste Condicional (Conditional Test Logic). Difícil verificar a exatidão dele e mais propenso a conter erros;

- ***Conditional Test Logic***: Testes contendo lógica condicional (instruções IF ou loops). O Anexo A.7 ilustra através de exemplo, esse test smell;

- ***Magic Numbers ou Hard Coded Test Data***: Muitos "Números Mágicos" ou Strings usados ao criar objetos que provavelmente resultam em um teste irrepetível. O Anexo A.14 ilustra através de exemplo, esse test smell;

- ***Fragile Tests***: Toda vez que você altera o *System Under Test* (SUT), os testes não são compilados ou eles falham. Você precisa modificar muitos testes para tornar as coisas "verdes" novamente. Isso aumenta muito o custo de manutenção do sistema;

- ***Fragile Fixture***: Os testes começam a falhar quando um dispositivo compartilhado é modificado (por exemplo, novos registros são colocados no banco de dados). Isso ocorre porque os testes estão fazendo suposições sobre o conteúdo do dispositivo compartilhado;

- ***Interdependent Tests/Dependent Test***: Quando um teste falha, vários outros testes falham sem motivo aparente, porque dependem dos efeitos colaterais de um teste executado anteriormente. Os testes não podem ser executados sozinhos e são difíceis de manter;

- ***Unrepeatable Tests***: Os testes não podem ser executados repetidamente sem intervenção manual. Isso é causado por testes que não são limpos após sua execução e impedem que eles ou outros testes sejam executados novamente.

Peruma et al. (2019) também apresentaram alguns novos tipos de test smells são eles:

- ***Constructor Initialization***: Métodos de teste que apresentam um construtor. Idealmente, o conjunto de testes não deve ter um construtor. A inicialização dos campos deve estar no método setUp(). Os desenvolvedores que desconhecem o objetivo do método setUp() permitiriam esse test smell criando um construtor para o conjunto de testes. O Anexo A.8 ilustra através de exemplo, esse test smell;

- ***Default Test***: Por padrão, o uma IDE cria classes de teste padrão quando um projeto é criado. Essas classes de teste de modelo devem servir como um exemplo para os desenvolvedores ao escrever testes de unidade e devem ser removidas ou renomeadas. A presença desses arquivos no projeto fará com que os desenvolvedores comecem a adicionar métodos de teste a esses arquivos, tornando a classe de teste padrão um contêiner de todos os casos de teste e violando as boas práticas de teste. Os problemas também surgiriam quando as classes precisassem ser renomeadas no futuro. O Anexo A.9 ilustra através de exemplo, esse test smell;

- ***Duplicate Assert***: Esse smell ocorre quando um método de teste testa a mesma condição várias vezes no mesmo método de teste. O Anexo A.10 ilustra através de exemplo, esse test smell;

- ***Empty Test***: Métodos de teste que não contêm instruções executáveis. O Anexo A.11 ilustra através de exemplo, esse test smell;

- ***Exception Handling***: Esse test smell ocorre quando a aprovação ou reprovação de um método de teste depende explicitamente do método de produção que gera uma exceção. O Anexo A.12 ilustra através de exemplo, esse test smell;

- ***Ignored Test***: A partir do JUnit 4 é fornecido aos desenvolvedores a capacidade de impedir a execução de métodos de teste. No entanto, esses métodos de teste ignorados resultam em sobrecarga no que diz respeito ao tempo de compilação e um aumento na complexidade do código e no tempo de compreensão. O Anexo A.13 ilustra através de exemplo, esse test smell;

- ***Redundant Print***: As instruções de impressão nos testes de unidade são redundantes, pois os testes de unidade são executados como parte de um script automatizado. Consome recursos ou aumentar o tempo de execução. O Anexo A.15 ilustra através de exemplo, esse test smell;

- ***Redundant Assertion***: Esse smell ocorre quando os métodos de teste contêm declarações de asserção sempre verdadeiras ou falsas. Um teste destina-se a retornar um resultado binário, independentemente de o resultado pretendido estar correto ou não, e não deve retornar a mesma saída, independentemente da entrada. O Anexo A.16 ilustra através de exemplo, esse test smell;

- ***Sleepy Test***: Os desenvolvedores introduzem esse test smell quando precisam pausar a execução de instruções em um método de teste por um certo período e continuar a execução. O Anexo A.17 ilustra através de exemplo, esse test smell;
- ***Unknown Test***: Um método de teste sem uma condição de *assertion*, o teste sempre vai resultar como válido, não resultando em uma exceção. Essa prática de programação dificulta a compreensibilidade do teste. O Anexo A.18 ilustra através de exemplo, esse test smell;

2.4 FERRAMENTAS DE GERAÇÃO DE TESTE

De acordo com Somé e Cheng (2008), a utilização de ferramentas de geração automática de testes de unidade é essencial, trazendo os benefícios de reduzir o esforço de codificação e garantir a eficácia dos testes. Entretanto a geração de testes esbarra nas possibilidades de entradas de dados que são possíveis. Para conseguir assegurar que um software não possui defeitos, seria necessário testar todas as possibilidades de entrada do mesmo, até atingir o grau de qualidade desejado, algo que é impraticável.

Para exemplificar, considere um método em Java que recebe 2 parâmetros, do tipo int (32 bits). As possibilidades de entrada do método é de 2^{64}. Supondo que seja executado em um processador com capacidade de 2^{40} instruções por segundo, o tempo de geração de dados para todas as possibilidades seria de $2^{64}/2^{40} = 2^{24}$, ou seja, 194 dias para percorrer todas as entradas de um simples método.

A partir do exemplo podemos averiguar que a maior dificuldade da geração de testes é como criar uma quantidade de dados de entrada para identificar a maior quantidade de defeitos, no menor tempo e com menor custo.

Como uma possibilidade de solução, para o problema de geração de dados de entrada, uma estratégia que atraiu um grande interesse foi a aplicação e a adaptação de algoritmos de busca e otimização (OSMAN; KELLY, 1996), onde se percebeu que os problemas de geração de dados poderiam muitas vezes ser descritos como problemas de otimização.

Dentre os Algoritmos de Otimização, os algoritmos genéticos ganharam destaque para o propósito de geração de dados de entrada dos testes. Os algoritmos genéticos são os principais recursos para a geração automática de dados de testes, que simulam situações de biologia evolutiva (mutação, hereditariedade, seleção natural, etc.), buscando soluções aproximadas de otimização.

Podemos citar o Evosuite como uma ferramenta que utiliza de algoritmos genéticos para a geração dos dados. Além do Evosuite existe o Randoop que se utiliza de outra metodologia para a geração dos seus dados, aleatoriedade direcionada.

Nesta seção são descritas as ferramentas de geração de testes para a linguagem Java selecionados para o estudo.

2.4.1 EvoSuite - *Automatic Test Suite Generation for Java*

A EvoSuite é uma ferramenta que gera automaticamente casos de teste para classes escritas na linguagem Java utilizando o *framework* JUnit[12]. Ela se utiliza de técnicas

[12]<https://junit.org>

evolucionárias parar cobrir o máximo de possibilidades possíveis, além de utilizar de testes de mutação para criar testes menores, com o intuito de encontrar a maior quantidade de defeitos.

A EvoSuite tem como objetivo simplificar e melhorar a técnica de teste de software. Diversas técnicas de geração de testes são implementadas como execução simbólica dinâmica, buscas hibridas, transformação da testabilidade, pesquisa evolucionária e testes de mutação. A ferramenta começou seu percurso de criação e implementação em 2010, a fim de participar de competições de testes, que analisavam o desempenho (FRASER; ARCURI, 2011).

Em 2015, a EvoSuite teve seu código disponibilizado no GitHub, e atualmente é possível encontrar todas as publicações em relação a ferramenta em seu site[13], assim como a versão mais nova da ferramenta e tutoriais sobre execução e instalação.

A Evosuite tem uma abordagem híbrida, gerando conjuntos de testes otimizados para satisfazer os critérios de cobertura estabelecidos. São sugeridos possíveis oráculos nos testes produzidos, adicionando pequenos e efetivos conjuntos de *assertions* que resumem de maneira concisa o comportamento atual. Esses *assertions* permitem que o desenvolvedor detecte desvios do comportamento esperado e capture o comportamento atual para proteger contra defeitos futuros que quebrem esse comportamento.

As características da Evosuite são: geração de testes JUnit para as classes selecionadas, otimização de diferentes critérios de cobertura, como linhas, ramificações, saídas e testes de mutação. Uma das etapas do processamento da Evosuite é minimizar os testes, apenas os que contribuem para obter cobertura não são retirados. O sistema de geração de classes captura o comportamento real das classes testadas. Os testes são executados em uma estrutura de proteção para evitar operações potencialmente perigosas.

Outro ponto é que a Evosuite utiliza teste de mutação para reduzir o máximo possível o número de testes, mas que por outro lado, garanta o máximo de descoberta de defeitos no programa que está sendo testado.

2.4.2 Randoop - RANDom tester for Object-Oriented Programs

O Randoop gera testes de unidade na linguagem Java utilizando o *framework* JUnit a partir da geração de testes aleatórios direcionados por *feedback*, ou seja, utiliza-se dos *feedbacks* das execuções à medida que as entradas são geradas, evitando desta maneira a redundância de dados (PACHECO; ERNST, 2007).

Essa técnica, aleatoriamente, mas de maneira inteligente, gera sequências de invocações de método/construtor para as classes em teste. O Randoop executa as sequências que foram criadas por ele, usando os resultados da execução para gerar *assertions* que capturam o comportamento do programa que será testado, ou seja, o Randoop cria testes a partir das sequências de código e *assertions* (RANDOOP, 2018).

Como podemos notar na Figura 2.2, o Randoop pode ser usado para dois propósitos: encontrar *bugs* no programa e criar testes de regressão para indicar alterações de comportamento.

[13]<http://www.evosuite.org/>

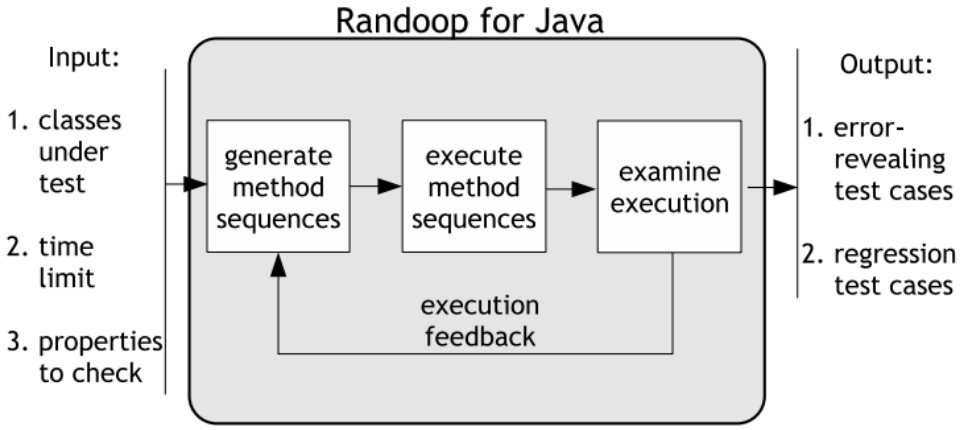


Figura 2.2 Processo do Randoop (PACHECO; ERNST, 2007)

O Randoop possui duas versões: uma para Java[14], desenvolvida por um grupo do MIT – Estados Unidos, e outra para .NET[15], desenvolvida pela Microsoft.

A combinação da geração de testes randômicos e execução de testes no Randoop resulta em uma técnica de geração de testes altamente eficaz. Como exemplo da sua eficácia, em testes executados, o Randoop revelou erros desconhecidos em bibliotecas amplamente utilizadas, incluindo o core do JDK e componentes do .NET (RANDOOP, 2018).

O Randoop possui ampla documentação[16] e código-fonte disponível no github[17]. Também possui alguns desenvolvedores que estão continuamente evoluindo o software como pode ser comprovado no repositório dele no GitHub[18]. Existem diversas publicações sobre a ferramenta (PACHECO; ERNST, 2005; PACHECO; LAHIRI; BALL, 2007; PACHECO; ERNST, 2007; PACHECO; LAHIRI; BALL, 2008), mostrando que o Randoop tem bons resultados em encontrar falhas em diversos programas.

2.5 SÍNTESE DO CAPÍTULO

Neste capítulo apresentamos os conceitos fundamentais que nortearam o desenvolvimento do presente estudo, iniciamos com a explanação sobre a qualidade de software, o qual se divide em qualidade do produto e do processo. Nos aprofundamos na questão da qualidade falamos sobre os testes de software, suas abordagens, estágios, e tipos de testes. Continuamos com o assunto de refatoração de software, e como os códigos podem conter *test smells*, e é aprofundado o assunto descrevendo os *test smells*, que são *code smells* que ocorrem em códigos de testes de unidade. São descritas as ferramentas de geração de

[14]https://github.com/randoop/randoop
[15]https://github.com/abb-iss/Randoop.NET
[16]<https://randoop.github.io/randoop/manual/index.html>
[17]<https://github.com/randoop/randoop>
[18]https://github.com/randoop/randoop/commits/master

testes ”Randoop e Evosuite”e suas características. No próximo capítulo apresentaremos o desenvolvimento da ferramenta responsável pela coleta dos dados.

Capítulo

3

JNOSE TEST - JAVA TEST SMELL DETECTOR

Este capítulo descreve a ferramenta JNose Test, com os estudos utilizados como base e as melhorias implementadas, assim como as bibliotecas utilizadas.

3.1 INTRODUÇÃO

A ferramenta `JNose Test` foi desenvolvida com o intuito de prover detecção automatizada de *test smells* em código de teste desenvolvido na linguagem Java. A Jnose Test é uma extensão da ferramenta Test Smells Detector (tsDetect)[1] (BAVOTA et al., 2015). tsDetect é uma ferramenta desenvolvida em Java, executável em modo terminal. Ela funciona da seguinte maneira: recebe como entrada a lista das classes de testes do projeto e as classes testadas pelas classes de testes e apresenta como saída a lista de *test smells* identificados automaticamente.

Além da tsDetect, a construção da JNose Test utilizou outros dois projetos, que precisam ser executados em sequência: (1) *Test File Detector*[2], responsável por detectar os arquivos de teste do projeto, e (2) *Test File Mapping*[3], responsável por vincular os arquivos de teste com os arquivos que estão sendo testados.

A `JNose Test`[4] é uma refatoração dos três projetos em um. A ferramenta apresenta uma interface gráfica web e inclui a coleta automatizada de métricas de código e de cobertura.

A ferramenta, além de automatizar todo o processo de execução, consegue executar a análise de vários projetos que estejam em um mesmo diretório, gerando somente um arquivo de saída. A seguir, apresentamos uma síntese das *features* implementadas na `JNose Test`, destacando as melhorias em relação ao tsDetect:

- Interface gráfica;

[1]Disponível em <https://testsmells.github.io/>
[2]<https://github.com/TestSmells/TestFileDetector>
[3]<https://github.com/TestSmells/TestFileMapping>
[4]Disponível em: <https://github.com/tassiovirginio/jnosetest>

- Detecção de projetos em subdiretórios;
- Automação do fluxo de entrada e saída de dados em cada etapa;
- Coleta automatizada de métricas de código e de cobertura;
- Retorna o quantitativo de cada tipo de *test smells*;
- Resultado unificado dos projetos selecionados;
- Utilização de *threads* para cada detecção de classe de test, projeto e processo de cobertura. Com o intuito de melhorar a volocidade de processamento.

3.2 ARQUITETURA DA FERRAMENTA JNOSE TEST

A JNose Test foi desenvolvida na linguagem Java e seu funcionamento, como na ferramenta do estudo base, tem seu foco em código Java. Para gerenciar o projeto da ferramenta, foi utilizado o Apache Maven[5], responsável pelo gerenciamento das dependências de bibliotecas, compilação e execução do projeto. A implementação da JNose Test utiliza as seguintes bibliotecas:

- **Apache Wicket**[6]: *Framework* para o desenvolvimento de aplicações Web, desenvolvido na linguagem Java. A versão 8 é a mais estável, as aplicações são em Java, HTML5 e CSS3. Na JNose Test, o Apache Wicket[7] foi utilizado para o desenvolvimento da interface web;
- ***Java Code Coverage Library* (JaCoCo)**[8]: Biblioteca de cobertura de código aberto para Java, criada pela equipe do EclEmma[9] com base nas lições aprendidas do uso e na integração de bibliotecas existentes por muitos anos. Neste projeto, a biblioteca JaCoCo foi utilizada para gerar a cobertura de testes na aplicação;
- **JavaParser**[10]: É um conjunto simples e leve de ferramentas para gerar, analisar e processar o código Java. Atualmente, é utilizado em diversos projetos comerciais e de código aberto. Neste projeto, o JavaParser foi utilizada para a detecção dos *test smells*.

O processo de funcionamento da JNose é realizado utilizando processos paralelos *i.e.*, para cada projeto é criada uma *thread*, e para cada classe de teste uma *thread* é criada para o fluxo de detecção de *test smells* e uma outra *thread* para o fluxo da cobertura de código, que é opcional. A Figura 3.1 apresenta a visão geral de funcionamento da ferramenta `JNose Test`.

[5]<https://wicket.apache.org/>
[6]<https://wicket.apache.org/>
[7]<https://wicket.apache.org/>
[8]<https://www.eclemma.org/jacoco/>
[9]<https://www.eclemma.org/jacoco/>
[10]<https://javaparser.org/>

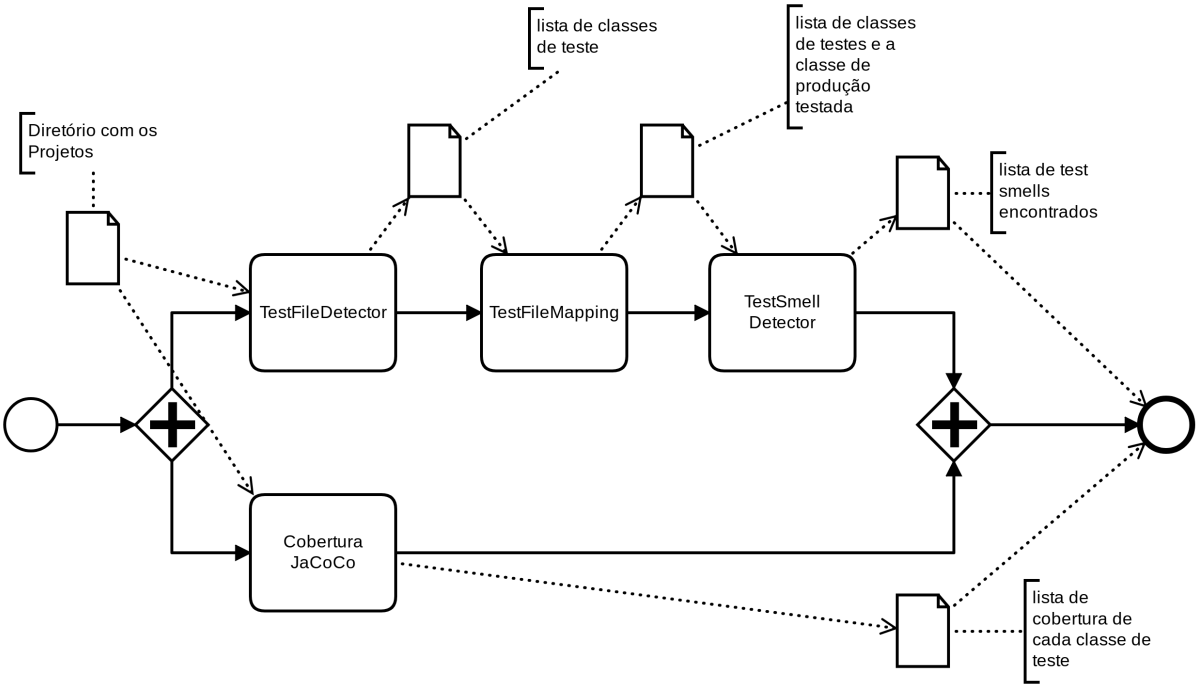


Figura 3.1 Visão Geral de funcionamento da JNose Test

A partir da refatoração dos projetos, criação da interface gráfica e a adição das duas outras bibliotecas supracitadas, as classes com as regras de negócio foram re-organizadas em pacotes de acordo com suas atividades. A estrutura dos pacotes do projeto ficou como mostra a Figura 3.2. Cada pacote tem seu conjunto de classes e é responsável por cada etapa do processo da `JNose Test`, lembrando que não são exibidos os pacotes da camada de interface.

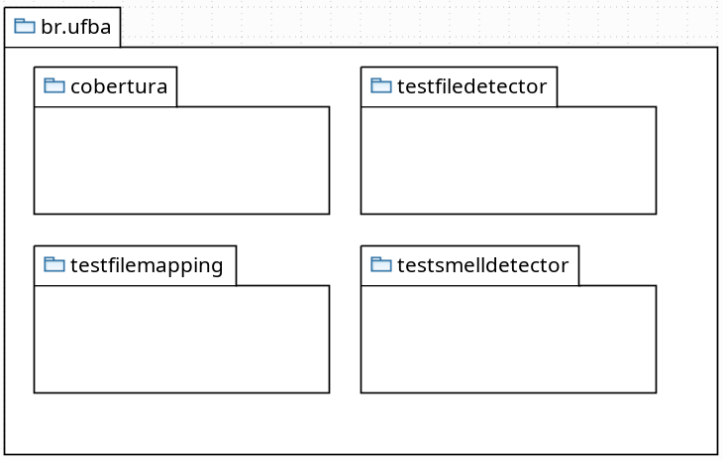


Figura 3.2 Divisão dos pacotes de negócio do sistema

Foram criados os seguintes pacotes:

- **cobertura**: responsável pelas regras de negócio e utilização da biblioteca JaCoCo para conseguir os valores das coberturas de código das classes de testes. Na Figura 3.3, são apresentadas as classes do pacote;
- **testfiledetector**: é o pacote responsável por detectar as classes de teste dos projetos. A Figura 3.4 apresenta as classes do pacote;
- **testfilemapping**: Após a detecção das classes de testes, é necessário saber qual o alvo do teste. As classes do pacote `testfilemapping` são as responsáveis por essa atividade e são apresentadas na Figura 3.5;
- **testsmelldetector**: Após ter os dados das classes de testes e as classes de produção sendo testadas, realiza a análise dos códigos e identifica os *test smells*. Suas classes estão listadas nas Figuras 3.6 e 3.7 (*Smells* identificáveis). Na Figura 3.7, são apresentadas as classes com as regras de detecção dos *test smells*. Para cada *test smell*, encontra-se uma classe de detecção.

