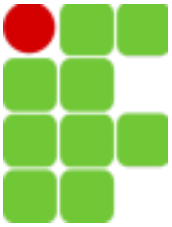


INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E
TECNOLOGIA DA PARAÍBA *CAMPUS* CAMPINA GRANDE
COORDENAÇÃO DO CURSO DE ENGENHARIA DE COMPUTAÇÃO

KILVIA DA SILVA CARVALHO
PAMELA VICTORIA SOARES LIMA

AVALIAÇÃO DE FERRAMENTAS DE GERAÇÃO DE CASOS DE TESTE DE SOFTWARE: UMA ANÁLISE COMPARATIVA

CAMPINA GRANDE – PB
2022

KILVIA DA SILVA CARVALHO
PAMELA VICTORIA SOARES LIMA

AVALIAÇÃO DE FERRAMENTAS DE GERAÇÃO DE CASO DE TESTE DE SOFTWARE: UMA ANÁLISE COMPARATIVA

Monografia apresentada à Coordenação do Curso de Engenharia de Computação do IFPB - Campus Campina Grande, como requisito parcial para conclusão do curso de Engenharia de Computação.

ORIENTADOR: DR. ALYSSON FILGUEIRA MILANEZ

CAMPINA GRANDE – PB
2022

KILVIA DA SILVA CARVALHO
PAMELA VICTORIA SOARES LIMA

AVALIAÇÃO DE FERRAMENTAS DE GERAÇÃO DE CASOS DE TESTE DE SOFTWARE: UMA ANÁLISE COMPARATIVA

Monografia apresentada à Coordenação do Curso de Engenharia de Computação do IFPB - Campus Campina Grande, como requisito parcial para conclusão do curso de Engenharia de Computação.

APROVADO EM: 26/ Ago / 2022

Prof. Dr. Alysson Filgueira Milanez
Instituto Federal de Educação, Ciência e Tecnologia da Paraíba *Campus* Campina Grande
Orientador

Prof. Me. Daniel Marques Vasconcelos Guimarães
Instituto Federal de Educação, Ciência e Tecnologia da Paraíba *Campus* Campina Grande
Membro da Banca

Profa. Ma. Iana Daya Cavalcante Facundo Passos
Instituto Federal de Educação, Ciência e Tecnologia da Paraíba *Campus* Campina Grande
Membro da Banca

CAMPINA GRANDE – PB
2022

C331a Carvalho, Kilvia da Silva.
Avaliação de ferramentas de geração de casos de teste de software: uma análise comparativa / Kilvia da Silva Carvalho, Pamela Victoria Soares Lima. - Campina Grande, 2022.
56f.:il.

Trabalho de Conclusão de Curso - Monografia (Curso de Bacharelado em Engenharia da Computação) - Instituto Federal da Paraíba, 2022.
Orientador: Prof. Dr.Alysson Filgueira Milanez.

1.Engenharia da computação - avaliação de ferramenta. 2. Software - teste. 3.Engenharia de software I. Lima, Pamela Victoria Soares. II. Título.

CDU 004.4

AGRADECIMENTOS

Agradecemos primeiramente a Deus, que nos deu forças para a conclusão do curso.

Ao professor e orientador Dr. Alysson Filgueira Milanez, pela confiança depositada em nós, e que através da sua sabedoria, conhecimento e determinação nos orientou durante todo o desenvolvimento deste trabalho.

Aos demais professores pelo conhecimento transmitido.

Aos nossos pais e familiares, pelo apoio e encorajamento ao longo do curso.

Às amizades construídas no decorrer da graduação, em especial João Igor, José Domingos, Josenildo Simões, Isaque Melo, Micael Marques, Iasmin Santos e Myrlla Lucas, pela companhia e amparo imensurável.

Aos nossos amigos, Victor Medeiros, Fabrício Pires, Fabrício Rodrigues, Philipe Luna, Kaio Carvalho, Vitória Lopes, Annyzabele Barbosa e Josefa Maria pelo apoio emocional nessa caminhada.

Ao LaTIn, em especial ao professor Dr. Katyusco Farias e a professora Dra. Ianna Sodré, que contribuíram com nosso crescimento pessoal e profissional.

Por fim, ao Instituto Federal da Paraíba, pela oportunidade da realização do curso.

“Filho meu, se aceitares as minhas palavras, e esconderes contigo os meus mandamentos,

Para fazeres o teu ouvido atento à sabedoria; e inclinares o teu coração ao entendimento;

Se clamares por conhecimento, e por inteligência alcançares a tua voz,

Se como prata buscares e como a tesouros escondidos procurares,

Então entenderás o temor do Senhor, e acharás o conhecimento de Deus.

Porque o Senhor dá a sabedoria; da sua boca é que vem o conhecimento e o entendimento.” (Provérbios 2: 3-6)

RESUMO

Devido a sua complexidade, é compreensível que sistemas de software estejam sujeitos a diversos tipos de erros e inconsistências. Para evitar que esses erros cheguem aos usuários finais e causem prejuízos de valor incalculável, é fundamental introduzir atividades de teste em projetos de desenvolvimento de software que garantam sua qualidade. É possível garantir a qualidade na etapa de desenvolvimento do software com a constância e repetição dos testes, contudo, manualmente essa seria uma atividade custosa de ser realizada, devido à quantidade de tempo necessária para executar todos os processos. Uma vez que os testes automatizados sejam criados, eles podem ser executados quantas vezes forem necessárias. Em outros termos, testes automatizados fazem o que os testes manuais não são capazes, por exemplo, simular a execução de milhares de usuários em um sistema. Outro benefício dos testes automatizados é a possibilidade de serem programados para executar após cada modificação do código, informando caso haja defeitos, e auxiliando os desenvolvedores a encontrar falhas antes de enviar o produto para a equipe de qualidade. Nesse contexto, existem diversas ferramentas que geram casos de teste automaticamente para estes fins, no entanto, identificar a melhor ferramenta para cada cenário é uma tarefa custosa. O presente trabalho tem como objetivo auxiliar a escolha da ferramenta mais adequada, dentre o Evosuite e o Randoop, para o contexto do teste de software, baseado na cobertura de código e teste de mutação como métricas qualitativas estudadas em dois sistemas operacionais. Ao final do estudo, os resultados obtidos mostram que o Evosuite gerou casos de teste que cobriram uma porcentagem maior de código do que a cobertura dos casos de teste gerados pelo Randoop, tendo um melhor desempenho no Windows que no MacOS nas duas ferramentas. Considerando a métrica de mutação, os testes do Randoop mataram mais mutantes quando comparados aos testes gerados pelo Evosuite em ambos os sistemas operacionais, e o percentual de mutantes mortos foi equivalente no Windows e no MacOS.

Palavras-chave: Teste de software; Geração automática de teste; Evosuite; Randoop; Cobertura de Código; Teste de Mutação.

ABSTRACT

Due to your complexity, it is understandable that software systems are susceptible to different types of errors and inconsistencies. To prevent these errors from reaching end users and causing incalculable losses, it is essential to introduce testing activities into software development projects that guarantee their quality. It is possible to guarantee the quality in the software development stage with the constancy and repetition of the tests, however, manually this would be a costly activity to be carried out, due to the amount of time required to execute all the processes. Once automated tests are created, they can be run as often as needed. In other words, automated tests do what manual tests cannot, for example, simulate the execution of thousands of users on a system. Another benefit of automated tests is that they can be programmed to run after each code modification, informing you if there are defects, and helping developers to find faults before sending the product to the quality team. In this context, there are several tools that generate test cases automatically for these purposes, however, identifying the best tool for each scenario is a costly and time-consuming task. The present work aims to support the choice of the most suitable tool, among Evosuite and Randoop, for the context of software testing, based on code coverage and mutation testing as qualitative metrics studied in two operating systems. At the end of the study, the results obtained show us that Evosuite generated test cases that covered a higher percentage of code than the coverage of test cases generated by Randoop, performing better on Windows than on MacOS in both tools. Considering the mutation metric, the Randoop tests killed more mutants when compared to the tests generated by Evosuite on both operating systems, and the percentage of mutants killed was equivalent to Windows and MacOS.

Keywords: Software testing; Automatic test generation; Evosuite; Randoop; Code Coverage; Mutation Test.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ABNT	Associação Brasileira de Normas Técnicas
HTML	*HyperText Markup Language*
IA	Inteligência Artificial
IEC	*International Electrotechnical Commission*
JDK	*Java Development Kit*
JML	*Java Modeling Language*
ISO	*International Organization for Standardization*
ISTQB	*International Software Testing Qualifications Board*
NBR	Norma Brasileira
OTAN	Organização do Tratado do Atlântico Norte
SBTVD	Sistema Brasileiro de Televisão Digital
XML	*Extensible Markup Language*

SUMÁRIO

1 INTRODUÇÃO

Em virtude de o desenvolvimento de software ser uma tarefa árdua e complicada, existe a possibilidade de ocorrerem problemas que passam despercebidos pelos desenvolvedores (SUÁREZ, 2015). Por isso, os testes de software são de extrema importância para garantir a qualidade e confiabilidade do que foi desenvolvido.

De acordo com Pressman (2016), o teste de software pode ser compreendido como o processo de executar um programa com a intenção de descobrir erros. Molinari (2012) complementa esse pensamento afirmando que os testes possuem como principal objetivo sanar ou diminuir de forma considerável a ocorrência de erros, quando o sistema estiver funcionando em produção.

1.1 ABORDAGENS DE TESTE

Desse modo, a execução dos testes precisa ser contínua, evitando que um produto chegue com problemas às mãos do consumidor ou cliente. Com isso em mente, Crespo (2013) afirma que adotar técnicas de teste de software é fundamental para a avaliação do software desenvolvido. Dentre essas técnicas, as principais são:

Teste de Caixa Branca: Pressman (2016) define o teste de caixa branca como o teste com o objetivo de testar a estrutura interna do software, focando em como o código fonte foi construído e a lógica de programação utilizada no desenvolvimento dos métodos, percorrendo todos os caminhos lógicos possíveis.

Teste de Caixa Preta: Segundo Inthurn (2001), o teste de caixa preta possui o objetivo de verificar se todos os requisitos do sistema estão funcionando de acordo com a regra de negócio do projeto. São testes que focam nos aspectos funcionais.

Teste de Caixa Cinza: De acordo com Lewis (2000), o teste de caixa cinza é uma combinação do teste de caixa branca com o teste de caixa preta. Nessa abordagem, quem irá testar consegue ter um entendimento melhor do sistema, tornando possível a otimização dos casos de teste que vão ser executados. Essa abordagem é comumente aplicada a testes de aplicações web.

1.2 TÉCNICAS DE TESTE DE SOFTWARE

Diante destas abordagens, pode-se citar as principais fases de testes de software utilizados atualmente:

Teste Unitário: segundo Delamaro e col. (2016), o teste unitário é o processo de testar os componentes do programa, as menores partes do sistema, dentre eles, pode-se citar: os métodos e funções das classes ou pacotes utilizados no projeto. Esse tipo de teste possui como objetivo verificar essas unidades do sistema de forma isolada, com o intuito de garantir que a lógica de cada uma delas esteja correta e funcionando de acordo com o esperado. O teste unitário geralmente adota uma abordagem de teste caixa branca.

Teste de Integração: diferentemente do teste de unidade que busca verificar se os elementos individuais estão funcionando corretamente, segundo Delamaro e col. (2016), o teste de integração busca garantir que os componentes interagem na forma que está especificada no projeto do sistema, com o objetivo de expor faltas nas interfaces. O teste de integração geralmente adota uma abordagem de teste caixa preta, mas também pode adotar as abordagens caixa branca e caixa cinza.

Teste de Sistema: Lewis (2000) diz que essa estratégia consiste na verificação do software, quando é incorporado a outros elementos do sistema, ou seja, ele permanece sem a presença de falhas após a integração. Ainda de acordo com Pressman (2016), o teste de sistema consiste em uma série de diferentes testes, cujo objetivo principal é colocar à prova todo o sistema, verificando se todos os elementos do sistema foram adequadamente integrados e realizam suas funções adequadamente. Costuma ser realizado segundo uma abordagem caixa preta.

Testes de Regressão: segundo Pressman (2016), o Teste de Regressão é a reexecução de algum subconjunto de testes que já foram conduzidos para garantir que as modificações não propagaram efeitos colaterais indesejáveis. Efeitos colaterais, nesse caso, são problemas causados em partes do sistema que estavam funcionando antes que tais modificações fossem feitas. O teste de regressão geralmente adota uma abordagem de teste caixa preta.

1.3 TESTE MANUAL E TESTE AUTOMATIZADO

De modo geral, os testes podem ser executados de dois modos: manual ou automatizado. Silva (2021) diz que o teste manual comumente tem um alto custo financeiro, permitindo que o profissional que os realize vivencie condições equivalentes às do ambiente

de produção, já que é possível definir manualmente os parâmetros dos testes. Por outro lado, são mais demorados e necessitam totalmente da interação humana.

Os testes automatizados trazem a facilidade de ter *scripts*, que agilizam o processo, ajudando a descobrir prontamente se o sistema está com o desempenho desejado, e, por serem automatizados, não necessitam de uma pessoa para a execução dos testes (SILVA, 2021). Segundo Izabel (2014), a automação dos testes de software apresenta-se como uma forma de agilizar o processo e criar ferramentas mais eficazes que sejam capazes de garantir a qualidade de uma forma mais completa.

Embora os testes automatizados tragam vantagens em relação ao aumento da produtividade, reuso, diminuição de erros e confiabilidade, eles também possuem desvantagens em relação ao custo de implantação, inclusão em cenários complexos e específicos, geração de dados de teste e a dificuldade no desenvolvimento dos *scripts* de teste (DIAS, 2020).

De acordo com Inthurn (2011), é possível garantir a qualidade na etapa de desenvolvimento do software com a constância e repetição dos testes. Manualmente, seria uma atividade custosa, devido à quantidade de tempo necessária para realizar todos os processos, e segundo Valente (2020), uma vez que os testes automatizados sejam criados, eles precisam ser executados sempre que o sistema sofrer uma modificação, no entanto, a dificuldade está em desenvolver um *script* que seja capaz de realmente testar a aplicação da forma mais completa possívelsso ajuda a reduzir custos e garante a qualidade do projeto do código. Portanto, os testes automatizados fazem o que os testes manuais não são capazes, por exemplo, de simular a execução de milhares de usuários em um sistema. Em contrapartida, uma automação de qualidade é capaz de fazer isso com facilidade.

Bartié (2009) afirma que os testes têm por objetivo identificar o maior número possível de erros tanto nos componentes isolados quanto na solução tecnológica como um todo, e para ser possível descobrir *bugs* no sistema, é necessário ter bons casos de teste com o intuito de elevar a probabilidade de revelar um erro ainda não descoberto, pois, conforme Mats (2001), um dos principais problemas nas atividades de testes de software é a ausência de critérios para seleção dos casos de teste, definição da sua completude e estabelecimento de um ponto de parada, dificultando a revelação de falhas no produto.

