JULIANA BOTELHO DE CARVALHO

UMA INVESTIGAÇÃO DO CUSTO ENVOLVIDO NA IDENTIFICAÇÃO MANUAL DE MUTANTES EQUIVALENTES

LAVRAS – MG

2019

JULIANA BOTELHO DE CARVALHO

UMA INVESTIGAÇÃO DO CUSTO ENVOLVIDO NA IDENTIFICAÇÃO MANUAL DE MUTANTES EQUIVALENTES

Dissertação apresentada à Universidade Federal de Lavras, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, área de concentração em Engenharia de Software, para obtenção do título de Mestre em Ciência da Computação.

Prof. DSc. Rafael Serapilha Durelli (UFLA)
Orientador

Prof. DSc. Vinicius Humberto Serapilha Durelli (UFSJ)
Coorientador

LAVRAS – MG

2019

Ficha catalográfica elaborada pelo Sistema de Geração de Ficha Catalográfica da Biblioteca Universitária da UFLA, com dados informados pelo(a) próprio(a) autor(a).

Carvalho, Juliana Botelho de.
Uma Investigação do Custo Envolvido na Identificação Manual de Mutantes Equivalentes / Juliana Botelho de Carvalho. – Lavras : UFLA, 2019.
111 p. : il.

Dissertação(mestrado acadêmico–Universidade Federal de Lavras, 2019.
Orientador: Prof. DSc. Rafael Serapilha Durelli (UFLA).
Bibliografia.

1. Teste de mutação. 2. Mutantes equivalentes. 3. Custo da análise manual. I. Durelli, Rafael Serapilha. II. Durelli, Vinicius Humberto Serapilha. III. Título.

JULIANA BOTELHO DE CARVALHO

UMA INVESTIGAÇÃO DO CUSTO ENVOLVIDO NA IDENTIFICAÇÃO MANUAL DE MUTANTES EQUIVALENTES

Dissertação apresentada à Universidade Federal de Lavras, como parte das exigências do Programa de Pós-Graduação em Ciência da Computação, área de concentração em Engenharia de Software, para obtenção do título de Mestre em Ciência da Computação.

APROVADA em 22 de outubro de 2019.

Prof. DSc. Rafael Serapilha Durelli	UFLA
Prof. DSc. Vinicius Humberto Serapilha Durelli	UFSJ
Prof. DSc. Lucas Bueno Ruas de Oliveira	IFSP
Prof. DSc. Paulo Afonso Parreira Junior	UFLA
Profa. Dsc. Marluce Rodrigues Pereira	UFLA

Prof. DSc. Rafael Serapilha Durelli (UFLA)
Orientador

Prof. DSc. Vinicius Humberto Serapilha Durelli (UFSJ)
Co-Orientador

LAVRAS – MG
2019

Dedico esta dissertação à Deus pelo cuidado e sustento, sendo minha força e minha fé. E à minha família, em especial aos meus pais, por sempre me apoiarem e serem meu auxílio.

AGRADECIMENTOS

Durante os anos do mestrado, algumas pessoas se fizeram presentes e contribuíram de alguma forma para a realização desta dissertação. Meu maior agradecimento se estende a Deus, que fez com que tudo acontecesse no momento certo, conforme a Tua vontade. Obrigada Senhor por me ensinar a ter paciência, a confiar e ter fé. Obrigada por cuidar de mim e me preparar para este momento, por me permitir errar e aprender, e assim me tornar uma pessoa mais forte! Nada disso seria possível sem a mão de Deus, sem o Seu amor. Gratidão e fé resumem o meu sentimento neste momento. À minha família, em especial aos meus pais, que me proporcionaram grandes ensinamentos, obrigada pela pessoa que sou hoje. Obrigada por serem exemplos de família, de caráter, de dignidade, de responsabilidade, de amor e união. Ao meu irmão Leandro, meu eterno agradecimento, por ser meu companheiro, por estar ao meu lado e me apoiar. Esta dissertação não seria possível sem a orientação do Rafael Durelli. Obrigada pela paciência, competência, profissionalismo e amizade. Obrigada por sempre acreditar em mim e me incentivar! Vinicius Durelli, meu coorientador, obrigada pelos ensinamentos e pela sua excelência profissional! Obrigada pela paciência e dedicação. Agradeço aos meus professores das disciplinas, em especial ao professor Ricardo Terra, por ser exemplo de professor, aprendi muito com você! Agradeço à minha companheira e amiga Elena, minha dupla! Obrigada pelas madrugadas no DCC, pelos finais de semana fazendo trabalhos e escrevendo artigos. Aos meus companheiros de mestrado, obrigada por fazerem meus dias mais felizes e alegres. Aos meus queridos amigos dos outros laboratórios, como faria sem o cafézinho de todo dia? E também dos nossos *happy hour*, obrigada pela parceria! Vânia, minha querida, obrigada pelo companheirismo e por ser exemplo de força e de fé. Agradecer ao meu grande amigo de graduação, Diego Vinícius, por ser o grande incentivador para o ingresso no mestrado. Agradeço ao Carlos Henrique, por contribuir com seus conhecimentos no desenvolvimento do projeto. Agradeço à MDA Pesquisa, por me permitir seguir nesta caminhada. Agradeço à Universidade Federal de Lavras, especialmente ao Departamento de Ciência da Computação, pela oportunidade de realizar este sonho. Agradeço à secretária Luiza, por ser tão solícita em nos ajudar. Agradeço aos professores Heitor e Paulo, por compor a banca de qualificação e serem instrumentos para a melhoria desta dissertação e também aos professores Lucas e Marluce por compor a banca de defesa e contribuir com a finalização do mestrado. Não poderia deixar de agradecer à minha segunda família, Emaús, obrigada por me proporcionar momentos de fé e esperança. Meu "Samuzinho da Terê", Gabriela e Lívia, vocês foram essenciais em minha vida, obrigada por todo apoio! Por fim, agradeço à todos que contribuíram durante este período do mestrado.