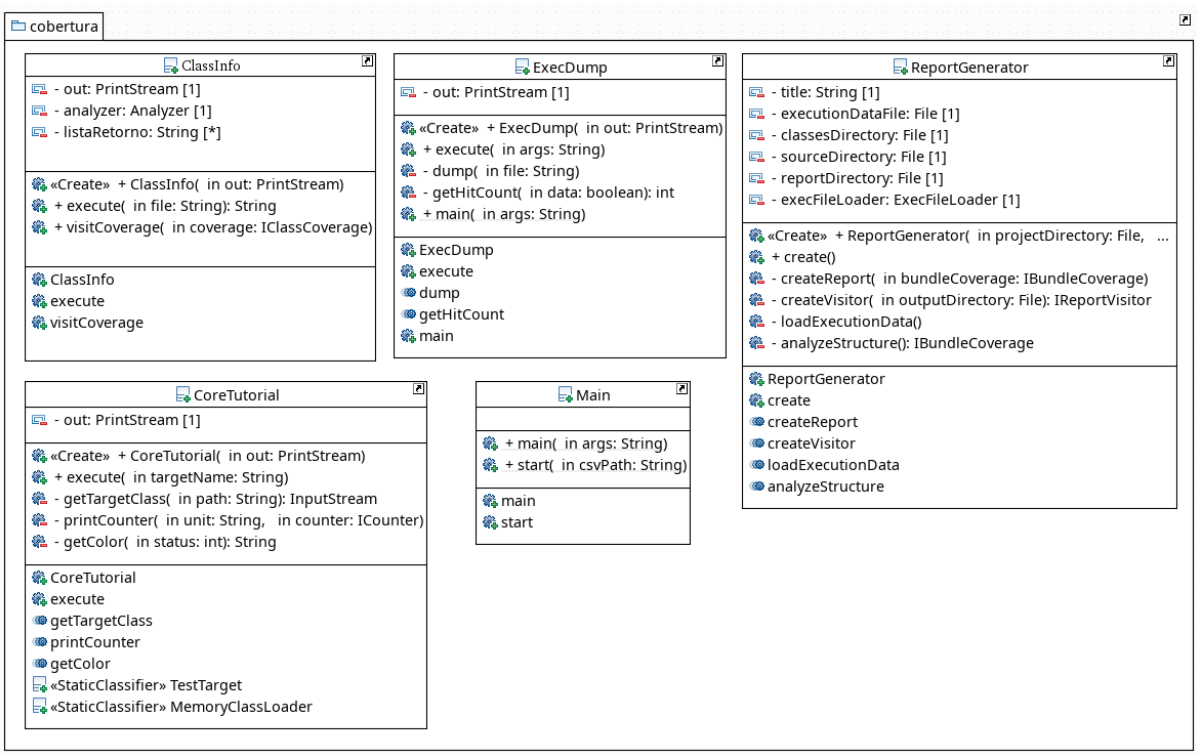


Figura 3.3 Estrutura de classes do Pacote `cobertura`

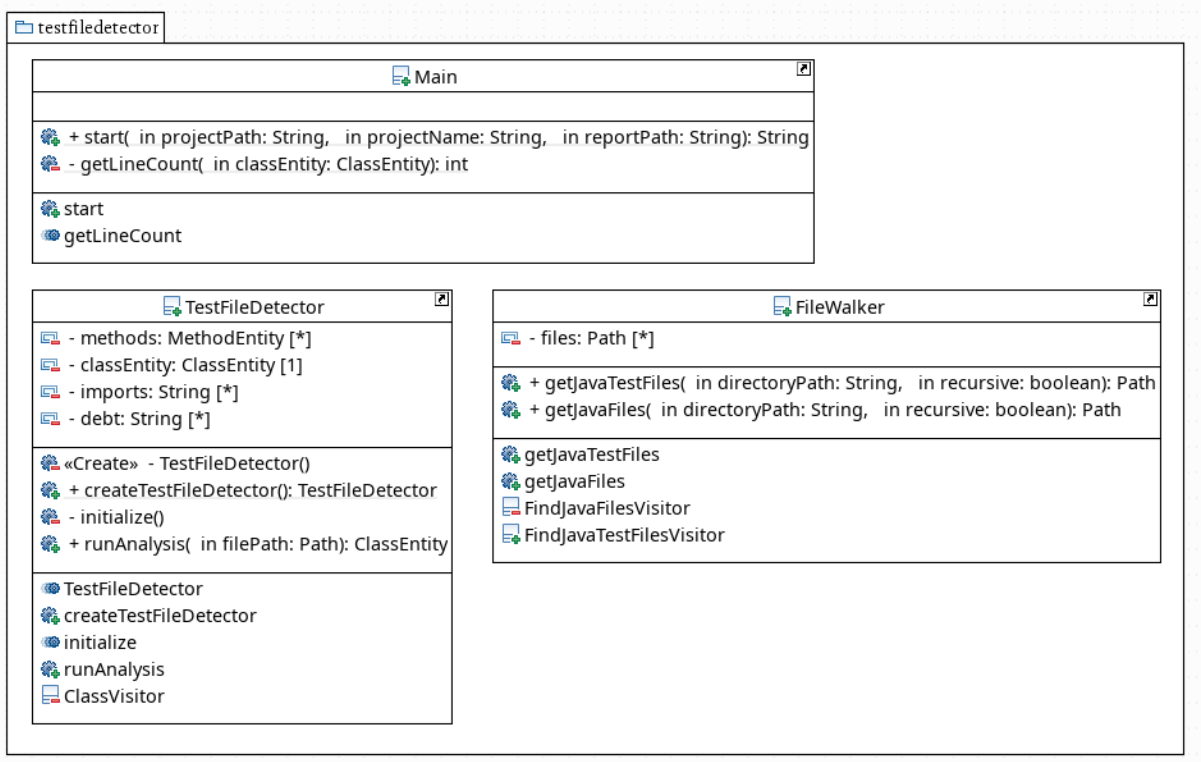


Figura 3.4 Estrutura de classes do Pacote `testfiledetector`

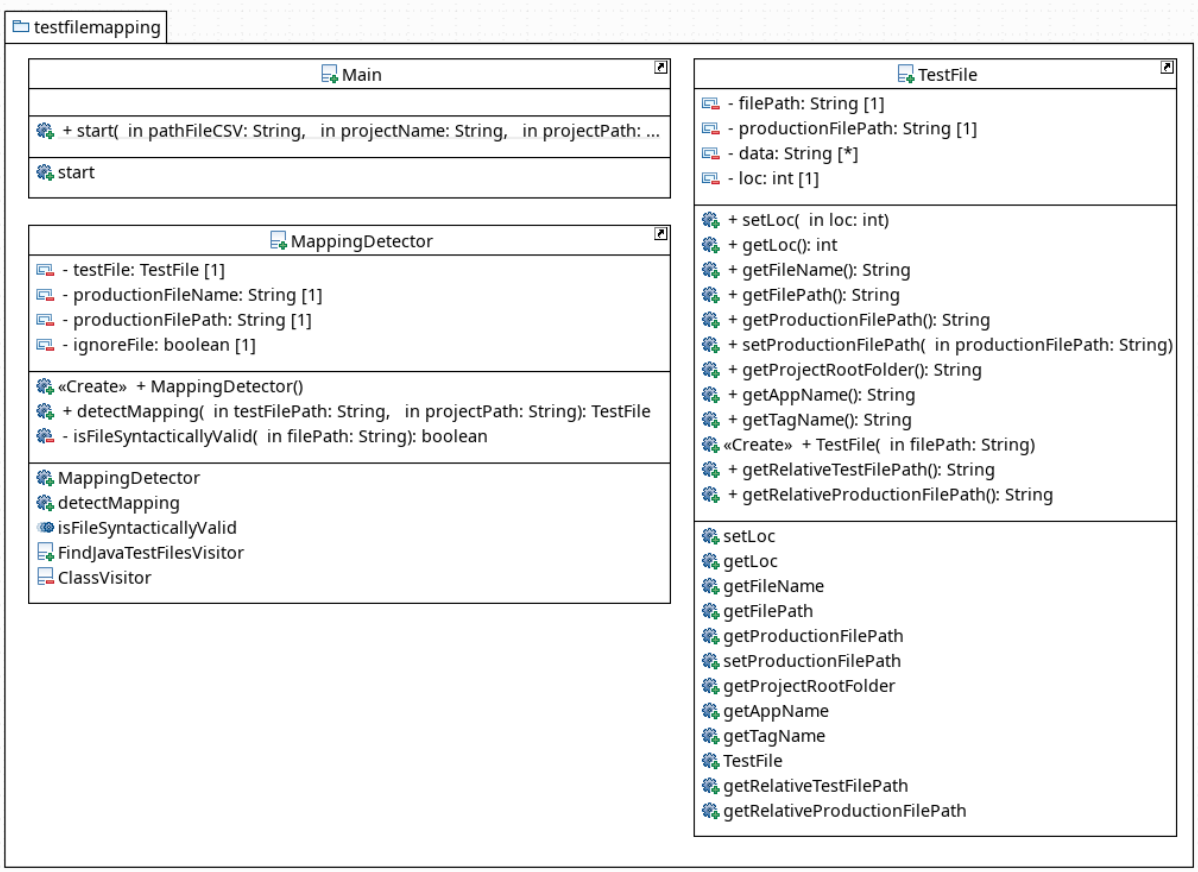


Figura 3.5 Estrutura de classes do Pacote `testfilemapping`

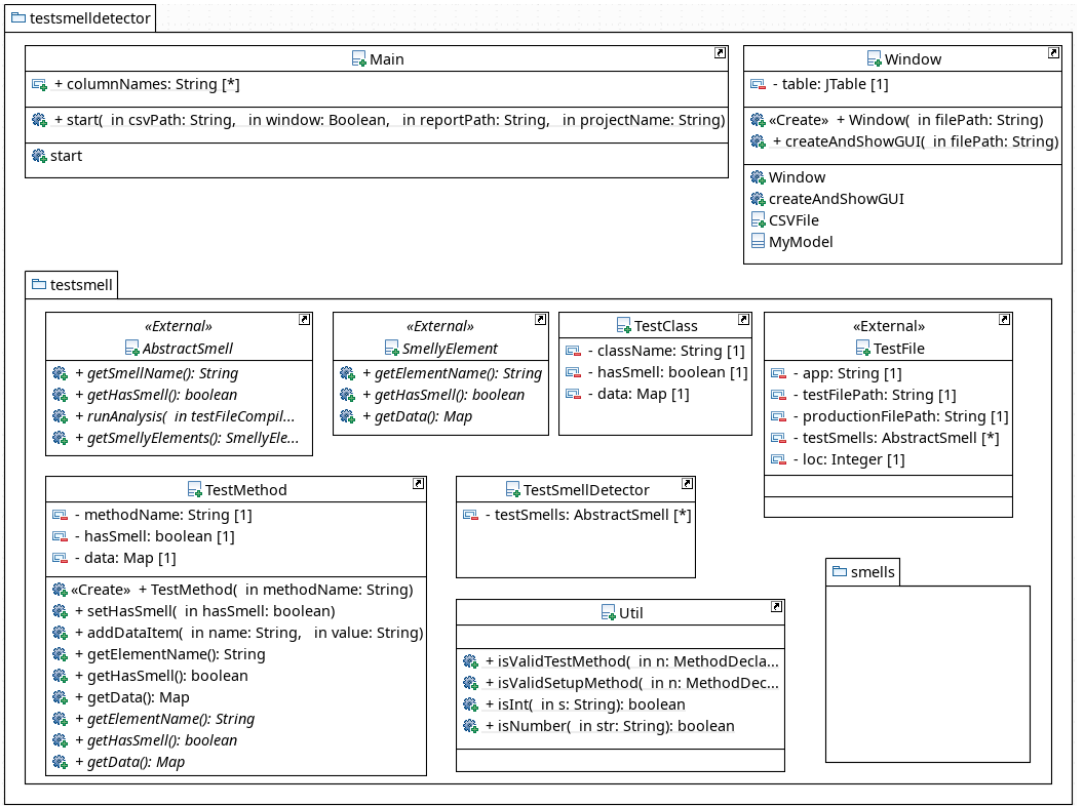


Figura 3.6 Estrutura de classes do Pacote `testsmelldetector`

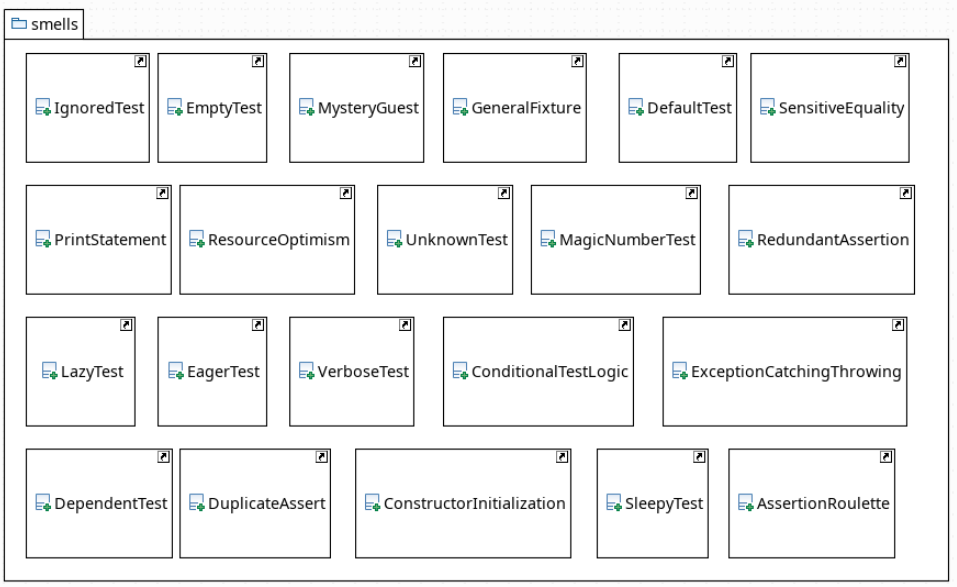


Figura 3.7 Estrutura de classes do Pacote `smells`

3.3 DADOS DE SAÍDA

A JNose Test reduz o tempo de processamento e das etapas manuais e aumenta a quantidade de dados obtidos. Além dos dados gerados no estudo anterior (BAVOTA et al., 2015), foram acrescentados um novo conjunto de dados, tais como, Linhas de Código (LOC), Quantidade de Métodos de Testes (QMT) e dados de cobertura do teste, que foram utilizados no estudo Virgínio et al. (2019).

A Tabela 3.1 lista os dados gerados pela `JNose Test`: metadados do projeto (linhas 1 a 3), os *test smells* identificáveis (linhas 4 a 24), tamanho da classe de teste - LOC (linha 25), quantidade de métodos de testes - QMT (linha 26) e as métricas de cobertura (linhas 27 a 36).

Tabela 3.1 Retorno da execução do JNose Test

#	**Parâmetro**	**Descrição**
1	App	Nome do Projeto
2	TestFileName	Nome da Classe de Teste
3	ProductionFileName	Nome da Classe de Produção
4	Assertion Roulette (AR)	Test Smells
5	Conditional Test Logic (CTL)	Test Smells
6	Constructor Initialization (CI)	Test Smells
7	Default Test (DT)	Test Smells
8	EmptyTest (ET)	Test Smells
9	Exception Catching Throwing/Exception Handling (ECT)	Test Smells
10	General Fixture (GF)	Test Smells
11	Mystery Guest (MG)	Test Smells
12	Print Statement/Redundant Print (PS)	Test Smells
13	Redundant Assertion (RA)	Test Smells
14	Sensitive Equality (SE)	Test Smells
15	Verbose Test/Complex Test (VT)	Test Smells
16	Sleepy Test (ST)	Test Smells
17	Eager Test (ET)	Test Smells
18	Lazy Test (LT)	Test Smells
19	Duplicate Assert (DA)	Test Smells
20	Unknown Test (UT)	Test Smells
21	IgnoredTest (IT)	Test Smells
22	Resource Optimism (RO)	Test Smells
23	Magic Number Test (MNT)	Test Smells
24	Dependent Test/Interdependent Tests (DpT)	Test Smells
25	LOC	Quantidade de Linhas de Teste
26	Number of methods	Quantidade de Métodos de Teste
27	Instruction Missed (Opcional)	Instruções Perdidas
28	Instruction Covered (Opcional)	Instruções Cobertas
29	Branch Missed (Opcional)	Ramos Perdidos
30	Branch Covered (Opcional)	Ramos Cobertos
31	Line Missed (Opcional)	Linhas Perdidas
32	Line Covered (Opcional)	Linhas Cobertas
33	Complexity Missed (Opcional)	Complexidade Perdida
34	Complexity Covered (Opcional)	Complexidade Coberta
35	Method Missed (Opcional)	Métodos Perdidos
36	Method Covered (Opcional)	Métodos Cobertos

3.4 JNOSE TEST

Para facilitar a utilização da ferramenta pelos possíveis usuários do sistema, como por exemplo, analistas, engenheiros de software, testadores, entre outros, foi desenvolvida uma interface web. Essa seção apresenta as telas da JNose Test, e descreve as principais funcionalidades.

A Figura 3.8 apresenta a tela principal da ferramenta, (1) com o campo de texto para o caminho do diretório onde se encontra os projetos a serem analisados e o botão `Selecionar Diretório`. Logo a baixo (2) é onde serão apresentados os projetos detectados automaticamente, os quais poderão ser ou não selecionados para o processamento. Abaixo da lista de projetos encontra-se o botão `Processar` (3). Uma observação necessária é sobre o *checkbox* logo no início da tela (4), o qual ativa ou desativa a detecção de cobertura utilizada no estudo anterior realizado (VIRGíNIO et al., 2019) (não foi utilizado no estudo corrente). No `textarea` (5), são apresentados os *logs* do sistema (sem os *logs* de erro).No `textarea` (6) são apresentos os *logs* de erro. Na parte de baixo da tela (7), é apresentado o botão para *download* dos resultados. A barra (8) apresenta o progresso do processamento.

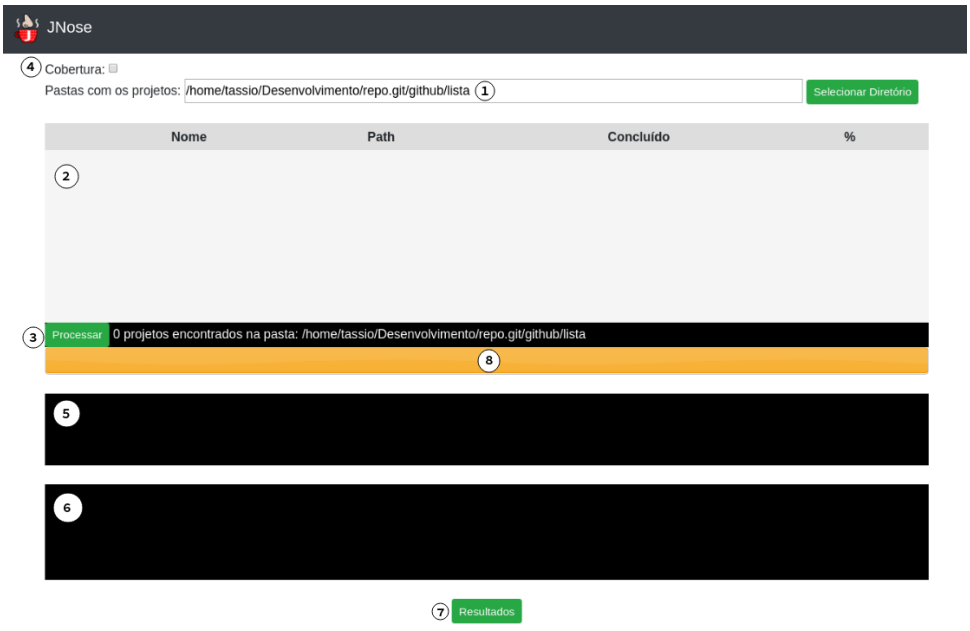


Figura 3.8 Captura da tela principal da JNose Test

A Figura 3.9 apresenta um exemplo de uso da JNose Test com os campos carregados.

A Figura 3.10 apresenta o exemplo com o funcionamento do `textarea` de *log* do sistema. Os resultados da JNose Test são retornados como arquivos CSV para cada projeto: **jacoco.csv** - possui os dados de cobertura dos testes; **testfiledetector.csv** - possui a lista de arquivos de testes do projeto detectados; **testfilemapping.csv** - possui a lista das classes de produção a serem testadas pelos teste; **testsmellsdetector.csv** -

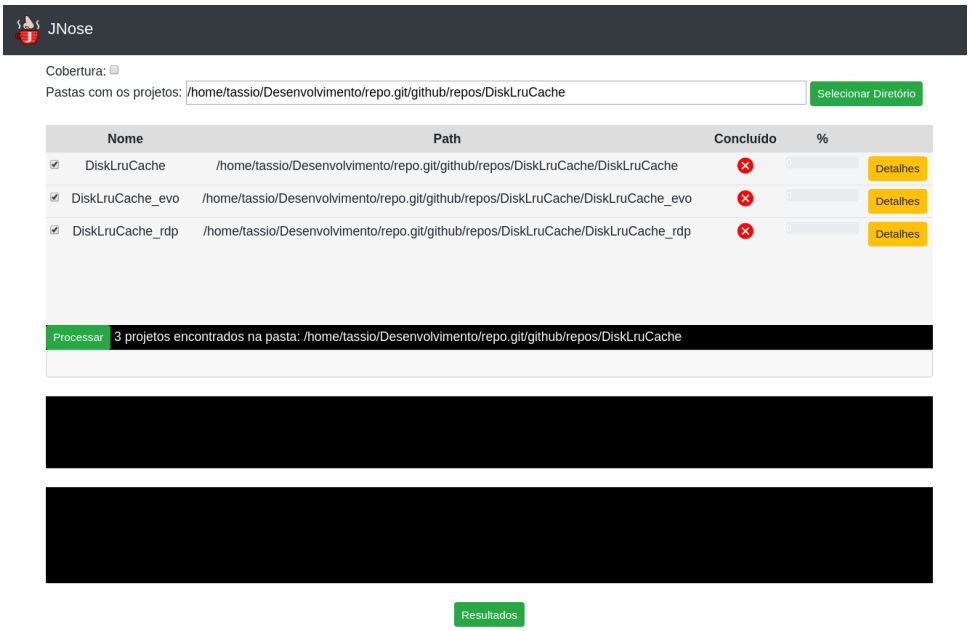


Figura 3.9 Captura de Tela com o JNose Test funcionando

possui a lista de *test smells* encontrados. Esses arquivos podem ser obtidos ao clicar no botão `Detalhes` de cada projeto, como mostra a tela da Figura 3.11.

Além de fornecer os dados de forma separada, a JNose Test possui a opção da geração de dados de forma reunida, ou seja, de todos os projetos em um único arquivo, como descrito na Tabela 3.1 e como mostra a tela da Figura 3.12.

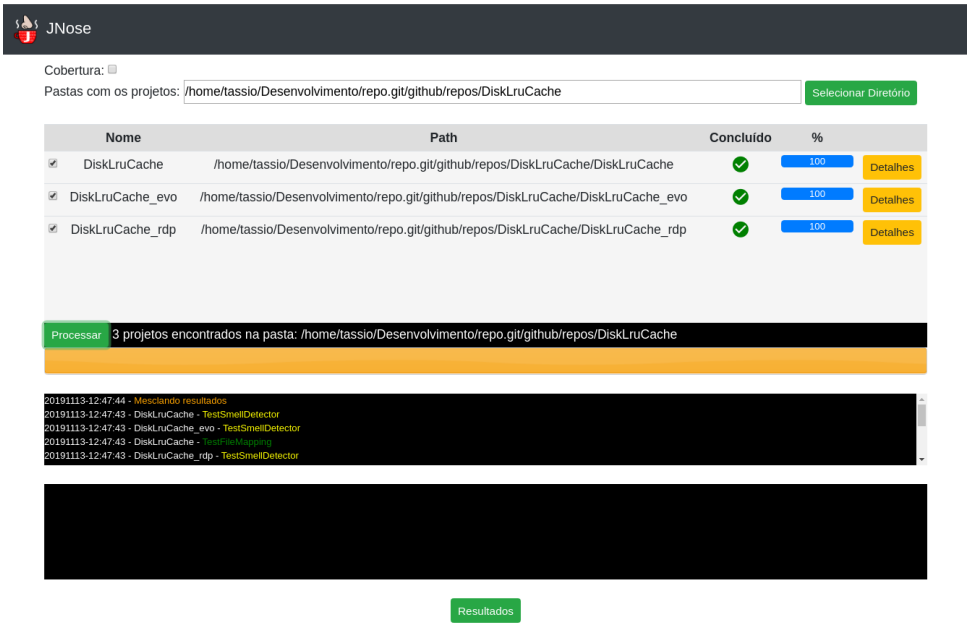


Figura 3.10 Captura de tela após processamento do JNose Test

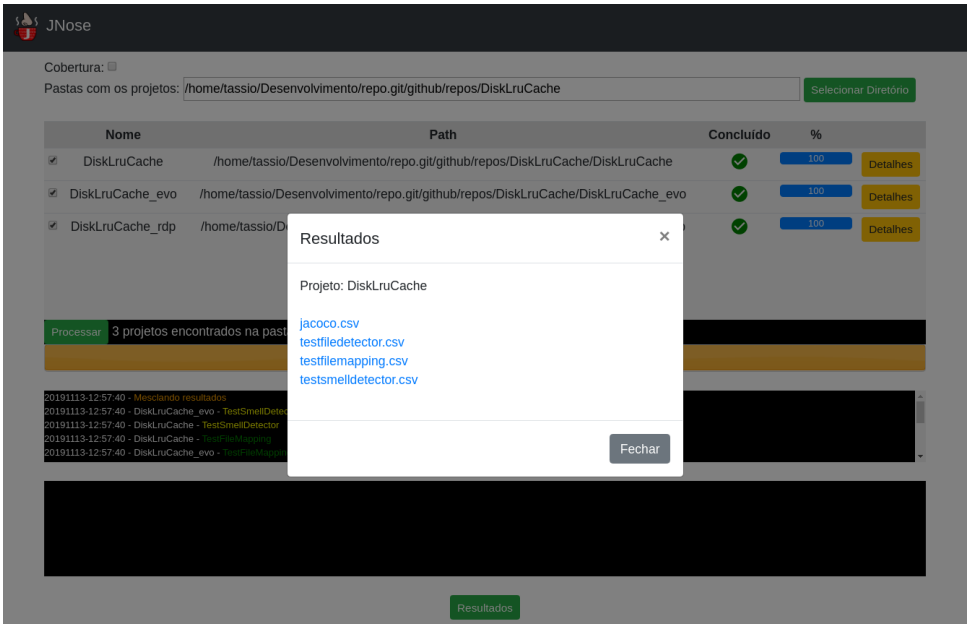


Figura 3.11 Captura de tela com a opção de *download* dos resultados por projeto

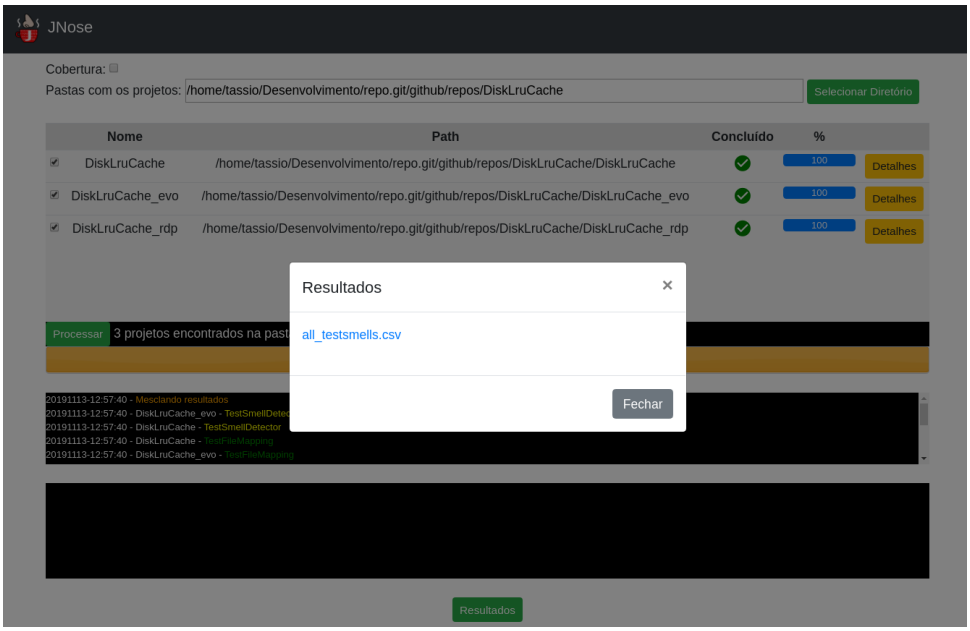


Figura 3.12 Captura de tela com a opção de *download* do resultado geral

3.5 EXECUTANDO O JNOSE TEST: TEST SMELL X COBERTURA

No decorrer do desenvolvimento do JNose Test foi realizado um estudo para analisar a correlações entre os *test smells* e as métricas de cobertura de código (VIRGíNIO et al., 2019), o qual será descrito a seguir. As métricas de cobertura são utilizadas para avaliar a qualidade dos testes por meio da mensuração da cobertura de elementos estruturais, como funções, instruções, ramificações e condicionais (GOPINATH; JENSEN; GROCE, 2014).

No entanto, mesmo com a alta cobertura dos códigos de teste, sua implementação pode conter *test smells*. A presença de *test smells* afeta negativamente a qualidade dos códigos de teste e do código de produção (DEURSEN et al., 2001). Eles indicam problemas no *design* do código de teste que podem afetar a capacidade de manutenção do teste, mais especificamente a legibilidade e a compreensão.

De modo a investigar se há correlação entre as métricas de cobertura e a presença de *test smells*, foi realizado um estudo quantitativo para responder a questão de pesquisa a seguir:

Q1 Existe uma correlação entre *test smells* e as métricas de cobertura de código?

Para responder essa questão de pesquisa, foram analisados 11 projetos de código aberto, buscamos 21 tipos de *test smells* e 10 métricas de cobertura. A ferramenta JNose Test foi utilizada para a coleta dos dados de cobertura e *test smells*. Em seguida, foram calculadas as correlações entre os valores obtidos dos *test smells* e os valores da métricas de cobertura. São descritas as etapas para a execução do experimento a seguir.

3.5.1 Coleta dos Dados

De modo a coletar os dados necessários para responder a **Q1**, a ferramenta JNose Test foi executada. A Tabela 3.2 lista os 11 projetos de software selecionados para o experimento a patir dos seguintes critérios: (i) ser um projeto Maven, (ii) desenvolvido na linguagem Java, (iii) utilizar o *framework* JUnit e (iv) ser *opensource*. A partir dos dados coletados, foram aplicados testes estatísticos para analisar se há correlação entre os *test smells* e as métricas de cobertura obtidas dos projetos selecionados.

Tabela 3.2 Resumo dos projetos de código aberto analisados

Projeto	Versão	LOC	JUnit LOC
Joda Time	2.9.7	29,272	55,554
Apache Commons Text	1.6	9,526	13,467
Apache Commons Lang	3.9	28,006	47,833
Apache Commons Email	1.5.1	2,947	3,989
Apache Commons CSV	1.6	1,787	4,783
Apache Commons Colletion	4.3	29,309	34,200
Apache Commons Codec	1.11	8,392	12,040
Apache Commons DBCP	2.6.0	14,458	16,846
Apache Commons Math	3.4.1	86,481	89,016
Apache Commons Pool	2.6.2	5,532	8,813
Apache Commons DbUtils	1.7	3,101	3,789

3.5.2 Análise de Dados

Para descrever e caracterizar as amostras dos dados, foram utilizadas medidas de tendência central e medidas de dispersão. Em seguida, foi verificado se os dados da amostra seguem a distribuição normal, utilizando o teste de Kolmogorov-Smirnov (WOOD; ALTAVELA, 1978). Este teste retorna a distância máxima entre a distribuição de probabilidades acumulada teórica e observada. O resultado nessa análise apresentou uma distribuição não normal, por tanto, o teste de correlação de Spearman foi escolhido para aceitar ou negar uma das hipóteses de pesquisa:

- Hipótese nula ($H_0 0$): *Não há correlação entre as métricas de cobertura e os test smells.*
- Hipótese alternativa ($H_1 0$): *Há correlação entre as métricas de cobertura e os test smells.*

3.5.3 Resultados

A Tabela 3.3 e a Tabela 3.4 apresentam as estatísticas descritivas para os *test smells* e métricas de cobertura de teste, respectivamente. Por exemplo, em relação ao *test smells* de *lazy test* (LT), o número de LT variou de 0 (mínimo) a 1268 (máximo). O 1º quartil representa $\frac{1}{4}$ do conjunto de dados e é composto apenas por classes sem LT (zero). O terceiro quartil representa $\frac{3}{4}$ do conjunto de dados e é composto por classes que contêm entre 45 e 1268 LT. As classes analisadas apresentaram a mediana 6 LT, com valor médio de 50,8 e desvio padrão de 120,5, indicando alta dispersão dos valores. A Tabela 3.3 apresenta os valores descritivos dos tipos de *test smells* retornados a partir da execução do JNose Test.

Os resultados mostram que o LT tem a maior média e o maior desvio padrão. LT foi seguido por ET com um valor médio de 8,7 e AR com 7,2. Uma alta ocorrência de LT indica que um método da classe de produção executa várias funções, ou seja, um

Tabela 3.3 Estatísticas descritivas dos *test smells*

test smells	Min	1st Qu.	Median	Mean	3st Qu.	Max.	SD
AR	0,0	0,0	2,0	7,2	7,0	219,0	14,6
CTL	0,0	0,0	0,0	1,4	1,0	54,0	3,7
CI	0,0	0,0	0,0	0,3	1,0	2,0	0,5
DT	0,0	0,0	0,0	0,0	0,0	0,0	0
DepT	0,0	0,0	0,0	0,0	0,0	0,0	0
DA	0,0	0,0	0,0	2,1	2,0	83,0	4,9
ET	0,0	0,0	2,0	8,7	9,0	195,0	18,2
EpT	0,0	0,0	0,0	0,0	0,0	6,0	0,2
ECT	0,0	0,0	1,0	5,0	4,0	195,0	13,7
GF	0,0	0,0	0,0	2,0	0,0	196,0	12,2
IgT	0,0	0,0	0,0	0,1	0,0	24,0	0,9
LT	0,0	0,0	6,0	50,8	45,0	1.268,0	120,5
MNT	0,0	0,0	1,0	4,3	0,0	126,0	9,6
MG	0,0	0,0	0,0	0,0	0,0	5,0	0,2
PS	0,0	0,0	0,0	0,0	0,0	2,0	0,1
RA	0,0	0,0	0,0	0,2	0,0	23,0	1,0
RO	0,0	0,0	0,0	0,0	0,0	11,0	0,4
SE	0,0	0,0	0,0	1,1	0,0	67,0	5,3
ST	0,0	0,0	0,0	0,1	0,0	21,0	0,9
UT	0,0	0,0	0,0	1,6	0,0	194,0	10,0
VT	0,0	0,0	0,0	0,0	0,0	1,0	0,1

método tem muitas responsabilidades. Então, é necessário desenvolver vários métodos para testá-lo.

Com relação às métricas de cobertura apresentadas na Tabela 3.4, a *Instruction Coverage* (IC) e a *Line Coverage* (LC) apresentaram os maiores valores médios, assumindo os valores de 445,3 e 80,0, respectivamente. Além disso, a *Instruction Coverage* (IC) apresentou a maior dispersão, 1010,9, seguida pela *Line Coverage* (LC), com 142,4. Esses resultados indicam que a ocorrência de *test smells* são maiores em IC e LC do que nas restantes.

Portanto, como os dados seguem uma distribuição não normal, escolheu-se o teste de correlação não paramétrica de Spearman. A correlação é calculada entre 21 *test smells* e 10 métricas de coberturas, o que resulta em 210 correlações calculadas. A Tabela 3.5 apresenta os resultados obtidos pelo teste de correlação de Spearman. Os resultados foram interpretados de acordo com a escala de valores sugerida por Salkind e Rainwater (SALKIND; RAINWATER, 2003) descrita na Tabela 3.6.

Em geral, foram identificadas correlações entre as métricas de cobertura e os *test smells*. Os valores na Tabela 3.5 indicam que não há correlação entre os *test smells*: *Empty Test* (EpT), *Ignore Test* (IgT), *Mystery Guest* (MG), *Print Declaration* (PS), *Resource Optimism* (RO), *Sleep Test* (ST) e *Test Verbose* (TV). Além disso, os *test smells Dependent Test* (DepT) e *Default Test* (DT) não estão presentes nos projetos analisados, portanto, a correlação não pode ser calculada para esses dois *test smells*. No entanto, os demais *test smells* tiveram correlação fraca a forte com a cobertura de teste. Os test smells: *Conditional Test Logic* (CTL), *Execption Catching Throwing* (ECT),

Tabela 3.4 Estatísticas descritivas por cobertura de código

Coverage	Min	1st Qu.	Median	Mean	3st Qu.	Max.	SD
Branch Covarege (BC)	0,0	3,0	12,0	36,8	37,0	1.495,0	93,5
Branch Missed (BM)	0,0	0,0	1,0	5,0	4,0	212,0	13,8
Complexity Covarege (CC)	0,0	7,0	14,0	34,1	37,0	997,0	64,5
Complexity Missed (CM)	0,0	0,0	2,0	5,5	5,0	193,0	13,4
Instruction Covarege (IC)	0,0	68,0	167,0	445,3	461,0	18.146,0	1010,9
Instruction Missed (IM)	0,0	0,0	5,0	34,7	25,0	2.046,0	115,9
Line Covarege (LC)	0,0	16,0	35,0	80,0	87,0	2.010,0	142,4
Line Missed (LM)	0,0	0,0	1,0	8,4	6,0	529,0	32,9
Method Covarege (MC)	0,0	4,0	8,0	17,2	18,0	312,0	25,6
Method Missed (MM)	0,0	0,0	0,0	1,3	1,0	192,0	6,9

Tabela 3.5 Correlação entre *test smells* e as métricas de cobertura

Test Smell	**BC**	**BM**	**CC**	**CM**	**IC**	**IM**	**LC**	**LM**	**MC**	**MM**
AR	0,42★	0,29❖	0,55★	0,25❖	0,41★	0,18	0,46★	0,18	0,59★	0,06
CI	0,02	0,02	0,15	0,02	-0,01	-0,01	0,05	0,00	0,26❖	-0,02
CTL	0,25❖	0,27❖	0,20❖	0,26❖	0,29❖	0,26❖	0,26❖	0,24❖	0,15	0,17
DA	0,47★	0,35❖	0,52★	0,33❖	0,47★	0,26❖	0,50★	0,25❖	0,49★	0,14
ECT	0,24❖	0,21❖	0,34❖	0,19	0,29❖	0,19	0,32❖	0,19	0,38❖	0,09
ET	0,51★	0,32❖	0,67◆	0,29❖	0,51★	0,23❖	0,57★	0,22❖	0,70◆	0,11
GF	0,15	0,15	0,18	0,18	0,16	0,23❖	0,18	0,23❖	0,16	0,19
LT	0,56★	0,34❖	0,65◆	0,28❖	0,52★	0,23❖	0,57★	0,21❖	0,62◆	0,07
MNT	0,40❖	0,26❖	0,44★	0,21❖	0,40❖	0,17	0,41★	0,14	0,43★	0,06
RA	0,05	0,05	0,16	0,02	0,11	0,00	0,13	0,00	0,22❖	-0,04
SE	0,28❖	0,14	0,38❖	0,09	0,28❖	0,02	0,31❖	0,03	0,40❖	-0,06
UT	0,14	0,15	0,16	0,11	0,28❖	0,09	0,25❖	0,09	0,14	0,02

labels: (▲) muito forte, (◆) forte, (★) moderada, (❖) fraca, ( ) sem correlação.