Levando em consideração a problemática da seleção de boas suítes de teste, algumas ferramentas de automação foram lançadas no mercado para automatizar essa tarefa. Com o objetivo de investigar dentre as ferramentas de geração automática de suítes de teste para

softwares codificados na linguagem de programação Java, selecionamos o *Randoop*[1] *e Evosuite*[2], a fim de analisar quais delas conseguem gerar uma suíte de teste que garante uma maior qualidade ao software que está sendo testado. Ao fim do estudo, a pesquisa serve como ambiente de consulta para que membros de equipes de desenvolvimento de softwares, sejam eles desenvolvedores, testadores ou estudantes da área de informática, consigam determinar, de forma mais eficiente, durante o processo de testes, a ferramenta mais adequada a ser utilizada no contexto de geração de testes.

1.4 OBJETIVOS

Nesta seção, são apresentados o objetivo geral e os específicos do presente trabalho.

1.4.1 Objetivo Geral

- Executar um plano de identificação e investigação de ferramentas de geração de casos de teste *Open Source*, com o intuito de facilitar a escolha da ferramenta mais adequada para determinados cenários/projetos de desenvolvimento de software.

1.4.2 Objetivos Específicos

- Identificar ferramentas de teste *Open Source* utilizadas por profissionais da área de desenvolvimento;
- Aplicar as ferramentas em cenários reais de sistemas de desenvolvimento de software;
- Apresentar resultados comparativos entre as ferramentas de geração de casos de teste, em conformidade às suas respectivas documentações, e as métricas de cobertura de código e mutação.

[1] Disponível em: https://randoop.github.io/randoop/. Acesso em 28 jul. 2022.
[2] Disponível em: https://www.evosuite.org/. Acesso em 28 jul. 2022.

2 FUNDAMENTAÇÃO TEÓRICA

No capítulo a seguir serão abordados conceitos sobre engenharia e teste de software, critérios de qualidade de software e estratégias e técnicas de teste, com ênfase em teste de caixa branca e caixa preta.

2.1 ENGENHARIA DE SOFTWARE

Nos dias de hoje, o software está presente em diversas áreas da nossa vida. É possível encontrá-los em diferentes dispositivos e produtos de engenharia, incluindo automóveis, aviões, satélites, entre outros. Assim, tem-se que o software está contribuindo para renovar indústrias e serviços tradicionais, como telecomunicações, transporte, hospedagem, lazer e publicidade. Portanto, devido a sua relevância, a Engenharia de Software é a área na computação destinada a investigar e propor soluções que permitam desenvolver sistemas de software, de forma produtiva e com qualidade (VALENTE, 2020).

A Engenharia de Software é a criação e a utilização de sólidos princípios de engenharia a fim de obter software de maneira econômica, que seja confiável e que trabalhe em máquinas reais (BAUER, 1972), e abrange um processo, um conjunto de métodos e ferramentas que possibilitam aos profissionais desenvolverem software de alta qualidade (PRESSMAN, 2016).

2.2 CONFERÊNCIA DA OTAN

Em meados dos anos 70, na Conferência da OTAN, surgiram os primeiros elementos dessa área, devido ao evento chamado “crise do software”: em que os computadores evoluíram cada vez mais rápido com a introdução dos microchips, e os softwares não estavam acompanhando no mesmo ritmo essa evolução. Sommerville (2011) relata os maiores problemas enfrentados à época, como o estouro de orçamento, prazos não cumpridos, softwares que não atendiam aos requisitos do usuário e projetos com poucos elementos para permitir sua gestão e código fonte de difícil manutenção. Diversos foram os esforços e progressos de sistematização, automatização da documentação do programa e testes nessa época, e os avanços obtidos em técnicas e métodos para construção de software são notáveis atualmente. De acordo com Valente (2020), hoje já se tem conhecimento de que software, na maioria das vezes, não deve ser construído em fases estritamente sequenciais, e prosseguindo, as mais variadas técnicas de testes podem ser usadas para garantir que os sistemas em construção tenham

qualidade e que falhas não ocorram quando eles entrarem em produção e forem usados por clientes reais.

2.3 DEFINIÇÃO DE TESTE DE SOFTWARE

Devido a sua complexidade, é compreensível que sistemas de software estejam sujeitos a diversos tipos de erros e inconsistências. Para evitar que esses erros cheguem aos usuários finais e causem prejuízos de valor incalculável, é fundamental introduzir atividades de teste em projetos de desenvolvimento de software que garantam sua qualidade. Pressman (2016) define a qualidade como item preponderante para o sucesso de um software.

Por definição, o teste de software consiste na execução de um sistema com um conjunto finito de casos, com o objetivo de verificar se ele possui o comportamento esperado, sintetizando não apenas os benefícios de testes, mas também suas limitações, mostrando a presença de *bugs*, e não a sua ausência (DIJKSTRA, 1982).

Sommerville (2011) define o teste como uma atividade destinada a descobrir os defeitos do programa antes do uso e mostrar se um programa faz o que lhe foi proposto. Os resultados do teste são verificados à procura de erros, anomalias ou informações sobre os atributos do programa.

2.4 QUALIDADE DE SOFTWARE

De acordo com o *International Software Testing Qualifications Board* (ISTQB), um selo internacional de qualidade para testadores de software, realizar os testes é de extrema importância, pois permite identificar erros durante as etapas de desenvolvimento, garante a confiança do usuário final e sua satisfação ao utilizar o software e permite assegurar a qualidade do produto e seu funcionamento correto. Entre seus objetivos específicos estão a verificação da integração adequada dos componentes, análise de todos os requisitos, garantido que foram implementados corretamente e que os defeitos encontrados sejam corrigidos antes da implantação do software e a redução de custos de manutenção corretiva e retrabalho, já que o custo médio da indústria de software para corrigir um defeito durante a geração de código é de aproximadamente US$ 977 por erro, e custo médio para corrigir o mesmo erro, caso ele tenha sido descoberto durante os testes do sistema, passa a ser de US$ 7.136 por erro (PRESSMAN, 2016).

Os problemas com a qualidade de software foram descobertos na década de 1960, com

o desenvolvimento do primeiro grande sistema de software. Quando foi entregue, era lento, pouco confiável, difícil de manter e de reusar, e buscando uma solução, passaram a adotar técnicas formais de gerenciamento de qualidade do software, desenvolvidas a partir dos métodos usados na indústria manufatureira, em um esforço para melhorar a qualidade dos produtos em produção. Como parte disso, eles desenvolveram uma definição de 'qualidade', baseada na conformidade com uma especificação detalhada (CROSBY, 1979) e na noção de tolerâncias. Essas técnicas de gerenciamento de qualidade, junto a melhores testes de software, conduziram melhoras significativas no nível geral de qualidade de software (SOMMERVILLE, 2011).

Em uma entrevista publicada na Internet[3], Bertrand Meyer discute o que se chama de dilema da qualidade: se produzimos um sistema de software de péssima qualidade, nós o perdemos, porque ninguém vai querer comprá-lo. Por outro lado, se gastamos com um esforço extremamente grande e grandes quantidades de dinheiro para construir um software absolutamente perfeito, ele levará muito tempo para ser concluído, e o custo de produção será tão alto que iremos à falência. Ou perdemos a oportunidade de mercado, ou esgotamos todos os nossos recursos. Assim, os profissionais dessa área tentam encontrar um meio-termo em que o produto é suficientemente bom para não ser rejeitado durante uma avaliação, por exemplo, mas também não possui tamanho trabalho que levaria muito tempo ou custo para ser finalizado (MEYER, 2003).

Mas o que é exatamente "bom o suficiente"? Um software bom o suficiente fornece funções e características de alta qualidade que os usuários desejam, mas, ao mesmo tempo, fornece outras funções e características mais especializadas, e ainda sim, contém erros conhecidos. O fornecedor de software espera que a grande maioria dos usuários ignore os erros pelo fato de estarem muito satisfeitos com as outras funcionalidades oferecidas pela aplicação (PRESSMAN, 2016).

De acordo com Pressman (2016), para tentar mitigar esses erros, é necessário um conjunto de fatores que nos leve a uma indicação da qualidade, garantindo que cada produto resultante atinja suas metas de qualidade. Podemos classificar esses fatores em duas categorias: a primeira, em que os fatores podem ser medidos diretamente (por exemplo, defeitos revelados durante os testes) e a segunda, onde fatores podem ser medidos apenas indiretamente (por exemplo, usabilidade ou facilidade de manutenção).

---

[3] Disponível em: www.artima.com/intv/contracts.html. Acesso em 16 fev. 2022.

2.5 ESTRATÉGIAS E TÉCNICAS DE TESTE

A respeito das estratégias de teste, muitas delas podem ser utilizadas para testar um software. Segundo Pressman (2016), o teste de software é envolvido por basicamente quatro etapas: o planejamento de testes, projeto de casos de teste, a execução e avaliação dos resultados dos testes. Tais atividades devem estar integradas ao próprio processo de desenvolvimento de software, que em geral se concretizam em três fases de teste: de unidade, de integração e de sistema. Esses testes podem ser usados tanto para verificação, que garante que um sistema atende às suas especificações, como para validação de sistemas, que garante que um sistema atende às necessidades de seus clientes (VALENTE, 2020).

Para aplicações convencionais, o software é testado a partir de duas perspectivas diferentes: primeiro, a lógica interna do programa é exercitada usando técnicas de projeto de caso de teste “caixa-branca”, uma filosofia de projeto de casos de teste que usa a estrutura de controle descrita como parte do projeto no nível de componentes para derivar casos de teste, e, posteriormente, os requisitos de software são exercitados usando técnicas de projeto de casos de teste “caixa-preta”, que permite derivar séries de condições de entrada que utilizarão completamente todos os requisitos funcionais para um programa. Casos de teste de uso ajudam no projeto de testes para descobrir erros no nível de validação do software. Em todos os casos, a intenção é encontrar o número máximo de erros com o mínimo de esforço e tempo (PRESSMAN, 2016).

Usando métodos de teste caixa-branca, também chamado de teste da caixa-de-vidro ou teste estrutural, o engenheiro de software pode criar casos de teste que garantam que todos os caminhos independentes de um módulo foram exercitados pelo menos uma vez, que exercitem todas as decisões lógicas nos seus estados verdadeiro e falso, que executem todos os ciclos em seus limites e dentro de suas fronteiras operacionais e que exercitam estruturas de dados internas para assegurar a sua validade. Já o teste caixa-preta, também chamado de teste comportamental ou teste funcional, em vez disso, é uma abordagem complementar, com possibilidade de descobrir uma classe de erros diferente daquela obtida com métodos caixa-branca, pois tenta encontrar erros em funções incorretas ou ausentes, erros de interface, erros em estruturas de dados ou acesso a bases de dados externas, erros de comportamento ou de desempenho e erros de inicialização e término (PRESSMAN, 2016).

Também podemos citar a técnica de teste de caixa-cinza, que mescla o uso das técnicas de caixa-preta e de caixa-branca. Esta técnica analisa a parte lógica mais a funcionalidade do sistema, fazendo uma comparação do que foi especificado com o que está sendo realizado.

Usando esse método, o testador comunica-se com o desenvolvedor para entender melhor o sistema e otimizar os casos de teste que serão realizados. Isso envolve ter acesso a estruturas de dados e algoritmos do componente a fim de desenvolver os casos de teste, que são executados como na técnica da caixa-preta. A caixa-cinza pode incluir também o uso de engenharia reversa para determinar por exemplo os limites superiores e inferiores das classes, além de mensagens de erro (BEQUE, 2009).

Existe um risco real de que determinado componente de um sistema tenha sido modificado (incluído um novo parâmetro ou modificado o nome de uma função, por exemplo), deixando-o incompatível com o restante do sistema ao qual está inserido. Nessa abordagem, a integração entre a unidade a ser testada e o componente modificado poderia não ser atualizada e, mesmo assim, passaria nos testes unitários. Dessa forma, a abordagem do teste de integração de unidades se torna essencial, pois é somente nessa etapa de testes que é possível detectar o problema (BARTIÉ, 2009).

Há casos de teste que podem ser projetados para verificar se as especificações funcionais do software estão corretamente implementadas, o que é referido dentro da literatura como testes de conformidade, testes de correção ou testes funcionais. No entanto, várias outras propriedades não funcionais também devem ser levadas em consideração nas abordagens de teste, incluindo desempenho, confiabilidade e usabilidade, entre muitas outras (BOURQUE e FAIRLEY, 2014).

3 ESTADO DA ARTE

As publicações científicas a seguir apresentam resultados significativos, que estão relacionados com o tema deste trabalho: Amorim et al. (2017), Pereira (2019), Parreira e Cardeal (2015), Milanez (2014), Oliveira (2016), Souza e Machado (2019) e Jia e Harman (2010).

3.1 AMORIM ET AL. (2017)

Encontrar a ferramenta certa para gestão de testes ainda é um problema e levando em consideração que esse tipo de ferramenta acrescenta qualidade e confiança ao desenvolvimento, Amorim et al. (2017) realizaram um estudo com o objetivo de diminuir essa problemática. O trabalho faz uma análise das ferramentas *Open Source* voltadas para a gestão de teste de software, visando realizar um comparativo entre as ferramentas analisadas, com base nas características de qualidade da NBR ISO/IEC 9126. Por esse motivo, foi realizado um comparativo entre as ferramentas TestLink, Rth, TestiTool e TestMaster, destinadas ao gerenciamento de testes de software.

No fim da execução do estudo, foram apresentados dois quadros de resultados. O primeiro faz um comparativo entre as ferramentas estudadas com as características gerais do procedimento de desenvolvimento de software; e, por fim, foi observado que as ferramentas TestLink e Rth atendem a uma maior quantidade de características analisadas, destacando-se das demais. No entanto, a ferramenta TestLink tem mais vantagem em relação a ferramenta Rth, pois ela possui uma interface de fácil utilização. Além disso, a ferramenta TestLink possui integração com as ferramentas de bug track, aumentando seu desempenho em relação à gestão de testes de software. Portanto, neste primeiro comparativo, os autores chegaram à conclusão de que a ferramenta TestLink é a melhor entre as ferramentas analisadas.

O segundo faz um comparativo entre as características de qualidade, usabilidade e funcionalidade presentes na NBR ISO/IEC 9126 e as ferramentas selecionadas para análise.

Em relação a funcionalidade foram analisados os pontos de segurança de acesso, acurácia, adequação e interoperabilidade. Concluiu-se então que para essa característica as ferramentas TestLink e Rth se destacaram das demais, no entanto, o TestLink se destaca um pouco mais nas características específicas do quesito funcionalidade. Levando em consideração as ferramentas que atendem menos características desse quesito, temos que TestiTool e a TestMaster se encontram empatadas. Tratando-se da característica usabilidade, os autores

analisaram a operacionalidade, atratividade, inteligibilidade e a apreensibilidade. Como conclusão, mais uma vez a ferramenta TestLink é a melhor, pois ela proporciona um melhor entendimento e possui maior facilidade de manuseio. Por fim, na análise das características de modelo de qualidade em uso, as ferramentas Rth e TestLink se destacaram por possuírem todas as características desse quesito. Com isso em mente, a ferramenta TestiTool se destaca como a menos recomendável por não proporcionar uma interface amigável e não conter integração com ferramentas externas de rastreamento de defeitos. Ao fim da análise, Amorim et al. (2017) concluíram que TestLink satisfaz uma maior quantidade de características de qualidade da NBR ISO/IEC 9126, sendo considerada a melhor entre as ferramentas analisadas.