"É preciso abandonar o futuro nas mãos do Bom Deus...
Nada acontece que Deus não tenha previsto desde toda a eternidade..."
(Santa Terezinha do Menino Jesus)

RESUMO

O teste de mutação consiste em mudar o sistema de software a ser testado por meio da aplicação de operadores de mutação. Como resultado, a aplicação de tais operadores gera versões diferentes desse sistema. Essas versões são denominadas mutantes. Os mutantes são utilizados para verificar se os casos de teste, construídos para o sistema de software, são capazes de identificar essas alterações efetuadas no código original, verificando a qualidade do conjunto de testes. Caso os testes de unidade sejam capazes de detectar todas as alterações, os mutantes são mortos e o conjunto de testes é suficiente; caso alguns mutantes não sejam mortos, é necessária a análise desses mutantes para verificar se novos casos de teste são necessários para detectar as alterações ou se os mutantes vivos são equivalentes ao código original, ou seja, se geram a mesma saída que o código original. Um problema do teste de mutação é o custo envolvido em identificar se esses mutantes são equivalentes ao sistema de software original. A identificação dos mutantes equivalentes é indecidível, portanto, normalmente a análise é conduzida manualmente, o que consome horas de um analista. Dessa forma, esta pesquisa investigou o custo humano envolvido na análise manual dos mutantes para a verificação da equivalência entre os códigos original e mutante. Para verificar o custo manual da análise, a ferramenta DiffMutAnalyze foi desenvolvida, a fim de comparar dois tipos de análise: utilizando a DiffMutAnalyze com a manual, ou seja, sem o uso de uma ferramenta específica para auxiliar a inspeção dos mutantes com os códigos originais. Na DiffMutAnalyze está incluída a ferramenta Major para geração dos mutantes, desse modo, somente projetos Java podem ser inseridos na ferramenta, pois a Major é uma ferramenta para realização da mutação em projetos Java. Por meio do experimento realizado, é possível verificar o tempo gasto com a análise manual e verificar a diferença dessa análise com a análise por meio de uma ferramenta de apoio computacional. Tal ferramenta tem como propósito possibilitar que programas originais e mutantes sejam comparados lado a lado, facilitando a análise manual dos mutantes, uma vez que na ferramenta, o local da mutação é evidenciado para o usuário. Para realizar essa comparação das análises manual e por meio da DiffMutAnalyze, um experimento foi conduzido. Como resultado do experimento, em geral, os sujeitos participantes gastaram em média 31 minutos e 30 segundos para realizar a análise manual dos mutantes, quando a análise foi realizada por meio da DiffMutAnalyze, o tempo médio foi de 14 minutos. Dessa forma, a análise realizada por meio da DiffMutAnalyze foi reduzida em mais de 50% do tempo. Portanto, a utilização de uma ferramenta para auxiliar a verificação da identificação dos mutantes equivalentes, reduz o custo dessa análise. Além disso, foi verificado que a ferramenta ajuda na realização do teste de mutação, uma vez que a mutação do código original é realizada automaticamente e os mutantes sobreviventes aos testes de unidades são disponibilizados para inspeção. Com isso, a comparação dos códigos original e mutante se torna mais rápida, pois na ferramenta, os códigos são disponibilizados automaticamente e a alteração efetuada é identificada pela DiffMutAnalyze. Assim, o analista não necessita procurar o local da alteração realizada pelos operadores de mutação.

Palavras-chave: Teste de mutação; Mutantes equivalentes; Custo da análise manual.

ABSTRACT

Mutation testing consists of changing the software system to be tested by applying mutation operators. As a result, applying such operators generates different versions of this system. These versions are called mutants. Mutants are used to verify that test cases built for the software system are able to identify these changes made to the original code by checking the quality of the test suite. If unit tests are able to detect all changes, the mutants are killed and the test set is sufficient; If some mutants are not killed, analysis of these mutants is required to check whether new test cases are needed to detect changes or whether live mutants are equivalent to the original code, ie generate the same output as the original code. The problem with mutation testing is the cost involved in identifying whether these mutants are equivalent to the original software system. Identification of equivalent mutants is undecidable, so analysis is usually conducted manually, which consumes hours of an analyst. Thus, this research investigated the human cost involved in the manual analysis of mutants to verify equivalence between the original and mutant codes. To verify the manual cost of analysis, the DiffMutAnalyze tool was developed in order to compare two types of analysis: using DiffMutAnalyze with the manual, ie without the use of a specific tool to aid inspection of mutants with the original codes. DiffMutAnalyze includes the Major tool for mutant generation, so only Java projects can be inserted into the tool, as Major is a tool for mutation in Java projects. Through the experiment, it is possible to verify the time spent with the manual analysis and to verify the difference between this analysis and the analysis by means of a computational support tool. The purpose of this tool is to enable original and mutant programs to be compared side by side, facilitating the manual analysis of the mutants, since in the tool the location of the mutation is evident to the user. To perform this comparison of the analyzes manually and through DiffMutAnalyze, an experiment was conducted. As a result of the experiment, in general, the participants took an average of 31 minutes and 30 seconds to perform manual mutant analysis, when the analysis was performed using DiffMutAnalyze, the average time was 14 minutes. Thus, the analysis performed using DiffMutAnalyze was reduced by more than 50% of the time. Therefore, the use of a tool to assist in the identification of equivalent mutant identification reduces the cost of this analysis. In addition, it has been found that the tool assists in performing the mutation test as the original code mutation is performed automatically and the surviving unit test mutants are made available for inspection. This makes the comparison of the original and mutant codes faster because in the tool the codes are automatically made available and the change made is identified by DiffMutAnalyze. Thus, the analyst need not look for the location of the change made by the mutation operators.

Keywords: Mutation Testing; Equivalents Mutants; Cost of manual analysis.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Desenvolver sistemas de software é um processo composto por diversas atividades e uma das atividades é a aplicação de testes nesses sistemas. O teste tem como principal objetivo garantir a qualidade do software (SOMMERVILLE, 2011). Na literatura (AMMANN; OFFUTT, 2008; DELAMARO et al., 2016), é possível identificar diversas atividades de teste, por exemplo: *i)* teste de funcionalidade; *ii)* teste de desempenho; *iii)* teste de unidade; e *iv)* teste de mutação. Neste projeto, o objetivo é investigar e utilizar o teste de mutação. Neste capítulo é apresentada a contextualização, a motivação, o problema, os objetivos e a justificativa para a condução da pesquisa em questão.