Magic Number Test (MNT), *Sensitive Test* (SE) e *Unknown Test* (UV) tiveram correlação fraca ou moderada com as métricas de cobertura, principalmente nas métricas *Branch Coverage*, *Instruction Coverage*, e *Line Coverage*. O *test smells Assertion Roulette* (AR), *Duplicate Assert* (DA), *Eager Test* (ET) e *Lazy Test* (LT) tem correlações de moderada a forte com as métricas de cobertura *Branch Coverage*, *Complexity Coverage*, *Instruction Coverage*, *Line Coverage* e *Method Coverage*.

3.5.4 Conclusão e Ameaças a Validade

Nesse estudo, foi utilizada a ferramenta JNose Test para a obtenção dos dados e posteriormente realizou-se 210 cálculos de correlação, e obteve 4 correlações fortes, 17 correlações moderadas, 42 correlações fracas, 147 sem correlação.

As ameaças a validade encontradas foram:

- Validade Interna: Neste estudo, o conjunto de métricas de cobertura e o conjunto de *test smells* selecionados foram de acordo com o suporte das ferramentas existentes.

Tabela 3.6 Escala de valores (SALKIND; RAINWATER, 2003)

Intervalo	**descrição**
0,8 a 1,0 ou -0,8 a -1,0	correlação muito forte
0,6 a 0,8 ou -0,6 a -0,8	correlação forte
0,4 a 0,6 ou -0,4 a -0,6	correlação moderada
0,2 a 0,4 ou -0,2 a -0,4	correlação fraca
0,2 a -0,2	correlação fraca ou sem correlação

Não foram cobertos outras métricas de cobertura e *test smells* fora do escopo das ferramentas.

- Validade Externa: Neste estudo, apenas o *framework* JUnit foi abordado, não sendo analisadas outras possibilidades, os projetos selecionados são desenvolvidos na linguagem Java, ficando de fora outras linguagens.

Esse estudo contribuiu para a detecção de *test smells* e o cálculo de métricas de cobertura, através do desenvolvimento da ferramenta `JNose Test`. Como trabalho futuro, pode-se refinar os algoritmos para identificar a linha na qual os *test smells* foram encontrados. Além disso, analisou quais partes do código são mais afetadas pelos *test smells*.

3.6 SÍNTESE DO CAPÍTULO

Ao longo deste capítulo, foi apresentada a ferramenta JNose Test, desenvolvida com o propósito de auxiliar a geração e a coleta de dados, automatizando o processo de seleção de projetos. Um estudo preliminar sobre a correlação dos *test smells* e as métricas de cobertura que a ferramenta disponibiliza encontra-se disponível em Virgínio et al. (2019).

Capítulo

4

ESTUDO EXPERIMENTAL

Este capítulo descreve o estudo experimental realizado para o levantamento e análise de dados sobre os *test smells*, suas motivações bem como discute as potenciais ameaças à validade. O estudo experimental foi concebido e estruturado com base nos conceitos de Engenharia de Software Experimental e na avaliação de métodos e ferramentas de engenharia de software fornecidos por Jedlitschka, Ciolkowski e Pfahl (2008).

A Seção 4.1 apresenta o objetivo do experimento, as questões de pesquisa, métricas e o contexto do experimento. A Seção 4.2 detalha o projeto do experimento, define as unidades experimentais, o material, as atividades realizadas, o projeto piloto, as hipóteses, as variáveis e o *design* do experimento. A Seção 4.3 apresenta os dados, a preparação do *dataset* e os testes de hipóteses. A Seção 4.4 apresenta os resultados encontrados a partir do experimento, suas implicações, as ameaças à validade do estudo, inferências e lições aprendidas.

4.1 DEFINIÇÃO DO ESTUDO EXPERIMENTAL

4.1.1 Objetivo

Avaliar empiricamente na perspectiva dos *test smells* a qualidade do código de teste gerado pelas ferramentas Randoop e Evosuite e dos códigos de teste existentes nos projetos.

4.1.2 Questões de Pesquisa

A seguir, são descritas as questões de pesquisa definidas para atingir o objetivo:

QP1 Existe diferença (estatisticamente significativa) na qualidade do código de teste gerado pelas ferramentas Randoop e Evosuite?

QP2 Existe diferença (estatisticamente significativa) na qualidade do código de teste gerado pela ferramenta Randoop e os códigos de testes pré-existentes?

QP3 Existe diferença (estatisticamente significativa) na qualidade do código de teste gerado pela ferramenta Evosuite e os códigos de testes pré-existentes?

4.1.3 Métricas

A seguintes métricas foram avaliadas:

M1 Ocorrência dos *test smells* nos pacotes de testes (Existente, Randoop e Evosuite) (OTSP): A ocorrência (%) dos *test smells* por classe de teste nos pacotes de testes. Essa métrica foi utilizada para responder as questões de pesquisa **QP1**, **QP2** e **QP3**;

M2 Quantidade de *test smells* por métodos de teste nos pacotes de testes (Existente, Randoop e Evosuite) (QTSM): Essa métrica apresenta a média de *test smells* por método de teste. Essa métrica foi utilizada para responder as questões de pesquisa **QP1**, **QP2** e **QP3**;

M3 Quantidade de *test smells* (QTS): A quantidade de *test smells* encontrados pela ferramenta JNose Test por classe/projeto. Essa métrica foi utilizada para responder as questões de pesquisa **QP1**, **QP2** e **QP3**;

M4 Quantidade de *test smells* por tipo (QTSTP): Essa métrica representa a quantidade de vezes que um *test smell* é encontrado por classe/projeto. Essa métrica foi utilizada para responder as questões de pesquisa**QP1**, **QP2** e **QP3**;

M5 Quantidade de tipos de *test smells* (QTiTS): Essa métrica representa a quantidade de *test smells* diferentes que foram encontrados por classe/projeto. Essa métrica foi utilizada para responder as questões de pesquisa **QP1**, **QP2** e **QP3**;

M6 Ocorrência de *test smells* por classe de teste (OTSC): A ocorrência (quantidade) dos *test smells* por classe de teste. Essa métrica foi utilizada para responder as questões de pesquisa **QP1**, **QP2** e **QP3**;

4.1.4 Seleção do Contexto

O contexto do projeto se caracteriza por ser *on-line* pois os projetos utilizados como base foram projetos do mundo real, ou seja, problemas reais, todos *opensource* presentes no repositório GitHub. O contexto se caracteriza de forma específica por considerar projetos desenvolvidos na linguagem Java e que utilizam o *framework* JUnit.

4.2 PLANEJAMENTO DO EXPERIMENTO

Esta seção descreve o passo-a-passo utilizado para a realização do experimento. Conforme mencionado anteriormente, o estudo utilizou projetos de código aberto, disponíveis no repositório GitHub. Neste estudo, a fase de planejamento considera as etapas descritas a seguir, seguindo o modelo proposto em Jedlitschka, Ciolkowski e Pfahl (2008).

4.2.1 Unidades Experimentais

Para este estudo as unidades experimentais foram 21 projetos desenvolvidos na linguagem Java, selecionados de forma aleatória a partir do repositório do GitHub. Os critérios de escolha dos projetos foram: desenvolvidos na linguagem Java, de código aberto, testes unitários escritos com o *framework* JUnit e projetos que utilizam o Maven. Não foram utilizados critérios de quantidade de linhas de código, classes, datas de alterações, datas de *commits* e quantidade de *commits*. A Tabela 4.1 apresenta os projetos selecionados. Após a seleção dos projetos utilizamos as ferramentas Evosuite e Randoop para gerar os testes a partir do código fonte.

Tabela 4.1 Projetos Selecionados

#	**Projeto**	**nº classes**	**nº teste**	**URL (https://github.com)**
1	Algorithms	86	80	/pedrovgs/Algorithms.git
2	sling-org-apache-sling-api	116	15	/apache/sling-org-apache-sling-api.git
3	blade	191	84	/lets-blade/blade.git
4	checkstyle	377	1395	/checkstyle/checkstyle.git
5	commons-collections	326	217	/apache/commons-collections.git
6	commons-jexl	102	63	/apache/commons-jexl.git
7	DiskLruCache	3	2	/JakeWharton/DiskLruCache.git
8	failsafe	56	47	/jhalterman/failsafe.git
9	fastjson	178	2642	/alibaba/fastjson.git
10	GCViewer	155	67	/chewiebug/GCViewer.git
11	HanLP	472	155	/hankcs/HanLP.git
12	mybatis-generator-gui	31	2	/zouzg/mybatis-generator-gui.git
13	Mybatis-PageHelper	48	66	/pagehelper/Mybatis-PageHelper.git
14	lettuce-core	643	349	/lettuce-io/lettuce-core.git
15	reflections	36	12	/ronmamo/reflections.git
16	spark	99	81	/perwendel/spark.git
17	uid-generator	22	2	/baidu/uid-generator.git
18	xmlgraphics-commons	314	73	/apache/xmlgraphics-commons.git
19	ysoserial	59	18	/frohoff/ysoserial.git
20	commons-csv	12	22	/apache/commons-csv.git
21	commons-text	101	80	/apache/commons-text.git

4.2.2 Material Experimental

O experimento foi realizado em uma estação de trabalho com as seguintes configurações: processador Core I5; 16G de memoria RAM, HD de 240GB SSD.

4.2.3 Atividades

Inicialmente, um projeto piloto foi realizado, a partir das características iniciais do experimento. No decorrer do desenvolvimento do projeto piloto, foi necessária a codificação de um *script* para automatizar a seleção dos projetos. Após a seleção dos projetos, Evosuite e Randoop foram executados para cada projeto selecionado, gerando pacotes de testes. Ao final foi obtido 3 pacotes de testes: um dos testes existentes, outro da Randoop e outro da Evosuite.

O projeto selecionado para executar o piloto foi o Apache Commons-io[1], selecionado

[1]<https://github.com/apache/commons-io>

de forma aleatória. O projeto piloto serviu para identificar inconsistências e problemas com o planejamento do experimento, reduzindo eventuais prejuízos à coleta e à análise de dados. A Figura 4.1 apresenta o fluxo executado no projeto piloto.

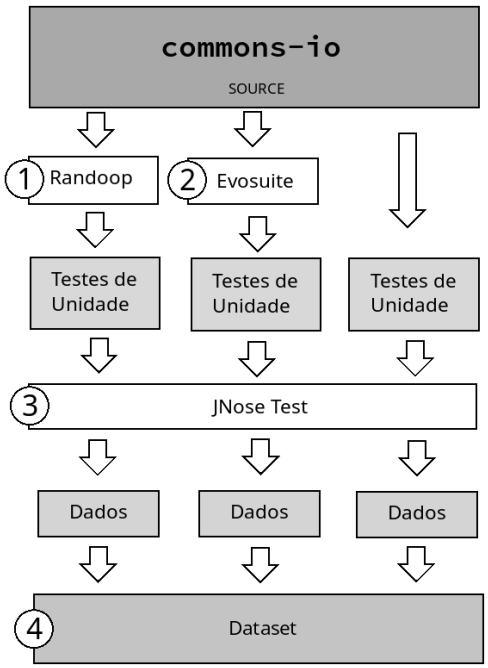


Figura 4.1 Fluxo do Projeto Piloto

As duas primeiras etapas consistem na execução da Randoop e da Evosuite com o código fonte do `Apache Commons-io`. Com a execução, três pacotes de testes de unidade foram gerados, considerando o pacote existente do projeto. A terceira etapa consiste na execução do JNose Test nesses pacotes de testes, que resultará em três pacotes de dados a serem utilizados na quarta etapa (análise e discussão). A partir do projeto piloto, gerou um *script* de seleção de projetos de forma aleatória no repositório do GitHub, o qual segue os critérios definidos, além de nos ter proporcionado testar o fluxo descrito na Figura 4.1 e mostrar alguns pontos de melhorias e correções no código da JNose Test.

4.2.4 Formulação de hipóteses

Em um experimento, é necessário declarar formal e claramente o que se pretende avaliar. Neste estudo experimental, optou por focar em três hipóteses. Conforme descrição a seguir:

- **Hipótese nula ($H_0$1)**: Não há diferença significativa entre a quantidade de *test smells* "por classe de teste" (M6) detectados no código gerado pela ferramenta Randoop e os códigos existentes.
- **Hipótese alternativa ($H_1$1)**: Há diferença significativa entre a quantidade de *test smells* "por classe de teste"(M6) detectados no código gerado pela ferramenta Randoop e os códigos existentes.

- **Hipótese nula ($H_0$2)**: Não há diferença significativa entre a quantidade de *test smells* "por classe de teste"(M6) detectados no código gerado pela ferramenta Evosuite e os códigos existentes.

- **Hipótese alternativa ($H_1$2)**: Há diferença significativa entre a quantidade de *test smells* "por classe de teste"(M6) detectados no código gerado pela ferramenta Evosuite e os códigos existentes.

- **Hipótese nula ($H_0$3)**: Não há diferença significativa entre a quantidade de *test smells* "por classe de teste"(M6) detectados no código gerado pelas ferramentas Randoop e Evosuite.

- **Hipótese alternativa ($H_1$3)**: Há diferença significativa entre a quantidade de *test smells* "por classe de teste"(M6) detectados no código gerado pelas ferramentas Randoop e Evosuite.

4.2.5 Variáveis

- As variáveis independentes são a plataforma do sistema, linguagem utilizada, biblioteca de teste utilizada;

- As variáveis dependentes são a distribuição de *test smells*, a quantidade total de *test smells*, a quantidade de cada tipo de *test smells* e a quantidade de tipos de *test smells*.

Utilizou a distribuição, a quantidade total, a quantidade por tipos e a quantidade de tipos de *test smells* para caracterizar a qualidade dos testes na perspectiva dos *test smells*.

4.2.6 *Design*

O experimento consiste inicialmente na execução da Evosuite e da Randoop para gerar códigos de teste de unidade. O tipo de desenho a ser usado nesse experimento é *um fator com dois tratamentos*, no qual compara os dois tratamentos entre si (JEDLITSCHKA; CIOLKOWSKI; PFAHL, 2008).

Os tratamentos são pacote de teste (existentes, evosuite, randoop). Foram realizados três comparativos utilizando os pacotes de testes para verificar os dados sobre os *test smells*. A Figura 4.2 apresenta o fluxo do experimento de forma macro.

Na Figura 4.2, a primeira e a segunda etapa, consistem em executar as ferramentas Randoop e Evosuite com os códigos-fontes dos projetos selecionados. Com essa execução haverá três pacotes de testes de unidade, contando com o pacote já existente, para cada projeto selecionado. A terceira etapa consiste em executar o JNose Test nestes pacotes de testes, que resultará em três pacotes de dados para cada projeto selecionado, gerando um único *dataset* com as informações dos *test smells* para todos os pacotes de testes. Esse *dataset* foi utilizado na quarta etapa de análise estatística, descrito mais detalhadamente na próxima seção.

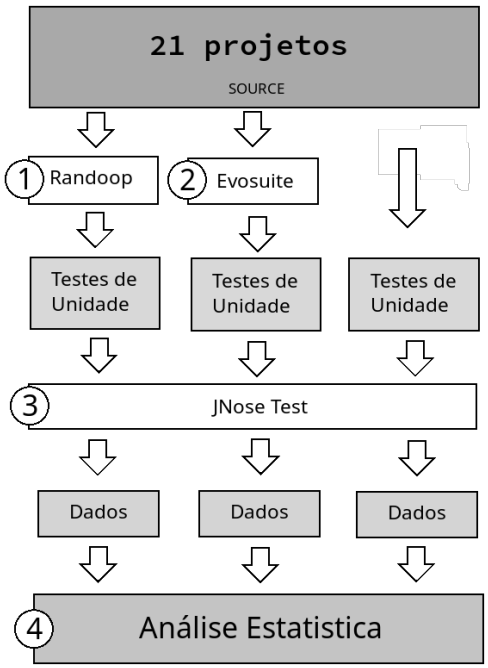


Figura 4.2 Fluxo do Experimento

4.3 ANÁLISE

Nesta seção apresenta o resumo dos dados coletados e suas características, bem como os processos adicionais necessários para a utilização desses dados.

4.3.1 Estatística Descritiva

Apresentou inicialmente os dados encontrados com a execução do experimento a partir dos 21 projetos selecionados. Ao final gerou um *dataset* com 5445 linhas, disponibilizado em Virgínio (2020), onde cada linha trouxe os valores descritos na Tabela 3.1 (retorno da ferramenta JNose Test). Em seguida descreveu de forma mais específica os dados e suas características.

A partir dos valores encontrados (com base nos 21 projetos selecionados), notou-se que os testes existentes nos projetos foram os que tiveram menor quantidade de linhas de código Linhas de Código (LOC) com 168.839, menor quantidade de métodos de testes com 10.141 e de *test smells* encontrados com 28.828. Os dados encontrados para o Evosuite foram 639.979 linhas de código, 31.017 métodos de testes e 228.492 *test smells*. A Tabela 4.2 apresenta os valores descritos, tem-se os somatórios de LOC das classes de teste, quantidade de métodos de testes e a quantidade total de *test smells* encontrados, todos os valores por pacotes de testes (Existentes, Evosuite e Randoop).

Os dados da Randoop foram os mais discrepantes encontrados, as linhas de código ficaram 22 vezes maior do que os testes existentes, aproximadamente 6 vezes maior que os da Evosuite, a quantidade de métodos ficou 14 vezes maior que a quantidade dos códigos existentes. Uma diferença significativa em relação à quantidade de *test smells*

Tabela 4.2 Valores totais por pacote de teste

#	Pacote	LOC	Qtd.Métodos	TestSmells
1	Existente	168.839 (4%)	10.141 (6%)	28.828 (3%)
2	Evosuite	639.979 (14%)	31.017 (16%)	228.492 (20%)
3	Randoop	3.733.485 (82%)	148.284 (78%)	859.130 (77%)

encontrados, aproximadamente 30 vezes maior do que o valor dos códigos existentes.

Continuando com o levantamento dos dados obtidos a partir do *dataset*, encontramos as informações dos *test smells* pelos tipos de *test smell*. A Tabela 4.3 apresenta os valores do somatório de cada tipo de *test smell* por pacote de testes.

Tabela 4.3 Quantidade de tipos *test smells* por pacote (M4)

Test Smell	Existente	Randoop	Evosuite
Lazy Test (LT)	13797	407019	172563
Exception Catching Throwing (ECT)	4256	148284	30767
Conditional Test Logic (CTL)	485	148284	28
Assertion Roulette (AR)	1852	86233	4953
Eager Test (ET)	2637	61101	10713
Unknown Test (UT)	1977	5987	135
Magic Number Test (MNT)	1137	297	6850
Constructor Initialization (CI)	140	0	0
Default Test (DT)	0	0	0
Empty Test (EmT)	20	0	102
General Fixture (GF)	660	0	26
Mystery Guest (MG)	128	0	176
Print Statement (PS)	108	0	0
Redundant Assertion (RA)	32	0	1
Sensitive Equality (SE)	518	1682	533
Verbose Test (VT)	3	243	259
Sleepy Test (ST)	8	0	0
Duplicate Assert (DA)	747	0	1236
Ignored Test (IgT)	41	0	0
Resource Optimism (RO)	282	0	150
Dependent Test (DpT)	0	0	0

A Figura 4.3 apresenta a quantidade de *test smells* por tipo e pacote de teste, a partir do qual podemos verificar qual tipo de *test smell* mais se destacou no *dataset* (*Lazy Test, Exception Catching Throwing, Conditional Test Logic, Assertion Roulette*), sendo o *Lazy Test* o que teve maior ocorrência nos três pacotes analisados.

Na Figura 4.4 apresenta quais os *test smells* encontrados por ordem decrescente, ficando o *Lazy Test* com os maiores valores.

Encontrados 19 tipos de *test smells* no pacote de testes existentes. A Tabela 4.3 apresenta os quantitativos dos *test smells* encontrados. Além do *Lazy Test* destacou-se o *Exception Catching Throwing* e o *Eager Test* que ficou em segundo e terceiro lugar

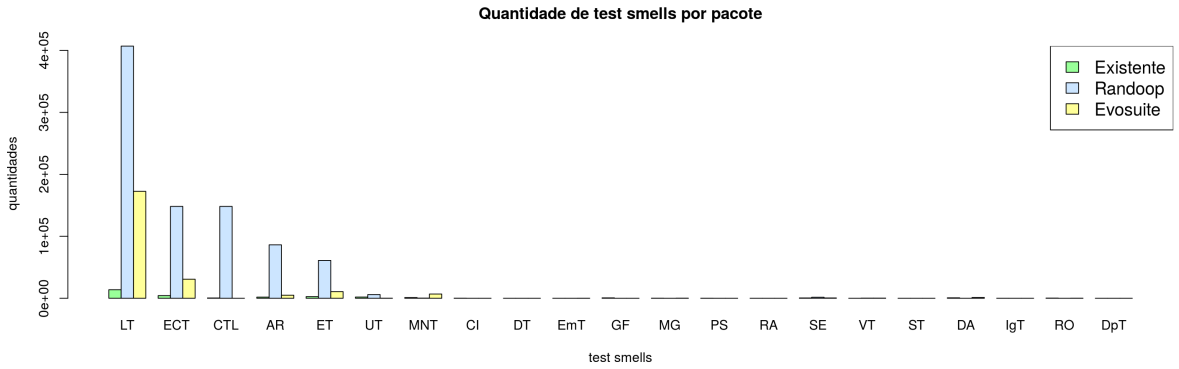


Figura 4.3 Quantidade de *test smells* por Tipo/Pacote

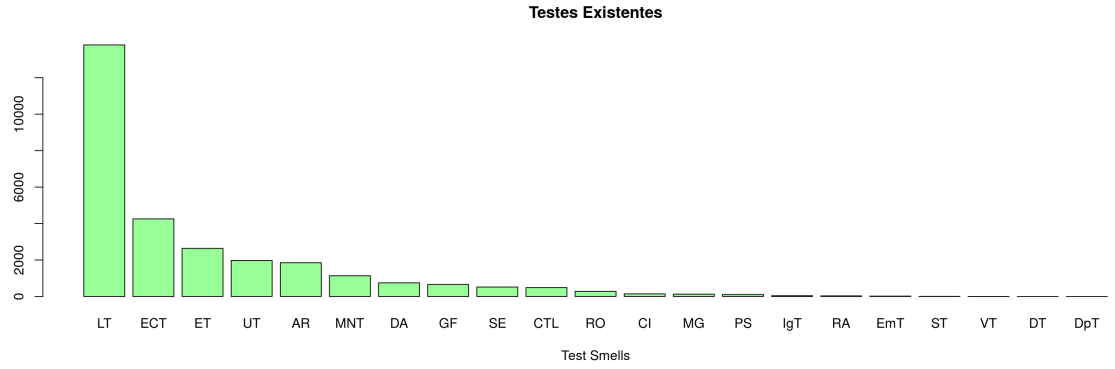


Figura 4.4 Quantidade de *test smells* (Existentes)

respectivamente.

No experimento o *Lazy Test* destacou-se por ter sido o *test smell* mais encontrado em todos os pacotes de testes. Novamente, nota-se essa característica na Figura 4.5, do pacote da Randoop. A Tabela 4.3 apresenta os dados quantitativos dos *test smells* encontrados no pacote de testes do Randoop. Foram 9 tipos de *test smells* encontrados, quantidade menor de tipos do que os testes existentes. Além do *Lazy Test*, destacou-se o *Conditional Test Logic* e *Exception Catching Throwing*.

O *Lazy Test* apresenta os maiores valores no pacote da Evosuite. A Figura 4.6 apresenta os dados da ferramenta Evosuite, de forma bem discrepante dos demais, sendo 5 vezes maior do que o *Exception Catching Throwing*. A Tabela 4.3 mostra os dados encontrados para cada tipo de *test smell* no pacote de dados da Evosuite, além de 15 tipos de *test smells*.

Com a análise inicial nota-se que os testes existentes foram os que tiveram maior ocorrência de tipos(M5) de *test smells*, e a Randoop apresentou a menor quantidade de tipos distintos(M5) de *test smell*, mas por outro lado, foi a que mais gerou código de teste

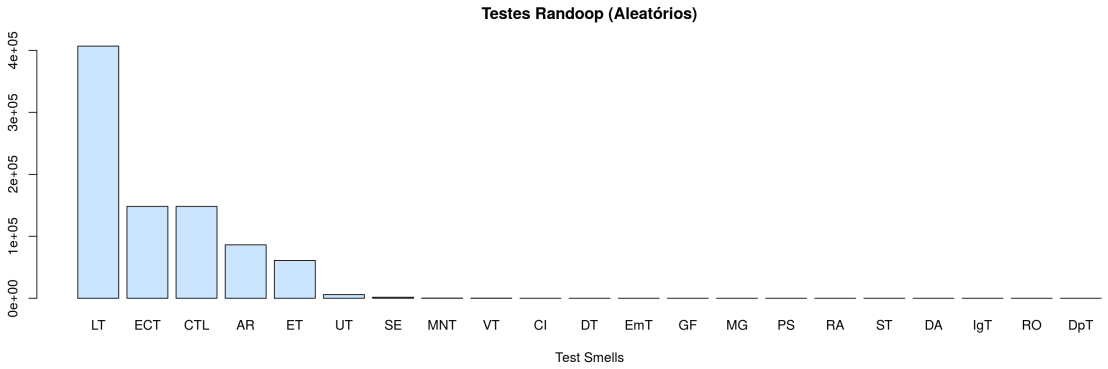


Figura 4.5 Quantidade de *test smells* (Randoop)

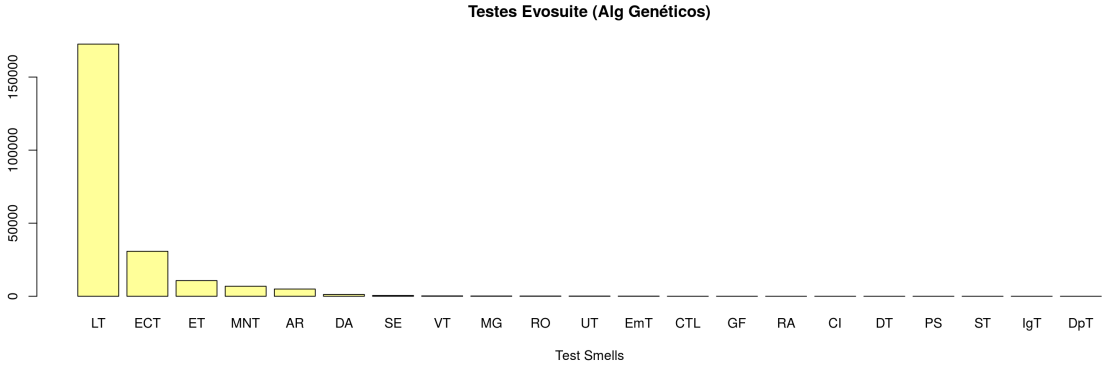


Figura 4.6 Quantidade de *test smells* (Evosuite)

e quantidade de métodos de teste. O pacote da Evosuite aproximou-se do quantitativo de tipos(M5) de *test smells* do pacote de testes existentes, como mostra a Tabela 4.4.

Tabela 4.4 Métricas por pacote de teste

Pacote	qtd tipos(M5)	ocorrência(M1)	*test smells*(M3)	smells/testes(M2)
Existente	19	97,03%	206622	7,19
Evosuite	15	99,95%	228492	7,37
Randoop	9	99,53%	681336	5,25

Encontrou alta ocorrência de *test smells* nas classes de testes de todos os pacotes analisados (M1). A Tabela 4.4 apresenta os dados do pacote da Randoop por tipo de *test smell*, que resultou nos maiores valores. Nota-se que as classes do pacote de testes existente tem variação de ocorrência de *test smells* menor, aproximada de 2% menor em comparação com os pacotes da Evosuite e da Randoop os quais ficaram com a taxa de ocorrência dos *test smells* acima de 99% das classes de testes.

Esses dados mostram que os testes existentes tiveram menor ocorrência de classes afetadas por *test smells*(M1). Além disso, na Tabela 4.4 apresenta o valor de quantos *test smells* ocorreram por método de teste(M2). Com isso, pode-se realizar um comparativo mais aprofundado das características entre os pacotes de teste.

Dentre os resultados apresentados notou-se que a Evosuite introduziu praticamente a mesma quantidade de *test smells* que os testes existentes nos métodos de teste e a Randoop (geração aleatória) conseguiu introduzir quantidade menor de *test smells* por método de teste(M2). Por outro lado, o quantitativo de métodos de testes(M2) é aproximadamente quatro vezes maior que os testes existentes e a Evosuite. A Tabela 4.4 apresenta os valores encontrados.