3.2 JIA E HARMAN (2010)

Atualmente, há uma grande busca por otimização para reduzir o custo do processo de teste de mutação, uma técnica de teste de software baseada em falhas. A literatura sobre testes de mutação tem contribuído com uma série de abordagens, ferramentas e resultados empíricos, e partindo dessa premissa, os autores fornecem uma análise e levantamento abrangente sobre testes de mutação, através de uma pesquisa detalhada de análise de tendências de desenvolvimento e resultados dos testes. Essas análises fornecem evidências de que as técnicas e ferramentas de teste de mutação estão atingindo um estado de maturidade e aplicabilidade cada vez maior, enquanto o tópico de teste de mutação em si é objeto de crescente interesse.

Com a frequente busca por otimização para reduzir o custo do processo de teste de mutação, o artigo abrange teorias, técnicas de otimização e detecção de mutantes equivalentes, aplicações, estudos empíricos e ferramentas de mutação, e a partir dos dados coletados sobre a literatura de testes de mutação, o estudo revela uma tendência cada vez mais prática sobre o assunto. Além do mais, também foi encontrado evidências de que há um número crescente de novas aplicações, e existindo programas maiores e mais realistas, os testes de mutação podem ser passíveis de manipulá-los.

Jia e Harman (2010) explicam que com o surgimento de novas ferramentas de código aberto, é possível fornecer indicadores de apoio à afirmação de que o campo do teste de mutação está agora atingindo um estado maduro. Trabalhos recentes tendem a se concentrar em formas mais elaboradas de mutação do que em simples falhas, havendo um interesse nos efeitos semânticos da mutação, em vez da realização sintática dela. Essa migração da realização sintática da mutação para o efeito semântico desejado, aumentou o interesse na mutação para gerar falhas sutis e encontrar as mutações que retornam falhas reais. Os autores esperam que

no futuro haja ainda uma maior maturidade, com o fornecimento de ferramentas práticas de geração de mutantes mais realistas e casos de teste para matá-los.

3.3 MILANEZ (2014)

Segundo Milanez (2014), os testes de software são utilizados para verificar conformidade em programas baseados em contrato, já que a verificação por provas formais tem baixo poder de escalabilidade e análise estática e, em alguns casos, limitada para identificar não-conformidades mais gerais. O autor também afirma que os casos de teste tradicionais, com dados de teste providos manualmente, podem ser ineficazes para detectar não-conformidades sutis que surgem apenas após diversas criações e modificações nos objetos sob teste. Em contrapartida, os casos de teste com dados de teste gerados automática- e aleatoriamente, são uma abordagem promissora quando testes mais substanciais são necessários.

Por isso, o trabalho do autor propõe avaliar uma abordagem para detecção e categorização de não-conformidades, cujo objetivo é auxiliar o programador no processo de correção de não-conformidades.

A detecção é suportada por uma abordagem de Testes Gerados Aleatoriamente, e a categorização por uma abordagem baseada em heurísticas. Milanez (2014) realizou duas avaliações. A primeira avaliação utilizou a abordagem criada pelo autor para detecção de não-conformidades e do processo para categorização manual, obtendo como resultado, que a maioria das não-conformidades detectadas foram classificadas como erros de pós-condição. Além disso, também foi observado pelo autor a necessidade de uma estrutura de testes mais elaborada para detecção de não-conformidades. Ademais, o autor comparou a sua abordagem com a ferramenta JET, uma ferramenta para detecção de não-conformidades em programas JML baseada em testes, utilizando um subconjunto dos programas usados no primeiro estudo.

Na segunda avaliação, foi realizado um comparativo entre a categorização manual e a automática, obtendo como resultado um valor de coincidências de 0.73, indicando que há similaridade entre essas duas abordagens. Além do mais, através da comparação dos resultados da categorização automática com a categorização realizada por experts em JML, também foi observado algumas similaridades.

3.4 OLIVEIRA (2016)

Considerando o processo de desenvolvimento de software, uma grande parcela de

tempo é dedicada à atividade de geração de casos de teste. Por isso, surge a necessidade da utilização de ferramentas de geração desses casos, pois segundo Oliveira (2016), a atividade de geração das suítes de teste será mais ágil e menos onerosa, para que as empresas que atuam na área de desenvolvimento de software possam gerar produtos com mais qualidade, atingindo dessa forma seus objetivos.

Com isso em mente, o autor traz o seguinte questionamento: como adotar um processo de desenvolvimento e teste de software de forma que alcance os objetivos e obtenha um bom resultado, apesar das dificuldades em gerar dados de teste devido aos domínios de entrada dos programas serem, em geral, infinitos? Por isso, o principal objetivo do autor foi realizar uma análise experimental de ferramentas automáticas geradoras de casos de teste, com o objetivo de identificar qual delas apresentam a melhor relação custo/benefício, e para isso, Oliveira (2016) utilizou as seguintes métricas: número de dados de teste gerados, defeitos detectados e cobertura de código determinada pelos conjuntos de teste.

Em relação ao estudo feito, foi dividido em duas fases: na primeira, foi selecionado um conjunto de 32 programas codificados na linguagem Java, e as ferramentas Randoop e CodePro foram analisadas em relação a esse conjunto de programas. Os primeiros resultados obtidos da análise foram que a ferramenta CodePro apresentou um melhor custo-benefício em relação ao Randoop, levando em consideração apenas a geração de suítes de teste. No entanto, em relação aos testes de mutação, o Randoop apresentou melhores resultados. Na segunda fase do estudo, o autor incluiu a ferramenta Evosuite, e casos de testes gerados de forma manual. No final da pesquisa, foram apresentados os resultados em termos de eficácia e eficiência, através do comparativo entre os quatro conjuntos de teste. Por fim, ao analisar os dados obtidos durante a pesquisa, Oliveira (2016) concluiu que a ferramenta que obteve os melhores resultados foi o Evosuite, pois cobre uma maior quantidade de linhas de código.

3.5 PARREIRA E CARDEAL (2015)

A busca para automatizar os testes de software é cada vez maior visto que diminui os custos do projeto e acaba gerando uma melhora na produtividade. No entanto, para que seja possível ter sucesso na utilização das ferramentas de automação, as empresas precisam de maturidade no processo de testes e profissionais qualificados com o perfil adequado nas etapas de planejamento e execução dos testes de software. Visando isso, o estudo de Parreira e Cardeal (2015) apresenta os principais resultados obtidos na empresa Softbox após a adoção da prática de automação dos testes durante o desenvolvimento de um determinado projeto.

De acordo com os autores, umas das maiores vantagens da automação de testes não é somente executar os casos de testes de forma mais rápida ou a qualquer momento do dia, mas também, ter uma maior amplitude de cobertura dos testes e o aumento da complexidade das suítes de teste. No entanto, automatizar todos os testes manuais nem sempre é mais vantajoso, visto que, alguns testes podem ser muito complexos de automatizar, e outros exigem integrações entre sistemas que podem necessitar de uma validação mais subjetiva, ou seja, uma análise mais rigorosa/criteriosa.

Como resultado da pesquisa, a empresa decidiu automatizar os testes em seus projetos, já que demandam muito esforço dos testes manuais a cada alteração crítica feita no sistema. Ademais, devido às limitações humanas e naturais, os analistas de testes nem sempre validam com a mesma eficácia os resultados obtidos, principalmente nos testes regressivos. Além do investimento na automação de testes, segundo o autor, a empresa também criou o seu próprio framework para programação de testes utilizando o *PHPUnit*.

Levando em consideração o tempo de esforço para o desenvolvimento dos scripts de teste, Parreira e Cardeal (2015) afirmam que é grande em relação a execução de testes manuais, contudo, isso vai diminuindo a cada execução automática desses testes, diminuindo a necessidade de ter sempre um teste manual para realizar tal função.

Por fim, de acordo com Parreira e Cardeal (2015) a aplicação de testes automatizados no projeto foi de grande ganho, a execução dos testes automatizados é feita várias vezes pelos desenvolvedores sempre que necessário, seja após uma implantação, ou após um ajuste necessário para a correção de um bug. Consequentemente, houve um aumento na qualidade das entregas dos desenvolvedores e na diminuição de bugs relatados pelos usuários.

3.6 PEREIRA (2019)

Devido ao fato de a qualidade de software ser um fator fundamental durante o processo de desenvolvimento e possuir como premissa a seleção de boas suítes de teste, medir a qualidade de diversos tipos de testes é um desafio importante a ser atingido. Por isso, a pesquisa realizada por Pereira (2019) fez uma avaliação da efetividade de testes de sistemas realizados pela empresa MOPA Embedded Systems na utilização do middleware Ginga para TV Digital. Com o intuito de avaliar a suíte de teste de sistema executada pela empresa, foi utilizado para avaliação o critério de cobertura de código, seguido de um estudo exploratório para analisar as possíveis justificativas para os resultados obtidos da cobertura. Por último, realizaram testes exploratórios para confirmar as descobertas obtidas durante o estudo exploratório. Através

dessas etapas, foi possível obter uma conclusão sobre a qualidade dos testes realizados.

Ao final do estudo foi possível concluir que a cobertura das suítes de teste do fórum SBTVD não é boa e não garante a qualidade e conformidade do Ginga com as diretivas da norma ABNT. De acordo com o autor, a equipe de desenvolvimento de software da empresa já tinha essa percepção antes mesmo do estudo ser iniciado, e para garantir o bom funcionamento do Ginga, a empresa realizava testes adicionais. No entanto, esses testes eram realizados baseados em falhas encontradas durante a execução dos testes manuais, logo, não tinham como foco a cobertura de código.

O autor também afirma no estudo, que apesar da automatização ainda não estar em um nível de maturidade ideal, é fundamental ser acrescentado em todo o ciclo de testes da empresa, já que antes era executado somente de maneira manual.

Com os resultados obtidos da pesquisa, foi possível fazer a equipe de desenvolvimento perceber o quanto o processo de teste de software precisa ser executado constantemente, e que automatizar os testes pode auxiliar e melhorar o processo de desenvolvimento em relação à produtividade e qualidade do software. Além disso, foi realizado um treinamento na empresa voltado para testes do desenvolvimento de software e a cobertura de código passou a ser utilizada largamente na empresa nos testes unitário, integração e de sistemas para verificar a cobertura dos testes que passaram a ser implementados.

3.7 SOUZA E MACHADO (2019)

Segundo Souza e Machado (2019), constantemente estão sendo desenvolvidas ferramentas cujo objetivo é a geração automática de suítes de teste. No entanto, de acordo com os estudos realizados, a efetividade dos conjuntos de testes gerados nem sempre têm apresentado resultados satisfatórios.

Por isso, o estudo das autoras traz uma análise entre as suítes de testes geradas automaticamente e os casos de teste modelados manualmente pelos desenvolvedores. Para tal análise, Souza e Machado (2019) selecionaram 10 programas codificados na linguagem de programação Java que têm coleções de testes manuais e aplicaram as ferramentas Randoop e Evosuite, no intuito de analisar a qualidade das suítes de testes geradas utilizando as métricas de mutação e cobertura por linha.

Como resultado da pesquisa, as autoras concluíram que, em relação às métricas de coberturas de linha e de mutação, há uma diferença estatística entre as suítes de testes geradas de forma manual e as geradas de forma automática. Ademais, Souza e Machado (2019) também

perceberam que em relação à cobertura de mutação, existe uma diferença estatística entre as suítes de teste geradas pelo Evosuite e Randoop. Além disso, através do estudo, as autoras afirmam que uma grande quantidade de casos de testes gerados não significa necessariamente a efetividade de coleções de teste.

Ao fim do estudo, através da análise dos resultados, as autoras concluíram que as ferramentas de geração automática de casos de teste ainda precisam evoluir para que possam produzir suítes de testes equivalentes às manualmente escritas por desenvolvedores.

4 METODOLOGIA

Neste capítulo, é descrito o desenvolvimento do trabalho, estabelecendo objetivos principais da pesquisa, critérios para a seleção dos artefatos necessários para o projeto, tais como ferramentas de teste e programas.

Conforme descrito no capítulo de Introdução, o objetivo geral é a execução de um plano de identificação e investigação de ferramentas de geração de casos de teste *Open Source*, assim, criando parâmetros para facilitar a seleção da ferramenta mais adequada para determinados cenários/projetos de desenvolvimento de software, levando em consideração seus critérios.

Para o êxito do objetivo geral, os três objetivos específicos foram definidos e são retomados agora, são eles a identificação das ferramentas de teste *Open Source* utilizadas por profissionais da área de desenvolvimento, a aplicação das ferramentas em cenários reais de sistemas de desenvolvimento de software e a apresentação dos resultados comparativos entre as ferramentas de geração de casos de teste, em conformidade às suas respectivas documentações, e as métricas de cobertura de código e cobertura de mutação. A partir deles, são criadas métricas e atividades que auxiliarão no desenvolvimento dos estudos.

4.1 QUESTÕES DE PESQUISA

Com o intuito de contribuir com a área de teste de software, foram levantadas duas questões de pesquisa que foram respondidas ao final deste estudo. São elas:

- *Q1: Dentre as ferramentas de geração de casos de teste selecionadas, em relação à métrica de cobertura de código, qual ferramenta gera uma melhor suíte de teste?*
- *Q2: Dentre as ferramentas de geração de casos de teste selecionadas, em relação à métrica de mutação, qual ferramenta gera uma melhor suíte de teste?*

4.2 DESENVOLVIMENTO

Nesta seção é apresentado o ambiente no qual os experimentos da pesquisa foram realizados, as ferramentas utilizadas, os critérios utilizados para definir qual das ferramentas geram uma suíte de teste melhor, os projetos dos quais as suítes de testes foram geradas pelas ferramentas escolhidas e os recursos utilizados para obtenção dos resultados.

4.2.1 Ambiente de Experimentação

Para a realização dos experimentos, foi levado em consideração fatores que eram passíveis de causar impacto nos experimentos, como, por exemplo, a conFiguração dos computadores utilizados para a experimentação.

Foram utilizados dois computadores para a execução dos experimentos, e eles possuem as seguintes conFigurações:

Tabela 1: ConFigurações dos computadores utilizados

ConFigurações	Computador 1	Computador 2
Sistema Operacional	Windows 11	MacOS Monterey
Processador	Intel Core i7-10610U	Chip Apple M1
Memória Interna	SSD 256GB	SSD 256GB
Memória RAM	16 GB	8 GB
Versão do JDK	1.8.0	1.8.0

Fonte: Autoral.

Para a experimentação, o escopo foi sintetizado em ferramentas *Open Source* que gerem suítes de testes para sistemas codificados na linguagem Java, e que possibilitem a utilização de testes de mutação.

No início da elaboração do estudo foram selecionadas quatro ferramentas: Randoop, Evosuite, Daikon e Feed4jUnit. No entanto, com o aprofundamento da pesquisa sobre as ferramentas, foi possível perceber que o Feed4jUnit e o Daikon fogem do escopo do projeto, pois não são ferramentas de geração de suítes de teste, e, portanto, foram descartadas da pesquisa.

4.2.2 Definição da Experimentação

Com o intuito de alcançar o objetivo da presente pesquisa, ou seja, identificar as vantagens de adotar uma determinada ferramenta para geração de casos de teste, foram levadas em consideração duas características fundamentais, conforme segue:

- Cobertura de código;
- Critério de mutação.

As ferramentas foram escolhidas com base na sua vasta utilização para geração de casos de teste codificados na linguagem Java dentro da academia, e são elas:

- Evosuite: ferramenta que utiliza recursos de IA para geração dos casos de teste;
- Randoop: ferramenta que utiliza a aleatoriedade direcionada por um mecanismo de feedback para geração de casos de teste.