1.1 Contextualização e Motivação

Os desafios da Engenharia de Software aumentaram nos últimos anos diante da crescente importância do software e dos novos desenvolvimentos tecnológicos. Conforme a Engenharia de Software evolui, mais pesquisas nessa área são conduzidas. Assim, métodos, ferramentas e técnicas têm sido propostos para facilitar e apoiar o desenvolvimento de sistemas de software. Em conjunto com essa evolução, foi fundamental que engenheiros de software se preocupassem em produzir produtos com qualidade. Atividades de Verificação, Validação e Teste (VV & T) (DELAMARO et al., 2016) surgiram com o propósito de acompanhar o processo da produção de sistemas de software, não se restringindo somente ao produto final, mas em todas as atividades conduzidas desde sua concepção.

Uma das atividades do desenvolvimento de sistemas de software envolve o teste desses sistemas em desenvolvimento. A combinação de técnicas de teste permite produzir esses sistemas com qualidade. Para que a qualidade seja alcançada, a atividade de teste consiste em encontrar defeitos ao longo do ciclo do desenvolvimento de sistemas de software. O teste pode ser dividido em fases (DELAMARO et al., 2016): *i)* teste de unidade; *ii)* teste de integração; e *iii)* teste de sistemas. No teste de unidade, o ponto central são as menores unidades de um programa, que podem ser funções, procedimentos, métodos ou classes. Visto que um teste bem-sucedido revela a presença de um defeito, um bom caso de teste é aquele que tem alta probabilidade de detectar um erro ainda não descoberto (MYERS et al., 2011). Dessa forma, conforme o sistema evolui, novos testes de unidade são construídos para verificar a qualidade do sistema de software. No teste de integração, o foco é nas interfaces das unidades que compõem

o sistema. O teste de sistema verifica o sistema como um todo, usualmente testa-se uma *release* de sistema a ser entregue ao cliente.

Considerar que o teste é eficaz quando ele é capaz de detectar falhas, pode ser difícil de mensurar, uma vez que, o conjunto de falhas em um sistema de software é incognoscível. Dessa forma, os desenvolvedores não possuem total compreensão da eficácia dos casos de teste (ALVIN et al., 2019). Uma maneira de verificar a efetividade do conjunto de casos de teste é utilizando técnicas que revelam defeitos no código. Assim, é possível avaliar a eficácia dos testes medindo a capacidade deles em encontrar falhas no código do sistema de software. Essas falhas podem ser introduzidas artificialmente no sistema de software original e, a partir de uma versão modificada do código original, é possível verificar a capacidade dos testes em detectar tais falhas (AMMANN; OFFUTT, 2008; DELAMARO et al., 2016).

Esse critério de teste é chamado de teste de mutação. O teste de mutação oferece evidências para avaliar a eficácia do conjunto de testes (USAOLA; MATEO, 2010; HU et al., 2011; ZHU et al., 2018; FERRARI et al., 2018; PAPADAKIS et al., 2019). A ideia desse critério de teste é inserir falhas artificiais ao sistema de software original, gerando uma nova versão desse sistema, denominada mutante. As mudanças realizadas no código são alterações sintáticas e essas alterações são semelhantes às falhas reais (PAPADAKIS et al., 2019). A premissa da análise de mutação é que, se um conjunto de testes puder revelar um comportamento diferente entre cada mutante e o sistema de software original, o conjunto de testes é eficaz em encontrar defeitos reais no software em teste (ALVIN et al., 2019). O trabalho do testador é, basicamente, criar casos de teste que detectem a diferença de comportamento entre o programa original e os mutantes (DELAMARO et al., 2016). Assim, quanto melhor o conjunto de casos de teste, maior a chance de encontrar erros. Se o conjunto de testes não encontrar essas falhas que foram introduzidas no código, é provável que não encontre falhas reais, portanto os testes devem ser melhorados (SCHULER; ZELLER, 2013).

Do ponto de vista da pesquisa, o teste de mutação é uma técnica madura (JIA; HARMAN, 2011; USAOLA; MATEO, 2010; PAPADAKIS et al., 2019). Esse critério de teste é considerado uma avaliação efetiva para a qualidade do conjunto de testes. No entanto, vários obstáculos estão associados a sua realização, dificultando a sua adoção como prática de teste. No trabalho de revisão sistemática de Zhu et al. (ZHU et al., 2018), os autores identificaram que grande parte das pesquisas sobre teste de mutação estão relacionadas aos processos de garantia de qualidade. Assim, o teste de mutação é importante porque fornece um mecanismo que avalia

a eficácia dos casos de teste. Além disso, os autores relatam os principais interesses na área de teste de mutação (ZHU et al., 2018): *i)* definir operadores de mutação; *ii)* desenvolvimento de sistemas de testes de mutação; *iii)* redução do custo do teste de mutação; *iv)* superação do problema dos mutantes equivalentes; e *v)* estudos empíricos sobre teste de mutação. No contexto das pesquisas que envolvem o teste de mutação citado anteriormente, este projeto de pesquisa está relacionado com estudo empírico para investigar o custo em identificar os mutantes equivalentes. Os mutantes equivalentes são aqueles que geram a mesma saída que o sistema de software original (JIA; HARMAN, 2011).

Como observado no trabalho de Zhu et al. (ZHU et al., 2018), o custo associado à identificação de mutantes equivalentes ainda é um dos principais desafios enfrentados por pesquisadores e profissionais. No trabalho de Papadakis et al. (PAPADAKIS et al., 2019), os autores apresentam um levantamento abrangente sobre teste de mutação, foi identificado o avanço e evolução do nível de interesse nesse critério de teste. Os autores pesquisaram as várias aplicações do teste de mutação e citam que o problema dos mutantes equivalentes permanece em grande parte não resolvido. Vale ressaltar que neste projeto de pesquisa, o custo está associado ao esforço humano para análise da equivalência entre os códigos, dessa maneira, não está sendo abordado o custo computacional da geração da mutação. Na próxima seção, está descrito o principal problema investigado neste estudo.