Tabela 4.5 Ocorrência por tipos de *test smells*

Test Smell	Existente(%)	Randoop(%)	Evosuite(%)
Assertion Roulette (AR)	6,42	10,04	2,17
Conditional Test Logic (CTL)	1,68	17,26	0,01
Constructor Initialization (CI)	0,49	0,00	0,00
Default Test (DeT)	0,00	0,00	0,00
Empty Test (EmpT)	0,07	0,00	0,04
Exception Catching Throwing (ECT0	14,76	17,26	13,47
General Fixture (GF)	2,29	0,00	0,01
Mystery Guest (MG)	0,44	0,00	0,08
Print Statement (PS)	0,37	0,00	0,00
Redundant Assertion (RA)	0,11	0,00	0,00
Sensitive Equality (SE)	1,80	0,20	0,23
Verbose Test (VerT)	0,01	0,03	0,11
Sleepy Test (SlpT)	0,03	0,00	0,00
Eager Test (EgT)	9,15	7,11	4,69
Lazy Test (LT)	47,86	47,38	75,52
Duplicate Assert (DA)	2,59	0,00	0,54
Unknown Test (UkT)	6,86	0,70	0,06
Ignored Test (IgT)	0,14	0,00	0,00
Resource Optimism (RsOpt)	0,98	0,00	0,07
Magic Number Test (MNT)	3,94	0,03	3,00
Dependent Test (DpT)	0,00	0,00	0,00

A Tabela 4.5 apresenta as ocorrências de *test smells* nas classes de testes para cada pacote de teste. Cabe destaque nos dados da Evosuite para o *Lazy Test* por estar presente em 75,53% das classes de testes deste pacote, bem diferente do que pode-se observar nos outros pacotes de testes, ficando 47.86% para os testes existentes e 47.38% para a Randoop. Esses dados mostram oportunidade de melhorias possível na ferramenta Evosuite, sugerindo a refatoração dos códigos de geração de teste foco na eliminação dos *Lazy Tests*, de forma bem significativa para a ferramenta Evosuite.

4.3.2 Preparação do *dataset*

Para realizar um comparativo com a mesma quantidade de dados entre a Randoop, a Evosuite e os testes existentes, foram gerados pacotes de dados pareados a partir das classes de produção, parâmetro "3" exibido na tabela 3.1. Foram criados 3 *datasets* a partir do *dataset* principal, um para cada par de análise, entre os testes existentes e o Evosuite, um para os testes existentes e a Randoop, e outro para a Randoop e a Evosuite. A Figura 4.7 ilustra o processo de pareamento dos dados.

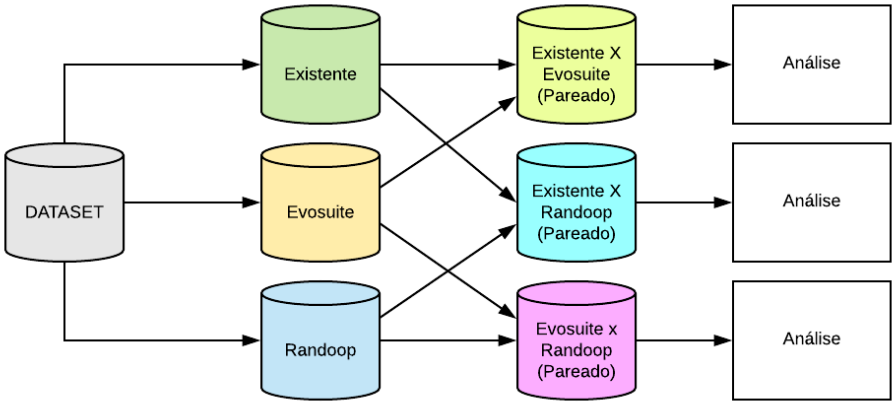


Figura 4.7 Geração dos pacotes pareados por classes de produção

4.3.3 Verificação da normalidade dos dados

A partir dos 3 novos *datasets* pareados, realizou uma verificação da normalidade dos dados para identificar qual teste de hipótese que seria utilizado. Os resultados são discutidos a seguir.

- Existentes x Randoop

  - Verificação dos dados dos testes existentes
  Utilizamos o teste de Shapiro-Wilk, o qual deu valor a baixo de 2.2^{-16} como podemos visualizar no resultado do cálculo realizado com o R Studio na Lista 4.1.

  Também foi desenvolvido um histograma com os dados obtidos para prover uma melhor visualização dos resultados, como pode ser visto na Figura 4.8. Como visualizado na figura o histograma tende a esta com os dados mais a esquerda, o que descreve uma distribuição não normal. Com os resultados podemos confirmar que os dados não estão em uma distribuição normal, o que ocasiona na utilização do teste de Wilcoxon para distribuições não-normais.

Lista 4.1 Resultado do Teste de Shapiro-Wilk

```
## Shapiro-Wilk normality test
##
## data: lista_existente$Total
## W = 0.29063, p-value < 2.2e-16
```

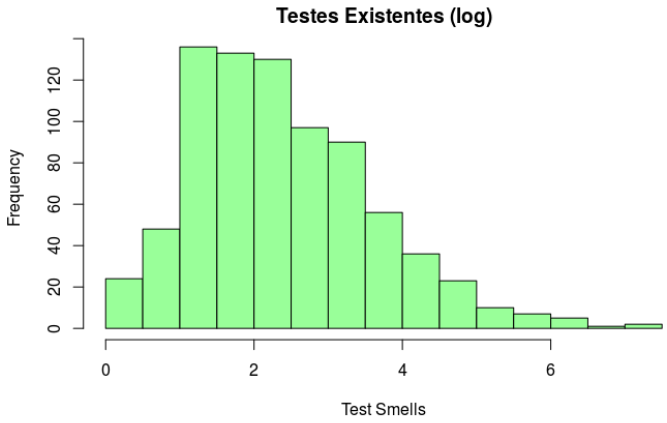


Figura 4.8 Histograma dos testes existentes

- Verificação dos dados dos testes gerados pelo Randoop
Utilizamos o teste de Shapiro-Wilk, o qual deu valor a baixo de 2.2^{-16} como podemos visualizar no resultado do cálculo realizado com o R Studio na Lista 4.2.

Também foi desenvolvido um histograma com os dados obtidos para prover uma melhor visualização dos resultados, como pode ser visto na Figura 4.9. Como visualizado na figura o histograma tende a esta com os dados mais a direita, o que descreve uma distribuição não normal. Com os resultados podemos confirmar que os dados não estão em uma distribuição normal, o que ocasiona na utilização do teste de Wilcoxon para distribuições não-normais.

Lista 4.2 Resultado do Teste de Shapiro-Wilk

```
## Shapiro-Wilk normality test
##
## data: lista_randoop$Total
## W = 0.53809, p-value < 2.2e-16
```

- Existentes x Evosuite

- Verificação dos dados dos testes existentes
Utilizou o teste de Shapiro-Wilk, o qual deu valor a baixo de 2.2^{-16} como visualizado no resultado do cálculo realizado com o R Studio na Lista 4.3.

Também foi desenvolvido um histograma com os dados obtidos para prover uma melhor visualização dos resultados, como pode ser visto na Figura 4.10. Como

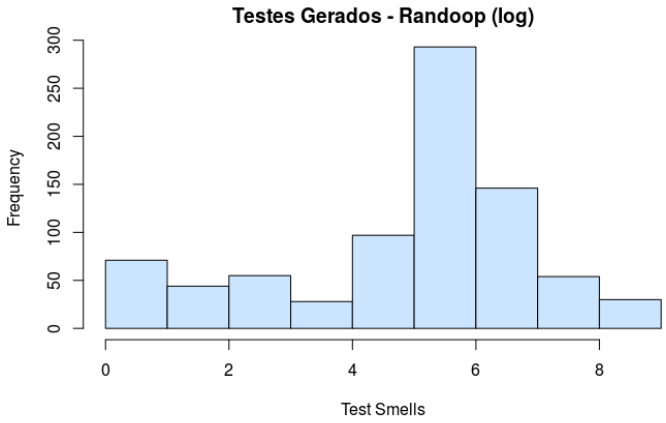


Figura 4.9 Histograma dos testes gerados pelo Randoop

visualizado na figura o histograma tende a esta com os dados mais a esquerda, o que descreve uma distribuição não normal. Com os resultados podemos confirmar que os dados não estão em uma distribuição normal, o que ocasiona na utilização do teste de Wilcoxon para distribuições não-normais.

Lista 4.3 Resultado do Teste de Shapiro-Wilk

```
## Shapiro-Wilk normality test
##
## data: lista_existente2$Total
## W = 0.30637, p-value < 2.2e-16
```

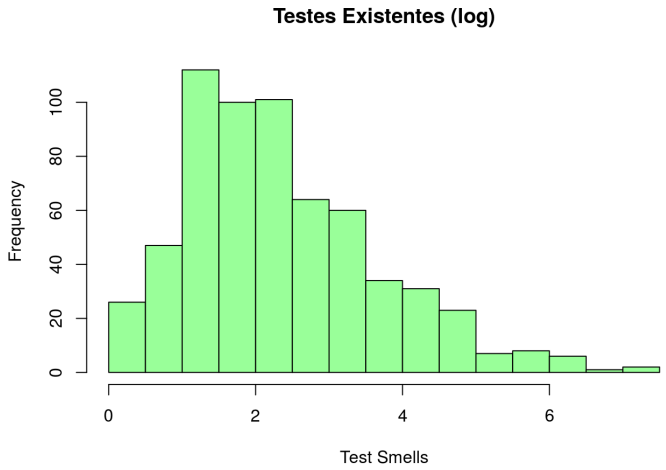


Figura 4.10 Histograma dos testes existentes

- Verificação dos dados dos testes gerados pela Evosuite

Utilizou o teste de Shapiro-Wilk, o qual deu valor a baixo de 2.2^{-16} como visualizado no resultado do cálculo realizado com o R Studio na Lista 4.4.

Também, foi desenvolvido um histograma com os dados obtidos para prover melhor visualização dos resultados, como pode ser visto na Figura 4.11. Como visualizado na figura o histograma tende a esta com os dados mais a esquerda, o que descreve uma distribuição não normal. Os resultados confirmam que os dados não estão em uma distribuição normal, o que ocasiona na utilização do teste de Wilcoxon para distribuições não-normais.

Lista 4.4 Resultado do Teste de Shapiro-Wilk

```
## Shapiro-Wilk normality test
##
## data: lista_evosuite$Total
## W = 0.026245, p-value < 2.2e-16
```

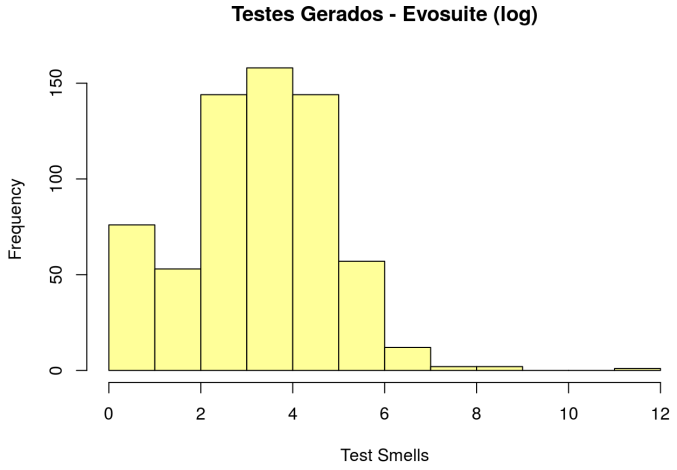


Figura 4.11 Histograma dos testes gerados pela Evosuite

- Randoop x Evosuite

  - Verificação dos dados dos testes da Randoop

  Utilizou o teste de Shapiro-Wilk, o qual deu valor a baixo de 2.2^{-16} como observado no resultado do cálculo realizado com o R Studio na Lista 4.5.

  Também, foi desenvolvido um histograma com os dados obtidos para prover melhor visualização dos resultados, como pode ser visto na Figura 4.12. Como visualizado na figura o histograma tende a esta com os dados mais a direita, o que descreve uma distribuição não normal. Os resultados confirmam que os dados não estão em

uma distribuição normal, o que ocasiona na utilização do teste de Wilcoxon para distribuições não-normais.

Lista 4.5 Resultado do Teste de Shapiro-Wilk

```
## Shapiro-Wilk normality test
##
## data: lista_randoop2$Total
## W = 0.53187, p-value < 2.2e-16
```

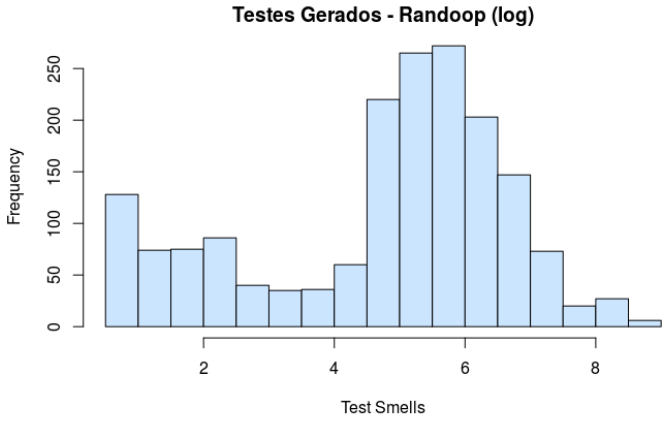


Figura 4.12 Histograma dos testes gerados pelo Randoop

- Verificação dos dados dos testes gerados pela Evosuite

Utilizou o teste de Shapiro-Wilk, o qual deu valor a baixo de 2.2^{-16} observado no resultado do cálculo realizado com o R Studio na Lista 4.6.

Também, foi desenvolvido um histograma com os dados obtidos para prover uma melhor visualização dos resultados, como pode ser visto na Figura 4.13. Como visualizado na figura o histograma tende a esta com os dados mais a esquerda, o que descreve uma distribuição não normal. Os resultados confirmam que os dados não estão em uma distribuição normal, o que ocasiona na utilização do teste de Wilcoxon para distribuições não-normais.

Lista 4.6 Resultado do Teste de Shapiro-Wilk

```
## Shapiro-Wilk normality test
##
## data: lista_evosuite2$Total
## W = 0.014813, p-value < 2.2e-16
```

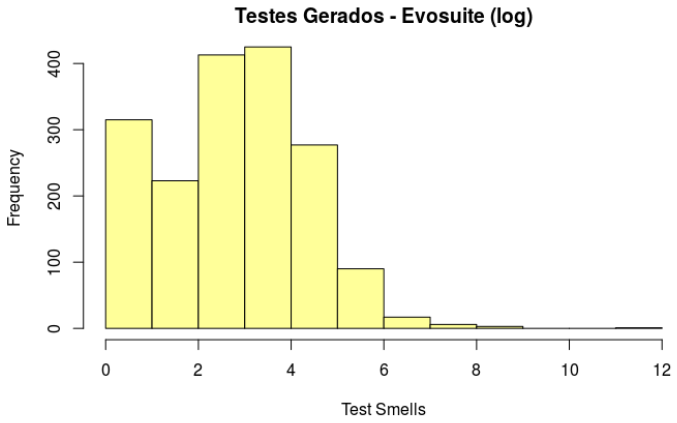


Figura 4.13 Histograma dos testes gerados pela Evosuite

4.3.4 Teste de Hipóteses

A partir dos resultados obtidos e descritos na seção anterior, percebe-se que os mesmos não seguem uma distribuição normal, e a partir desta informação o Teste de Wilcoxon foi escolhido para verificar as hipóteses.

Existentes x Randoop: Nessa seção apresentou as hipóteses para o estudo da relação dos pacotes de dados da Randoop e dos testes existentes, a seguir:

- Hipótese nula ($H_0 1$): Não há diferença significativa entre a quantidade de *test smells* "por classe de teste" (M6) detectados no código gerado pela ferramenta Randoop e os códigos existentes.

- Hipótese alternativa ($H_1 1$): Há diferença significativa entre a quantidade de *test smells* "por classe de teste"(M6) detectados no código gerado pela ferramenta Randoop e os códigos existentes.

Para responder as duas hipóteses foi realizado o teste de Wilcoxon, o qual obteve o resultado exibido na Lista 4.7 e na Figura 4.14.

Lista 4.7 Resultado do Teste de Wilcoxon

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista_randoop$Total and lista_existente$Total
## V = 311596, p-value < 2.2e-16
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
## 224.5 273.5
## sample estimates:
## (pseudo)median
## 247.4999
```

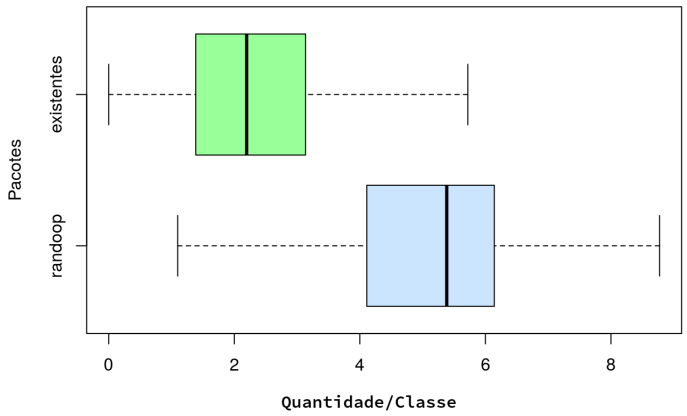


Figura 4.14 Boxplot dos dados da Randoop e dos dados existentes

Existente x Evosuite: Nesta seção apresentou as hipóteses para o estudo da relação dos pacotes de dados da Evosuite e dos testes existentes, a seguir:

- Hipótese nula ($H_0 2$): Não há diferença significativa entre a quantidade de *test smells* "por classe de teste"(M6) detectados no código gerado pela ferramenta Evosuite e os códigos existentes.

- Hipótese alternativa ($H_1 2$): Há diferença significativa entre a quantidade de *test smells* "por classe de teste"(M6) detectados no código gerado pela ferramenta Evosuite e os códigos existentes.

Para responder as duas hipóteses realizou o teste de Wilcoxon, o qual pode-se verificar o resultado na Lista 4.8 e na Figura 4.15.

Lista 4.8 Resultado do Teste de Wilcoxon

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista_evosuite$Total and lista_existente2$Total
## V = 160865, p-value < 2.2e-16
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
## 19.49992 28.50002
## sample estimates:
## (pseudo)median
## 24.00002
```

Randoop x Evosuite: Apresenta nesta seção as hipóteses para o estudo da relação dos pacotes de dados da Randoop e da Evosuite, a seguir:

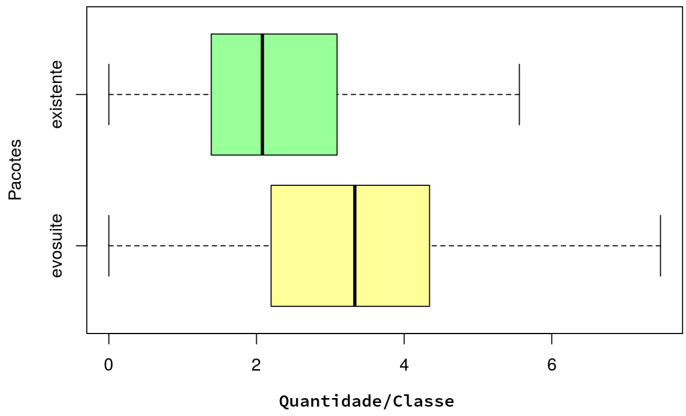


Figura 4.15 Boxplot dos dados da Evosuite e dos dados existentes

- Hipótese nula ($H_0$3): Não há diferença significativa entre a quantidade de *test smells* "por classe de teste"(M6) detectados no código gerado pelas ferramentas Randoop e Evosuite.
- Hipótese alternativa ($H_1$3): Há diferença significativa entre a quantidade de *test smells* "por classe de teste"(M6) detectados no código gerado pelas ferramentas Randoop e Evosuite.

Para responder as duas hipóteses realizou o teste de Wilcoxon, o qual pode-se verificar o resultado na Lista 4.9 e na Figura 4.16.