Com as ferramentas selecionadas, a seleção dos programas de domínios diversos para experimentação em conjunto com as ferramentas foi o próximo passo. A princípio, foram selecionados oito (8) sistemas de projetos *Open Source* reais do Apache, Mozilla e Microsoft que foram extraídos dos seus próprios repositórios de projetos abertos e do GitHub. A quantidade de programas foi definida visando uma primeira abordagem das ferramentas para definir se o estudo é aplicável para uma quantidade maior de sistemas.

Para validação dos critérios de teste funcionais, foi selecionada a ferramenta JUnit, pois com ela é possível validar as unidades dos programas selecionados para o estudo.

Tabela 2: Lista dos programas que foram utilizados no estudo

Id	Nome	Descrição	Disponível em:
1	Apache Commons Lang	O Apache Commons Lang é um pacote de classes utilitárias Java para as classes que estão na hierarquia do java.lang.	https://github.com/apache/commons-lang
2	Apache Commons Net	A biblioteca Apache Commons Net contém uma coleção de utilitários de rede e implementações de protocolo. Os protocolos suportados incluem: Echo, Finger, FTP, NNTP, NTP, POP3(S), SMTP(S), Telnet, Whois	https://github.com/apache/commons-net
3	JFR Streaming	O JFR Streaming fornece uma biblioteca	https://github.com/mi

		principal para conFigurar, iniciar, parar e ler arquivos Java Flight Recording de uma JVM.	crosoft/jfr-streaming
4	Apache Commons Cli	O Apache Commons CLI fornece uma API simples para apresentar, processar e validar uma interface de linha de comando.	https://github.com/apache/commons-cli
5	Apache Commons DbUtils	O pacote Apache Commons DbUtils é um conjunto de classes de utilitário Java para facilitar o desenvolvimento JDBC.	https://github.com/apache/commons-dbutils
6	Apache Commons Graph	O pacote Apache Commons Graph é um kit de ferramentas para gerenciar gráficos e estruturas de dados baseadas em gráficos.	https://github.com/apache/commons-graph
7	Microsoft Exploration Library	O objetivo principal é permitir que os indivíduos coletem os dados corretos para usar o aprendizado de máquina para intervenções em um sistema ativo com base no feedback do usuário (clique, pausa, correção etc.).	https://github.com/microsoft/mwt-ds-explore
8	Apache Sling Launchpad Comparator	Utilitário de linha de comando que ajuda a comparar os artefatos contidos por duas instâncias do launchpad/starter.	https://github.com/apache/sling-launchpad-comparator

Fonte: Autoral.

4.2.3 Recursos

Neste tópico, são descritas as ferramentas utilizadas para a realização do estudo e a utilização dos softwares para obtenção dos resultados.

Desenvolvido pelos professores Gordon Fraser e Andrea Arcuri, o Evosuite é uma ferramenta *Open Source* para geração automática de casos de teste com asserções para softwares codificados em Java. Segundo Fraser e Arcuri (2011), as suítes de testes são geradas utilizando uma abordagem híbrida que gera e otimiza as suítes para satisfazer um critério de cobertura. Para os conjuntos de testes produzidos, o Evosuite sugere possíveis oráculos, adicionando conjuntos pequenos e eficazes de asserções que resumem de forma concisa o comportamento atual. Essas asserções permitem que o desenvolvedor detecte desvios do comportamento esperado e capture o comportamento atual para se proteger contra defeitos futuros que interrompam esse comportamento. Além disso, de acordo com a documentação da ferramenta, a suíte de testes também é gerada utilizando a abordagem baseada em pesquisa, o que melhora significativamente a capacidade de satisfazer os critérios de cobertura selecionados.

O Randoop, segundo a sua própria documentação[4], é um gerador automático de testes de unidade para Java, no formato JUnit[5]. Para criar as suítes de testes, a ferramenta utiliza a técnica geração de teste aleatório direcionado por *feedback*. De acordo com Ernst e col. (2007), essa técnica consiste na geração aleatória das suítes de teste, utilizando o feedback obtido na execução à medida em que as entradas são criadas, evitando dessa forma a geração de entradas redundantes e ilegais. O Randoop trabalha somente via linha de comando, recebendo alguns parâmetros como um conjunto de classes, um tempo limite para geração dos testes e outros opcionais. A ferramenta gera dois conjuntos de teste, um que força a geração de erros e outro que não força e contém testes de regressão.

Para execução das suítes de testes citadas acima, foi selecionada a ferramenta JUnit a fim de validar os testes de unidade gerados pelos programas selecionados. O JUnit é um *framework* de código aberto utilizado para escrever e executar testes de unidade para a linguagem de programação Java. Atualmente é muito utilizado pois fornece anotações para a identificação de métodos de teste e asserções para testar os resultados esperados. Além disso, de acordo com a documentação do JUnit, é possível executar os testes na própria ferramenta, e

---

[4] Disponível em: https://randoop.github.io/randoop/. Acesso em 14 mai. 2022.
[5] Disponível em: https://junit.org/junit5/. Acesso em 14 mai. 2022.

ela fornece um *feedback* imediato através de uma barra verde, caso o teste tenha sido executado sem problemas, e uma barra vermelha caso tenha ocorrido falha em algum cenário de teste.

4.2.4 Plano de Experimentação

Para obter e mensurar o percentual de cobertura de código e a capacidade de identificar e eliminar mutações, serão utilizados os procedimentos a seguir explicitados.

Para a análise da métrica de cobertura de código dos softwares desenvolvidos em Java, utilizou-se a ferramenta EMMA. De acordo com a documentação, EMMA[6] é uma ferramenta *Open Source* usada para medir e relatar a cobertura de código Java, e pode instrumentar classes para cobertura *offline* (antes de serem carregadas) ou em tempo real (utilizando um *classloader*). O relatório de saída pode ser dos tipos: texto simples, HTML, XML, e destacando itens com níveis de cobertura abaixo dos limites fornecidos pelo usuário. Além disso, os dados presentes no relatório gerado pelo EMMA são calculados levando em consideração a quantidade de linhas de código que foram executadas durante a execução da aplicação, e sua análise é dividida em classes, métodos e linhas. As informações de cobertura de código ficam automaticamente disponíveis no ambiente de trabalho do ambiente de desenvolvimento. A ferramenta permite que lançamentos de dentro do ambiente de trabalho, como execuções de teste JUnit, possam ser analisadas diretamente para cobertura de código, e os resultados dessa cobertura são imediatamente resumidos e destacados nos editores de códigos-fonte Java.

Com relação à métrica de mutação, foi utilizada a ferramenta PIT *Mutation Testing*[7], que executa automaticamente os testes da unidade contra versões modificadas, pois quando o código de aplicação muda, deve ser produzido resultados diferentes, fazendo com que os testes da unidade falhem, e se um teste de unidade não falhar nesta situação, pode indicar um problema com a suíte de testes. Os relatórios produzidos pelo PIT estão em um formato de fácil leitura combinando cobertura de linha e informações de cobertura de mutação.

Os oito programas selecionados na etapa de definição da experimentação foram submetidos às ferramentas (Randoop e Evosuite) para a execução do processo de geração de

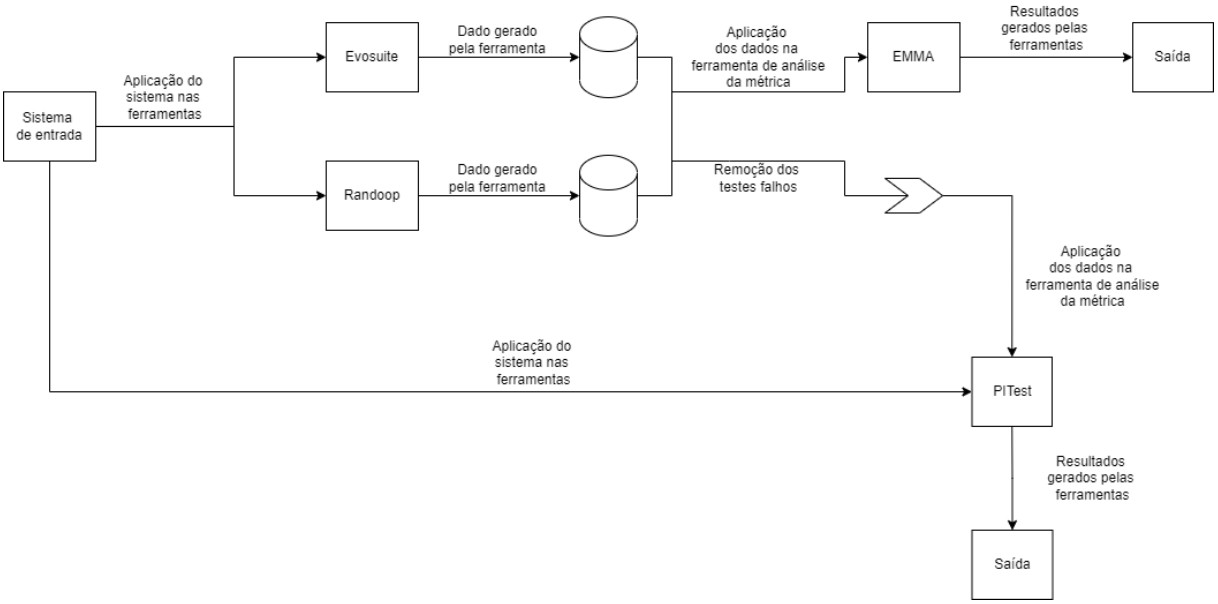
---

[6] Disponível em: http://emma.sourceforge.net/. Acesso em 14 mai. 2022.
[7] Disponível em: https://pitest.org/. Acesso em 23 mai. 2022.

casos de teste, de acordo com as abordagens definidas, sendo executados em dois computadores distintos com sistemas operacionais diferentes (MacOS e Windows).

Os dados obtidos foram coletados e aplicados aos procedimentos de análise das métricas de cobertura e mutação, para a análise e discussão dos resultados de cada ferramenta. A Figura 1 apresenta a esquematização da etapa de execução do experimento.

Figura 1: Esquema de execução do experimento



Fonte: Autoral.

5 RESULTADOS

No capítulo a seguir é apresentada uma descrição para as métricas de cobertura de código e mutação. Além disso, é discutido os resultados para cada questão de pesquisa de acordo com os dados coletados, indicando como os resultados da pesquisa respondem a cada uma delas.

5.1 MÉTRICAS

Com o intuito de analisar a qualidade dos casos de testes gerados pelas ferramentas, foram utilizadas as métricas de cobertura de código e mutação.

- Cobertura de código: neste estudo utilizou-se a cobertura de código baseada em fluxo de controle, ou seja, essa métrica irá medir o quantitativo de linhas do código que foram chamados durante a execução dos testes, em relação a quantidade de código não executado.
- Mutação: falhas (ou mutações) são automaticamente inseridas no código e, em seguida, seus testes são executados. Se seus testes falharem, a mutação é morta. Se seus testes passarem, a mutação sobreviveu. Na Tabela 5, pode-se ver os tipos de mutantes gerados nos códigos

Tabela 3: Operadores de mutação utilizados na ferramenta PIT

Mutante	Exemplo
Conditionals Boundary	1. Limite condicional alterado de i >= 1 para i < 1 2. Limite condicional alterado de i < 1 para i >= 1
Increments	1. Incremento alterado de -1 para 1 2. Incremento alterado de 1 para -1
Math	1. Subtração de inteiros substituída por adição 2. Substituição da adição de inteiros por subtração

Negate Conditionals	1. Condição negada de == para != 2. Condição negada de != para ==	
Void Method Calls	Antes	Depois
	public void voidMethod(int i) { }	public void voidMethod(int i) { }

	public int foo() {
	someVoidMethod(i);
	return i;
	}
		public int foo() {
		~~someVoidMethod(i);~~
		return i;
		}

Fonte: Autoral.

5.2 CONFIGURAÇÕES DE SIMULAÇÃO

Nas subseções a seguir, é descrito as características dos programas selecionados para o experimento, os passos seguidos para geração das suítes de testes e a coleta das métricas.

5.2.1 Seleção dos programas

Foram selecionados programas para compor nosso experimento com base nos seguintes requisitos:

- Projeto codificado na linguagem de programação Java
- O projeto é construído com Maven;
- É possível gerar casos de teste para o projeto usando o Randoop e Evosuite;
- É possível gerar mutantes do projeto usando o PIT

Foram selecionados softwares das organizações Apache e Mozilla que se adequam aos requisitos mencionados.

5.2.2 Simulação do experimento

Para a simulação do experimento, foram seguidos os passos:

- Foi instalado o *plugin* do Evosuite no ambiente de desenvolvimento Java da Jetbrains, o IntelliJ, e na sua execução foi utilizado o JDK da versão 1.8 do Java, definindo que o tempo de execução da ferramenta por cada classe seria de 1 minuto (tempo mínimo aceito pelo *plugin*);
- Para o Randoop, a ferramenta JMLOK2 (MILANEZ et. al, 2014) foi utilizada para gerar a linha de comando com todos os pacotes do projeto e então o Randoop foi executado no terminal para gerar os casos de teste;
- Em relação à métrica de cobertura, foi utilizado o *plugin* do Emma, que vem por padrão no IntelliJ, para obter a cobertura dos cenários de teste gerados em ambas as ferramentas;
- Com o intuito de coletar a métrica de mutação, a ferramenta PIT precisa ter uma *green suite test* para ser executada, ou seja, apenas os testes que passaram. Por isso, foi necessário a criação de um *script* para retirar os casos de teste que falharam para ser possível coletar essa métrica.

5.3 ANÁLISE DOS RESULTADOS

Os dados coletados dos resultados foram armazenados em tabelas, gradualmente incrementadas conforme a execução dos experimentos. Os primeiros dados obtidos são referentes à métrica de cobertura nos testes unitários gerados pelo Evosuite (Tabela 4) e pelo Randoop (Tabela 5), seguidos da métrica de mutação aplicada aos projetos contendo os testes unitários (Tabelas 6 e 7).