1.2 Problema e Objetivo

O teste de mutação é considerado uma técnica eficaz para localização de falhas (NATELLA et al., 2013; PAPADAKIS; TRAON, 2012; JUST et al., 2014), porém tem como desvantagem o alto custo de sua aplicação. Um dos maiores obstáculos à adoção do teste de mutação está em identificar quais mutantes se comportam como o código original, ou seja, produzem a mesma saída. Esses códigos mutantes que se comportam da mesma maneira que o código original são considerados equivalentes (OFFUTT; PAN, 1997). É possível que uma mutação deixe a semântica inalterada do sistema de software em teste.

No trabalho de Grün et al. (GRÜN et al., 2009), foi realizado um experimento para verificar a aplicação da mutação em sistemas de softwares em larga escala. O sistema de software utilizado foi o JAXEN, o qual possui 12.229 linhas de código Java. Para esse sistema de software, 9.819 mutantes foram gerados. Os mutantes que não foram detectados pelos casos de teste foram analisados manualmente, a análise de uma única mutação levou entre 1 e

50 minutos. A verificação da possível equivalência dos mutantes com o código original gastou um total de 10 horas dos analistas, o que resultou em média cerca de 15 minutos para verificar manualmente se um mutante é equivalente ao programa original.

Embora ainda não exista pesquisa capaz de encontrar uma solução abrangente para identificar mutantes equivalentes, vários estudos foram realizados para mitigar os custos envolvidos na análise desses mutantes (JIA; HARMAN, 2011; MA; KIM, 2016; PAPADAKIS et al., 2015). Ainda que existam técnicas para detectar algumas mutações equivalentes (OFFUTT; PAN, 1997; ARCAINI et al., 2017; PAPADAKIS et al., 2014; KINTIS; MALEVRIS, 2015; NICA; WOTAWA, 2012), é importante salientar que o problema, em geral, é indecidível (BUDD; ANGLUIN, 1982).

A identificação dos mutantes equivalentes é necessária para verificar se os mutantes não detectados pelos casos de teste são de fato equivalentes ao código original, ou se novos casos de teste devem ser construídos para conseguirem detectar esses mutantes. Dessa forma, a análise desses mutantes é valiosa para melhorar o conjunto de testes (GRÜN et al., 2009). Além disso, há uma medida para avaliar a qualidade dos casos de teste, denominada escore de mutação, essa medida utiliza a quantidade de mutantes gerados, a quantidade de mutantes mortos pelos casos de teste e a quantidade de mutantes equivalentes gerados. Dessa forma, para que o cálculo dessa medida seja realizado de maneira efetiva, a quantidade de mutantes equivalentes precisa ser definida.

Além da verificação da equivalência dos programas, os operadores de mutação desempenham um papel fundamental na determinação do custo da análise de mutantes. A principal desvantagem dos conjuntos de operadores de mutação é eles incluírem muitos operadores, o que implica na geração de muitos mutantes. Pesquisas têm sido conduzidas para abordar esse problema, algumas dessas pesquisas envolvem a investigação para reduzir a quantidade de mutantes gerados (DURELLI et al., 2017; MA; KIM, 2016; SUN et al., 2017). Diante da geração de grandes quantidades de mutantes, a aplicação da mutação é afetada, ou seja, quando a quantidade de mutantes gerados é alta, a complexidade computacional da execução de todos os mutantes aumenta e, consequentemente, aumenta o esforço para identificação dos mutantes equivalentes (DELAMARO et al., 2016).

Nesta pesquisa, o objetivo é responder as seguintes questões de pesquisa (QP):

QP1: *"Qual é o custo para determinar manualmente a equivalência de mutantes com o sistema de software original?"*
QP2: *"A ferramenta DiffMutAnalyze pode contribuir com a redução do custo da análise, auxiliando os analistas na identificação dos mutantes equivalentes?"*

Dessa forma, o objetivo principal é investigar o custo humano envolvido na atividade de identificação de mutantes equivalentes, visto que essa atividade deve ser realizada de forma manual. A geração de muitos mutantes implica em maior tempo de execução da ferramenta de mutação e, conforme a quantidade de mutantes sobreviventes aumenta, o custo humano para análise de tais mutantes aumenta. Para investigar o custo da análise manual, a ferramenta DiffMutAnalyze foi desenvolvida para verificar se o uso da ferramenta pode contribuir com a redução do custo da análise, comparando com a análise manual, ou seja, sem a utilização de uma ferramenta que auxilie essa análise. O custo refere-se ao tempo empregado de um analista durante a análise dos mutantes e, assim, além de investigar o custo da análise dos mutantes, este projeto de pesquisa, verificou se por meio da DiffMutAnalyze é possível contribuir com a redução do custo da análise durante a identificação dos mutantes equivalentes. Mais detalhes para responder às QPs estão descritos na seção a seguir.

1.3 Justificativa

Diante da observação realizada no trabalho de Grün et al. (GRÜN et al., 2009), foi identificado que são gastos em média, 15 minutos para verificar se um mutante é equivalente ao seu código original correspondente. Nesta pesquisa, o intuito é verificar o custo da análise manual desses mutantes, comparando com o uso de uma ferramenta de apoio computacional e assim estabelecer um paralelo a fim de verificar o custo da análise manual e também verificar se uma ferramenta pode auxiliar durante a análise e reduzir o custo da identificação manual.

Para responder às QPs, um experimento foi conduzido para investigar o custo envolvido na identificação manual de mutantes equivalentes. A análise desses mutantes, quando conduzida de forma manual, é feita por meio da análise do código original e da sua versão alterada. O analista compara as duas versões do código e verifica se produzem a mesma saída, o que traz dois resultados distintos por meio da análise: *i)* o mutante é equivalente ao código original; ou *ii)* o mutante não é equivalente, quando o mutante não é equivalente, novos casos de teste devem

ser construídos para identificar a mudança imposta no dado mutante. Desse modo, o analista necessita comparar as duas versões do código para detectar se há equivalência entre eles.