Lista 4.9 Resultado do Teste de Wilcoxon

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista__randoop2$Total and lista__evosuite2$Total
## V = 1362822, p-value < 2.2e-16
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
## 191.5 224.0
## sample estimates:
## (pseudo)median
## 207.5
```

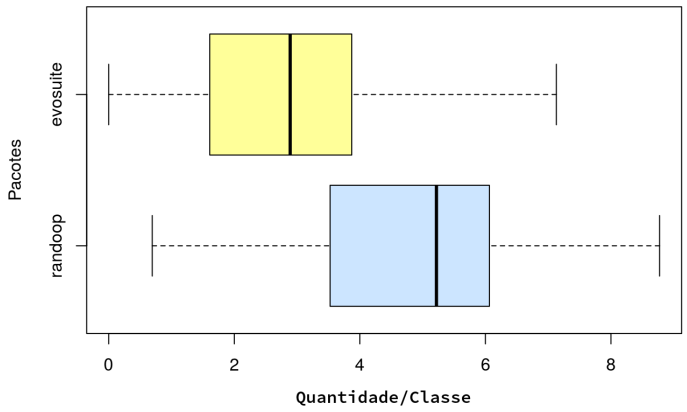


Figura 4.16 Boxplot dos dados da Evosuite e da Randoop

4.3.5 Análise de Coocorrência

Além das análises das hipóteses descritas na seção anterior, foi realizada uma análise das coocorrências dos *test smells* em cada pacote de teste. Com essa análise pode-se obter características de cada pacote de testes.

Verificou que os dados não estão distribuídos de forma normal a partir do teste de Shapiro-Wilk, que pode ser visualizado a partir dos resultados na Tabela 4.6 e disponibilizados no Apêndice A. Todos os resultados foram com *p-value* menores que 2.2^{-16} ou *identical*, o que significa que os dados não estão em uma distribuição normal. A partir desse resultado, foi utilizado o teste de correlação de Spearman para a verificação das correlações.

Tabela 4.6 Resultado do Teste de Shapiro-Wilk por *Test Smell* / Pacote

#	Test Smell (p-value)	Randoop	Existente	Evosuite
1	Assertion.Roulette	$< 2.2^{-16}$	$< 2.2^{-16}$	$< 2.2^{-16}$
2	Conditional.Test.Logic	$< 2.2^{-16}$	$< 2.2^{-16}$	$< 2.2^{-16}$
3	Constructor.Initialization	identical	$< 2.2^{-16}$	identical
4	Default.Test	identical	$< 2.2^{-16}$	identical
5	EmptyTest	identical	$< 2.2^{-16}$	$< 2.2^{-16}$
6	Exception.Catching.Throwing	$< 2.2^{-16}$	$< 2.2^{-16}$	$< 2.2^{-16}$
7	General.Fixture	identical	$< 2.2^{-16}$	$< 2.2^{-16}$
8	Mystery.Guest	identical	$< 2.2^{-16}$	$< 2.2^{-16}$
9	Print.Statement	identical	$< 2.2^{-16}$	identical
10	Redundant.Assertion	identical	$< 2.2^{-16}$	$< 2.2^{-16}$
11	Sensitive.Equality	$< 2.2^{-16}$	$< 2.2^{-16}$	$< 2.2^{-16}$
12	Verbose.Test	$< 2.2^{-16}$	$< 2.2^{-16}$	$< 2.2^{-16}$
13	Sleepy.Test	identical	$< 2.2^{-16}$	identical
14	Eager.Test	$< 2.2^{-16}$	$< 2.2^{-16}$	$< 2.2^{-16}$
15	Lazy.Test	$< 2.2^{-16}$	$< 2.2^{-16}$	$< 2.2^{-16}$
16	Duplicate.Assert	identical	$< 2.2^{-16}$	$< 2.2^{-16}$
17	Unknown.Test	$< 2.2^{-16}$	$< 2.2^{-16}$	$< 2.2^{-16}$
18	IgnoredTest	identical	$< 2.2^{-16}$	identical
19	Resource.Optimism	identical	$< 2.2^{-16}$	$< 2.2^{-16}$
20	Magic.Number.Test	$< 2.2^{-16}$	$< 2.2^{-16}$	$< 2.2^{-16}$
21	Dependent.Test	identical	identical	identical

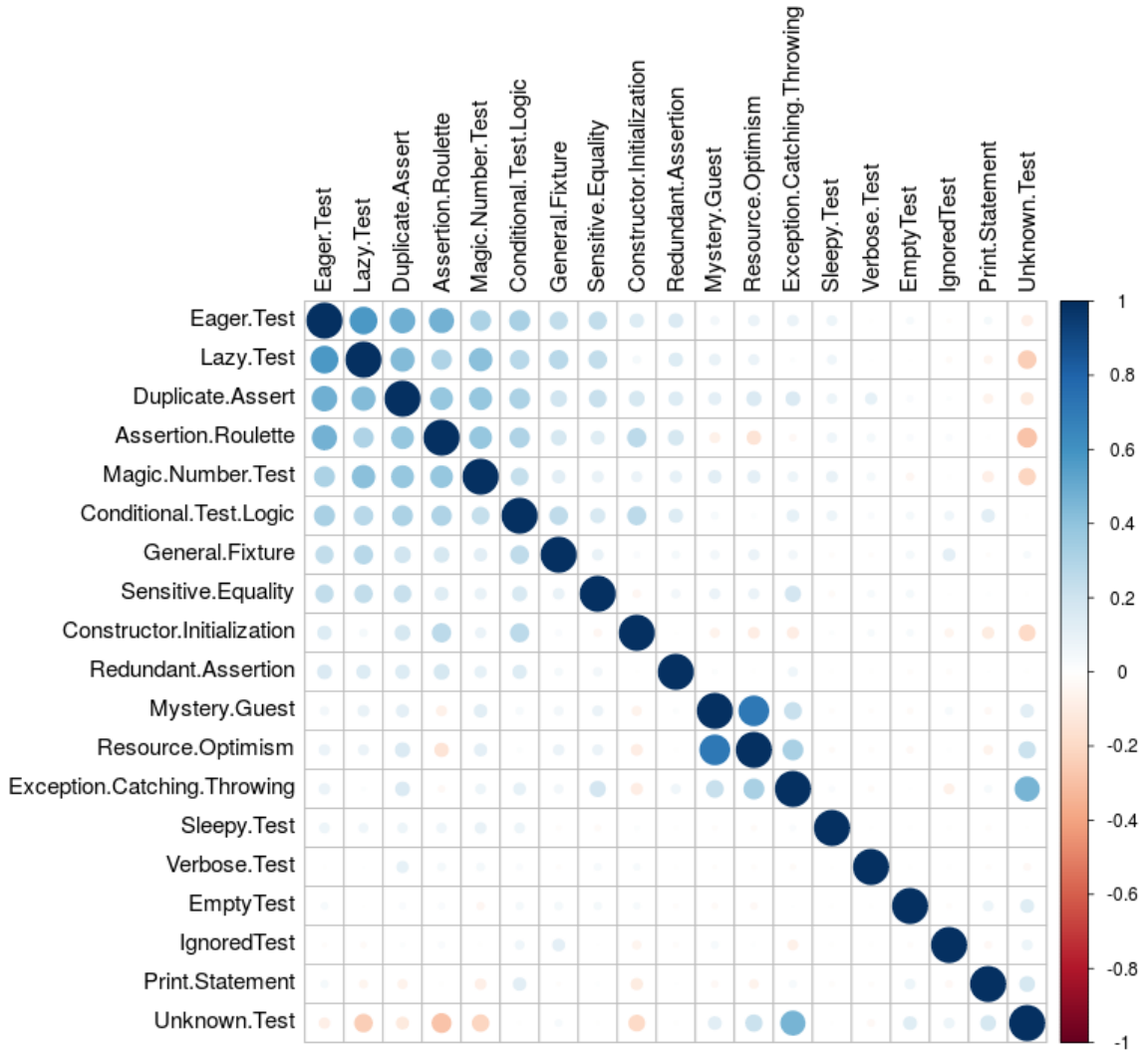


Figura 4.17 Coocorrência dos *test smells* nos testes existentes

A partir do gráfico da Figura 4.17 gerado com base nas correlações dos *test smells* do pacote existente de dados, é possível identificar que a maior parte das correlações encontradas são positivas, ficando as correlações moderadas a muito fortes concentradas entre os *test smells*: *Eager Test, Lazy Test, Duplicate Assert, Assertion Roulette, Magic Number Test.* É possível ainda verificar dois *outliers*: *Resource Optimism/Mystery Guest* e *Unknow Test/Exception Catching Throwing.*

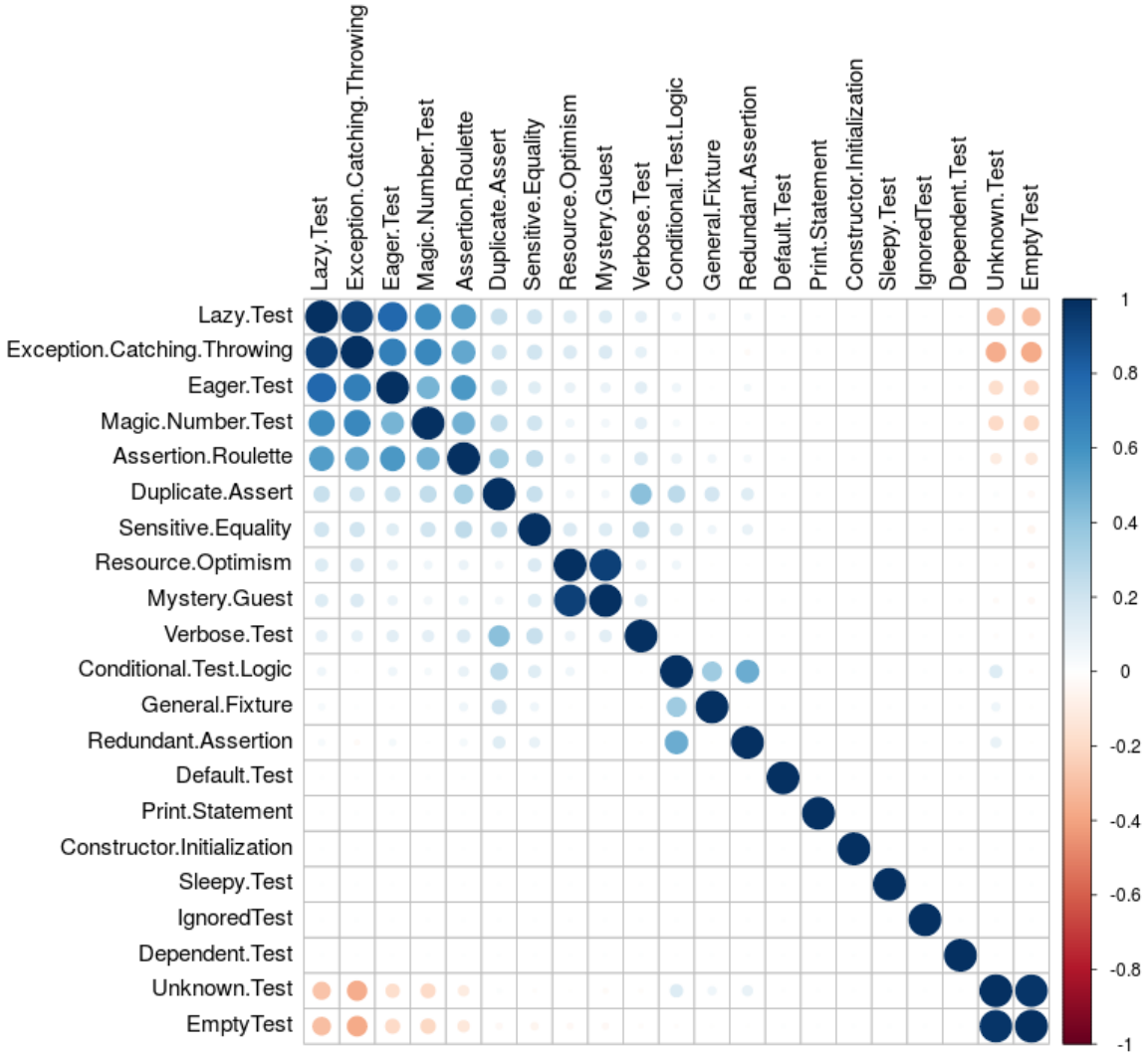


Figura 4.18 Coocorrência dos *test smells* nos testes gerados pela Evosuite

No gráfico da Figura 4.18 gerado a partir das correlações dos *test smells* do pacote de testes do Evosuite, é possível identificar que a maior parte das correlações encontradas são positivas, ficando as correlações moderadas a muito fortes concentradas entre os *test smells*: *Lazy Test, Exception Catching Throwing, Eager Test, Magic Number Test, Assertion Roulette.* É possível ainda verificar dois *outliers*: *Redundant Test/Conditional Test Logic* e *Verbose Test/Duplicate Assert.*

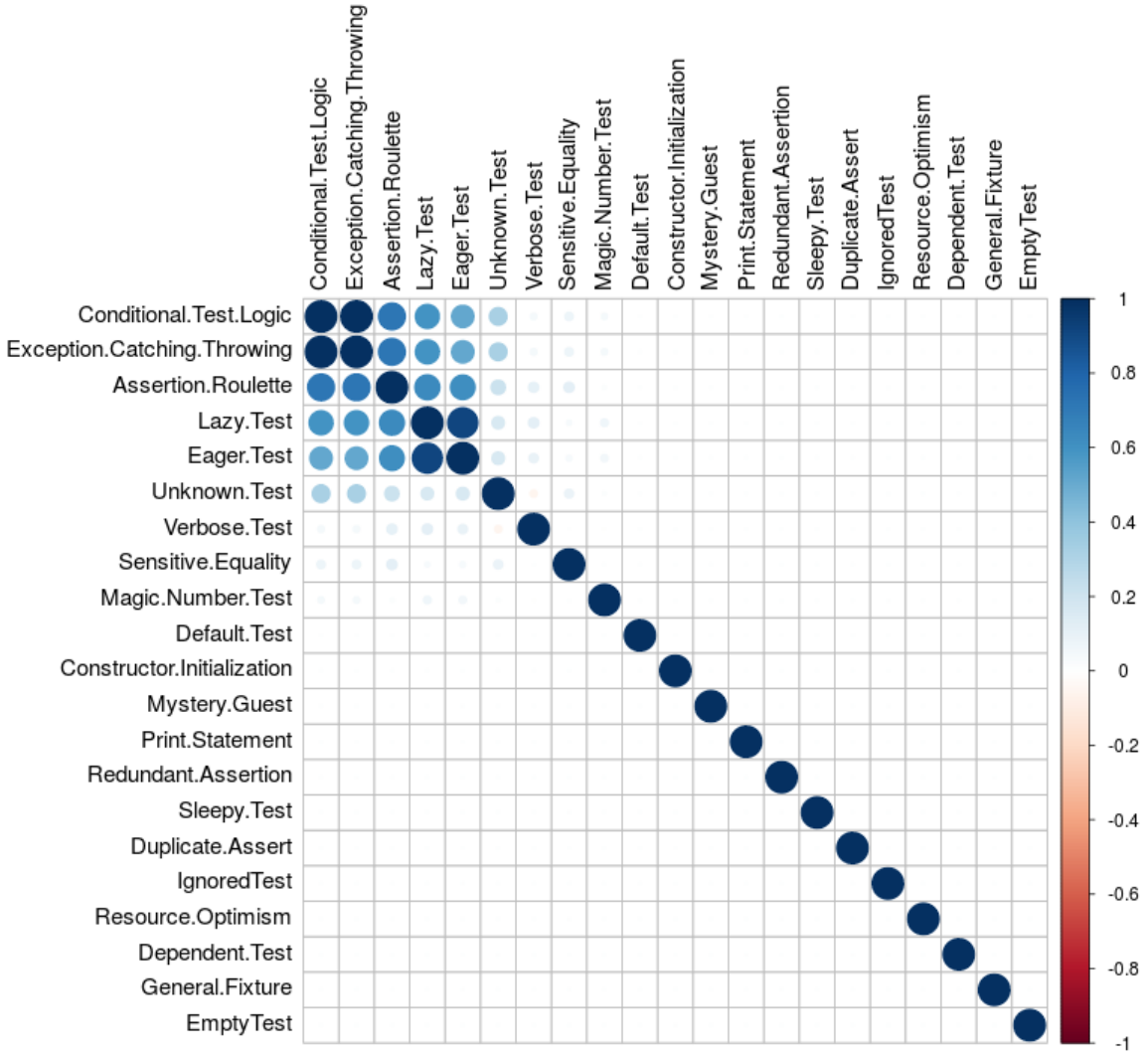


Figura 4.19 Coocorrência dos *test smells* nos testes gerados pela Randoop

Para o pacote de dados de testes da Randoop, tem-se o gráfico da Figura 4.19 que foi gerado a partir das correlações dos *test smells* do pacote, é possível identificar que a maior parte das correlações encontradas, como nos dados anteriores, são positivas, ficando as correlações moderadas a muito fortes concentradas entre os *test smells Conditional Test Logic, Exception Catching Throwing, Assertion Roulette, Lazy Test, Eager Test, Unknow Test*. Não foram encontrados *outliers* nesta amostra. A partir dos gráficos das Figuras 4.17, 4.18 e 4.19, nota-se características distintas para cada pacote nas coocorrências dos *test smells*.

4.4 DISCUSSÃO

Nessa seção ira discutir as descobertas apresentadas na seção anterior, descrevendo os resultados de forma geral e suas aplicações, as ameaças a validade do experimento e as lições aprendidas.

4.4.1 Avaliação dos resultados e implicações

A partir dos dados descritos na Seção 4.3 nota-se que as métricas estruturais (LOC e quantidade de métodos) tiveram uma diferença considerável entre os pacotes de testes. Esses dados descrevem as características estruturais dos pacotes de testes, onde o pacote de testes da Randoop teve quantidade maior de linhas de código e métodos de testes. O pacote de teste da Evosuite ficou em segundo lugar e o pacote de testes existentes obteve os menores valores para essas métricas.

Com esses dados, pode-se inferir, de acordo com a descrição da ferramenta na Seção 2.4, que as características estruturais estão de acordo com o esperado entre os pacotes de testes geradores e os existentes, pois as ferramentas não tem como prioridade manter um tamanho razoável de linhas de código e quantidade de métodos de testes, como descrito na seção 2.4.

Quando as métricas estruturais entre as ferramentas são comparadas, há uma diferença significativa, onde o pacote de teste da Randoop tem aproximadamente cinco vezes o tamanho do pacote de teste da Evosuite. Como mencionado no estudo de Palomba et al. (2016), os *test smells* têm forte correlação com características estruturais, o que se comprovou nos dados encontrados quando comparados ao quantitativo dos *test smells* encontrados com as métricas estruturais LOC e quantidade de métodos.

Com relação a dispersão dos *test smells* nos códigos de teste, o estudo encontrou resultados maiores do que os dois estudos relacionados de Palomba et al. (2016) e Bavota et al. (2015). Observou pelo menos um *test smell* em 97% dos testes existentes, o qual comparado com o estudo de Bavota et al. (2015) que encontrou uma dispersão de 85%, o atual estudo encontrou um valor maior. No estudo de Palomba et al. (2016) os códigos de testes foram gerados pela Evosuite e foi encontrada uma dispersão de 83%, enquanto que, no atual estudo encontrou-se dispersão de 99.95% para a Evosuite. Esses valores podem ter como motivo o quantitativo de tipos test smells possíveis de serem detectados pela ferramenta JNose Test, que difere dos estudos passados.

Como no estudo de Palomba et al. (2016), que encontrou co-ocorrência entre os *test smells*, o atual estudo, na Seção 4.3.5, realizou uma análise de correlação dos *test smells* nos pacotes de testes e conseguiu averiguar que determinados tipos de *test smells* ocorrem em conjunto, mas verificou-se diferentes características dessas correlações entre os pacotes, o que pode indicar que eles sofreram influência pela forma que foram desenvolvidos os testes.

4.4.2 Respostas das questões de pesquisa

A partir dos resultados das análises, as seguintes respostas das questões de pesquisa foram alcançadas a seguir:

- **Existe diferença (estatisticamente significativa) na qualidade do código de teste gerado pelas ferramentas Randoop e Evosuite?** Em comparação realizada a partir dos testes de hipóteses, da análise de coocorrência e da distribuição dos *test smells*, constatou-se que os testes da Randoop diferem de forma significativa dos testes da Evosuite, encontrou aproximadamente seis vezes mais *test smells* do que nos testes da Evosuite e a distribuição de *test smells* ficou em 99,53% para a Randoop e 99,95% para a Evosuite, mostrando diferença de 0,42%. Tendo os *test smells* como base para a análise da qualidade neste estudo, então conclui que os testes da Evosuite tem melhor qualidade do que os testes gerados pela Randoop, mas percebeu que a distribuição de *test smells* foi praticamente a mesma e a quantidade de tipos de *test smells* foi maior na Evosuite;

- **Existe diferença (estatisticamente significativa) na qualidade do código de teste gerado pela ferramenta Randoop e os códigos de testes pré-existentes?** Em comparação realizada a partir dos testes de hipóteses, da análise de coocorrência e da distribuição dos *test smells*, constatou-se que os testes existentes diferem de forma significativa dos testes da Randoop, encontrou trinta vezes mais *test smells* do que nos testes existentes, e a distribuição de *test smells* ficou em 99,53%, valor acima dos testes existentes. Utilizou os *test smells* como base para a análise da qualidade neste estudo, então conclui-se que os testes existentes tem melhor qualidade do que os testes gerados pela Randoop;

- **Existe diferença (estatisticamente significativa) na qualidade do código de teste gerado pela ferramenta Evosuite e os códigos de testes pré-existentes?** Em comparação realizada a partir dos testes de hipóteses, da análise de coocorrência e da distribuição dos *test smells*, constata-se que os testes existentes diferem de forma significativa dos testes da Evosuite, observou-se aproximadamente 4 vezes mais *test smells* do que nos testes existentes e a distribuição de *test smells* ficou em 99,95%, valor acima dos testes existentes. Utilizando os *test smells* como base para a análise da qualidade neste estudo, então conclui-se que os testes existentes tem melhor qualidade do que os testes gerados pela Evosuite.

Características distintas foram notadas para cada pacote nas coocorrências dos *test smells*, quantidade de linhas de código, tipos de *test smells*, quantidade de testes, e quantitativo de *test smells*. Os resultados encontrados mostram que os *test smells* estão presentes na maior parte dos testes, se destacando pelo quantitativo no pacote de teste da Randoop, o qual é um dos diferenciais deste estudo perante os anteriores. Foi alcançado resultados parecidos com os dos estudos anteriores, mas com a distribuição dos *test smells* em mais classes. Além disso, foi possível identificar correlações entre os *test smells*, mas essas correlações foram diferentes para cada pacote de teste, como também os tipos de *test smells* encontrados por pacote.

Esses dados mostram uma oportunidade de melhorias possíveis na ferramenta Evosuite, sugerindo a refatoração dos códigos de geração de teste com foco na eliminação dos

Lazy Tests, de forma bem significativa para a ferramenta.

4.4.3 Ameaças à validade

Uma questão fundamental sobre os resultados de um experimento é a validade dos resultados. Assim, é necessário antecipar as ameaças possivelmente envolvidas no contexto de um experimento. Jedlitschka, Ciolkowski e Pfahl (2008) adotam a categorização de quatro tipos das ameaças à validade dos resultados experimentais. São eles:

Validade Interna - diz respeito a questões que podem afetar a variável independente em relação à causalidade, sem o conhecimento dos pesquisadores (JEDLITSCHKA; CIOLKOWSKI; PFAHL, 2008). Neste estudo, observou duas ameaças principais à validade interna: as ferramentas de teste automatizadas selecionadas e a ferramenta de detecção de *smell* selecionada. Nos dois casos, contou-se com os dados fornecidos pelas ferramentas e não tendo controle sobre como elas foram implementadas;

Validade Externa - diz respeito à generalização do resultado do experimento para outros ambientes além daquele em que o estudo é conduzido (JEDLITSCHKA; CIOLKOWSKI; PFAHL, 2008). No experimento, foram selecionados de forma aleatória projetos de código aberto no repositório do GitHub. Embora acreditemos que essa escolha possa ser considerada uma boa estratégia, o que pode reduzir qualquer viés, não pode garantir que o conjunto de projetos recuperados seja o mais representativo. Portanto, um próximo passo para alcançar a validade externa é replicar esse experimento no futuro com dados de outros projetos;

Validade de Construção - diz respeito à generalização do resultado do experimento para um conceito ou teoria por trás do experimento (JEDLITSCHKA; CIOLKOWSKI; PFAHL, 2008). Como os projetos utilizados são do mundo real, não se notou alguma ameaça a essa validade neste projeto de estudo;

Validade de Conclusão - diz respeito à análise estatística dos resultados e à composição dos sujeitos (JEDLITSCHKA; CIOLKOWSKI; PFAHL, 2008). Neste experimento, a verificação de hipótese foi feita por meio de simples demonstração de presença quantificada de *test smells* como métrica de qualidade a partir dos dados gerados pela ferramenta JNose Test.

4.5 SÍNTESE DO CAPÍTULO

Este capítulo apresenta o estudo experimental, que buscou analisar a qualidade dos testes de unidade a partir dos *test smells*. O capítulo detalha o planejamento, o contexto, as hipóteses, e as variáveis. Apresenta também o projeto-piloto, realizado para validar os processos descritos no planejamento. Por fim, discute os resultados encontrados a partir dos pacotes de testes gerados pelas ferramentas Randoop e Evosuite, além dos testes existentes nos projetos selecionados.

Capítulo

5

CONSIDERAÇÕES FINAIS

"Quem testa os testes?" Diante de tal questionamento, neste trabalho buscou avaliar a qualidade dos testes de software, sob a perspectiva da existência de *test smells* no código de testes. A investigação consistiu em avaliar empiricamente a qualidade de testes gerados manual e automaticamente. Como dados observados, considerou um conjunto de 21 projetos de software de código-fonte aberto, disponíveis na Plataforma Github. Esses projetos foram executados em duas ferramentas de geração automatizada de testes: Randoop e Evosuite.

A avaliação empírica levou em consideração três hipóteses primárias: *"Não há diferença significativa entre a quantidade de test smells "por classe de teste" (M6) detectados no código gerado pela ferramenta Randoop e os códigos existentes.", "Não há diferença significativa entre a quantidade de test smells "por classe de teste"(M6) detectados no código gerado pela ferramenta Evosuite e os códigos existentes."* e *"Não há diferença significativa entre a quantidade de test smells "por classe de teste"(M6) detectados no código gerado pelas ferramentas Randoop e Evosuite."*. Como resultado da avaliação, todas as hipóteses foram refutadas. Uma outra observação bastante significativa do estudo foi a correlação entre os *test smells*, onde cada suíte apresentou pares de correlações de *test smells* distintas. Com isso, concluiu que as características dos *test smells* não são generalizáveis para as metodologias de geração de códigos de teste.

Os resultados aferidos com o estudo destacam a importância do processo de qualidade dos códigos de teste, gerados ou não de forma automatizada, com base nos *test smells* detectados. Observou que os *test smells* são distribuídos em mais que 97% das classes de teste, independentemente do pacote analisado.

As ferramentas de geração de teste usam metodologias diferentes e, portanto, apresentaram características diferentes de *test smells*, tais como: quantidade total de *test smells*, quantidade por tipos, quantidade de tipos e correlações entre os *test smells*.

O restante do capítulo está estruturado como segue: a Seção 5.1, apresenta os trabalhos relacionados, com destaque para as diferenças mais significativas entre a literatura

comparada e a proposta apresentada neste trabalho. A Seção 5.2, apresenta as contribuições deste trabalho para a área de Engenharia de Software. Na seção 5.3, apresenta as oportunidades de trabalhos futuros com relação ao tema abordado.

5.1 TRABALHOS RELACIONADOS

Nesta seção, os trabalhos relacionados são apresentados conforme as temáticas abordadas neste estudo.

5.1.1 Avaliação da qualidade de código de teste gerado automaticamente

O Evosuite e o Randoop são ferramentas bastante utilizadas na indústria para a geração automatizada de testes de software (ALMASI et al., 2017; SILVA; ALVES; ANDRADE, 2017; FRASER; ARCURI, 2014). Em Almasi et al. (2017), a eficácia das duas ferramentas são analisadas e comparadas. Um dos *insights* levantados é sobre a melhoria da legibilidade dos testes gerados, questão intrinsecamente ligada ao estudo com os *test smells*.

Em um outro estudo, Silva, Alves e Andrade (2017) investigaram a eficiência de suítes geradas automaticamente por ferramentas de automação de testes, quanto à capacidade de detecção de falhas de refatoração. Assim como o estudo anterior (cuja estratégia também foi seguida em nossa investigação), foram utilizadas as ferramentas Randoop e Evosuite, que resultaram em uma perda na detecção de cerca de 50% de todas as falhas. Isso mostra um *gap* para a melhoria dos códigos gerados.

A ferramenta Evosuite é também mencionada no estudo de Fraser e Arcuri (2014), que discute um experimento realizado com 100 projetos projetos de software de código-fonte aberto. O estudo confirma o bom alcance de níveis de cobertura da Evosuite. O estudo também exemplifica como a escolha de sistemas de software para um estudo empírico pode influenciar os resultados dos experimentos.

O presente estudo realizou um comparativo entre as ferramentas Evosuite e Randoop. Entretanto, ao invés de utilizar as métricas de cobertura como dados comparativos, foi utilizado os *test smells*, e ainda, utilizou no comparativo os códigos de testes existentes nos projetos. Isso possibilitou um estudo aprofundado sobre a existência dos *test smells* nos códigos de testes.

5.1.2 Detecção automatizada de *Test Smells*

Palomba et al. (2016) conduziram uma análise em larga escala da difusão de *test smells* em testes de unidade gerados automaticamente. A análise mostrou alta difusão dos *test smells* e frequente co-ocorrência de diferentes tipos nos projetos. O estudo verificou forte correlação positiva com características estruturais (tamanho e quantidade de classes). Na presente investigação, estendeu-se o estudo de Palomba et al. (2016) nas técnicas de geração de testes de unidade e sua influência nas ocorrências dos *test smells* encontrados. Para tal, foi acrescentado mais um gerador de teste, a Randoop, e foram utilizados os códigos de testes pré-existentes.

O trabalho Bavota et al. (2015) apresenta duas investigações empíricas sobre a preva-

lência e o impacto dos *test smells* nos projetos. A primeira investigação mostrou que há grande difusão de *test smells* em sistema de software de código aberto e industriais, com 85% dos testes de unidade contendo pelo menos um *test smell*. A segunda investigação fornece evidências do impacto negativo na compreensão e na manutenção dos códigos de teste do sistema, dificultando em 30% a compreensão daqueles que tinham *test smells*. Como uma contribuição do estudo, foi desenvolvido o Test Smells Detector (tsDetect), a ferramenta utilizada para o levantamento de dados do estudo. Essa ferramenta foi utilizada como base para o desenvolvimento da JNose Test, a qual foi desenvolvida a partir da necessidade de automatizar o processo de execução e facilitar o acesso a outros desenvolvedores, além de incluir novas métricas e requisitos os quais não estavam presentes na tsDetect, por exemplo, o quantitativo dos *test smells* encontrados e as métricas de cobertura de código.

5.2 CONTRIBUIÇÕES

A principal contribuição desse trabalho para a área de Engenharia de Software consistiu em investigar as diferenças entre a qualidade dos códigos de teste gerados de forma automática (ou não) em relação a presença de *test smells*. Em resumo, as contribuições do trabalho são:

- O desenvolvimento da ferramenta JNose Test, desenvolvida utilizando a linguagem Java, que tem como funcionalidade a detecção de *test smells* em códigos de testes de unidade desenvolvidos com o *framework* JUnit. Além de trazer métricas de cobertura dos testes, facilita as etapas de seleção e de processamento para o usuário final a partir de uma interface web amigável;

- Um estudo quantitativo para investigar a correlação entre a presença de *test smells* e a cobertura do código de teste (VIRGíNIO et al., 2019);

- Um estudo exploratório para identificar se há diferença significativa entre a quantidade de *test smells* inseridos em códigos de teste gerados a partir de ferramentas automatizadas e de códigos pré-existentes.

5.3 TRABALHOS FUTUROS

Diante das possibilidades trazidas por este estudo, a seguir apresenta as oportunidades para investigações futuras.

- Conduzir estudos que repliquem o *design* experimental apresentado nesta investigação, no sentido de incluir amostras de tamanho maior (maior quantidade de projetos avaliados e/ou projetos de maior tamanho que aqueles observados no presente estudo); ou ainda, considerar projetos industriais, cujo código-fonte não esteja disponível em repositórios abertos. Isso possibilitará coletar mais evidências, o que acarretará em oportunidades de generalização dos resultados encontrados;

- Realizar a análise de dados de outras ferramentas automatizadas para a geração de testes, de modo a observar se os resultados são semelhantes aos alcançados com as ferramentas Evosuite e Randoop e com os códigos de testes pré-existentes;
- Criação de um *plugin* do `IntelliJ IDEA` de detecção de *test smells* na etapa de codificação dos testes, para a realização de uma análise da qualidade dos testes e apresentação de *bugs* a partir de sua utilização;
- Realizar um estudo sobre o impacto da implantação do uso das refatorações baseadas nos *test smells* em projetos industriais, com acompanhamento da equipe de desenvolvimento (através do método de *action research*), e qual o impacto dessa implementação na qualidade do projeto;
- Realizar uma análise sobre a correlação de métricas de dívidas técnicas e os *test smells* ao longo do desenvolvimento do software;
- Realizar um estudo empírico que verifique a correlação entre a experiência do desenvolvedor com a inserção de *test smells* ao longo do desenvolvimento de um sistema, utilizando o histórico de projetos no GitHub;
- Melhoria da ferramenta JNose Test para funcionar como *plugin* em sistemas de integração contínua, tais como, o Jenkins[1] e Hudson[2]. Isso possibilitará a integração da JNose Test com projetos que utilizam essas ferramentas.

[1] <https://jenkins.io/>
[2] <https://www.eclipse.org/hudson/>

REFERÊNCIAS

ABBES, M.; KHOMH, F.; GUéHéNEUC, Y.-G.; ANTONIOL, G. An empirical study of the impact of two antipatterns on program comprehension outline introduction. *Empirical Software Engineering*, Springer International Publishing, 2011.

ALMASI, M. M.; HEMMATI, H.; FRASER, G.; ARCURI, A.; BENEFELDS, J. An industrial evaluation of unit test generation: Finding real faults in a financial application. May 2017.

BARTIé, A. *Garantia da qualidade de software*. [S.l.]: CAMPUS-RJ, 2002.

BAVOTA, G.; QUSEF, A.; OLIVETO, R.; LUCIA, A. D.; BINKLEY, D. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. p. 56–65, 2012.

BAVOTA, G.; QUSEF, A.; OLIVETO, R.; LUCIA, A. D.; BINKLEY, D. Are test smells really harmful? an empirical study. *Empirical Software Engineering*, Aug 2015.

BECK, K. *Extreme Programming Explained: Embrace Change*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

CISQ. *The Cost of Poor Quality Software in the US 2018 Report.* Consortium for Information & Software Quality, 2019. Acessado em fevereiro de 2020. Disponível em: <https://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2018-report/The-Cost-of-Poor-Quality-Software-in-the-US-2018-Report.pdf>.

CROSBY, P. *Quality is Free: The Art of Making Quality Certain*. [S.l.]: McGraw-Hill, 1979. (Mentor book).

DEURSEN, A.; MOONEN, L. M.; BERGH, A.; KOK, G. *Refactoring Test Code*. NLD, 2001.

EHA, B. P. *CNN-Is Knight's $440 million glitch the costliest computer bug ever?* 2012. Acessado em fevereiro de 2020. Disponível em: <https://money.cnn.com/2012/08/09/technology/knight-expensive-computer-bug/index.html>.

FOKAEFS, M.; TSANTALIS, N.; CHATZIGEORGIOU, A. Jdeodorant: Identification and removal of feature envy bad smells. p. 519–520, Oct 2007.

FOWLER, M. *Refactoring: Improving the Design of Existing Code (Object Technology Series)*. illustrated edition. Amsterdam: Addison-Wesley Longman, Amsterdam, 1999.

FRASER, G.; ARCURI, A. Evosuite: Automatic test suite generation for object-oriented software. *SIGSOFT/FSE 2011-Proceedings of the 19th ACM SIGSOFT Symposium on Foundations of Software Engineering*, p. 416–419, 09 2011.

FRASER, G.; ARCURI, A. A large-scale evaluation of automated unit test generation using evosuite. *ACM Transactions on Software Engineering and Methodology*, v. 24, p. 1–42, 12 2014.

GAROUSI, V.; KUCUK, B. Smells in software test code: A survey of knowledge in industry and academia. *Journal of Systems and Software*, v. 138, p. 52–81, 2018.

GOPINATH, R.; JENSEN, C.; GROCE, A. Code coverage for suite evaluation by developers. In: *Proceedings of the 36th International Conference on Software Engineering*. [S.l.: s.n.], 2014. (ICSE). ISBN 978-1-4503-2756-5.

GRAHAM, D.; VEENENDAAL, E. V.; EVANS, I.; BLACK, R. *Foundations of Software Testing: ISTQB Certification*. [S.l.]: Intl Thomson Business Pr, 2008. ISBN 9781844803552.

GRYNA, F. *Quality Planning and Analysis: From Product Development Through Use*. [S.l.]: McGraw-Hill, 2001. (McGraw-Hill series in industrial engineering and management science).

GUERRA, A.; COLOMBO, R. *Tecnologia da informação: qualidade de produto de software*. Ministério da Ciência e Tecnologia, 2009. Disponível em: <https://books.google.com.br/books?id=5j1sQwAACAAJ>.

HEDAYATI, A.; EBRAHIMZADEH, M.; SORI, A. A. Investigating into automated test patterns in erratic tests by considering complex objects. *International Journal of Information Technology and Computer Science*, v. 7, p. 54–59, 2015.

HIRAMA, K. *Engenharia de software: Qualidade e produtividade com tecnologia*. Rio de Janeiro: Elsevier Editora Ltda., 2012.

JEDLITSCHKA, A.; CIOLKOWSKI, M.; PFAHL, D. *Reporting Experiments in Software Engineering*. London: Springer London, 2008. 201–228 p.

KHOMH, F.; PENTA, M. D.; GUÉHÉNEUC, Y.-G.; ANTONIOL, G. An exploratory study of the impact of antipatterns on class change- and fault-proneness. *Empirical Software Engineering*, Springer International Publishing, 2011.

MESZAROS, G.; SMITH, S. M.; ANDREA, J. The test automation manifesto. In: MAURER, F.; WELLS, D. (Ed.). *Extreme Programming and Agile Methods-XP/Agile Universe 2003*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. p. 73–81.

MYERS, G. J.; SANDLER, C. *The Art of Software Testing*. USA: John Wiley Sons, Inc., 2004.

NAIK, K.; TRIPATHY, P. *Software Testing and Quality Assurance: Theory and Practice*. 2nd. ed. Miami: Wiley Publishing, 2018.

OPDYKE, W. F. *Refactoring Object-oriented Frameworks*. Tese (Doutorado) — University of Illinois, Champaign, IL, USA, 1992. UMI Order No. GAX93-05645.

OSMAN, I. H.; KELLY, J. P. *Meta-Heuristics: Theory and Applications*. USA: Springer US, 1996.

PACHECO, C.; ERNST, M. Randoop: Feedback-directed random testing for java. *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA*, p. 815–816, 01 2007.

PACHECO, C.; ERNST, M. D. Eclat: Automatic generation and classification of test inputs. Glasgow, Scotland, p. 504–527, jul. 2005.

PACHECO, C.; LAHIRI, S. K.; BALL, M. D. E. T. Feedback-directed random test generation. Minneapolis, MN, USA, p. 75–84, maio 2007.

PACHECO, C.; LAHIRI, S. K.; BALL, T. Finding errors in .net with feedback-directed random testing. Seattle, Washington, July 20–24, 2008.

PALOMBA, F.; NUCCI, D. D.; PANICHELLA, A.; OLIVETO, R.; LUCIA, A. D. On the diffusion of test smells in automatically generated test code: An empirical study. p. 5–14, May 2016.

PERUMA, A.; ALMALKI, K.; NEWMAN, C. D.; MKAOUER, M. W.; OUNI, A.; PALOMBA, F. On the distribution of test smells in open source android applications: An exploratory study. IBM Corp., Riverton, NJ, USA, p. 193–202, 2019.

RANDOOP. 2018. Acessado em fevereiro de 2020. Disponível em: <https://randoop.github.io/randoop/>.

RICADEL, A. *The State of Software Quality*. InformationWeek, 2001. Disponível em: <http://www.informationweek.com/838/quality.htm>.

ROMPAEY, B. V.; BOIS, B. D.; DEMEYER, S.; RIEGER, M. On the detection of test smells: A metrics-based approach for general fixture and eager test. *IEEE Transactions on Software Engineering*, v. 33, n. 12, p. 800–817, Dec 2007.

RUEDA, U.; JUST, R.; GALEOTTI, J. P.; VOS, T. E. J. Unit testing tool competition: Round four. ACM, New York, NY, USA, p. 19–28, 2016.

SALKIND, N. J.; RAINWATER, T. *Exploring research*. [S.l.]: Prentice Hall Upper Saddle River, NJ, 2003.

SCHMAUCH, C. H. *ISO 9000 for Software Developers*. 2nd. ed. [S.l.]: ASQ Quality Press, 1995.

SEDGWICK, K. *Bad Code Has Lost $500 Million of Cryptocurrency in Under a Year*. 2018. Acessado em fevereiro de 2020. Disponível em: <https://news.bitcoin.com/bad-code-has-lost-500-million-of-cryptocurrency-in-under-a-year/>.

SHARMA, T.; SPINELLIS, D. A survey on software smells. *Journal of Systems and Software*, v. 138, p. 158–173, 2018.

SILVA, I. P. S. C.; ALVES, E. L. G.; ANDRADE, W. L. Analyzing automatic test generation tools for refactoring validation. IEEE Press, Piscataway, NJ, USA, p. 38–44, 2017.

SOMÉ, S. S.; CHENG, X. An approach for supporting system-level test scenarios generation from textual use cases. ACM, New York, NY, USA, p. 724–729, 2008.

SOMMERVILLE, I. *Engenharia de software*. 9th. ed. Brasil: PEARSON BRASIL, 2011.

TRICENTIS. *The 5th Edition of the Software Fail Watch identified 606 recorded software failures, impacting half of the world's population (3.7 billion people), $1.7 trillion in assets, and 314 companies.* 2019. Acessado em fevereiro de 2020. Disponível em: <https://www.tricentis.com/resources/software-fail-watch-5th-edition/>.

VIRGíNIO, T. *DATASET - AVALIAÇÃO EMPÍRICA DA GERAÇÃO AUTOMATIZADA DE TESTES DE SOFTWARE SOB A PERSPECTIVA DE TEST SMELLS*. Zenodo, 2020. Disponível em: <https://doi.org/10.5281/zenodo.3715424>.

VIRGíNIO, T.; RAILANA, S.; MARTINS, L. A.; SOARES, L. R.; COSTA, H.; MACHADO, I. On the influence of test smells on test coverage. ACM, New York, NY, USA, p. 467–471, 2019.

WOOD, C. L.; ALTAVELA, M. M. Large-sample results for Kolmogorov-Smirnov statistics for discrete distributions. *Biometrika*, v. 65, n. 1, p. 235–239, 04 1978. Disponível em: <https://doi.org/10.1093/biomet/65.1.235>.

Anexo

A

EXEMPLOS DE *TEST SMELLS*

Este anexo apresenta exemplos de *smells* em código de testes.

A.1 *ASSERTION ROULETTE*

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

Lista A.1 Exemplo: *Assertion Roulette*

A.2 *EAGER TEST*