Tabela 4: Métrica de cobertura para os testes gerados pelo Evosuite

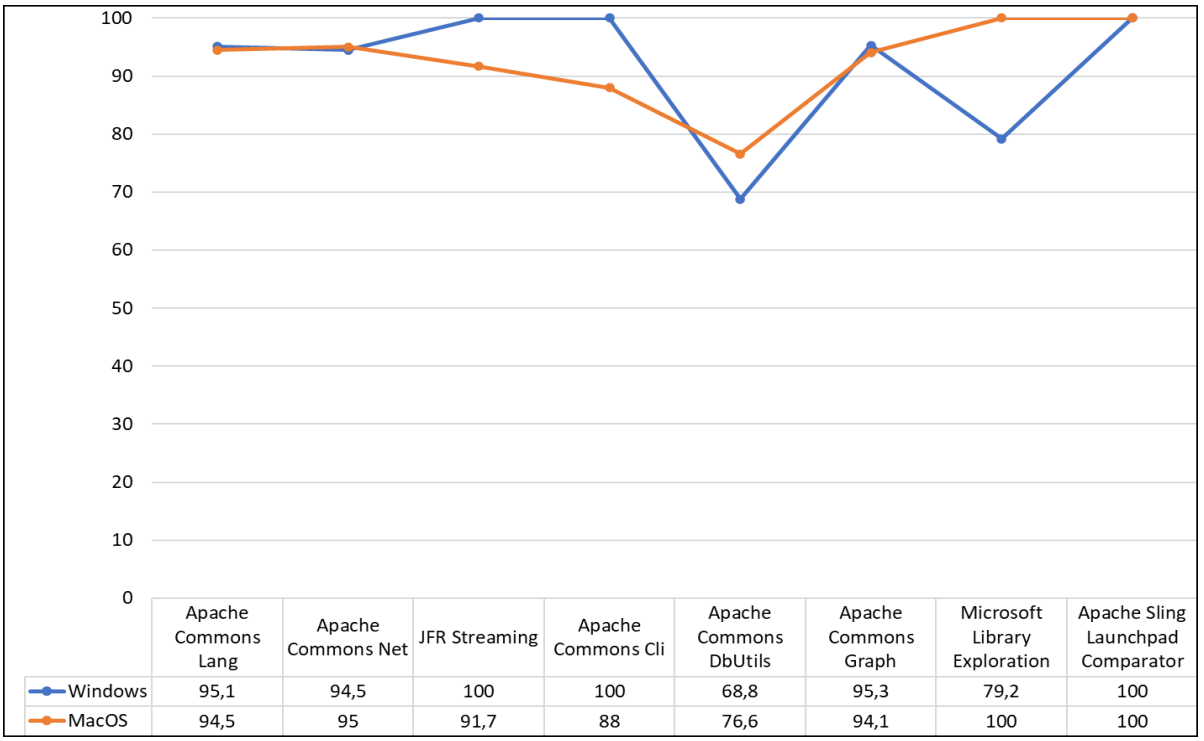
Métrica de Cobertura - EvoSuite						
Projeto	Windows			MacOS		
	Classes	Métodos	Linhas	Classes	Métodos	Linhas
Apache Commons Lang	95,10%	87,80%	77,80%	94,50%	70,80%	55,80%
Apache Commons Net	94,50%	81,40%	56,90%	95%	86,60%	57,50%
JFR Streaming	100%	98,70%	80%	91,70%	73,10%	52%
Apache Commons Cli	100%	91,10%	68,40%	88%	75,10%	60,80%
Apache Commons DbUtils	68,80%	47%	46,20%	76,60%	44,90%	43,20%
Apache Commons Graph	95,30%	91,60%	78,90%	94,10%	91,90%	81,10%
Microsoft Exploration Library	79,20%	76%	86,90%	100%	93,30%	79,90%
Apache Sling Launchpad Comparator	100%	79%	51%	100%	76,10%	47,90%
Média	91,76%	81,93%	64,46	92,12%	75,19%	59,19%

Fonte: Autoral.

De acordo com os resultados apresentados acima para a métrica de cobertura dos testes gerados automaticamente pela ferramenta Evosuite, é possível observar que o MacOs teve uma maior porcentagem cobertura de classes em relação ao Windows, no entanto, o Windows obteve melhores resultados na cobertura de métodos e linhas. As Figuras 2, 3 e 4 mostram visualmente os resultados, com os valores para o Windows em azul, e para o MacOs em laranja.

Para a cobertura de classes, a média de todos os projetos foi de 91,76% no Windows, com o menor resultado para o projeto Apache DbUtils, que cobriu 68.8%. Os melhores resultados são para os projetos JFR Streaming, Apache Commons Cli e Apache Sling Launchpad Comparator, atingindo os 100% de cobertura. Assim como no Windows, o MacOs teve seu menor resultado para o projeto Apache DbUtils, contudo com uma porcentagem maior, de 76,6%. Apenas o Microsoft Exploration Library e o Apache Sling Launchpad Comparator atingiram uma cobertura de 100%, totalizando dois dos oito projetos. A média de cobertura de todos os projetos testados no MacOs foi de 92,12%. É possível notar que tanto o Windows quanto o MacOs apresentam valores semelhantes para o Apache Commons Lang, Apache Commons Net, Apache Commons Graph e Apache Sling Launchpad Comparator.

Figura 2: Cobertura de classes para os testes do Evosuite

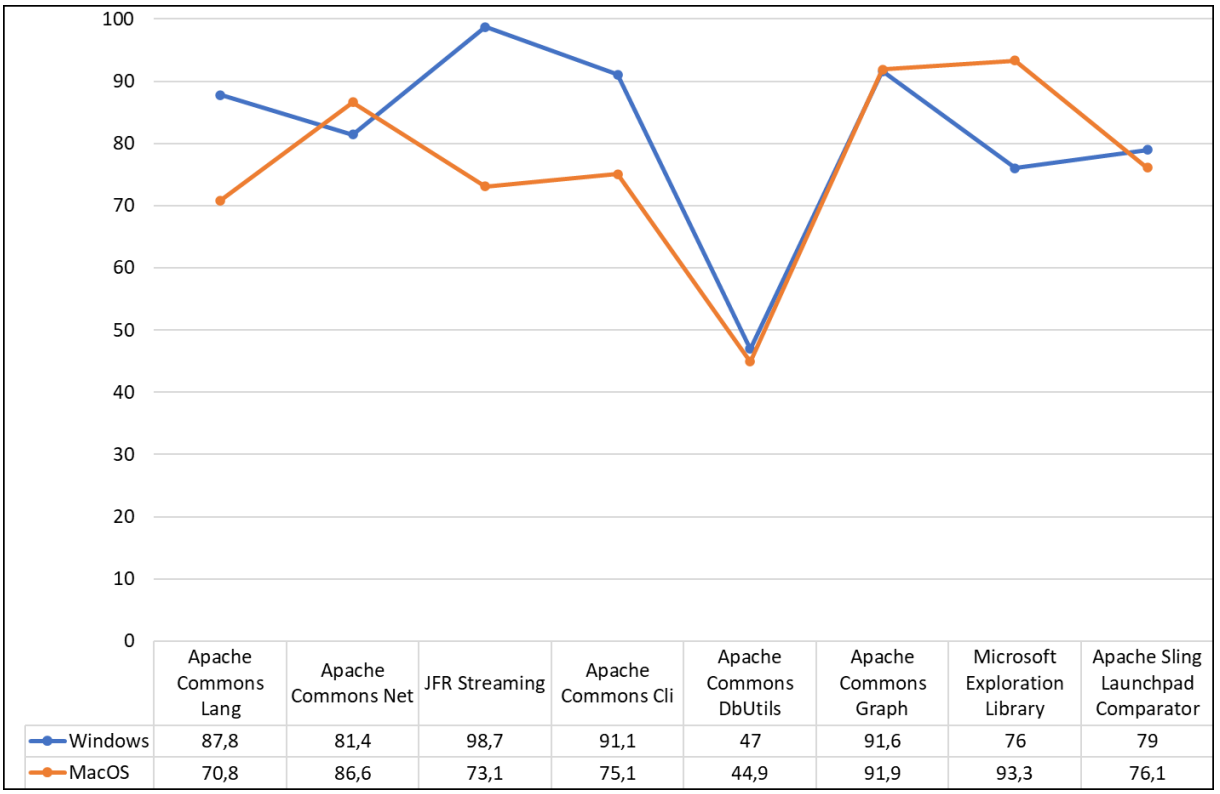


	Apache Commons Lang	Apache Commons Net	JFR Streaming	Apache Commons Cli	Apache Commons DbUtils	Apache Commons Graph	Microsoft Library Exploration	Apache Sling Launchpad Comparator
Windows	95,1	94,5	100	100	68,8	95,3	79,2	100
MacOS	94,5	95	91,7	88	76,6	94,1	100	100

Fonte: Autoral.

Para a cobertura de métodos, a média de todos os projetos foi de 81,93% no Windows, apresentando uma diminuição de 9,83% em relação à cobertura de classe. O menor resultado se mantém com o projeto Apache DbUtils, que cobriu 47%, tendo 21,8% a menos que a cobertura de classe. O melhor resultado é para os projetos JFR Streaming, atingindo os 98,7% de cobertura. Assim como no Windows, o MacOs teve seu pior resultado para o projeto Apache DbUtils, com 44,9%. O melhor resultado é do projeto Microsoft Exploration Library, que atingiu uma cobertura de 93,3%. A média de cobertura de todos os projetos testados no MacOs foi de 75,19%. Nesse contexto, é possível notar que o Windows e o MacOs apresentam valores semelhantes para o Apache DbUtils, Apache Commons Graph e Apache Sling Launchpad Comparator.

Figura 3: Cobertura de métodos para os testes do Evosuite

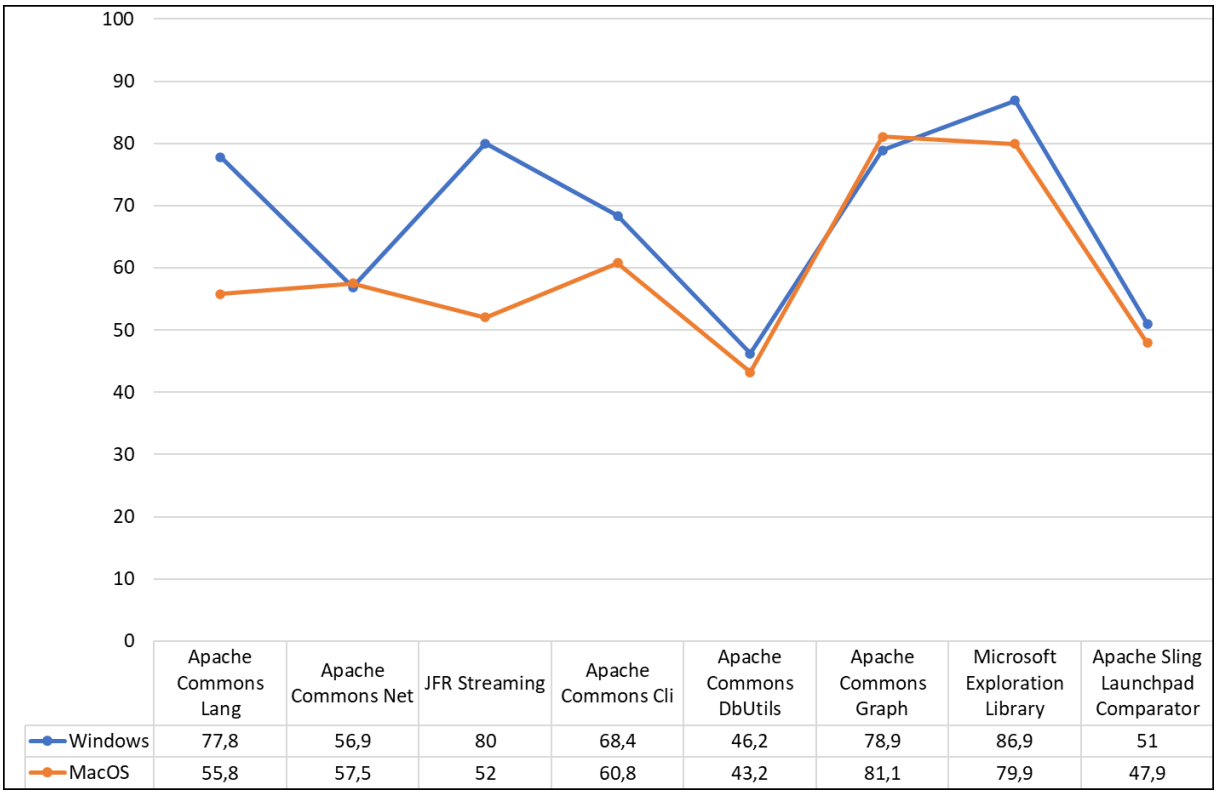


	Apache Commons Lang	Apache Commons Net	JFR Streaming	Apache Commons Cli	Apache Commons DbUtils	Apache Commons Graph	Microsoft Exploration Library	Apache Sling Launchpad Comparator
Windows	87,8	81,4	98,7	91,1	47	91,6	76	79
MacOS	70,8	86,6	73,1	75,1	44,9	91,9	93,3	76,1

Fonte: Autoral.

Na Figura 4 é exibido os resultados para a cobertura de linhas. No Windows, a média de todos os projetos foi de 64,46%, com uma diminuição de 17,47% em relação à cobertura de métodos. Novamente, o menor resultado é o projeto Apache DbUtils, que cobriu 46,2%, tendo 0,8% a menos que o seu valor para a cobertura de método. O melhor resultado é para o projeto Microsoft Exploration Library, atingindo os 86,9% de cobertura. Assim como no Windows, o MacOs teve seu menor resultado para o projeto Apache DbUtils, com 43,2%. O melhor resultado é do projeto Apache Commons Graph, que atingiu uma cobertura de 81,1%, e a média de cobertura de todos os projetos testados no MacOs foi de 59,19%. Nesse contexto, é possível notar que o Windows e o MacOs apresentam valores semelhantes para o Apache Commons Net, Apache DbUtils, Apache Commons Graph e o Apache Sling Launchpad Comparator.

Figura 4: Cobertura de linhas para os testes do Evosuite



Fonte: Autoral.

Os resultados da Tabela 5 abaixo se referem aos dados obtidos para a cobertura de classes, métodos e linhas dos testes gerados pela ferramenta Randoop para os oito projetos estudados nos sistemas Windows e MacOs.

Tabela 5: Métrica de cobertura para os testes gerados pelo Randoop

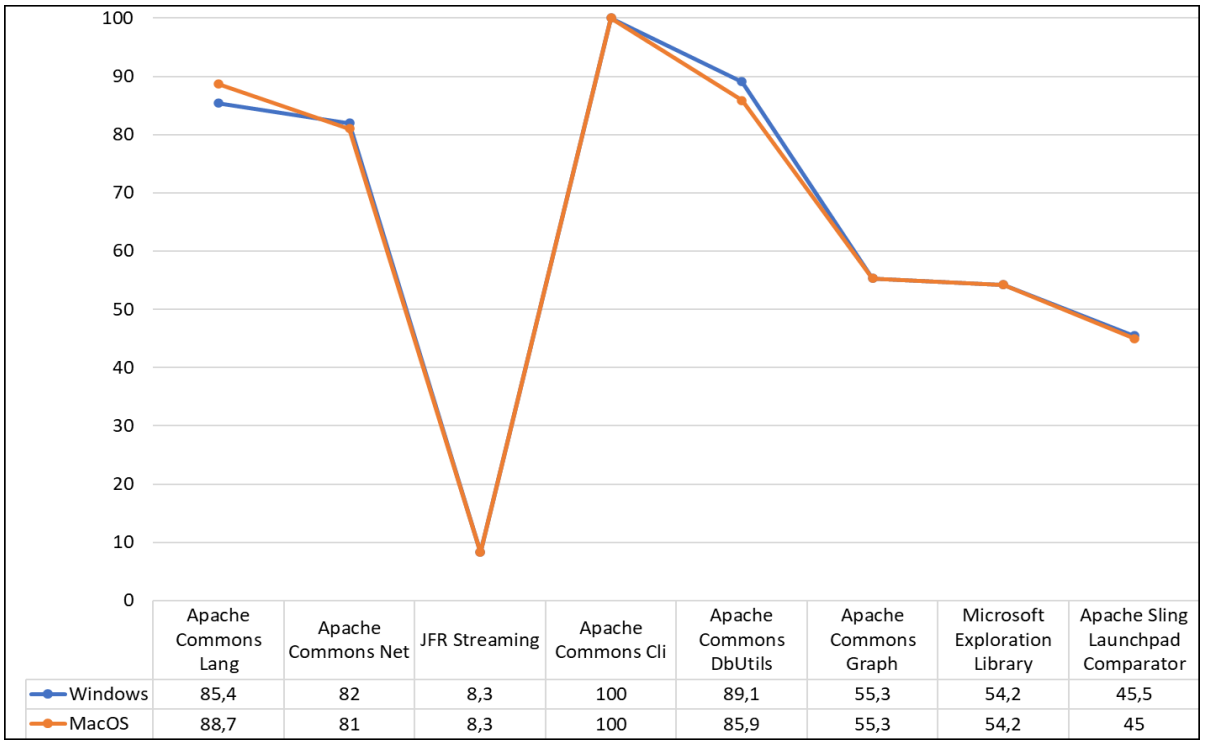
Métrica de Cobertura - Randoop						
Projeto	Windows			MacOS		
	Classes	Métodos	Linhas	Classes	Métodos	Linhas
Apache Commons Lang	85,40%	79,70%	64,10%	88,70%	81,30%	68,60%
Apache Commons Net	82%	75,50%	36,30%	81%	73,50%	35%
JFR Streaming	8,30%	2,40%	0,60%	8,30%	2,41%	0,70%
Apache Commons Cli	100%	89,40%	65,80%	100%	89,40%	65,70%
Apache Commons DbUtils	89,10%	52,30%	51,30%	85,90%	46,20%	33,30%
Apache Commons Graph	55,30%	32,10%	18,50%	55,30%	35,60%	20%
Microsoft Exploration Library	54,20%	59,50%	43%	54,20%	59,50%	43%
Apache Sling Launchpad Comparator	45,50%	19,70%	9%	45%	19,70%	9%
Média	64,98%	51,33%	36,08%	64,08%	50,95%	34,41%

Fonte: Autoral.