Alguns mutantes não podem ser "mortos" por nenhum caso de teste, porque são equivalentes (isto é, possuem o mesmo comportamento que o programa original) e sua identificação é realizada manualmente, somando o custo humano envolvido na análise dos mutantes. Para este projeto de pesquisa, o custo humano é o fator predominante na análise manual para a identificação dos mutantes equivalentes. Uma vez que, segundo Delamaro et al. (DELAMARO et al., 2016), uma parte significativa do ônus da realização dos testes de mutação está relacionada com a análise dos mutantes, uma vez que é necessária a intervenção humana. Portanto, o custo humano da análise de mutação geralmente domina. Dessa forma, para este projeto de pesquisa, o custo da análise é medido portanto, através do tempo gasto em analisar cada mutante. O experimento irá investigar quais operadores de mutação geram mutantes mais complexos de serem analisados, fornecendo evidências dos mutantes que mais contribuem para o custo manual da análise dos mutantes.

Analisando as ferramentas de mutação já existentes, como Major (JUST, 2014), MuJava (MA et al., 2006) e Pit (COLES, 2018), a DiffMutAnalyze diferencia dessas ferramentas no que diz respeito à análise e definição dos mutantes equivalentes. Uma vez que essas ferramentas apenas geram os mutantes e os disponibilizam para visualização, porém a análise para determinar se o dado mutante é equivalente e associar essa informação àquele mutante, não pode ser realizada nas ferramentas citadas anteriormente. Assim, a DiffMutAnalyze adiciona ao mutante sendo analisado a informação se ele é equivalente ao código original, além disso, é possível indicar o grau de dificuldade em analisar o dado mutante e contabilizar o tempo da análise na ferramenta proposta nesta pesquisa, no qual essas informações não são armazenadas nas ferramentas de mutação existentes.

Além das contribuições da ferramenta DiffMutAnalyze, por meio do experimento, esta pesquisa contribui com a fundamentação teórica sobre quais mutantes são difíceis de serem analisados e quais os operadores de mutação que os geraram. O tempo gasto com a análise dos mutantes é fornecido, assim, é possível perceber se os mutantes com maiores tempo de análise estão relacionados com o grau de dificuldade em analisá-los. Além disso, as contribuições técnicas desta pesquisa de mestrado se estabelece por meio da ferramenta desenvolvida. Por meio do uso da DiffMutAnalyze, outros pesquisadores podem utilizar a ferramenta para aplicação da mutação em outros projetos de pesquisa, além de poder incluir em sala de aula a utilização

prática do teste de mutação. É importante considerar a utilização de novas técnicas de teste de software no âmbito acadêmico, uma vez que o campo atual da indústria de software apresenta problemas cada vez mais complexos. Por essa razão, as empresas anseiam por profissionais com conhecimento adequado sobre desenvolvimento de software, dessa forma, elas esperam que as instituições educacionais forneçam conhecimento suficiente para que os novos alunos se adaptem instantaneamente a uma atmosfera industrial, sem que haja a necessidade de treinamento intensivo (RADHAKRISHNAN et al., 2015). Para acompanhar essa evolução constante da indústria, é sugerido que as instituições de ensino busquem se adequar ao conteúdo industrial e, empregar em sala de aula, conceitos que possam contribuir para o aprendizado dos futuros engenheiros de software (ZHENG et al., 2018; SPACCO; PUGH, 2006).

Motivar o aluno sobre a importância do aprendizado sobre teste de software vem sendo difundida por alguns autores (LEMOS et al., 2018; JANZEN; SAIEDIAN, 2006; CHRISTENSEN, 2003). O principal foco é mostrar a importância do teste e ensinar aos alunos que o teste deve ser empregado e desenvolvido naturalmente. Dessa forma, incentivar os alunos sobre a importância dos testes, é um tópico que deve ser trabalhado em sala de aula, é preciso encontrar maneiras de incentivar a prática da utilização de técnicas de testes em projetos de programação. Somente exigir que os alunos utilizem técnicas de teste durante as aulas pode não ser muito efetivo, uma vez que o intuito é fazer com que os alunos entendam a necessidade dos testes; é preciso fazer com que eles apreciem o valor que um caso de teste possui (ZHENG et al., 2018; SPACCO; PUGH, 2006) e instigá-los a produzir seus próprios casos de teste durante o desenvolvimento de um projeto, para que eles busquem desenvolver softwares com qualidade.

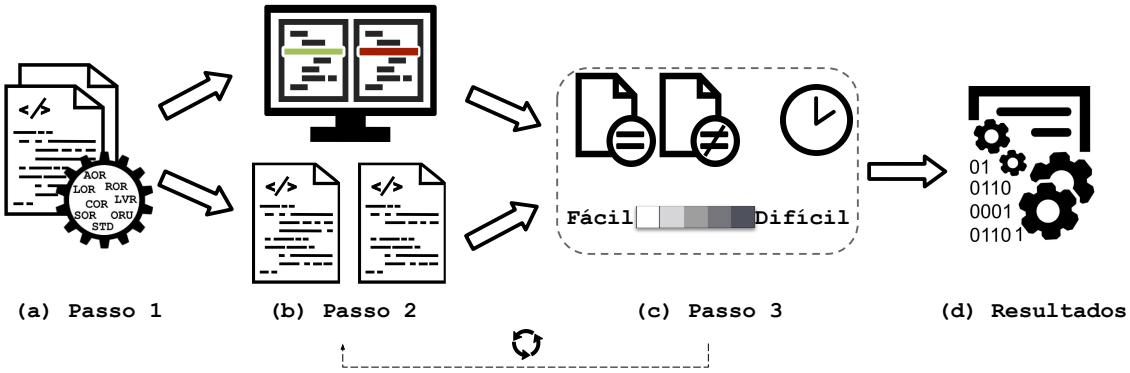
Ao verificar a importância de desenvolver novas habilidades e gerir novos conhecimentos para os alunos de engenharia de software, o critério de teste de mutação é promissor para ser empregado em disciplinas da engenharia de software. A ferramenta DiffMutAnalyze desenvolvida neste projeto de mestrado contribui com o ensino prático sobre o teste de mutação. Uma vez que ela disponibiliza um ambiente amplo de aprendizagem sobre teste de mutação, que contempla desde a geração dos mutantes até os resultados das análises dos mutantes.