```
1 @Test
2 public void NmeaSentence_GPGSA_ReadValidValues(){
3     NmeaSentence nmeaSentence = new NmeaSentence("$GPGSA,A,3,04,05,,09,12,,,24,,,,,2.5,1.3,2.1*39");
4     assertThat("GPGSA - read PDOP", nmeaSentence.getLatestPdop(), is("2.5"));
5     assertThat("GPGSA - read HDOP", nmeaSentence.getLatestHdop(), is("1.3"));
6     assertThat("GPGSA - read VDOP", nmeaSentence.getLatestVdop(), is("2.1"));
7 }
```

Lista A.2 Exemplo: *Eager Test*

A.3 *GENERAL FIXTURE*

```
1 protected void setUp() throws Exception {
2     assetManager = getInstrumentation().getContext().getAssets();
3     certificateFactory = CertificateFactory.getInstance("X.509");
4     infoDebianTestCA = loadCertificateInfo("DebianTestCA.pem");
5     infoDebianTestNoCA = loadCertificateInfo("DebianTestNoCA.pem");
```

```
6       infoGTECyberTrust = loadCertificateInfo("GTECyberTrustGlobalRoot.pem");
7       // user-submitted test cases
8    infoMehlMX = loadCertificateInfo("mehl.mx.pem");
9   }
10
11  public void testIsCA() {
12      assertTrue(infoDebianTestCA.isCA());
13      assertFalse(infoDebianTestNoCA.isCA());
14      assertNull(infoGTECyberTrust.isCA());
15      assertFalse(infoMehlMX.isCA());
16  }
```

Lista A.3 Exemplo: *General Fixture*

A.4 *LAZY TEST*

```
1
2   @Test
3   public void testDecrypt() throws Exception {
4       FileInputStream file = new FileInputStream(ENCRYPTED_DATA_FILE_4_14);
5       byte[] enfileData = new byte[file.available()];
6       FileInputStream input = new FileInputStream(DECRYPTED_DATA_FILE_4_14);
7       byte[] fileData = new byte[input.available()];
8       input.read(fileData);
9       input.close();
10      file.read(enfileData);
11      file.close();
12      String expectedResult = new String(fileData, "UTF-8");
13      assertEquals("Testing simple decrypt",expectedResult, Cryptographer.decrypt(enfileData, "test"));
14  }
15
16  @Test
17  public void testEncrypt() throws Exception {
18      String xml = readFileAsString(DECRYPTED_DATA_FILE_4_14);
19      byte[] encrypted = Cryptographer.encrypt(xml, "test");
20      String decrypt = Cryptographer.decrypt(encrypted, "test");
21      assertEquals(xml, decrypt);
22  }
```

Lista A.4 Exemplo: *Lazy Test*

A.5 *MYSTERY GUEST*

```
1   public void testPersistence() throws Exception {
2       File tempFile = File.createTempFile("systemstate-", ".txt");
3       try {
4           SystemState a = new SystemState(then, 27, false, bootTimestamp);
5           a.addInstalledApp("a.b.c", "ABC", "1.2.3");
6
7           a.writeToFile(tempFile);
8           SystemState b = SystemState.readFromFile(tempFile);
9
10          assertEquals(a, b);
11      } finally {
12          //noinspection ConstantConditions
13          if (tempFile != null) {
14              //noinspection ResultOfMethodCallIgnored
15              tempFile.delete();
16          }
17      }
18  }
```

Lista A.5 Exemplo: *Mystery Guest*

A.6 *RESOURCE OPTIMISM*

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

Lista A.6 Exemplo: *Resource Optimism*

A.7 *CONDITIONAL TEST LOGIC*

```
/* ** Test method contains multiple control statements ** */
@Test
public void testSpinner () {
/* ** Control statement #1 ** */
for ( Map . Entry < String , String > entry : sourcesMap . entrySet () ) {
.....
/* ** Control statement #2 ** */
if ( resultObject instanceof EventsModel ) {
EventsModel result = ( EventsModel ) resultObject ;
/* ** Control statement #3 ** */
if ( result . testSpinner . runTest ) {
.....
/* ** Control statement #4 ** */
for (int i = 0; i < spinnerAdapter . getCount () ; i ++) {
assertEquals ( spinnerAdapter . getItem ( i ) , result . testSpinner .
data . get ( i ) ) ;
.....
}
```

Lista A.7 Exemplo: *Conditional Test Logic*

A.8 *CONSTRUCTOR INITIALIZATION*

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

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

 @Test
```

```
12  public void testKeyAffectsTag() throws Exception {
13   Set set = new HashSet<>();
14   for (int i = 0; i < 100; i++) {
15    byte[] tag = new byte[TAG_LENGTH];
16    SecretKey tagKey = TestUtils.getSecretKey();
17    crypto.encodeTag(tag, tagKey, PROTOCOL_VERSION, streamNumber);
18    assertTrue(set.add(new Bytes(tag)));
19   }
20  }
21  ...
22 }
23 }
```

Lista A.8 Exemplo: *Constructor Initialization*

A.9 *DEFAULT TEST*

```
1  public class ExampleUnitTest {
2     @Test
3   public void addition_isCorrect() throws Exception {
4    assertEquals(4, 2 + 2);
5   }
6
7     @Test
8     public void shareProblem() throws InterruptedException {
9      .....
10   Observable.just(200)
11    .subscribeOn(Schedulers.newThread())
12   .subscribe(begin.asAction());
13   begin.set(200);
14   Thread.sleep(1000);
15   assertEquals(beginTime.get(), "200");
16      .....
17  }
18  .....
19 }
```

Lista A.9 Exemplo: *Default Test*

A.10 *DUPLICATE ASSERT*

```
1  @Test
2  public void testXmlSanitizer() {
3     boolean valid = XmlSanitizer.isValid("Fritzbox");
4     assertEquals("Fritzbox is valid", true, valid);
5     System.out.println("Pure ASCII test - passed");
6
7     valid = XmlSanitizer.isValid("Fritz Box");
8     assertEquals("Spaces are valid", true, valid);
9     System.out.println("Spaces test - passed");
10
11    valid = XmlSanitizer.isValid("Frutzbux");
12    assertEquals("Frutzbux is invalid", false, valid);
13    System.out.println("No ASCII test - passed");
14
15    valid = XmlSanitizer.isValid("Fritz!box");
16    assertEquals("Exclamation mark is valid", true, valid);
17    System.out.println("Exclamation mark test - passed");
18
19    valid = XmlSanitizer.isValid("Fritz.box");
20    assertEquals("Exclamation mark is valid", true, valid);
```

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

Lista A.10 Exemplo: *Duplicate Assert*

A.11 *EMPTY TEST*

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

Lista A.11 Exemplo: *Empty Test*

A.12 *EXCEPTION HANDLING*

```
1   @Test
2   public void realCase() {
3       Point p34 = new Point("34", 556506.667, 172513.91, 620.34, true);
4       Point p45 = new Point("45", 556495.16, 172493.912, 623.37, true);
5       Point p47 = new Point("47", 556612.21, 172489.274, 0.0, true);
6       Abriss a = new Abriss(p34, false);
7       a.removeDAO(CalculationsDataSource.getInstance());
8       a.getMeasures().add(new Measure(p45, 0.0, 91.6892, 23.277, 1.63));
9       a.getMeasures().add(new Measure(p47, 281.3521, 100.0471, 108.384, 1.63));
10
11      try {
12          a.compute();
13      } catch (CalculationException e) {
14          Assert.fail(e.getMessage());
15      }
16
17      // test intermediate values with point 45
18      Assert.assertEquals("233.2405",
19          this.df4.format(a.getResults().get(0).getUnknownOrientation()));
20      Assert.assertEquals("233.2435",
21          this.df4.format(a.getResults().get(0).getOrientedDirection()));
22      Assert.assertEquals("-0.1", this.df1.format(
23          a.getResults().get(0).getErrTrans()));
24
25      // test intermediate values with point 47
26      Assert.assertEquals("233.2466",
27          this.df4.format(a.getResults().get(1).getUnknownOrientation()));
28      Assert.assertEquals("114.5956",
29          this.df4.format(a.getResults().get(1).getOrientedDirection()));
30      Assert.assertEquals("0.5", this.df1.format(
31          a.getResults().get(1).getErrTrans()));
32
33      // test final results
34      Assert.assertEquals("233.2435", this.df4.format(a.getMean()));
35      Assert.assertEquals("43", this.df0.format(a.getMSE()));
36      Assert.assertEquals("30", this.df0.format(a.getMeanErrComp()));
```

```
37 }
```

Lista A.12 Exemplo: *Exception Handling*

A.13 *IGNORED TEST*

```
1  @Ignore("disabled for now as this test is too flaky")
2  public void peerPriority() throws Exception {
3   final List addresses = Lists.newArrayList(
4       new InetSocketAddress("localhost", 2000),
5          new InetSocketAddress("localhost", 2001),
6          new InetSocketAddress("localhost", 2002)
7   );
8      peerGroup.addConnectedEventListener(connectedListener);
9      .....
10 }
```

Lista A.13 Exemplo: *Ignored Test*

A.14 *MAGIC NUMBER TEST*

```
1  @Test
2  public void testGetLocalTimeAsCalendar() {
3      Calendar localTime = calc.getLocalTimeAsCalendar(BigDecimal.valueOf(15.5D), Calendar.getInstance());
4      assertEquals(15, localTime.get(Calendar.HOUR_OF_DAY));
5      assertEquals(30, localTime.get(Calendar.MINUTE));
6  }
```

Lista A.14 Exemplo: *Magic Number Test*

A.15 *REDUNDANT PRINT*

```
1
2  @Test
3  public void testTransform10mNEUAndBack() {
4      Leg northEastAndUp10M = new Leg(10, 45, 45);
5      Coord3D result = transformer.transform(Coord3D.ORIGIN, northEastAndUp10M);
6      System.out.println("result = " + result);
7      Leg reverse = new Leg(10, 225, -45);
8      result = transformer.transform(result, reverse);
9      assertEquals(Coord3D.ORIGIN, result);
10 }
```

Lista A.15 Exemplo: *Redundant Print*

A.16 *REDUNDANT ASSERTION*

```
1
2  @Test
3  public void testTrue() {
4      assertEquals(true, true);
5  }
```

Lista A.16 Exemplo: *Redundant Assertion*

A.17 *SLEEPY TEST*

```
1
2  public void testEdictExternSearch() throws Exception {
3     final Intent i = new Intent(getInstrumentation().getContext(), ResultActivity.class);
4     i.setAction(ResultActivity.EDICT_ACTION_INTERCEPT);
5     i.putExtra(ResultActivity.EDICT_INTENTKEY_KANJIS, "XX");
6   tester.startActivity(i);
7   assertTrue(tester.getText(R.id.textSelectedDictionary).contains("Default"));
8   final ListView lv = getActivity().getListView();
9   assertEquals(1, lv.getCount());
10  DictEntry entry = (DictEntry) lv.getItemAtPosition(0);
11  assertEquals("Searching", entry.english);
12  Thread.sleep(500);
13  final Intent i2 = getStartedActivityIntent();
14  final List result = (List) i2.getSerializableExtra(ResultActivity.INTENTKEY_RESULT_LIST);
15  entry = result.get(0);
16  assertEquals("(adj-na,n,adj-no) blank space/vacuum/space/null (NUL)/(P)", entry.english);
17  assertEquals("###", entry.getJapanese());
18  assertEquals("####", entry.reading);
19  assertEquals(1, result.size());
20 }
```

Lista A.17 Exemplo: *Sleepy Test*

A.18 *UNKNOWN TEST*

```
1
2  @Test
3  public void hitGetPOICategoriesApi() throws Exception {
4     POICategories poiCategories = apiClient.getPOICategories(16);
5     for (POICategory category : poiCategories) {
6       System.out.println(category.name() + ": " + category);
7     }
8  }
```

Lista A.18 Exemplo: *Unknown Test*

Apêndice

A

TESTES DE HIPÓTESES

Este anexo apresenta os testes de hipóteses gerados para cada pacote de testes pareado por cada test smells.

A.1 EXISTENTE X RANDOOP

A.1.1 Lazy Test

Lista A.1 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.0 2.0 50.5 263.0 216.8 5318.0
```

Lista A.2 Sumário dos dados do Randoop

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.00 0.00 0.00 14.17 7.00 880.00
```

Lista A.3 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista_comp1$Lazy.Test and lista_comp2$Lazy.Test
## V = 232445, p-value < 2.2e-16
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
## 100.9999 138.0000
## sample estimates:
## (pseudo)median
## 118.5
```

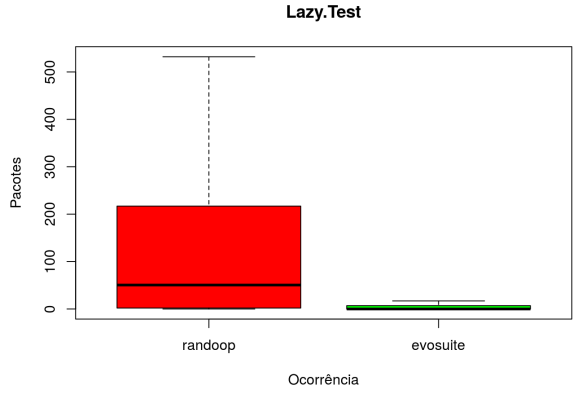


Figura A.1 Boxplot Lazy Test (Testes Existentes x Randoop)

A.1.2 Eager Test

Lista A.4 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.00 0.00 11.00 36.47 39.00 469.00
```

Lista A.5 Sumário dos dados do Randoop

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.000 0.000 1.000 2.603 2.000 94.000
```

Lista A.6 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista__comp1$Eager.Test and lista__comp2$Eager.Test
## V = 200240, p-value < 2.2e-16
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
## 20.50002 26.99999
## sample estimates:
## (pseudo)median
## 23.49998
```

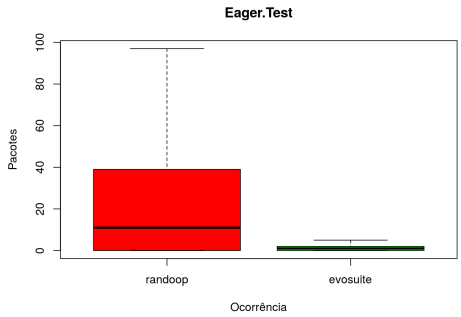


Figura A.2 Boxplot Eager Test (Testes Existentes x Randoop)

A.1.3 Exception Catching Throwing

Lista A.7 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.00 19.00 47.00 69.39 77.00 975.00
```

Lista A.8 Sumário dos dados do Randoop

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.000 0.000 1.000 4.063 4.000 96.000
```

Lista A.9 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista__comp1$Exception.Catching.Throwing and lista__comp2
      ↪ $Exception.Catching.Throwing
## V = 302295, p-value < 2.2e-16
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
## 45.50008 52.49997
## sample estimates:
## (pseudo)median
## 48.99999
```

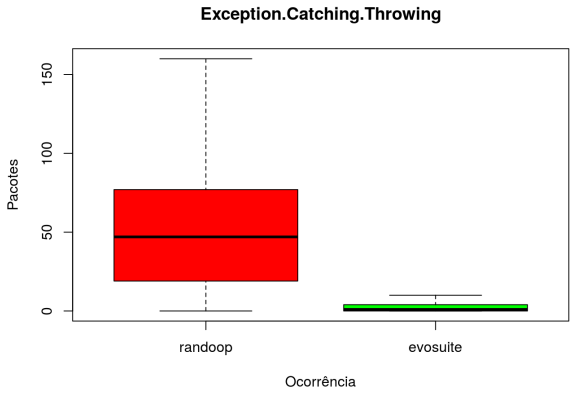


Figura A.3 Boxplot Exception Catching Throwing (Testes Existentes x Randoop)

A.1.4 Magic Number Test

Lista A.10 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.00000 0.00000 0.00000 0.09489 0.00000 34.00000
```

Lista A.11 Sumário dos dados do Randoop

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.00 0.00 0.00 1.12 1.00 41.00
```

Lista A.12 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista_comp1$Magic.Number.Test and lista_comp2$Magic.
    ↪ Number.Test
## V = 716, p-value < 2.2e-16
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
## -2.999936 -2.000024
## sample estimates:
## (pseudo)median
## -2.499955
```

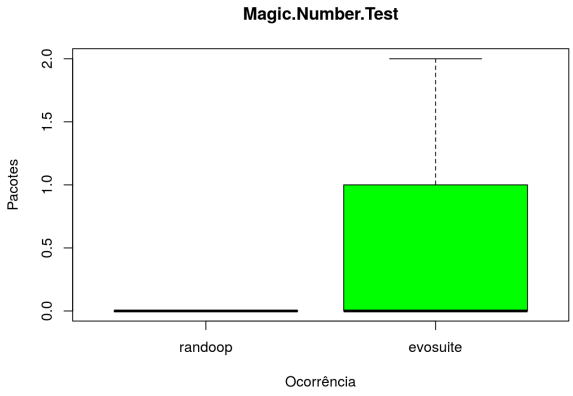


Figura A.4 Magic Number Test (Testes Existentes x Randoop)

A.1.5 Assertion Roulette

Lista A.13 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.00 0.00 27.00 44.31 50.00 472.00
```

Lista A.14 Sumário dos dados do Randoop

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.000 0.000 0.000 1.887 1.000 78.000
```

Lista A.15 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista_comp1$Assertion.Roulette and lista_comp2$Assertion
      ↪ .Roulette
## V = 240442, p-value < 2.2e-16
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
## 32.00003 37.49993
## sample estimates:
## (pseudo)median
## 34.99994
```

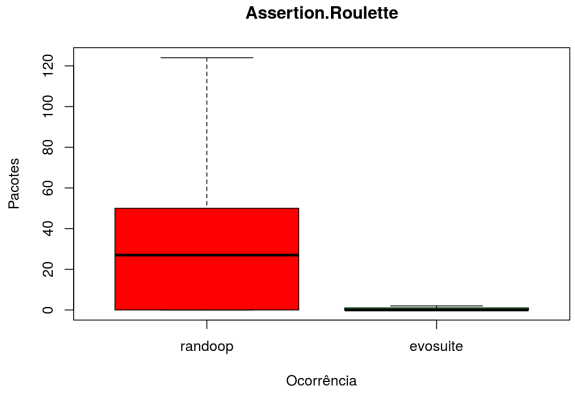


Figura A.5 Assertion Roulette (Testes Existentes x Randoop)

A.1.6 Duplicate Assert

Lista A.16 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0 0 0 0 0 0
```

Lista A.17 Sumário dos dados do Randoop

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.0000 0.0000 0.0000 0.7895 0.0000 33.0000
```

Lista A.18 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista_comp1$Duplicate.Assert and lista_comp2$Duplicate.
      ↪ Assert
## V = 0, p-value < 2.2e-16
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
## -2.999964 -2.000058
## sample estimates:
## (pseudo)median
## -2.499974
```

A.1.7 Conditional Test Logic

Lista A.19 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.00 19.00 47.00 69.39 77.00 975.00
```

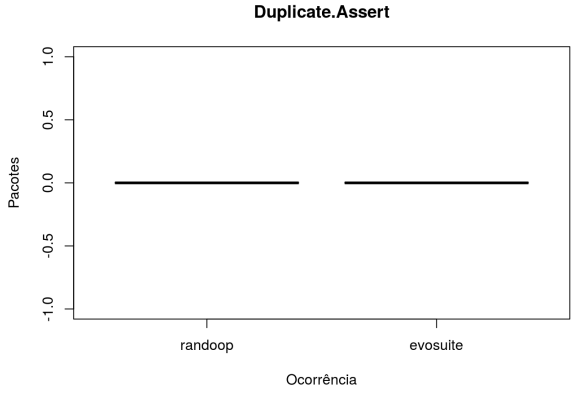


Figura A.6 Duplicate Assert (Testes Existentes x Randoop)

Lista A.20 Sumário dos dados do Randoop

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.0000 0.0000 0.0000 0.4526 0.0000 25.0000
```

Lista A.21 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista__comp1$Conditional.Test.Logic and lista__comp2$
    ↪ Conditional.Test.Logic
## V = 327452, p-value < 2.2e-16
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
## 47.50005 54.50006
## sample estimates:
## (pseudo)median
## 51.00004
```

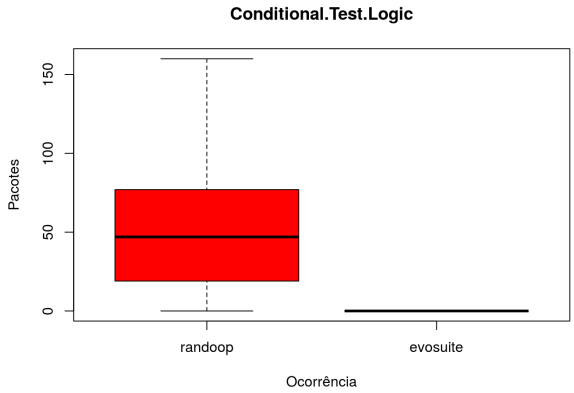


Figura A.7 Conditional Test Logic (Testes Existentes x Randoop)

A.1.8 Unknown Test

Lista A.22 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.000 0.000 0.000 3.158 2.000 153.000
```

Lista A.23 Sumário dos dados do Randoop

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.000 0.000 0.000 2.019 2.000 80.000
```

Lista A.24 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista_comp1$Unknown.Test and lista_comp2$Unknown.
      ↪ Test
## V = 41212, p-value = 0.6937
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
## -0.4999864 0.5000592
## sample estimates:
## (pseudo)median
## 3.136869e-05
```

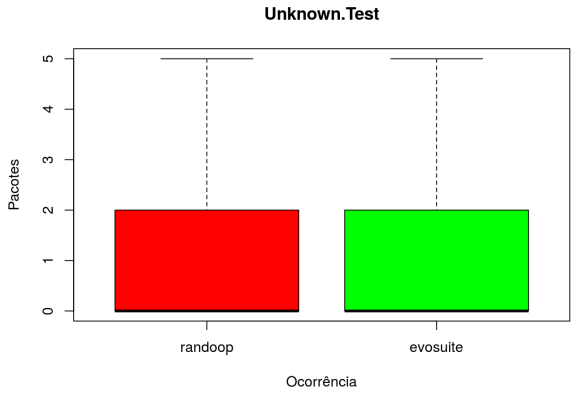


Figura A.8 Unknown Test (Testes Existentes x Randoop)

A.1.9 Sensitive Equality

Lista A.25 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.0000 0.0000 0.0000 0.9246 0.0000 266.0000
```

Lista A.26 Sumário dos dados do Randoop

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.0000 0.0000 0.0000 0.5097 0.0000 75.0000
```

Lista A.27 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista_comp1$Sensitive.Equality and lista_comp2$Sensitive.
      ↪ Equality
## V = 2135.5, p-value = 0.00244
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
## -1.4999831 -0.9999567
## sample estimates:
## (pseudo)median
## -1.000093
```

A.1.10 Resource Optimism

Lista A.28 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0 0 0 0 0 0
```

Lista A.29 Sumário dos dados do Randoop

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.0000 0.0000 0.0000 0.3187 0.0000 27.0000
```

Lista A.30 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista_comp1$Resource.Optimism and lista_comp2$Resource
      ↪ .Optimism
```

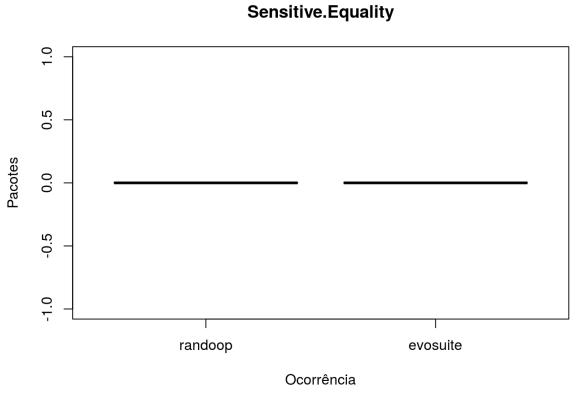


Figura A.9 Sensitive Equality (Testes Existentes x Randoop)

```
## V = 0, p-value = 3.014e-14
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
## -2.500067 -1.500010
```

```
## sample estimates:
## (pseudo)median
## -1.999949
```

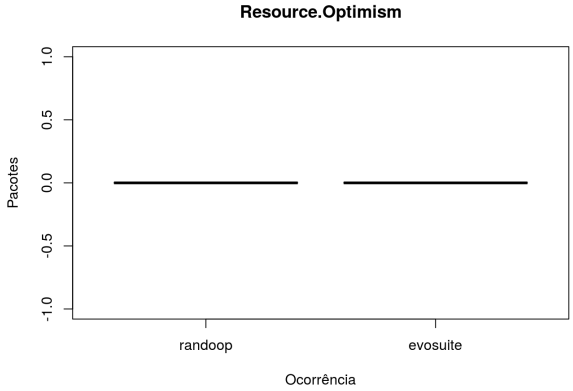


Figura A.10 Resource Optimism (Testes Existentes x Randoop)

A.1.11 Mystery Guest

Lista A.31 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0 0 0 0 0 0
```

Lista A.32 Sumário dos dados do Randoop

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.0000 0.0000 0.0000 0.1411 0.0000 20.0000
```

Lista A.33 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista__comp1$Mystery.Guest and lista__comp2$Mystery.
    ↪ Guest
## V = 0, p-value = 7.961e-08
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
## -3.000051 -1.499973
## sample estimates:
## (pseudo)median
## -2.000003
```

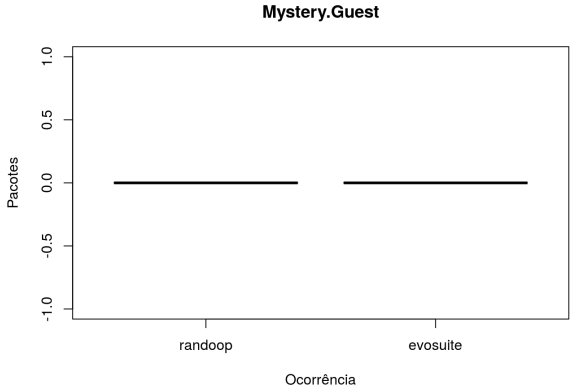


Figura A.11 Mystery Guest (Testes Existentes x Randoop)

A.1.12 Verbose Test

Lista A.34 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.0000 0.0000 0.0000 0.2202 0.0000 28.0000
```

Lista A.35 Sumário dos dados do Randoop

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.00000 0.00000 0.00000 0.00365 0.00000 1.00000
```

Lista A.36 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista_comp1$Verbose.Test and lista_comp2$Verbose.Test
## V = 160.5, p-value = 0.001119
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
## 3.499968 14.500039
## sample estimates:
## (pseudo)median
## 9.499998
```

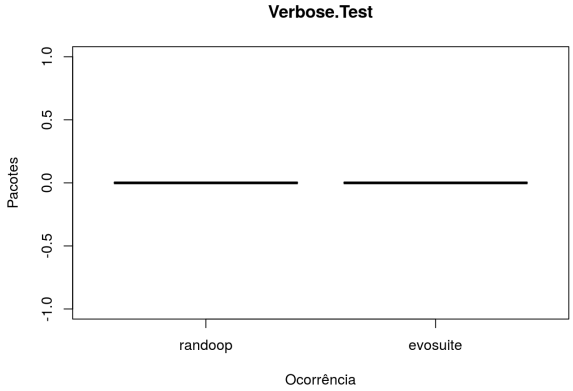


Figura A.12 Verbose Test (Testes Existentes x Randoop)

A.1.13 General Fixture

Lista A.37 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.0000 0.0000 0.0000 0.2202 0.0000 28.0000
```

Lista A.38 Sumário dos dados do Randoop

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.00000 0.00000 0.00000 0.00365 0.00000 1.00000
```

Lista A.39 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista__comp1$Verbose.Test and lista__comp2$Verbose.Test
## V = 160.5, p−value = 0.001119
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
## 3.499968 14.500039
## sample estimates:
## (pseudo)median
## 9.499998
```

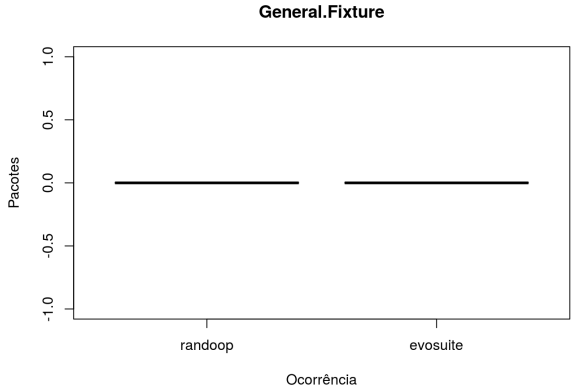


Figura A.13 General Fixture (Testes Existentes x Randoop)

A.1.14 Redundant Assertion

Lista A.40 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0 0 0 0 0 0
```