Ao analisar os resultados apresentados acima para a métrica de cobertura dos testes gerados automaticamente pela ferramenta Randoop, é possível observar que o Windows obteve mais bem resultados na cobertura de classes, métodos e linhas, apesar da pouca diferença em relação à média percentual dos casos de teste gerados no MacOs. As Figuras 5, 6 e 7 mostram visualmente os resultados.

Em relação à cobertura de classes, as suítes de teste dos dois sistemas operacionais apresentaram percentuais de cobertura parecidos, destacando-se as porcentagens iguais de 8,3% no JFR Streaming, que foi o projeto com menor percentual de cobertura, 55,3% no Apache Commons Graph, 54,2% no Microsoft Exploration Library e 100% de cobertura no projeto Apache Commons Cli. A média de cobertura de todos os projetos testados no MacOs foi de 64,08% e no Windows 64,98%, ou seja, uma diferença percentual de 0,9%. É possível notar que tanto o Windows quanto o MacOs apresentam valores semelhantes para o Apache Commons Lang, Apache Commons Net, e Apache Sling Launchpad Comparator.

Figura 5: Cobertura de classes para os testes do Randoop



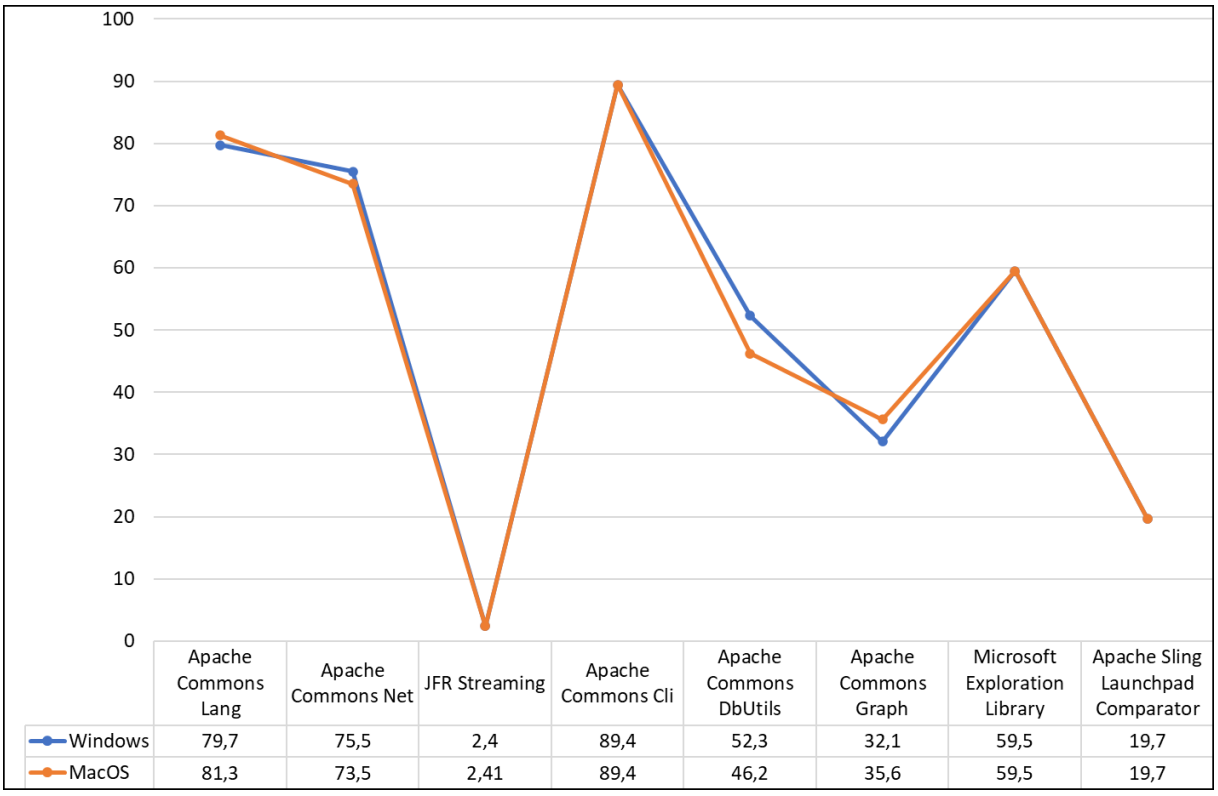
	Apache Commons Lang	Apache Commons Net	JFR Streaming	Apache Commons Cli	Apache Commons DbUtils	Apache Commons Graph	Microsoft Exploration Library	Apache Sling Launchpad Comparator
Windows	85,4	82	8,3	100	89,1	55,3	54,2	45,5
MacOS	88,7	81	8,3	100	85,9	55,3	54,2	45

Fonte: Autoral.

Com base na cobertura de métodos, a média de todos os projetos foi de 51,33% no Windows, apresentando uma diminuição de 13,65% em relação à cobertura de classe. O menor resultado se mantém com o projeto JFR Streaming, que cobriu 2,4%, obtendo 5,9% a menos

que a cobertura de classe. O melhor resultado também se mantém no projeto Apache Commons Cli, atingindo os 89,4% de cobertura. Assim como no Windows, o MacOs teve seu menor resultado para o projeto JFR Streaming, com 2,41%, e o melhor resultado também com o mesmo percentual de cobertura no projeto Apache Commons Cli. A média de cobertura de todos os projetos testados no MacOs foi de 50,95% e nesse contexto, é possível notar que o Windows e o MacOs apresentam valores semelhantes de cobertura para os projetos Apache Commons Lang e Apache Commons Net. Em relação ao Apache Sling Launchpad Comparator e Microsoft Exploration Library, ambos os sistemas operacionais apresentaram o mesmo percentual de cobertura.

Figura 6: Cobertura de métodos para os testes do Randoop



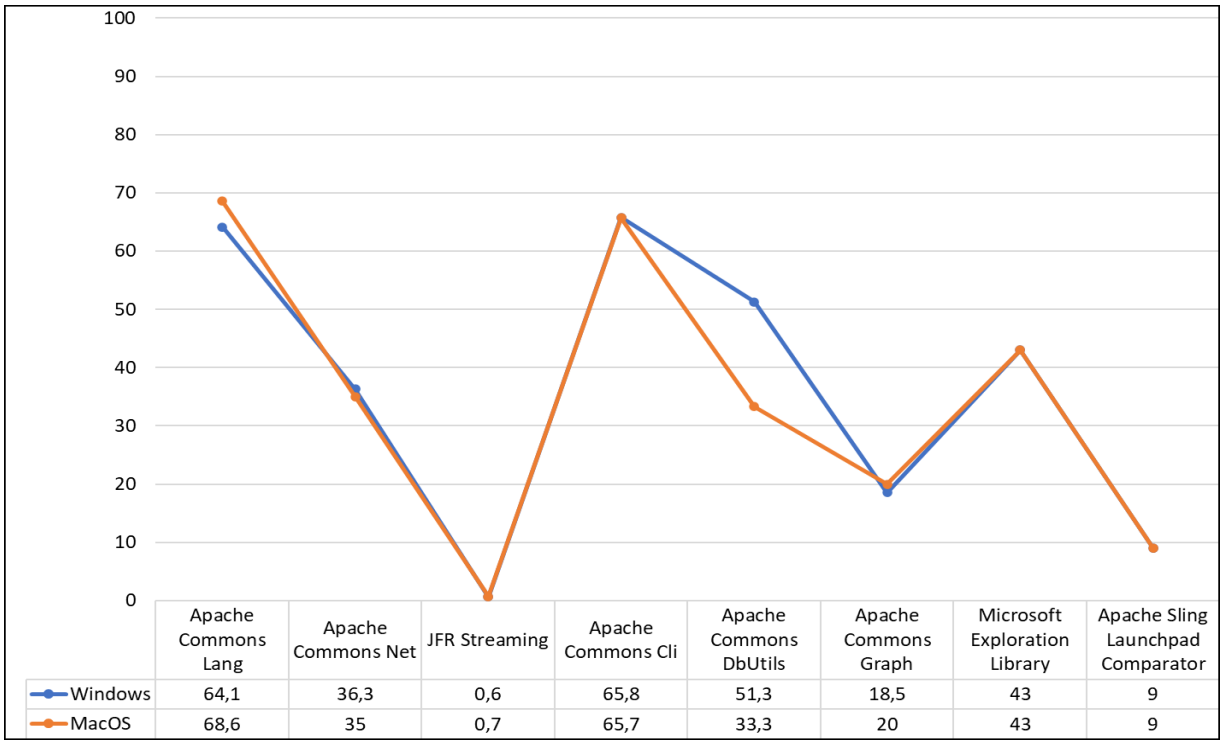
	Apache Commons Lang	Apache Commons Net	JFR Streaming	Apache Commons Cli	Apache Commons DbUtils	Apache Commons Graph	Microsoft Exploration Library	Apache Sling Launchpad Comparator
Windows	79,7	75,5	2,4	89,4	52,3	32,1	59,5	19,7
MacOS	81,3	73,5	2,41	89,4	46,2	35,6	59,5	19,7

Fonte: Autoral.

Na Figura 7 é exibido os resultados para a cobertura de linhas. No Windows, a média de todos os projetos foi de 36,08%, com uma diminuição de 15,25% em relação à cobertura de métodos. Novamente, o menor resultado é o projeto JFR Streaming, que cobriu 0.6%, tendo 1,8% a menos que o seu valor para a cobertura de método. O melhor resultado é para o projeto Apache Commons Cli, atingindo os 65,8% de cobertura. Assim como no Windows, o MacOs teve seu menor resultado para o projeto JFR Streaming, com 0,7%. O melhor resultado é do

projeto Apache Commons Lang, que atingiu uma cobertura de 68,6%, e a média de cobertura de todos os projetos testados no MacOs foi de 34,41%. Nesse contexto, é possível notar que o Windows e o MacOs apresentam valores semelhantes para o Apache Commons Net, Apache Commons Cli e Apache Commons Graph. Em relação ao Apache Sling Launchpad Comparator e Microsoft Exploration Library, ambos os sistemas operacionais apresentaram o mesmo percentual de cobertura de linhas.

Figura 7: Cobertura de linhas para os testes do Randoop



	Apache Commons Lang	Apache Commons Net	JFR Streaming	Apache Commons Cli	Apache Commons DbUtils	Apache Commons Graph	Microsoft Exploration Library	Apache Sling Launchpad Comparator
Windows	64,1	36,3	0,6	65,8	51,3	18,5	43	9
MacOS	68,6	35	0,7	65,7	33,3	20	43	9

Fonte: Autoral.

A divergência nos números da cobertura de classes, métodos e linhas pode ser justificada pela especificidade de cada entidade. Em valores, uma linha coberta de um método significa que tanto o método quanto a classe desse método foram cobertos, mas não o contrário.

As tabelas a seguir apresentam a quantidade de mutantes dos tipos *Conditionals Boundary, Increments, Math, Negate Conditionals e Void Method Calls,* gerados e mortos para os oito projetos, contendo os testes unitários gerados pelo Evosuite (Tabela 6) e os testes gerados pelo Randoop (Tabela 7).

Tabela 6: Métrica de mutação para os testes do Evosuite

Métrica de Mutação - EvoSuite					
Projeto	Mutante	Windows		MacOS	
		Quant. De Mutantes Gerados	Mutantes Mortos	Quant. De Mutantes Gerados	Mutantes Mortos
Apache Commons Lang	CONDITIONALS_BOUNDARY	1337	20(1%)	1337	27(2%)
	INCREMENTS	436	10(2%)	436	1(0%)
	MATH	1592	47(3%)	1592	16(1%)
	NEGATE_CONDITIONALS	4747	269(6%)	4747	24(1%)
	VOID_METHOD_CALLS	1065	90(8%)	1065	5((0%)
Apache Commons Net	CONDITIONALS_BOUNDARY	330	24 (7%)	330	5(2%)
	INCREMENTS	143	5(3%)	143	13(9%)
	MATH	552	29 (6%)	522	11(2%)
	NEGATE_CONDITIONALS	1784	92(5%)	1784	28(2%)
	VOID_METHOD_CALLS	1602	0(0%)	1602	16(1%)
JFR Streaming	CONDITIONALS_BOUNDARY	4	0 (0%)	4	0(0%)
	INCREMENTS	N/A	N/A	N/A	N/A
	MATH	3	0 (0%)	3	0(0%)
	NEGATE_CONDITIONALS	45	0 (0%)	45	0(0%)
	VOID_METHOD_CALLS	7	0 (0%)	7	0 (0%)
Apache Commons Cli	CONDITIONALS_BOUNDARY	31	20 (63%)	31	0(0%)
	INCREMENTS	8	8 (100%)	8	0(0%)
	MATH	25	24 (96%)	25	0(0%)
	NEGATE_CONDITIONALS	340	326(96%)	340	0(0%)
	VOID_METHOD_CALLS	116	102(88%)	116	0(0%)
Apache Commons DbUtils	CONDITIONALS_BOUNDARY	14	0(0%)	14	0(0%)
	INCREMENTS	6	0(0%)	6	0(0%)
	MATH	8	0(0%)	8	0(0%)
	NEGATE_CONDITIONALS	206	15(7%)	206	0(0%)
	VOID_METHOD_CALLS	269	3(1%)	269	0(0%)
Apache Commons Graph	CONDITIONALS_BOUNDARY	30	2 (7%)	30	0(0%)
	INCREMENTS	10	0 (0%)	10	0(0%)
	MATH	41	0 (0%)	41	0(0%)
	NEGATE_CONDITIONALS	275	7 (3%)	275	0(0%)
	VOID_METHOD_CALLS	226	0(0%)	226	0(0%)
Microsoft Exploration Library	CONDITIONALS_BOUNDARY	28	0(0%)	28	0(0%)
	INCREMENTS	9	0(0%)	9	0(0%)
	MATH	91	0(0%)	91	0(0%)
	NEGATE_CONDITIONALS	48	0(0%)	48	0(0%)
	VOID_METHOD_CALLS	6	0(0%)	6	0(0%)
Apache Sling Launchpad Comparator	CONDITIONALS_BOUNDARY	2	0(0%)	2	0(0%)
	INCREMENTS	N/A	N/A	N/A	N/A
	MATH	N/A	N/A	N/A	N/A
	NEGATE_CONDITIONALS	22	1(5%)	22	5(23%)
	VOID_METHOD_CALLS	15	0(0%)	15	0(0%)
Média	-	417.38	13,71%	417.38	1.16%

Fonte: Autoral.