1.4 Considerações finais

Na Figura 1.1, é ilustrado como a avaliação para verificar o custo envolvido na identificação dos mutantes equivalentes foi executada. Inicialmente o sistema de software foi selecionado para realização do teste de mutação e o primeiro passo (1.1(a)) implica em aplicar os operadores

de mutação ao código a ser testado. Alguns mutantes são detectados (mortos) pelos casos de teste construídos e outros permanecem vivos; para esses mutantes, uma análise é necessária. Assim, no segundo passo (1.1(b)), as versões do sistema de software (i. e., original e mutantes) são disponibilizadas para análise, seja por meio da DiffMutAnalyze ou de forma manual, para identificação da possível equivalência entre tais versões. Após a inspeção dos códigos, o analista indica, no terceiro passo (1.1(c)), quais mutantes são equivalentes ao código original ou se mais casos de teste são necessários para matar os mutantes vivos. Além disso, o tempo de análise é contabilizado e o analista indica o grau de dificuldade em inspecionar os códigos. Caso a análise seja realizada na DiffMutAnalyze, o tempo é contabilizado automaticamente e o grau de dificuldade é informado na ferramenta; para a análise manual, essas informações são anotadas a parte. Os passos dois e três são executados até que todos os mutantes sejam analisados. Ao final da análise, os resultados (1.1(d)) da avaliação são apresentados e comparados, verificando o tempo gasto com a análise manual e com a análise por meio da DiffMutAnalyze. Dessa forma, o tempo é medido e o custo da análise é avaliado, além disso, é possível perceber se a DiffMutAnalyze pode contribuir com a redução do custo da análise.

Figura 1.1 – Descrição da abordagem proposta.



Fonte: Elaborado pela autora (2018).

1.5 Publicações e Submissões

Durante o desenvolvimento desta pesquisa, artigos foram publicados e submetidos em revistas, conferências e *workshops*. A seguir estão listados os trabalhos publicados e os que foram enviados para análise.

1.5.1 Trabalhos diretamente relacionados com o tema da pesquisa

A - PUBLICADO

- BOTELHO, J.; PEREIRA, C. H.; DURELLI, V. H. S.; DURELLI, R. S. *DiffMutAnalyze: Uma abordagem para auxiliar a identificação de mutantes equivalentes*. In VI Workshop on Software Visualization, Evolution and Maintenance (VEM). 2018.
  (Qualis: B5) (BOTELHO et al., 2018)

B - SUBMETIDO

- BOTELHO, J.; DURELLI, V. H. S.; DURELLI, R. S. **DiffMutAnalyze: A Tool to Assist the Analysis of Equivalent Mutants in Mutation Test Teaching**. In ACM Transactions on Computing Education - 2019.
  (Qualis: A1)

C - A SER SUBMETIDO

- BOTELHO, J.; DURELLI, V. H. S.; BORGES, S. S.; DURELLI, R. S. ***"Do fewer"* apenas mutantes suficientes: uma revisão sistemática sobre técnicas de redução de custos**. In Brazilian Symposium on Systematic and Automated Software Testing (SAST) - 2020
  (Qualis: B5)

1.5.2 Trabalhos indiretamente relacionados com o tema da pesquisa

A - PUBLICADO

- BOTELHO, J.; DURELLI, V. H.; BORGES, S. S.; ENDO, A. T.; ELER, M. M.; DELAMARO, M. E.; DURELLI, R. S. **On the costs of applying logic-based criteria to mobile applications: An empirical analysis of predicates in real-world objective-c and swift applications.** In: ACM. 2nd Brazilian Symposium on Systematic and Automated Software Testing (SAST). [S.l.], 2017. p. 4.
  (Qualis: B5) (BOTELHO et al., 2017) [1]

B - A SER SUBMETIDO

- Elena A. Araujo, Juliana Botelho, Rafael S. Durelli e Ricardo Terra. **Uma Abordagem Híbrida para Visualização da Comunicação entre Microsserviços**. In 14rd Brazilian Symposium on Components, Architectures and Software Reuse (SBCARS)
  (Qualis: B3)

[1] Trabalho premiado como terceiro melhor artigo do 2nd Brazilian Symposium on Systematic and Automated Software Testing (SAST - 2017).

1.6 Estrutura da dissertação

A estrutura da dissertação está organizada da seguinte forma: no Capítulo 2, é apresentado o referencial teórico, fornecendo uma visão geral sobre testes de software, teste de mutação, mutantes equivalentes, ferramentas de apoio ao teste de mutação, a ferramenta de mutação escolhida para realização da mutação do sistema de software, bem como os operadores de mutação implementados por tal ferramenta. No Capítulo 3, está descrita a proposta da ferramenta DiffMutAnalyze, assim como sua arquitetura e suas funcionalidades. No Capítulo 4, está exposta a metodologia realizada para investigar o custo em analisar os mutantes e os procedimentos dos experimentos. Os resultados dos experimentos estão expostos no Capítulo 5. Os estudos relacionados ao teste de mutação são descritos no Capítulo 6. A conclusão e trabalhos futuros são apresentados no Capítulo 7. Por fim, no apêndice A desta dissertação contém alguns exemplos de mutantes gerados pela ferramenta Major, contida na DiffMutAnalyze. Além disso, consta o formulário de caracterização dos participantes do experimento.

2 REFERENCIAL TEÓRICO

2.1 Considerações iniciais

Com a crescente demanda da utilização de sistemas de software, a busca pela qualidade dos sistemas implementados está presente nas equipes de desenvolvimento. Pode-se argumentar que a qualidade de um produto final depende, em grande parte, dos métodos, das técnicas e das ferramentas que os desenvolvedores utilizam. Entre os procedimentos que buscam a qualidade de um sistema de software, os testes desempenham papel importante em qualquer projeto de desenvolvimento desses sistemas (PETERS; PEDRYCZ, 2001).

Certificar-se de que o software funciona pode ser caracterizado como a maior responsabilidade de um desenvolvedor de software. O teste de software é constituído por processos que visam garantir que o código implementado esteja em conformidade com o que foi especificado (MYERS et al., 2011). Inspecionar bases de código grandes e complexas para verificação dessa conformidade não é uma tarefa trivial no mundo real, muitas vezes os analistas precisam ter uma ampla compreensão de diferentes práticas de testes de software, que variam de simples testes exploratórios manuais, onde o próprio analista tenta encontrar falhas manualmente interagindo com o sistema, a técnicas avançadas de teste, como testes automatizados, em que os desenvolvedores programam máquinas para testar o código (ANICHE et al., 2019).