Lista A.41 Sumário dos dados do Randoop

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.00000 0.00000 0.00000 0.02555 0.00000 5.00000
```

Lista A.42 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista__comp1$Redundant.Assertion and lista__comp2$
    ↪ Redundant.Assertion
## V = 0, p−value = 0.001524
## alternative hypothesis: true location shift is not equal to 0
## 80 percent confidence interval:
## −2.499953 −1.000000
## sample estimates:
## (pseudo)median
## −1.000094
```

A.1.15 Default Test

```
## 0 0 0 0 0 0
```

Lista A.43 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
```

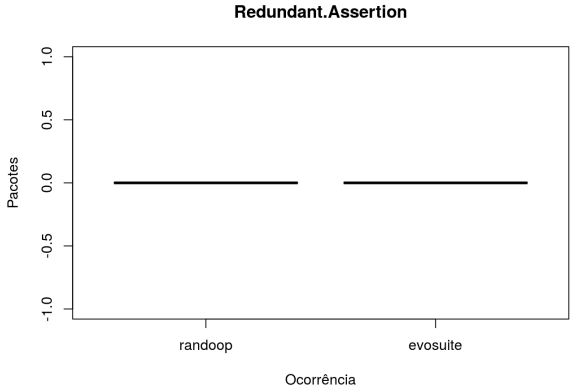


Figura A.14 Redundant Assertion (Testes Existentes x Randoop)

Lista A.44 Sumário dos dados do Randoop

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0 0 0 0 0 0
```

Lista A.45 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista__comp1$Default.Test and lista__comp2$Default.Test
## V = 0, p-value = NA
## alternative hypothesis: true location shift is not equal to 0
## 0 percent confidence interval:
## NaN NaN
## sample estimates:
## [1] NaN
```

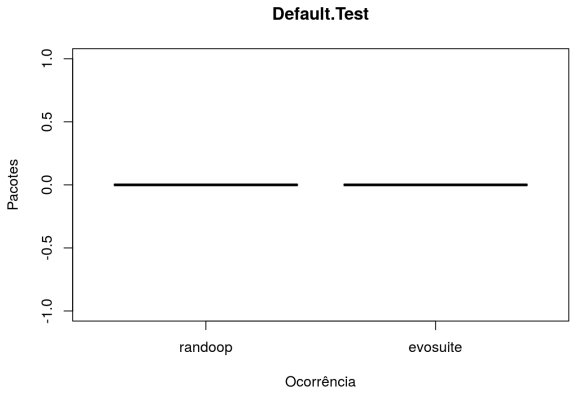


Figura A.15 Default Test (Testes Existentes x Randoop)

A.1.16 Print Statement

Lista A.46 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0 0 0 0 0 0
```

Lista A.47 Sumário dos dados do Randoop

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.00000 0.00000 0.00000 0.08759 0.00000 6.00000
```

Lista A.48 Resultado do Teste de Shapiro-

Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista__comp1$Print.Statement and lista__comp2$Print.
      ↪ Statement
## V = 0, p-value = 6.233e-10
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
## -1.500009 -1.000000
## sample estimates:
## (pseudo)median
## -1.000087
```

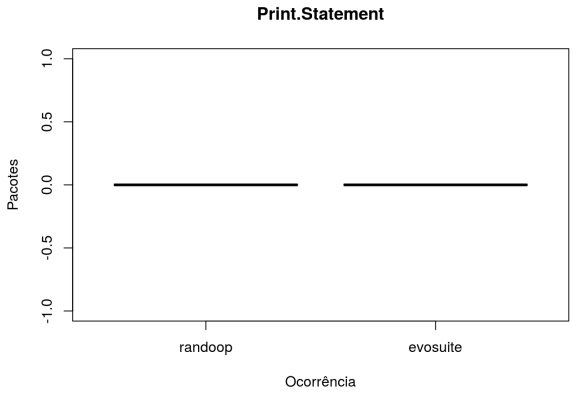


Figura A.16 Print Statement (Testes Existentes x Randoop)

A.1.17 Constructor Initialization

Lista A.49 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0 0 0 0 0 0
```

Lista A.50 Sumário dos dados do Randoop

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.0000 0.0000 0.0000 0.1509 0.0000 2.0000
```

Lista A.51 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista__comp1$Constructor.Initialization and lista__comp2$
      ↪ Constructor.Initialization
## V = 0, p-value < 2.2e-16
## alternative hypothesis: true location shift is not equal to 0
## 0 percent confidence interval:
## -1 -1
## sample estimates:
## (pseudo)median
## -1
```

A.1.18 Sleepy Test

Lista A.52 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0 0 0 0 0 0
```

Lista A.53 Sumário dos dados do Randoop

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.000000 0.000000 0.000000 0.008516 0.000000 4.000000
```

Lista A.54 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista__comp1$Sleepy.Test and lista__comp2$Sleepy.Test
## V = 0, p-value = 0.08897
## alternative hypothesis: true location shift is not equal to 0
## 0 percent confidence interval:
## -1 -1
## sample estimates:
## (pseudo)median
## -1.000097
```

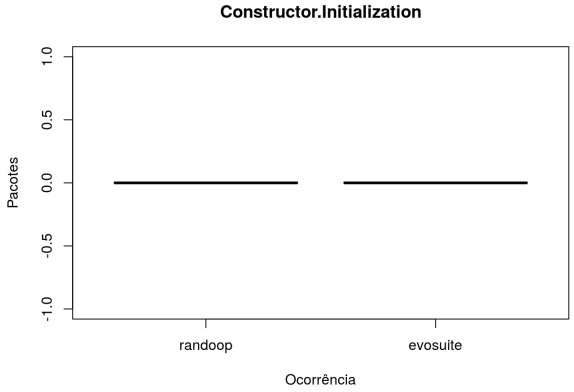


Figura A.17 Constructor Initialization (Testes Existentes x Randoop)

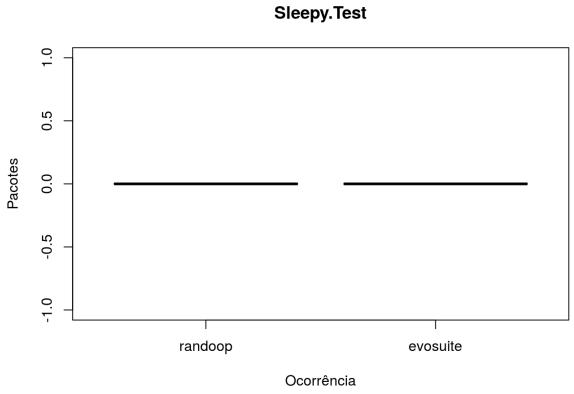


Figura A.18 Sleepy Test (Testes Existentes x Randoop)

A.1.19 Ignored Test

Lista A.55 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0 0 0 0 0 0
```

Lista A.56 Sumário dos dados do Randoop

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.00000 0.00000 0.00000 0.04501 0.00000 6.00000
```

Lista A.57 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista_comp1$IgnoredTest and lista_comp2$IgnoredTest
## V = 0, p-value = 0.0002669
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
## -2.999959 -1.000089
## sample estimates:
## (pseudo)median
## -2.000029
```

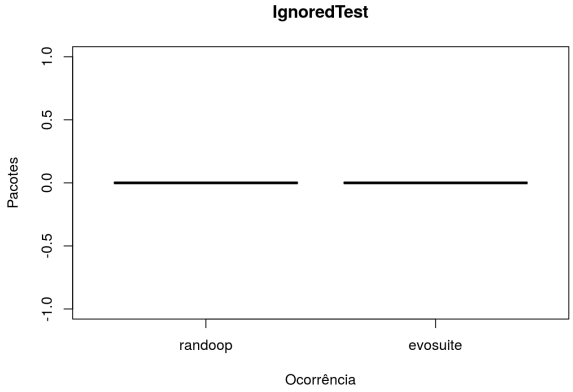


Figura A.19 Ignored Test (Testes Existentes x Randoop)

A.1.20 Dependent Test

Lista A.58 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0 0 0 0 0 0
```

Lista A.59 Sumário dos dados do Randoop

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0 0 0 0 0 0
```

Lista A.60 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista_comp1$Dependent.Test and lista_comp2$Dependent.
    ↪ Test
## V = 0, p-value = NA
## alternative hypothesis: true location shift is not equal to 0
## 0 percent confidence interval:
## NaN NaN
## sample estimates:
## [1] NaN
```

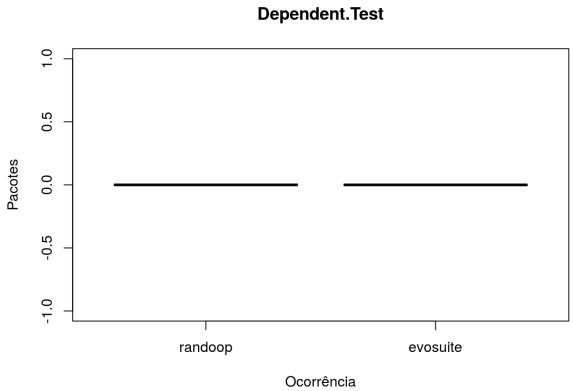


Figura A.20 Dependent Test (Testes Existentes x Randoop)

A.1.21 Empty Test

Lista A.61 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0 0 0 0 0 0
```

Lista A.62 Sumário dos dados do Randoop

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.00000 0.00000 0.00000 0.01825 0.00000 4.00000
```

Lista A.63 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista_comp1$EmptyTest and lista_comp2$EmptyTest
## V = 0, p-value = 0.0009212
## alternative hypothesis: true location shift is not equal to 0
## 0 percent confidence interval:
## -1 -1
## sample estimates:
## (pseudo)median
## -1
```

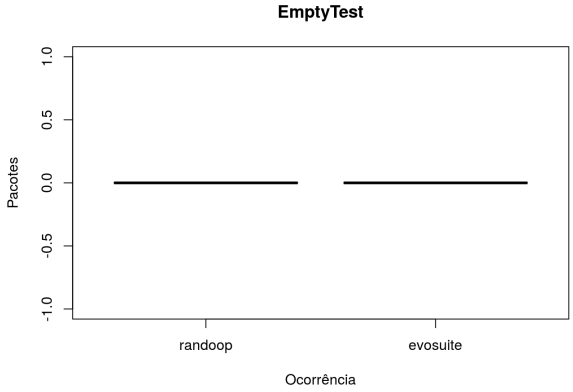


Figura A.21 Empty Test (Testes Existentes x Randoop)

A.2 EXISTENTE X EVOSUITE

A.2.1 Lazy Test

Lista A.64 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.0 4.0 12.0 210.2 35.0 104863.0
```

Lista A.65 Sumário dos dados do Evosuite

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.00 0.00 3.00 18.63 10.00 880.00
```

Lista A.66 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista_comp1$Lazy.Test and lista_comp2$Lazy.Test
## V = 121684, p-value < 2.2e-16
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
## 10.49999 15.50002
## sample estimates:
## (pseudo)median
## 12.99997
```

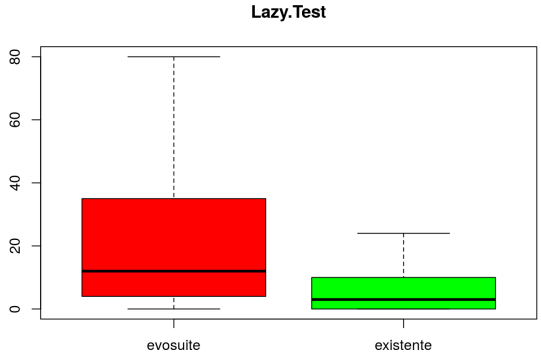


Figura A.22 Lazy Test (Testes Existentes x Evosuite)

A.2.2 Eager Test

Lista A.67 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.000 0.000 1.000 7.766 8.000 273.000
```

Lista A.68 Sumário dos dados do Evosuite

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.000 0.000 1.000 3.203 2.000 94.000
```

Lista A.69 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista_comp1$Eager.Test and lista_comp2$Eager.Test
## V = 65660, p-value = 1.747e-15
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
## 2.500070 4.999957
## sample estimates:
## (pseudo)median
## 3.500015
```

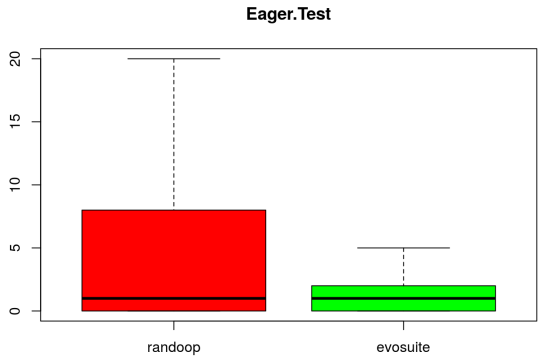


Figura A.23 Eager Test (Testes Existentes x Evosuite)

A.2.3 Exception Catching Throwing

Lista A.70 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.00 5.00 11.00 20.41 26.00 306.00
```

Lista A.71 Sumário dos dados do Evosuite

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.000 0.000 1.000 2.636 2.000 96.000
```

Lista A.72 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista_comp1$Exception.Catching.Throwing and lista_comp2
      ↪ $Exception.Catching.Throwing
## V = 185720, p-value < 2.2e-16
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
## 12.49998 15.00001
## sample estimates:
## (pseudo)median
## 13.50002
```

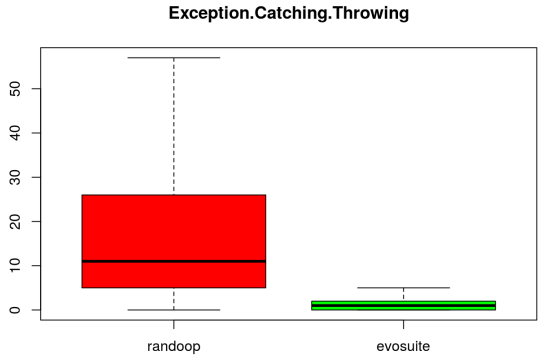


Figura A.24 Exception Catching Throwing (Testes Existentes x Evosuite)

A.2.4 Magic Number Test

Lista A.73 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.000 0.000 1.000 4.749 4.000 259.000
```

Lista A.74 Sumário dos dados do Evosuite

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.000 0.000 0.000 1.391 1.000 41.000
```

Lista A.75 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista_comp1$Magic.Number.Test and lista_comp2$Magic.
      ↪ Number.Test
## V = 64722, p-value < 2.2e-16
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
## 2.000078 3.000044
## sample estimates:
## (pseudo)median
## 2.500038
```

A.2.5 Assertion Roulette

Lista A.76 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
```

```
## 0.000 0.000 0.000 3.396 1.000 220.000
```

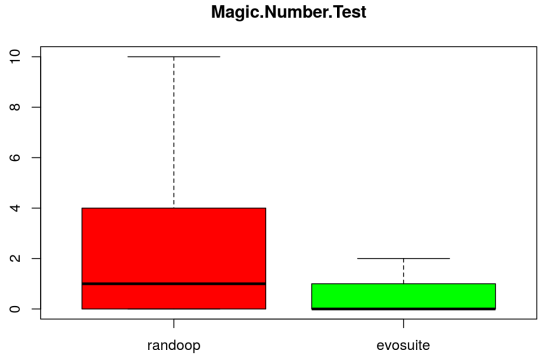


Figura A.25 Magic Number Test (Testes Existentes x Evosuite)

Lista A.77 Sumário dos dados do Evosuite

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.000 0.000 0.000 2.453 2.000 78.000
```

Lista A.78 Resultado do Teste de Shapiro-Wilk

```
##
```

```
## Wilcoxon signed rank test with continuity correction
##
## data: lista__comp1$Assertion.Roulette and lista__comp2$Assertion
      ↪ .Roulette
## V = 32049, p−value = 0.1062
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
## −9.999798e−01 3.483172e−05
## sample estimates:
## (pseudo)median
## −0.4999894
```

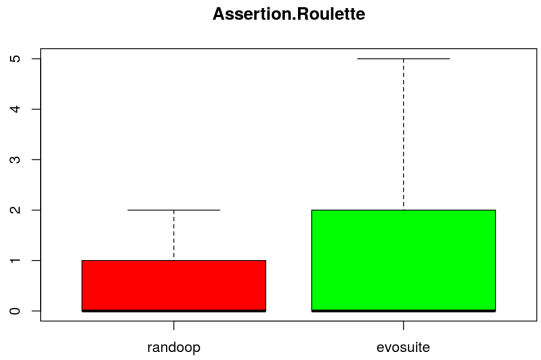


Figura A.26 Assertion Roulette (Testes Existentes x Evosuite)

A.2.6 Duplicate Assert

Lista A.79 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.000 0.000 0.000 1.129 0.000 168.000
```

Lista A.80 Sumário dos dados do Evosuite

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.0000 0.0000 0.0000 0.9291 1.0000 33.0000
```

Lista A.81 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista__comp1$Duplicate.Assert and lista__comp2$Duplicate.
      ↪ Assert
## V = 3008.5, p-value = 1.727e-14
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
## -2.499979 -1.500065
## sample estimates:
## (pseudo)median
## -1.999945
```

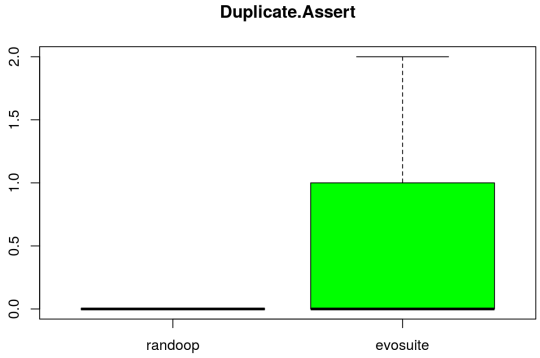


Figura A.27 Duplicate Assert (Testes Existentes x Evosuite)

A.2.7 Conditional Test Logic

Lista A.82 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0 0 0 0 0 0
```

Lista A.83 Sumário dos dados do Evosuite

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.0000 0.0000 0.0000 0.6194 0.0000 25.0000
```

Lista A.84 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista__comp1$Conditional.Test.Logic and lista__comp2$
      ↪ Conditional.Test.Logic
## V = 0, p-value < 2.2e-16
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
## -2.000028 -1.500030
## sample estimates:
## (pseudo)median
## -1.999961
```

A.2.8 Unknown Test

Lista A.85 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.00000 0.00000 0.00000 0.06471 0.00000 1.00000
```

Lista A.86 Sumário dos dados do Evosuite

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.0000 0.0000 0.0000 0.4977 0.0000 18.0000
```

Lista A.87 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista__comp1$Unknown.Test and lista__comp2$Unknown.
      ↪ Test
```

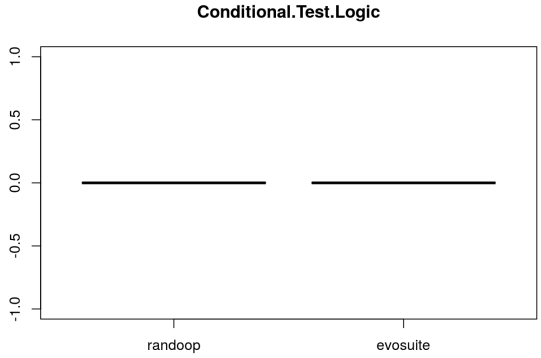


Figura A.28 Conditional Test Logic (Testes Existentes x Evosuite)

```
## V = 1881, p-value = 3.364e-15
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
## -1.500015 -1.000047
```

```
## sample estimates:
## (pseudo)median
## -1.499994
```

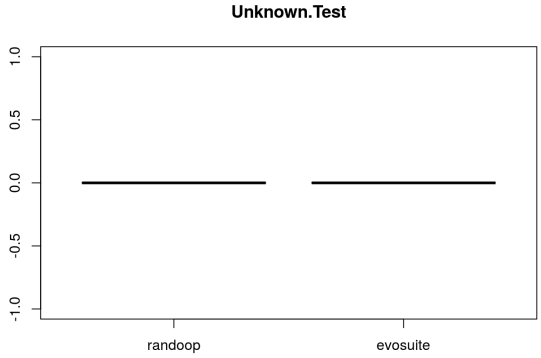


Figura A.29 Unknown Test (Testes Existentes x Evosuite)

A.2.9 Sensitive Equality

Lista A.88 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.0000 0.0000 0.0000 0.3128 0.0000 32.0000
```

Lista A.89 Sumário dos dados do Evosuite

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.0000 0.0000 0.0000 0.6302 0.0000 75.0000
```

Lista A.90 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista__comp1$Sensitive.Equality and lista__comp2$Sensitive.
      ↪ Equality
## V = 1822, p-value = 0.002752
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
## -1.4999671 -0.4999796
## sample estimates:
## (pseudo)median
## -0.9999759
```

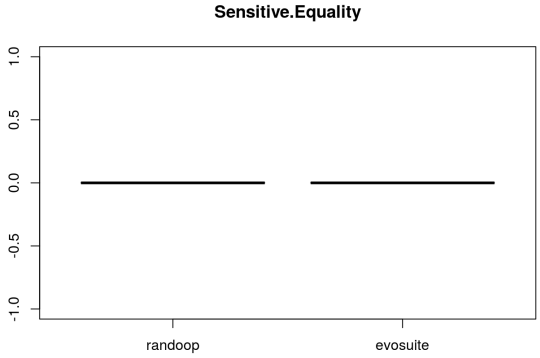


Figura A.30 Sensitive Equality (Testes Existentes x Evosuite)

A.2.10 Resource Optimism

Lista A.91 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.00000 0.00000 0.00000 0.09707 0.00000 13.00000
```

Lista A.92 Sumário dos dados do Evosuite

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.0000 0.0000 0.0000 0.1525 0.0000 21.0000
```

Lista A.93 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista_comp1$Resource.Optimism and lista_comp2$Resource
      ↪ .Optimism
## V = 329, p-value = 0.276
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
## -1.9999899 0.4999547
## sample estimates:
## (pseudo)median
## -0.5000184
```

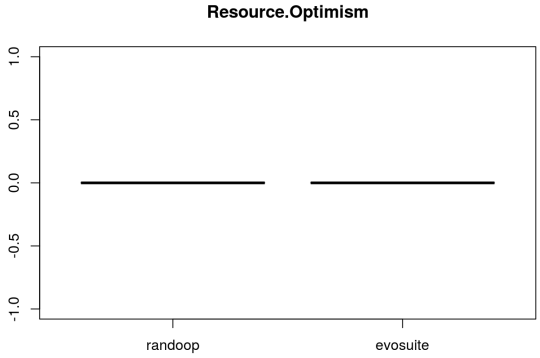


Figura A.31 Resource Optimism (Testes Existentes x Evosuite)

A.2.11 Mystery Guest

Lista A.94 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.0000 0.0000 0.0000 0.1094 0.0000 15.0000
```

Lista A.95 Sumário dos dados do Evosuite

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.00000 0.00000 0.00000 0.06471 0.00000 8.00000
```

Lista A.96 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista__comp1$Mystery.Guest and lista__comp2$Mystery.
    ↪ Guest
## V = 327.5, p-value = 0.4011
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
## -0.5000058 1.9999632
## sample estimates:
## (pseudo)median
## 0.4999805
```

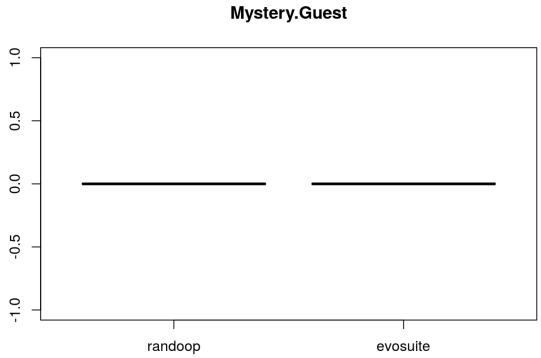


Figura A.32 Mystery Guest (Testes Existentes x Evosuite)

A.2.12 Verbose Test

Lista A.97 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.0000 0.0000 0.0000 0.3035 0.0000 155.0000
```

Lista A.98 Sumário dos dados do Evosuite

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.000000 0.000000 0.000000 0.004622 0.000000 1.000000
```

Lista A.99 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista__comp1$Verbose.Test and lista__comp2$Verbose.Test
## V = 20.5, p-value = 0.3017
## alternative hypothesis: true location shift is not equal to 0
## 80 percent confidence interval:
## -0.9999496 76.9999626
## sample estimates:
## (pseudo)median
## 9.632971
```

A.2.13 General Fixture

```
## 0 0 0 0 0 0
```

Lista A.100 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
```

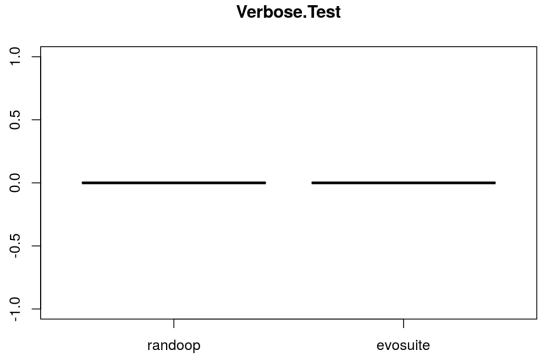


Figura A.33 Verbose Test (Testes Existentes x Evosuite)

Lista A.101 Sumário dos dados do Evosuite

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.00000 0.00000 0.00000 0.04314 0.00000 7.00000
```

Lista A.102 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista_comp1$Redundant.Assertion and lista_comp2$
      ↪ General.Fixture
## V = 0, p-value = 2.387e-09
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
## -10.500048 -5.000009
## sample estimates:
## (pseudo)median
## -7.499931
```

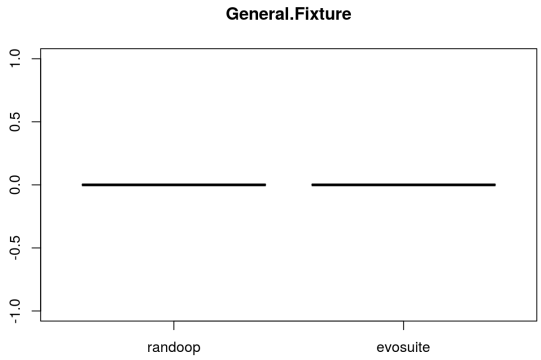


Figura A.34 General Fixture (Testes Existentes x Evosuite)

A.2.14 Redundant Assertion

Lista A.103 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0 0 0 0 0 0
```

Lista A.104 Sumário dos dados do Evosuite

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.00000 0.00000 0.00000 0.04314 0.00000 7.00000
```

Lista A.105 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista__comp1$Redundant.Assertion and lista__comp2$
    ↪ Redundant.Assertion
## V = 0, p-value = 0.001088
## alternative hypothesis: true location shift is not equal to 0
## 80 percent confidence interval:
## -2.999972 -1.000000
## sample estimates:
## (pseudo)median
## -1.999946
```

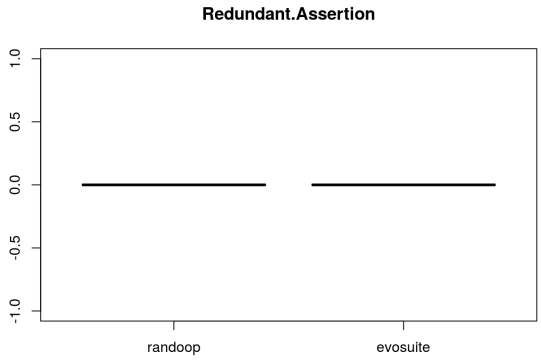


Figura A.35 Redundant Assertion (Testes Existentes x Evosuite)

A.2.15 Default Test

Lista A.106 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0 0 0 0 0 0
```

Lista A.107 Sumário dos dados do Evosuite

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0 0 0 0 0 0
```

Lista A.108 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista__comp1$Default.Test and lista__comp2$Default.Test
## V = 0, p-value = NA
## alternative hypothesis: true location shift is not equal to 0
## 0 percent confidence interval:
## NaN NaN
## sample estimates:
## [1] NaN
```

A.2.16 Print Statement

Lista A.109 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0 0 0 0 0 0
```

Lista A.110 Sumário dos dados do Evosuite

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.0000 0.0000 0.0000 0.1202 0.0000 6.0000
```

Lista A.111 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista__comp1$Print.Statement and lista__comp2$Print.
    ↪ Statement
## V = 0, p-value = 6.174e-10
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
## -1.500029 -1.000065
## sample estimates:
```

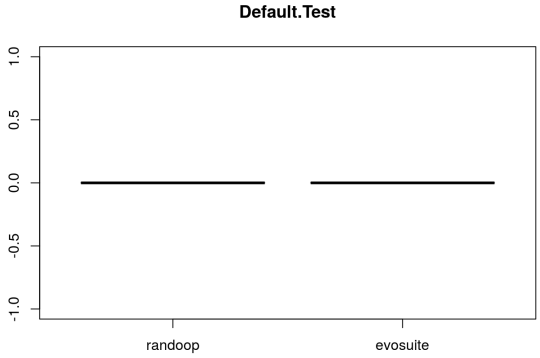


Figura A.36 Default Test (Testes Existentes x Evosuite)

```
## (pseudo)median
## -1.499954
```

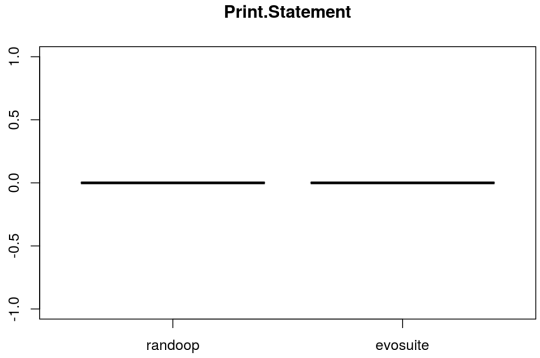


Figura A.37 Print Statement (Testes Existentes x Evosuite)

A.2.17 Constructor Initialization

Lista A.112 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0 0 0 0 0 0
```

Lista A.113 Sumário dos dados do Evosuite

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.0000 0.0000 0.0000 0.1941 0.0000 1.0000
```

Lista A.114 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista_comp1$Constructor.Initialization and lista_comp2$
    ↪ Constructor.Initialization
## V = 0, p-value < 2.2e-16
## alternative hypothesis: true location shift is not equal to 0
## 0 percent confidence interval:
## NaN NaN
## sample estimates:
## midrange
## -1
```

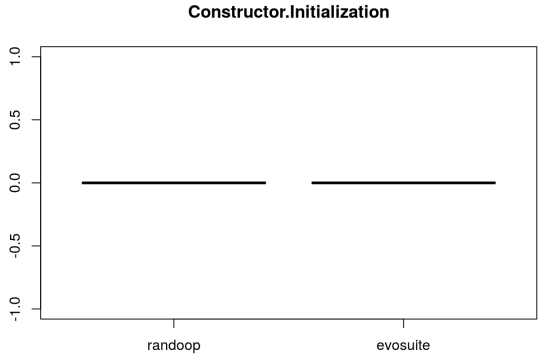


Figura A.38 Constructor Initialization (Testes Existentes x Evosuite)

A.2.18 Sleepy Test

Lista A.115 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0 0 0 0 0 0
```