Tabela 7: Métrica de mutação para os testes do Randoop

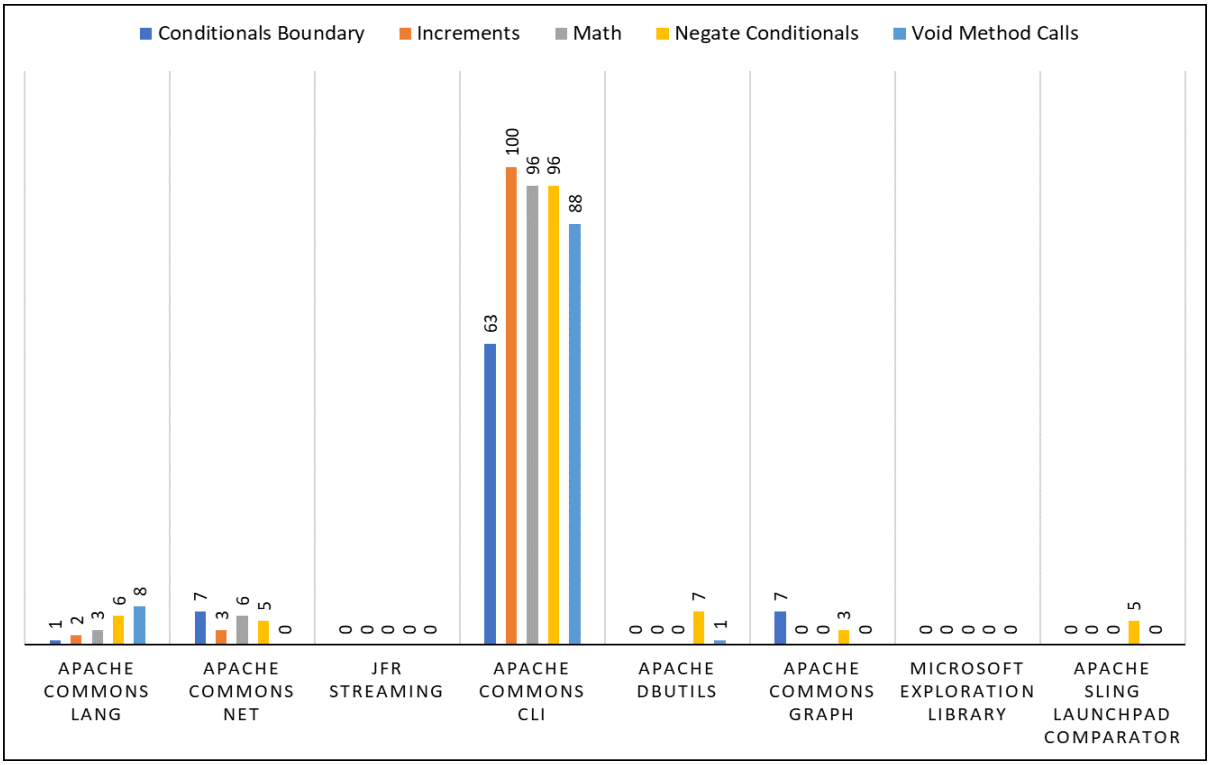
Métrica de Mutação - Randoop					
Projeto	Mutante	Windows		MacOS	
		Quant. De Mutantes Gerados	Mutantes Mortos	Quant. De Mutantes Gerados	Mutantes Mortos
Apache Commons Lang	CONDITIONALS_BOUNDARY	1337	285(21%)	1337	229(17%)
	INCREMENTS	436	191(44%)	436	160(37%)
	MATH	1592	447(28%)	1592	308(19%)
	NEGATE_CONDITIONALS	4747	1958(41%)	4747	1599(34%)
	VOID_METHOD_CALLS	1065	147(14%)	1065	99(9%)
Apache Commons Net	CONDITIONALS_BOUNDARY	330	24(7%)	330	22(7%)
	INCREMENTS	143	14(10%)	143	14(10%)
	MATH	522	156(30%)	522	189(36%)
	NEGATE_CONDITIONALS	1784	345(19%)	1784	351(20%)
	VOID_METHOD_CALLS	1602	15(1%)	1602	104(6%)
JFR Streaming	CONDITIONALS_BOUNDARY	4	0(0%)	4	0(0%)
	INCREMENTS	N/A	N/A	N/A	N/A
	MATH	3	0(0%)	3	0(0%)
	NEGATE_CONDITIONALS	45	0(0%)	45	0(0%)
	VOID_METHOD_CALLS	7	0(0%)	7	0(0%)
Apache Commons Cli	CONDITIONALS_BOUNDARY	31	9(29%)	31	9(29%)
	INCREMENTS	8	5(63%)	8	5(63%)
	MATH	25	7(28%)	25	7(28%)
	NEGATE_CONDITIONALS	340	106(31%)	340	112(33%)
	VOID_METHOD_CALLS	116	14(12%)	116	15(13%)
Apache Commons DbUtils	CONDITIONALS_BOUNDARY	14	2(14%)	14	2(14%)
	INCREMENTS	6	0(0%)	6	0(0%)
	MATH	8	0(0%)	8	0(0%)
	NEGATE_CONDITIONALS	206	35(17%)	206	35(17%)
	VOID_METHOD_CALLS	269	4(1%)	269	4(1%)
Apache Commons Graph	CONDITIONALS_BOUNDARY	30	1(3%)	30	1(3%)
	INCREMENTS	10	0(0%)	10	0(0%)
	MATH	41	2(5%)	41	2(5%)
	NEGATE_CONDITIONALS	275	21(8%)	275	21(8%)
	VOID_METHOD_CALLS	226	8(4%)	226	8(4%)
Microsoft Exploration Library	CONDITIONALS_BOUNDARY	28	9(32%)	28	6(21%)
	INCREMENTS	9	1(11%)	9	1(11%)
	MATH	91	65(71%)	91	61(67%)
	NEGATE_CONDITIONALS	48	1(0%)	48	14(29%)
	VOID_METHOD_CALLS	6	1(17%)	6	0(0%)
Apache Sling Launchpad Comparator	CONDITIONALS_BOUNDARY	2	0(0%)	2	0(0%)
	INCREMENTS	N/A	N/A	N/A	N/A
	MATH	N/A	N/A	N/A	N//A
	NEGATE_CONDITIONALS	22	1(5%)	22	1(5%)
	VOID_METHOD_CALLS	15	0(0%)	15	0(0%)
Média	-	417.38	15.29%	417.38	14.75%

Fonte: Autoral.

Com exceção do projeto Apache Commons Cli no Windows e dos mutantes do tipo *Negate Conditionals* no projeto Apache Sling Launchpad Comparator no MacOS, os resultados dos mutantes mortos pelos testes do Evosuite executados no Windows e no MacOS são irrisórios se compararmos aos valores do Randoop nos dois sistemas operacionais, que captura essas mutações para a maioria dos projetos investigados.

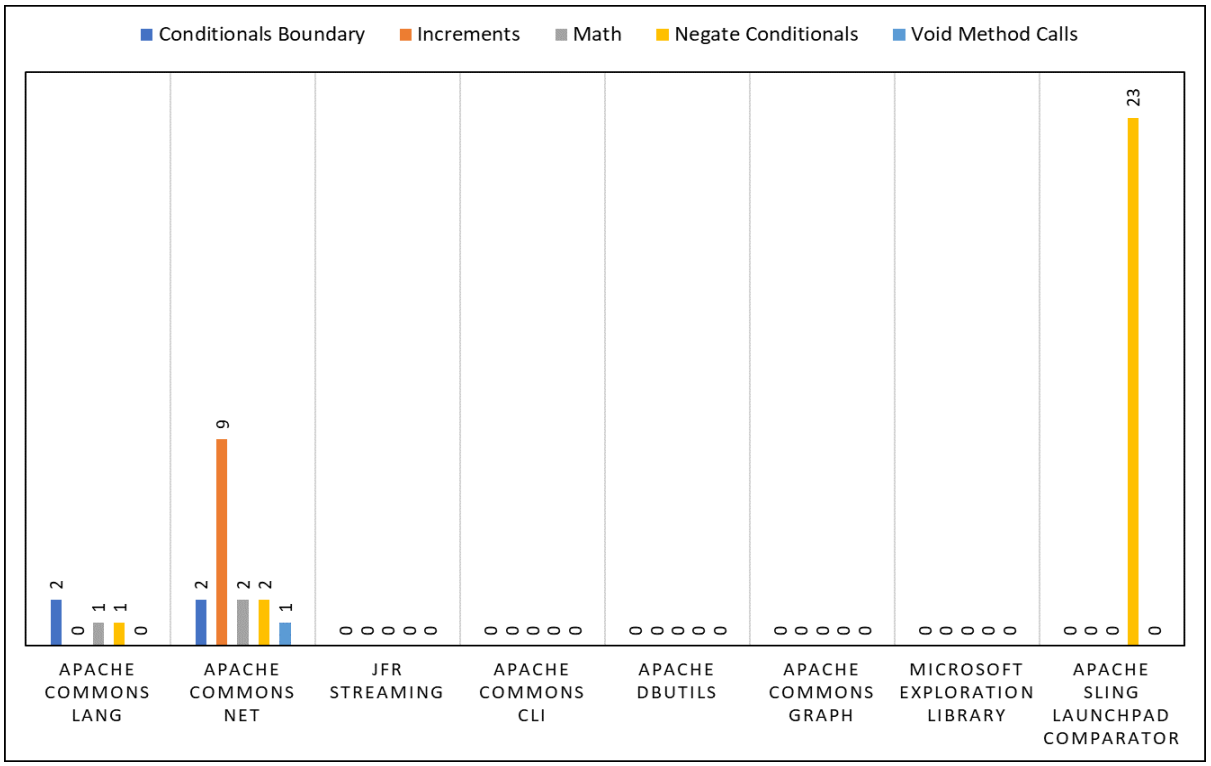
Adiante, nas Figuras 8, 9, 10 e 11, as análises podem ser constatadas nos gráficos de comparação entre os sistemas operacionais e as ferramentas Evosuite e Randoop.

Figura 8: Mutantes mortos com os testes do Evosuite no Windows



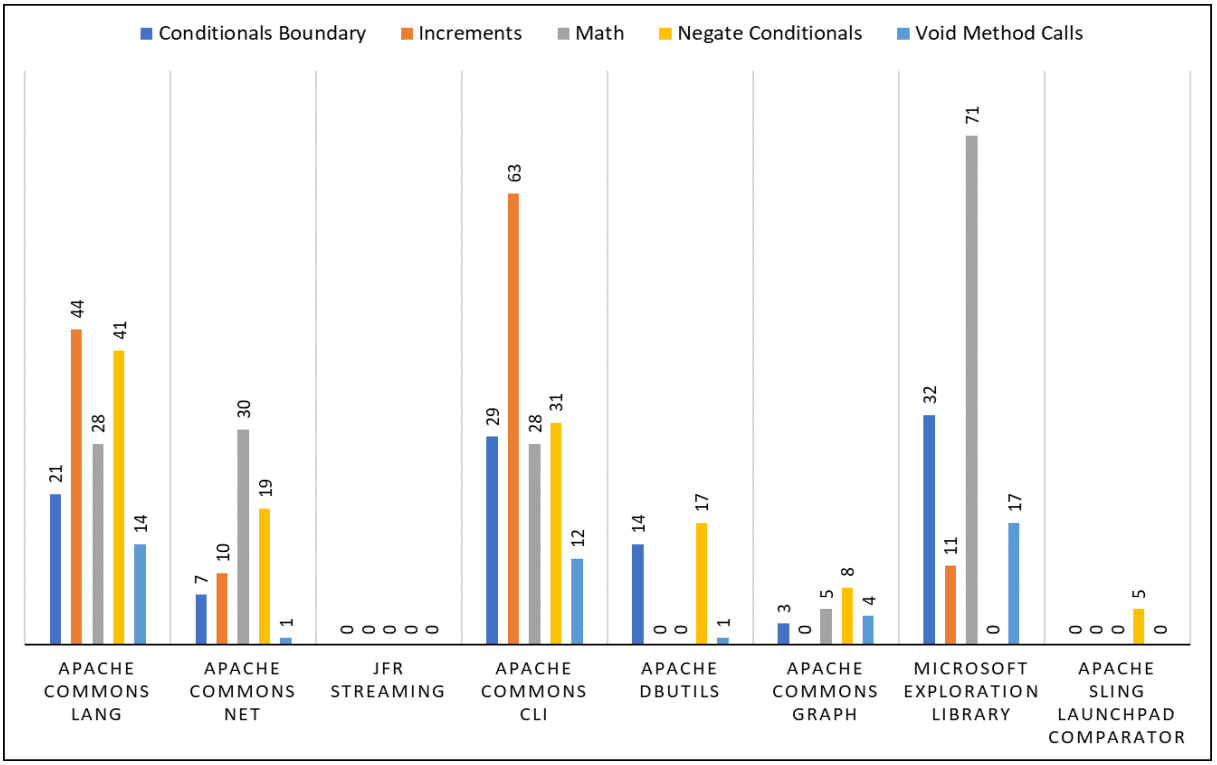
Fonte: Autoral.