Visando apresentar os principais conceitos do teste de software com ênfase no critério de teste denominado teste de mutação, este capítulo está organizado da seguinte forma: na Seção 2.2, são apresentados os conceitos fundamentais sobre o teste de software. Na Seção 2.3, os conceitos sobre teste de mutação e seus apoios ferramentais disponíveis na literatura são apresentados e elucidados.

2.2 Teste de Software

O campo atual da indústria de software apresenta problemas cada vez mais complexos. Com a crescente demanda de sistemas de software para utilização em diversas áreas, que vai além do âmbito da computação, a busca por qualidade e produtividade aumenta. Dessa forma, engenheiros de software buscam garantir a qualidade de sistemas de software desenvolvidos (PRESSMAN, 2016).

A visão geral do teste de software é que é uma atividade para "encontrar *bugs*". Para William E. Lewis (LEWIS, 2017), os objetivos do teste de software são qualificar a qualidade

de um sistema de software, medindo seus atributos e capacidades em relação às expectativas do cliente. Além disso, o teste de software também fornece informações importantes para o esforço de desenvolvimento de software.

Há problemas que podem ser caracterizados como defeito, erro ou falha no sistema de software que provoca seu mau funcionamento. Assim, no contexto deste projeto, é importante salientar que defeito, erro e falha seguem as seguintes definições (IEEE..., 1990): *i)* defeito é caracterizado por um defeito estático no software; *ii)* erro caracteriza-se por um estado interno incorreto que é a manifestação de alguma falha; e *iii)* falha é um comportamento externo incorreto com relação aos requisitos ou outra descrição do comportamento esperado.

Para que defeitos, erros e falhas sejam identificados durante o processo de desenvolvimento de sistemas de software e não somente ao final do projeto, a definição de Verificação, Validação e Teste (VV&T) garante a revisão contínua do projeto de software, minimizando a ocorrência de erros (MALDONADO et al., 1998), ou seja, identifica se o produto está em conformidade com o especificado, garantindo a qualidade do produto final (DELAMARO et al., 2016). A seguir, os conceitos relacionados à tarefa de VV&T são descritos.

2.2.1 Verificação, Validação e Teste

A atividade de verificação consiste em obter um conjunto de tarefas que envolvem avaliações dos requisitos do cliente e especificações do projeto, garantindo a implementação correta do sistema de software e podem ser vistas como atividades mais técnicas; essa atividade pode estar presente em todo o processo de desenvolvimento. A atividade de validação concentra-se no produto final, que pode ser testado pelos usuários durante o teste de aceitação, assegurando que o sistema de software desenvolvido está de acordo com o planejado, garantindo a conformidade com o uso pretendido. Na validação é possível verificar a correspondência entre o sistema e as expectativas do cliente, validando se está sendo desenvolvido o sistema de software correto (BARESI; PEZZE, 2006; AMMANN; OFFUTT, 2008).

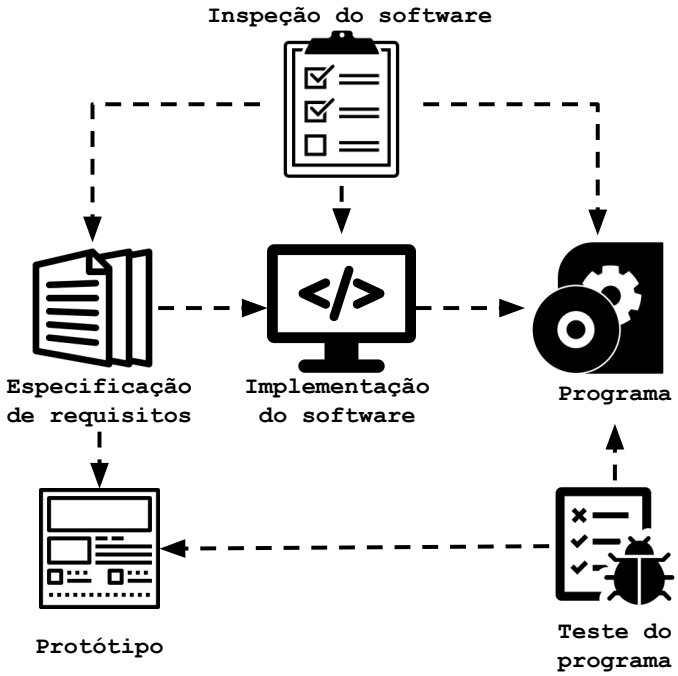
O intuito da atividade de teste do software é encontrar falhas no sistema em execução, utilizando algumas entradas para verificar se seu comportamento está conforme o especificado. Caso a execução apresente resultados diferentes do esperado, um erro ou defeito foi identificado e precisa ser corrigido. Assim, é possível identificar os erros que surgem durante o desenvolvimento de sistemas de software, verificando se o comportamento desse sistema está de acordo com o esperado (DELAMARO et al., 2016). Nesse contexto, é possível perceber que as ativi-

dades de verificação, de validação e de teste são complementares e estão presentes em vários estágios do desenvolvimento de sistemas de software.

Realizar testes em sistemas de software é um processo não trivial, uma vez que demanda esforços nos processos de VV&T. Dessa forma, é indicado que se tenha um planejamento para obter o máximo de inspeções necessárias, tanto na atividade de verificação quanto na de validação, sem atingir um custo elevado. O esforço dedicado a essas atividades de VV&T depende do tipo de sistema desenvolvido e da habilidade organizacional para realização dos testes. Quanto mais crítico o sistema, mais esforço é demandado à essas técnicas (SOMMERVILLE, 2011).

Como pode ser observado na Figura 2.1, o processo de VV&T desempenha um papel importante ao longo do projeto do software, onde a inspeção é fundamental desde a etapa inicial até a consolidação do produto final. As setas indicam que a inspeção está presente em todos os estágios do sistema de software. A fase de teste está presente desde o protótipo até que uma versão executável do sistema de software esteja disponível. Geralmente, as funções são testadas conforme são adicionadas ao sistema, não sendo necessário ter uma versão completa desse sistema (SOMMERVILLE, 2011).