Lista A.116 Sumário dos dados do Evosuite

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.00000 0.00000 0.00000 0.01079 0.00000 4.00000
```

Lista A.117 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista__comp1$Sleepy.Test and lista__comp2$Sleepy.Test
## V = 0, p-value = 0.08897
## alternative hypothesis: true location shift is not equal to 0
## 0 percent confidence interval:
## -1 -1
## sample estimates:
## (pseudo)median
## -1.000097
```

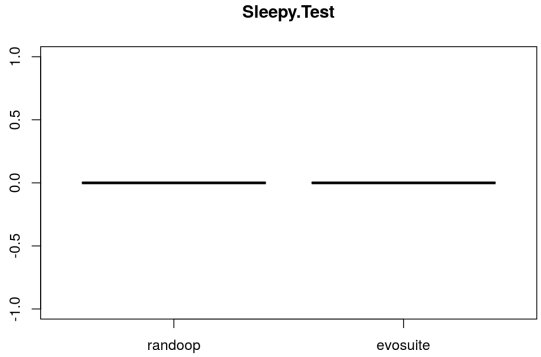


Figura A.39 Sleepy Test (Testes Existentes x Evosuite)

A.2.19 Ignored Test

Lista A.118 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0 0 0 0 0 0
```

Lista A.119 Sumário dos dados do Evosuite

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.0000 0.0000 0.0000 0.0416 0.0000 4.0000
```

Lista A.120 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista__comp1$IgnoredTest and lista__comp2$IgnoredTest
## V = 0, p-value = 0.0005735
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
## -2.499997 -1.000000
## sample estimates:
## (pseudo)median
## -1.723922
```

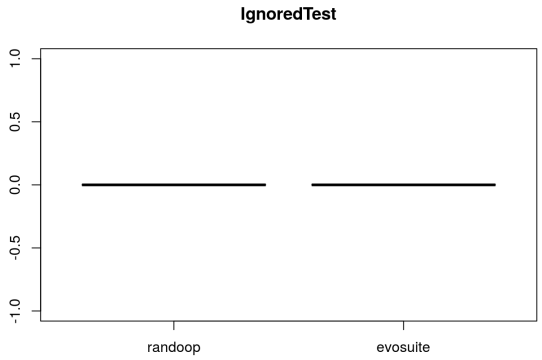


Figura A.40 Ignored Test (Testes Existentes x Evosuite)

A.2.20 Dependent Test

Lista A.121 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0 0 0 0 0 0
```

Lista A.122 Sumário dos dados do Evosuite

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0 0 0 0 0 0
```

Lista A.123 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista__comp1$Dependent.Test and lista__comp2$Dependent.
      ↪ Test
## V = 0, p-value = NA
## alternative hypothesis: true location shift is not equal to 0
## 0 percent confidence interval:
## NaN NaN
## sample estimates:
## [1] NaN
```

A.2.21 Empty Test

Lista A.124 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.00000 0.00000 0.00000 0.06471 0.00000 1.00000
```

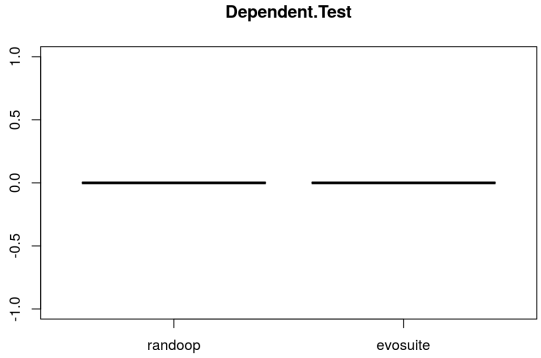


Figura A.41 Dependent Test (Testes Existentes x Evosuite)

Lista A.125 Sumário dos dados do Evosuite

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.00000 0.00000 0.00000 0.02157 0.00000 4.00000
```

Lista A.126 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista_comp1$EmptyTest and lista_comp2$EmptyTest
## V = 1066, p-value = 9.416e-05
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
## 1.636336e-05 1.000000e+00
## sample estimates:
## (pseudo)median
## 0.9999244
```

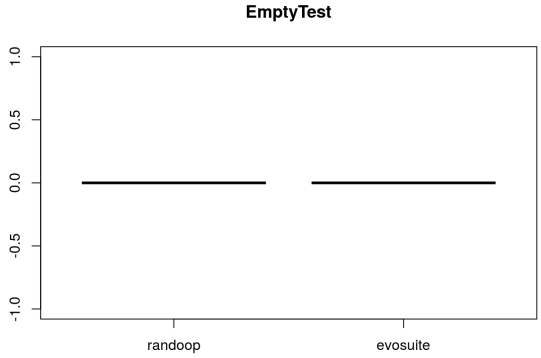


Figura A.42 Empty Test (Testes Existentes x Evosuite)

A.3 RANDOOP X EVOSUITE

A.3.1 Lazy Test

Lista A.127 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.0 0.0 27.0 189.9 171.0 5318.0
```

Lista A.128 Sumário dos dados do Evosuite

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.00 2.00 7.00 90.26 21.00 104863.00
```

Lista A.129 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista_comp1$Lazy.Test and lista_comp2$Lazy.Test
## V = 868053, p-value < 2.2e-16
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
## 72.00002 93.99995
## sample estimates:
## (pseudo)median
## 82.49993
```

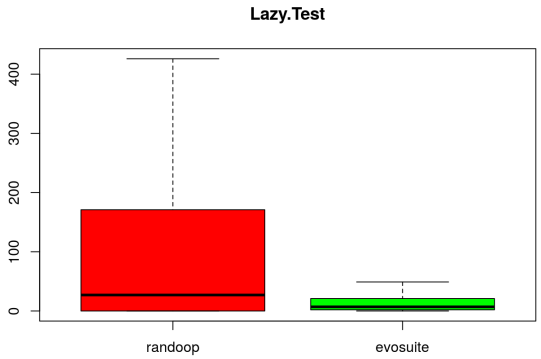


Figura A.43 Lazy Test (Randoop x Evosuite)

A.3.2 Eager Test

Lista A.130 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.00 0.00 0.00 27.55 34.00 469.00
```

Lista A.131 Sumário dos dados do Evosuite

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.000 0.000 0.000 5.025 4.000 273.000
```

Lista A.132 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista_comp1$Eager.Test and lista_comp2$Eager.Test
## V = 532363, p-value < 2.2e-16
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
## 20.50004 26.00008
## sample estimates:
## (pseudo)median
## 23.49997
```

A.3.3 Exception Catching Throwing

Lista A.133 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.00 13.00 46.00 64.46 84.50 975.00
```

Lista A.134 Sumário dos dados do Evosuite

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.00 3.00 8.00 14.42 17.00 306.00
```

Lista A.135 Resultado do Teste de Shapiro-Wilk

```
##
```

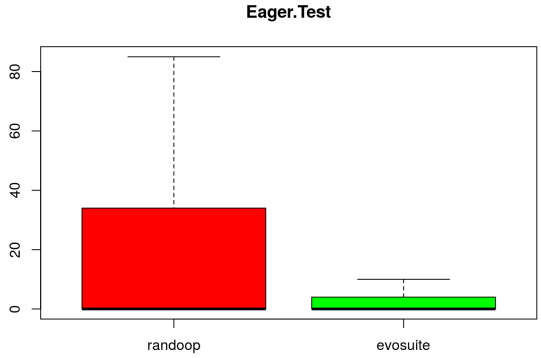


Figura A.44 Eager Test (Randoop x Evosuite)

```
## Wilcoxon signed rank test with continuity correction
##
## data: lista__comp1$Exception.Catching.Throwing and lista__comp2
      ↪ $Exception.Catching.Throwing
## V = 1338600, p-value < 2.2e-16
## alternative hypothesis: true location shift is not equal to 0
```

```
## 95 percent confidence interval:
## 36.99996 42.49996
## sample estimates:
## (pseudo)median
## 39.50001
```

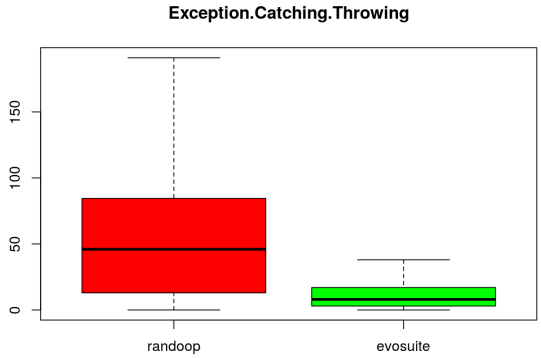


Figura A.45 Exception Catching Throwing (Randoop x Evosuite)

A.3.4 Magic Number Test

Lista A.136 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.0000 0.0000 0.0000 0.1677 0.0000 117.0000
```

Lista A.137 Sumário dos dados do Evosuite

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.000 0.000 0.000 3.133 3.000 259.000
```

Lista A.138 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista__comp1$Magic.Number.Test and lista__comp2$Magic.
      ↪ Number.Test
## V = 4901.5, p-value < 2.2e-16
```

```
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
## -4.000022 -3.499927
## sample estimates:
## (pseudo)median
## -3.999974
```

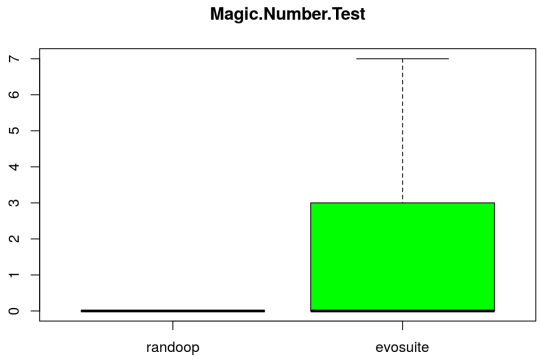


Figura A.46 Magic Number Test (Randoop x Evosuite)

A.3.5 Assertion Roulette

Lista A.139 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.0 0.0 22.0 37.9 49.5 749.0
```

Lista A.140 Sumário dos dados do Evosuite

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.000 0.000 0.000 2.245 0.000 236.000
```

Lista A.141 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista_comp1$Assertion.Roulette and lista_comp2$Assertion
      ↪ .Roulette
## V = 843600, p-value < 2.2e-16
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
## 34.50002 38.99998
## sample estimates:
## (pseudo)median
## 36.99999
```

A.3.6 Duplicate Assert

Lista A.142 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0 0 0 0 0 0
```

Lista A.143 Sumário dos dados do Evosuite

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.000 0.000 0.000 0.537 0.000 168.000
```

Lista A.144 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista_comp1$Duplicate.Assert and lista_comp2$Duplicate.
      ↪ Assert
## V = 0, p-value = 7.766e-08
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
## -27.999921 -6.500027
## sample estimates:
## (pseudo)median
## -18.00006
```

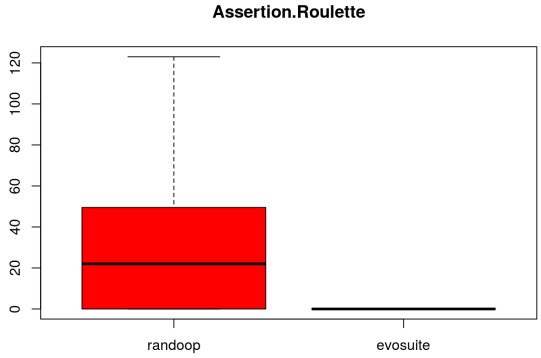


Figura A.47 Assertion Roulette (Randoop x Evosuite)

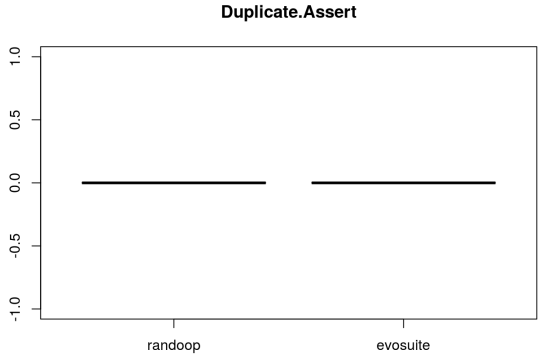


Figura A.48 Duplicate Assert (Randoop x Evosuite)

A.3.7 Conditional Test Logic

Lista A.145 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.00 13.00 46.00 64.46 84.50 975.00
```

Lista A.146 Sumário dos dados do Evosuite

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0 0 0 0 0 0
```

Lista A.147 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista__comp1$Conditional.Test.Logic and lista__comp2$
      ↪ Conditional.Test.Logic
## V = 1562028, p-value < 2.2e-16
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
## 49.00002 54.49996
## sample estimates:
## (pseudo)median
## 51.50005
```

A.3.8 Unknown Test

Lista A.148 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.000 0.000 0.000 2.717 2.000 153.000
```

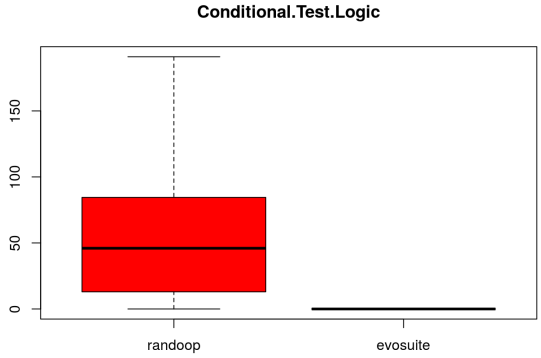


Figura A.49 Conditional Test Logic (Randoop x Evosuite)

Lista A.149 Sumário dos dados do Evosuite

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.00000 0.00000 0.00000 0.04404 0.00000 1.00000
```

Lista A.150 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista__comp1$Unknown.Test and lista__comp2$Unknown.
    ↪ Test
## V = 258678, p-value < 2.2e-16
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
## 3.000041 3.999951
## sample estimates:
## (pseudo)median
## 3.50005
```

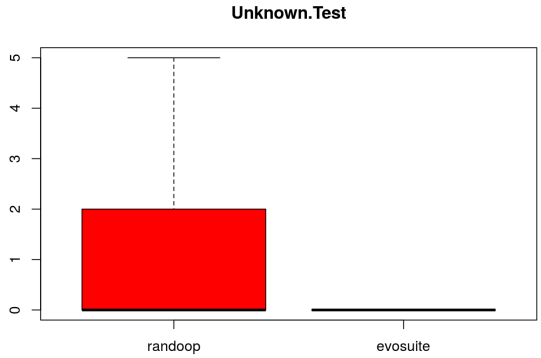


Figura A.50 Unknown Test (Randoop x Evosuite)

A.3.9 Sensitive Equality

Lista A.151 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.0000 0.0000 0.0000 0.8978 0.0000 266.0000
```

Lista A.152 Sumário dos dados do Evosuite

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.0000 0.0000 0.0000 0.1734 0.0000 32.0000
```

Lista A.153 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista_comp1$Sensitive.Equality and lista_comp2$Sensitive.
    ↪ Equality
## V = 8984, p-value = 0.002559
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
## 0.0000238706 2.0000058386
## sample estimates:
## (pseudo)median
## 1.000026
```

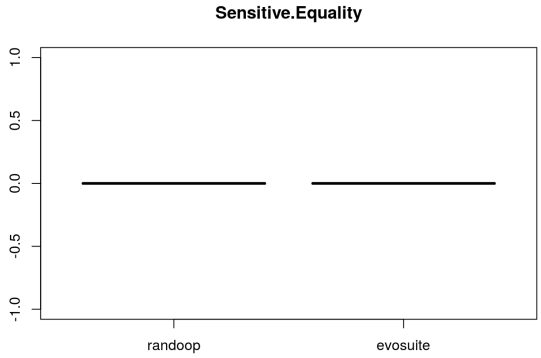


Figura A.51 Sensitive Equality (Randoop x Evosuite)

A.3.10 Resource Optimism

Lista A.154 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0 0 0 0 0 0
```

Lista A.155 Sumário dos dados do Evosuite

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.00000 0.00000 0.00000 0.05534 0.00000 11.00000
```

Lista A.156 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista_comp1$Resource.Optimism and lista_comp2$Resource
    ↪ .Optimism
## V = 0, p-value = 2.76e-07
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
## -3.000028 -1.500025
## sample estimates:
## (pseudo)median
## -2.000024
```

A.3.11 Mystery Guest

Lista A.157 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0 0 0 0 0 0
```

Lista A.158 Sumário dos dados do Evosuite

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.00000 0.00000 0.00000 0.06889 0.00000 11.00000
```

Lista A.159 Resultado do Teste de Shapiro-Wilk

```
##
```

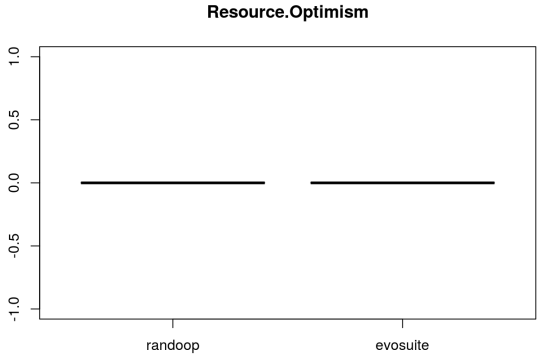


Figura A.52 Resource Optimism (Randoop x Evosuite)

```
## Wilcoxon signed rank test with continuity correction
##
## data: lista__comp1$Mystery.Guest and lista__comp2$Mystery.
      ↪ Guest
## V = 0, p-value = 6.202e-08
## alternative hypothesis: true location shift is not equal to 0
```

```
## 95 percent confidence interval:
## -3.999975 -1.500018
## sample estimates:
## (pseudo)median
## -2.500013
```

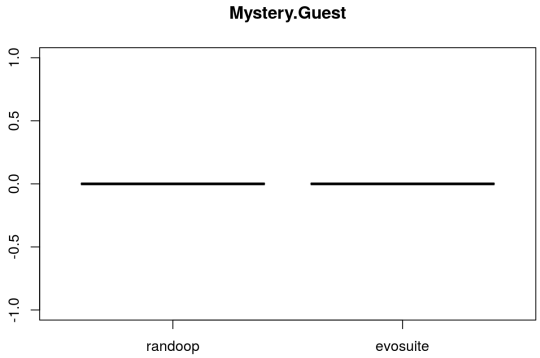


Figura A.53 Mystery Guest (Randoop x Evosuite)

A.3.12 Verbose Test

Lista A.160 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.0000 0.0000 0.0000 0.1203 0.0000 28.0000
```

Lista A.161 Sumário dos dados do Evosuite

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.0000 0.0000 0.0000 0.1237 0.0000 155.0000
```

Lista A.162 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista__comp1$Verbose.Test and lista__comp2$Verbose.Test
## V = 425, p-value = 0.009745
## alternative hypothesis: true location shift is not equal to 0
```

```
## 95 percent confidence interval:
## 0.9999508 7.5000539
## sample estimates:
```

```
## (pseudo)median
## 2.500006
```

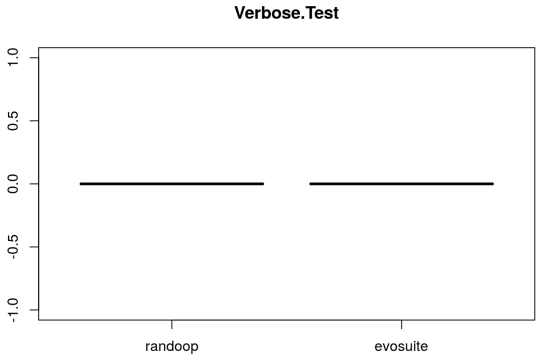


Figura A.54 Verbose Test (Randoop x Evosuite)

A.3.13 General Fixture

Lista A.163 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0 0 0 0 0 0
```

Lista A.164 Sumário dos dados do Evosuite

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0 0 0 0 0 0
```

Lista A.165 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista__comp1$General.Fixture and lista__comp2$General.
      ↪ Fixture
## V = 0, p-value = NA
## alternative hypothesis: true location shift is not equal to 0
## 0 percent confidence interval:
## NaN NaN
## sample estimates:
## [1] NaN
```

A.3.14 Redundant Assertion

Lista A.166 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0 0 0 0 0 0
```

Lista A.167 Sumário dos dados do Evosuite

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0 0 0 0 0 0
```

Lista A.168 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista__comp1$Redundant.Assertion and lista__comp2$
      ↪ Redundant.Assertion
## V = 0, p-value = NA
## alternative hypothesis: true location shift is not equal to 0
## 0 percent confidence interval:
## NaN NaN
## sample estimates:
## [1] NaN
```

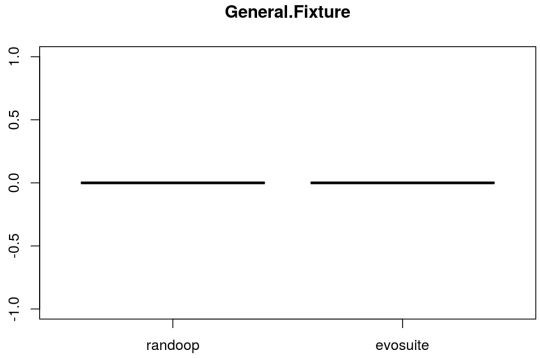


Figura A.55 General Fixture (Randoop x Evosuite)

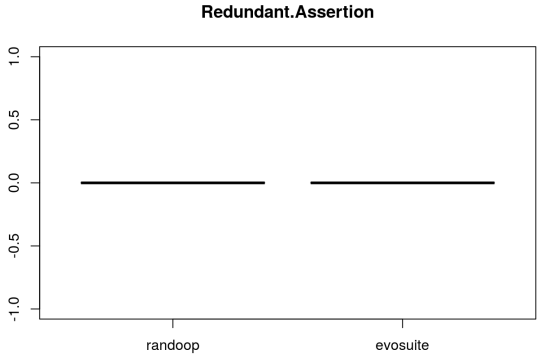


Figura A.56 Redundant Assertion (Randoop x Evosuite)

A.3.15 Default Test

Lista A.169 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0 0 0 0 0 0
```

Lista A.170 Sumário dos dados do Evosuite

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0 0 0 0 0 0
```

Lista A.171 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista_comp1$Default.Test and lista_comp2$Default.Test
## V = 0, p-value = NA
## alternative hypothesis: true location shift is not equal to 0
## 0 percent confidence interval:
## NaN NaN
## sample estimates:
## [1] NaN
```

A.3.16 Print Statement

Lista A.172 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0 0 0 0 0 0
```

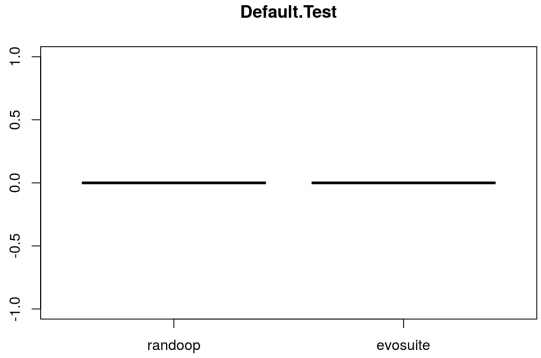


Figura A.57 Default Test (Randoop x Evosuite)

Lista A.173 Sumário dos dados do Evosuite

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0 0 0 0 0 0
```

Lista A.174 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista_comp1$Print.Statement and lista_comp2$Print.
      ↪ Statement
## V = 0, p-value = NA
## alternative hypothesis: true location shift is not equal to 0
## 0 percent confidence interval:
## NaN NaN
## sample estimates:
## [1] NaN
```

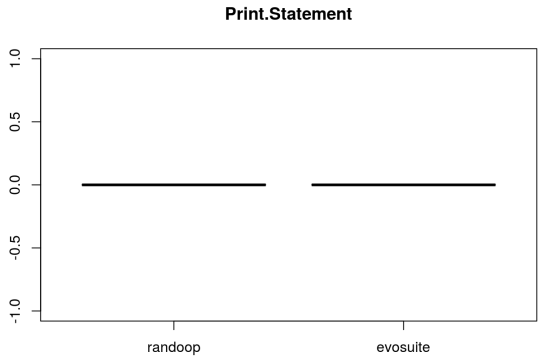


Figura A.58 Print Statement (Randoop x Evosuite)

A.3.17 Constructor Initialization

Lista A.175 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0 0 0 0 0 0
```

Lista A.176 Sumário dos dados do Evosuite

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0 0 0 0 0 0
```

Lista A.177 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista__comp1$Constructor.Initialization and lista__comp2$
      ↪ Constructor.Initialization
## V = 0, p-value = NA
## alternative hypothesis: true location shift is not equal to 0
## 0 percent confidence interval:
## NaN NaN
## sample estimates:
## [1] NaN
```

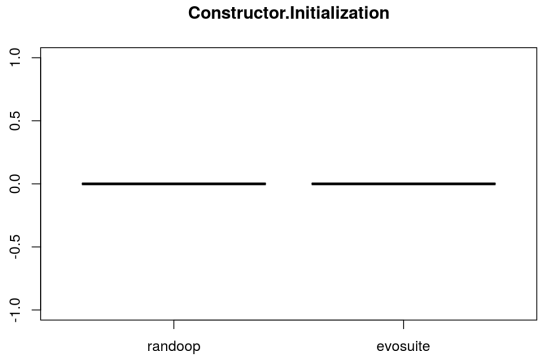


Figura A.59 Constructor Initialization (Randoop x Evosuite)

A.3.18 Sleepy Test

Lista A.178 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0 0 0 0 0 0
```

Lista A.179 Sumário dos dados do Evosuite

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0 0 0 0 0 0
```

Lista A.180 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista__comp1$Sleepy.Test and lista__comp2$Sleepy.Test
## V = 0, p-value = NA
## alternative hypothesis: true location shift is not equal to 0
## 0 percent confidence interval:
## NaN NaN
## sample estimates:
## [1] NaN
```

A.3.19 Ignored Test

Lista A.181 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0 0 0 0 0 0
```

Lista A.182 Sumário dos dados do Evosuite

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0 0 0 0 0 0
```

Lista A.183 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista__comp1$IgnoredTest and lista__comp2$IgnoredTest
## V = 0, p-value = NA
## alternative hypothesis: true location shift is not equal to 0
## 0 percent confidence interval:
## NaN NaN
## sample estimates:
## [1] NaN
```

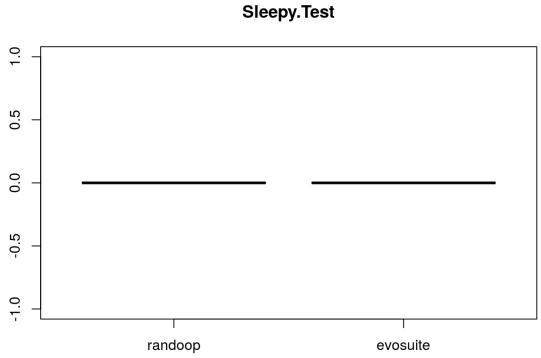


Figura A.60 Sleepy Test (Randoop x Evosuite)

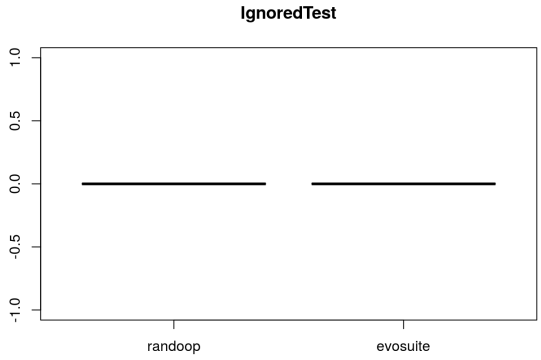


Figura A.61 Ignored Test (Randoop x Evosuite)

A.3.20 Dependent Test

Lista A.184 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0 0 0 0 0 0
```

Lista A.185 Sumário dos dados do Evosuite

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0 0 0 0 0 0
```

Lista A.186 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista_comp1$Dependent.Test and lista_comp2$Dependent.
    ↪ Test
## V = 0, p-value = NA
## alternative hypothesis: true location shift is not equal to 0
## 0 percent confidence interval:
## NaN NaN
## sample estimates:
## [1] NaN
```

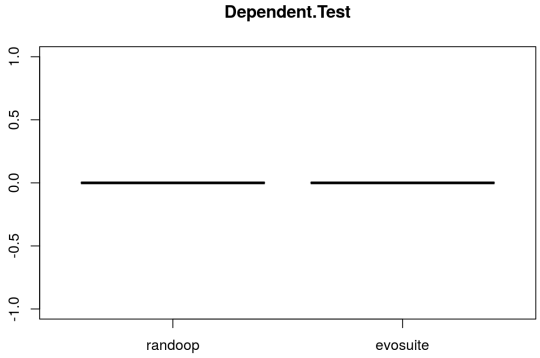


Figura A.62 Dependent Test (Randoop x Evosuite)

A.3.21 Empty Test

Lista A.187 Sumario dos dados Existentes

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0 0 0 0 0 0
```

Lista A.188 Sumário dos dados do Evosuite

```
## Min. 1st Qu. Median Mean 3rd Qu. Max.
## 0.00000 0.00000 0.00000 0.04404 0.00000 1.00000
```

Lista A.189 Resultado do Teste de Shapiro-Wilk

```
##
## Wilcoxon signed rank test with continuity correction
##
## data: lista__comp1$EmptyTest and lista__comp2$EmptyTest
## V = 0, p-value < 2.2e-16
## alternative hypothesis: true location shift is not equal to 0
## 0 percent confidence interval:
## NaN NaN
## sample estimates:
## midrange
## -1
```

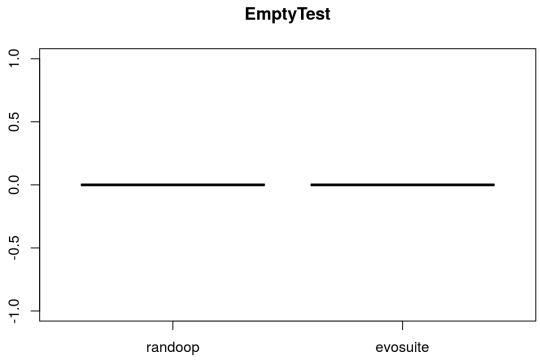


Figura A.63 Empty Test (Randoop x Evosuite)