Figura 9: Mutantes mortos com os testes do Evosuite no MacOS



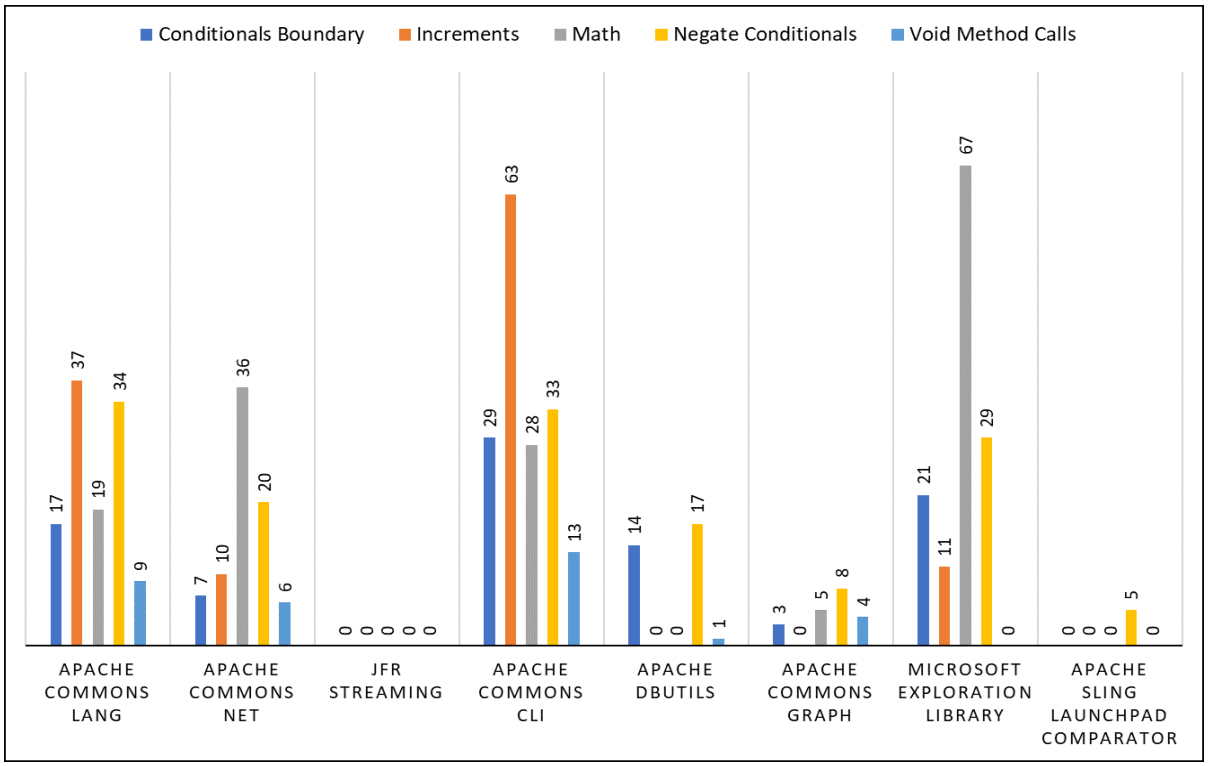
Fonte: Autoral.

Figura 10: Mutantes mortos com os testes do Randoop no Windows



Fonte: Autoral.

Figura 11: Mutantes mortos com os testes do Randoop no MacOS



Fonte: Autoral.

Após a análise dos dados expostos nas Tabelas 4, 5, 6 e 7, e nas Figuras 2, 3, 4, 5, 6,

7, 8, 9, 10 e 11, é possível responder às questões de pesquisa:

- RQ1: Em relação a métrica de cobertura, considerando a cobertura de linha que nos traz uma maior precisão em relação ao percentual de cobertura do código, as suítes de testes geradas pelo Evosuite obtiveram um percentual maior comparado às suítes de teste geradas pelo Randoop nos dois sistemas operacionais.
- RQ2: Se tratando da métrica de mutação, as suítes de testes geradas pelo Randoop mataram uma maior quantidade de mutantes quando comparado ao percentual de mutantes mortos nas suítes de teste geradas pelo Evosuite.

Através dos dados apresentados, que fundamentam as respostas das perguntas da pesquisa, é possível perceber que apesar da ferramenta Evosuite gerar casos de teste com um maior percentual de cobertura, quando é feita a análise do percentual de mutantes mortos nas suítes de teste geradas, é possível perceber que apesar da boa cobertura, o caso de teste pode ser muito simples (simplista) e em decorrência disso, essa cobertura boa não tem impacto de fato para quem está testando, pois é de extrema importância a capacidade do caso de teste de detectar modificações a fim de garantir a qualidade do software.

Por fim, através dos resultados coletados, é possível inferir que se o objetivo do projeto desenvolvido é ter uma suíte de teste que contenha uma maior cobertura de código, o Evosuite se destaca em relação ao Randoop por possuir um tempo fixo para gerar as suítes de teste de cada classe do projeto. No entanto, se o objetivo do projeto é ter uma suíte de teste capaz de detectar uma maior quantidade de modificações no código, ou seja, matar mutantes, o Randoop se destaca em relação ao Evosuite.

5.4 AMEAÇAS À VALIDADE

A pesquisa tem como objetivo avaliar os testes unitários gerados automaticamente pelas ferramentas Evosuite e Randoop, com base nas métricas de cobertura de código e mutação, para um conjunto de projetos *Open Source*, a fim de facilitar a escolha da ferramenta mais adequada de acordo com as necessidades do usuário.

Alguns fatores são considerados risco à validade dos experimentos. Dentre eles, pode-se citar a inexperiência do usuário das ferramentas utilizadas, a utilização de outras métricas quantitativas e qualitativas, como métrica de tempo, mas não foi possível aplicar devido à diferente forma que as ferramentas são executadas. O Evosuite e o Randoop foram selecionados como ferramentas de geração de casos de teste por serem as mais utilizadas nesse âmbito, mas também é possível utilizar um maior número de ferramentas, e em relação aos testes gerados,

não é possível afirmar que as ferramentas não geraram falsos positivos, ameaçando os resultados delas**.**

Outro aspecto importante a ser mencionado é a inviabilização de outras ferramentas de geração de casos de teste no decorrer do experimento, que não estavam em concordância com as premissas do experimento.

6 CONSIDERAÇÕES FINAIS

Este trabalho propôs uma análise da qualidade das suítes de teste geradas automaticamente por ferramentas de geração de casos de teste nos sistemas operacionais Windows e MacOs. Ao longo da pesquisa foram encontradas diversas ferramentas, dentre elas pode-se citar MuJava[8], Jumble[9], Feed4jUnit[10] e Daikon[11] que geram suítes de teste, no entanto não cumprem com os pré-requisitos necessários da pesquisa e foram descartadas do estudo.

Além disso, é importante citar a ferramenta TackleTest[12], lançada recentemente, que cumpre os pré-requisitos do estudo, contudo, não houve tempo hábil para incluí-la na pesquisa. Por isso, o presente estudo concentrou-se em analisar os casos de teste gerados pelas ferramentas Evosuite e Randoop, com o intuito de investigar qual dessas ferramentas gera uma melhor suíte de teste de acordo com o sistema operacional, com base nas métricas de cobertura e mutação.

Como resultado do estudo, baseado na métrica de cobertura, o Evosuite em geral, por ter um tempo fixo para gerar a suíte de teste de cada classe, nos dois sistemas operacionais, gerou casos de teste que cobrem uma porcentagem maior de linha de código do que a cobertura dos casos de teste gerados pelo Randoop. Levando em consideração o sistema operacional no qual a suíte de teste foi gerada, a ferramenta Evosuite teve um desempenho melhor no Windows do que no MacOS, ou seja, gerou uma suíte de teste com maior cobertura de linha. E os testes gerados pelo Randoop também obtiveram uma melhor cobertura no Windows com uma diferença percentual pequena em relação ao MacOs.

Quando analisados os resultados considerando a métrica de mutação, os testes do Randoop mataram mais mutantes do que os testes gerados pelo Evosuite em ambos os sistemas operacionais, e o percentual de mutantes mortos nas suítes geradas pelo Randoop foram equivalentes para o Windows e o MacOS. Considerando o percentual de mutantes mortos com os casos de testes gerados pelo Evosuite, a suíte de teste gerada pelo Windows matou mais mutantes do que no MacOS.

6.1 Sugestões para Trabalhos Futuros

[8] Disponível em: https://cs.gmu.edu/~offutt/mujava/. Acesso em 02 jun. 2022.
[9] Disponível em: http://jumble.sourceforge.net/. Acesso em 02 jun. 2022.
[10] Disponível em: http://databene.org/feed4junit.html. Acesso em 02 jun. 2022.
[11] Disponível em: https://plse.cs.washington.edu/daikon/. Acesso em 02 jun. 2022.
[12] Disponível em: https://github.com/konveyor/tackle-test-generator-cli. Acesso em 02 jun. 2022.

Como sugestões de trabalhos futuros, pode-se citar:

• Realização de um estudo com profissionais da área de TI para avaliar qual ferramenta gera uma suíte de teste mais legível e de fácil compreensão

• Avaliação das suítes de teste geradas pelas ferramentas TackleTest utilizando Evosuite, Randoop

• Analisar o impacto dos resultados quando uma suíte de teste gerada em um sistema operacional possui métricas geradas em outro sistema.

REFERÊNCIAS BIBLIOGRÁFICAS

BARTIÉ, A. **Garantia da qualidade de software: Adquirindo Maturidade Organizacional**. 1.ed. São Paulo: GEN LTC, 2002.

BAUER, F. L. Software Engineering. *In*: Information Processing: Proceedings of IFIP Congress, 71, Liubliana. **Anais** [...] Liubliana: Eslovênia, 1972. p. 530 - 538.

BEQUE, L. **Avaliação dos requisitos para teste de um sistema operacional embarcado**. Dissertação (Mestrado em Ciência da Computação) - Universidade Federal do Rio Grande do Sul. Porto Alegre, p. 59. 2009.

CRESPO, A. N. *et al*. Uma Metodologia para Teste de Software no Contexto da Melhoria de Processo. *In*: SIMPÓSIO BRASILEIRO DE QUALIDADE DE SOFTWARE (SBQS), 3., 2004, Brasília. **Anais** [...]. Porto Alegre: Sociedade Brasileira de Computação, 2004. p. 204-218. DOI: https://doi.org/10.5753/sbqs.2004.16194.

CROSBY, P. **Quality is Free**. Nova Iorque: McGraw-Hill, 1979.

DELAMARO, M. E; MALDONADO, J. C; JINO, M. **Introdução ao teste de software**. [S.l.: s.n.], 2016.

DIAS, L. D. Automatização de testes a cada entrega aumenta qualidade de um produto de TI de forma eficiente. **IT Forum**, 2021. Disponível em: https://itforum.com.br/noticias/automatizacao-de-testes-a-cada-entrega-aumenta-qualidade-de-um-produto-de-ti-de-forma-eficiente/. Acesso em: 15 jan. de 2022.

DIJKSTRA, E. W.; DAHL, O. J.; HOARE, C. A. R. **Structured Programming**. Londres: Academic Press, 1972.

FRASER, G; ARCURI, A. "EvoSuite: Automatic test suite generation for object-oriented software" *In*: SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13rd European Software Engineering Conference (ESEC-13), Szeged. **Anais** [...] Szeged: Hungria, 2011, doi: 10.1145/2025113.2025179.

INTHURN, C. **Qualidade & Teste de Software**. 1. ed. [S.l.]: Visual Books, 2001.

IZABEL, Leonardo Roxo Pessanha. Testes automatizados no processo de desenvolvimento de softwares. **Repositório da Politécnica UFRJ**, 2014. Disponível em: http://monografias.poli.ufrj.br/monografias/monopoli10012548.pdf. Acesso em: 10 jan. de 2022.

JIA, Y; HARMAN, M. An Analysis and Survey of the Development of Mutation Testing. **IEEE Transactions on Software Engineering,** Inglaterra**,** v. 37, n. 5, p. 649-678, Setembro, 2011. Disponível em: https://dl.acm.org/doi/10.1109/TSE.2010.62. Acesso em: 23 fev. 2022.

LEWIS, W. E. **Software Testing and Continuous Quality Improvement**. 2. ed. Boca Raton, USA: Auerbach Publications, 2000.

MATS, L. The top five software-testing problems and how to avoid them. **EDN**, 2001. Disponível em: https://www.edn.com/the-top-five-software-testing-problems-and-how-to-avoid-them/. Acesso em: 10 jan. de 2022.

MEYER, B. Design by Contract: A Conversation with Bertrand Meyer, Part II. [Entrevista concedida a] Bill Venners. **Artima**, Walnut Creek, 8 set. 2003.

MILANEZ, A. **Aprimorando a verificação de conformidade em programas baseados em contratos**. Dissertação (Mestrado em Ciência da Computação) - Universidade Federal de Campina Grande. Campina Grande, p. 123. 2014.

MILANEZ, A.; SOUSA, D.; MASSONI, T. & GHEYI, R. JMLOK2: A tool for detecting and categorizing nonconformances. In: Brazilian Conference on Software: Theory and Practice (CBSoft) - Tools Session, pages 69–76, 2014.

MOLINARI, L. **Teste de Software – Produzindo sistemas melhores e mais confiáveis**. 4. ed. São Paulo: Editora Érica Ltda, 2012.

OLIVEIRA, D. **Avaliação de Ferramentas de Geração Automática de Dados de Teste para Programas Java: Um Estudo Exploratório**. Dissertação (Mestrado em Ciência da Computação) - Universidade Federal de Goiás. Goiânia, p. 62. 2016.

PEREIRA, I. **Avaliação da efetividade de uma suíte de teste de sistema aplicada ao contexto do Middleware Ginga**. Dissertação (Mestrado em Engenharia de Software) - Universidade Federal do Rio Grande do Norte. Natal, p. 125. 2019.

PRESSMAN, R. S. **Engenharia de Software**. 8. ed. Porto Alegre: AMGH, 2016.

SILVA, G. Tipos de testes: quais os principais e por que utilizá-los? **Alura**, 2021. Disponível em: https://www.alura.com.br/artigos/tipos-de-testes-principais-por-que-utiliza-los. Acesso em: 13 jan. de 2022.

SOMMERVILLE, I. **Engenharia de Software**. 9. ed. São Paulo: Pearson Prentice Hall,2011.

SOUZA, B. B; MACHADO, P. D. L. Investigando a Efetividade de Ferramentas de Geração Automática de Casos de Teste. *In*: Congresso de Iniciação Científica da Universidade Federal de Campina Grande, XVI, Campina Grande. **Anais eletrônicos** [...] Campina Grande: Pró-Reitoria de Pós-Graduação e Pesquisa, 2019. Disponível em: https://posgraduacao.ufcg.edu.br/anais/2019/resumos/xvi-cic-ufcg-6073.pdf. Acesso em: 10 dez. 2021.

SUÁREZ, P. R. *et al*. Estudo e avaliação de ferramentas para desenvolvimento ágil. **Revista Principia - Divulgação Científica e Tecnológica do IFPB**, [S. l.], n. 26, p. 11-17, jun. 2015. Disponível em: https://periodicos.ifpb.edu.br/index.php/principia/article/view/13. Acesso em: 10 Jan. 2022.

VALENTE, M. T. **Engenharia de Software Moderna: Princípios e práticas para desenvolvimento de software com produtividade**. 1. ed. Minas Gerais: editora independente, 2020.

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DA PARAÍBA
Campus Campina Grande
R. Tranqüilino Coelho Lemos, 671, Dinamérica, CEP 58432-300, Campina Grande (PB)
CNPJ: 10.783.898/0003-37 - Telefone: (83) 2102.6200

Documento Digitalizado Ostensivo (Público)

Versão Final do TCC

Assunto: Versão Final do TCC
Assinado por: Kilvia Carvalho
Tipo do Documento: Projeto
Situação: Finalizado
Nível de Acesso: Ostensivo (Público)
Tipo do Conferência: Cópia Simples

Documento assinado eletronicamente por:

- **Kilvia da Silva Carvalho, ALUNO (201811250030) DE BACHARELADO EM ENGENHARIA DE COMPUTAÇÃO - CAMPINA GRANDE**, em 19/12/2022 11:56:22.

Este documento foi armazenado no SUAP em 19/12/2022. Para comprovar sua integridade, faça a leitura do QRCode ao lado ou acesse https://suap.ifpb.edu.br/verificar-documento-externo/ e forneça os dados abaixo:

Código Verificador: 696622
Código de Autenticação: 84c2475073