Figura 2.1 – Processo VV&T.



Fonte: Adaptado de Sommerville (2011).

2.2.2 Atividade do teste

O teste de software é uma atividade dinâmica e complexa e que necessita ser realizada durante o desenvolvimento, pois há fatores distintos que podem contribuir com a ocorrência de erros. Alguns erros que podem ocorrer durante o desenvolvimento, por exemplo, podem ser erro na utilização de um algoritmo ou a não implementação de um requisito do software, sendo dois erros de tipos distintos. Dessa forma, para que a atividade de teste seja melhor gerenciada, recomenda-se sua aplicação em fases distintas (AMMANN; OFFUTT, 2008).

O teste de software é um termo amplo que engloba diferentes atividades, desde o teste do código pelo desenvolvedor (teste de unidade) até a validação do cliente de um grande sistema de software (teste de aceitação). Assim, como há fatores distintos que podem ocasionar algum erro, há casos de teste que podem ser planejados com objetivos diferentes, como expor desvios dos requisitos do usuário ou avaliar a conformidade com a especificação, ou avaliar a robustez a condições estressantes de carga ou a entradas maliciosas, ou medir determinados atributos, como desempenho e usabilidade, ou estimar a confiabilidade operacional, dentre outros fatores que podem ocorrer (BERTOLINO, 2007).

É importante salientar que o foco principal desta pesquisa se encontra em avaliar a qualidade dos testes de unidade, cujo objetivo é avaliar as unidades de um sistema de software (DELAMARO et al., 2016). Fazendo a verificação dinamicamente, é requerido que o sistema de software seja executado com alguns casos de teste constituídos por conjuntos de dados de entrada e as saídas esperadas do software a ser testado. A meta do processo do teste de unidade é criar casos de teste capazes de descobrir falhas do sistema (SOMMERVILLE, 2011). Como cada unidade é testada separadamente, os casos de teste são implementados de acordo com a evolução do sistema sendo desenvolvido, sem a necessidade de o sistema estar totalmente finalizado para iniciar a construção desses casos de teste.

O objetivo da atividade de teste é definido pelo processo de execução de um sistema de software com a intenção de encontrar defeitos que façam com que o resultado obtido ao final da execução não esteja de acordo com o que foi especificado. Uma vez que um teste bem-sucedido é o que revela a presença de um defeito, um bom caso de teste é aquele que tem alta probabilidade de detectar um erro ainda não descoberto (MYERS et al., 2011).

Para auxiliar o uso automatizado do teste, existem alguns *frameworks* que são utilizados para apoiar a construção e execução dos testes de unidade. Zerouali e Mens (ZEROUALI; MENS, 2017) relacionaram os *frameworks* utilizados ao longo do tempo para testar sistemas de

software desenvolvidos em Java. Para realizar a análise dos *frameworks* utilizados para realização dos testes em sistemas de softwares, os autores verificaram os *frameworks* utilizados em mais de 4.000 projetos de código aberto; além disso, foi verificada a frequência de utilização dos *frameworks* de teste ao longo do tempo. Como resultado, têm-se que o principal *framework* utilizado é o JUnit [1], presente em 97% dos projetos analisados. O *framework* TesteNG é usado em 11,9% dos projetos. Stefan et al. fizeram um estudo semelhante e constataram que o *framework* JUnit está presente em maior parte dos projetos analisados (STEFAN et al., 2017)

O *framework* JUnit é utilizado para criar testes de unidade para aplicações desenvolvidas na linguagem Java. Esse *framework* possui o método *assert* que, com uma condição *booleana* (verdadeiro ou falso), permite relatar os testes que falharam (NOONAN; PROSL, 2002). Cada caso de teste é criado com base em uma função do sistema de software a ser testado. Para isso, cria-se um conjunto de entradas que execute tal função, sendo preciso criar as possíveis saídas esperadas. Conforme o sistema de software é desenvolvido, é necessário planejar a execução automática realizando as verificações e analisando se as saídas reais coincidem com as saídas esperadas definidas no caso de teste (SOMMERVILLE, 2011).

Em geral, os critérios de teste de sistemas de software combinam três técnicas (MALDONADO et al., 1998): *i)* funcional; *ii)* estrutural; e *iii)* baseada em erros. A técnica funcional relaciona os requisitos do sistema de software para ser a base para a concepção de casos de teste. Aplica-se a todos os níveis da especificação de requisitos e processo de *design*, do sistema ao teste de módulo, podendo ser aplicada a qualquer descrição do comportamento do sistema. Uma especificação funcional é uma descrição do comportamento esperado do sistema de software. A técnica estrutural é utilizada para complementar a funcional, cobrindo a estrutura do código e, portanto, lida com casos não incluídos nos testes funcionais. Os testes estruturais são frequentemente aplicados em duas etapas: *i)* inicialmente, os programadores verificam a completude dos casos de teste, medindo a cobertura do código por meio de ferramentas de cobertura, que indicam a quantidade de código coberto pelos testes de unidade e destacam os elementos descobertos do código; e *ii)* em seguida, os desenvolvedores geram casos de teste que possam contemplar todos os elementos descobertos (BARESI; PEZZE, 2006). Na técnica baseada em erros, o objetivo é verificar se o sistema de software está livre de erros cometidos pelo desenvolvedor. Essa técnica utiliza informações sobre os erros cometidos com frequência pelos desenvolvedores para estabelecer os critérios e requisitos de teste (HOWDEN, 1986).

[1] JUnit. Disponível em: http://junit.org/junit5/

Uma técnica de teste que visa avaliar diretamente o código implementado, é a técnica denominada *Test Driven Development* (TDD) (AMBLER, 2002), assim o estado do código-fonte de um sistema de software é verificado. Por meio dessa técnica, testes de unidade são construídos a fim de verificar o código do sistema. Porém, os testes de unidade também são suscetíveis a alguns problemas, uma vez que casos de teste são gerados manualmente ou automaticamente por ferramentas especializadas (ARCURI et al., 2016) e, mesmo com a geração automatizada dos testes, os casos de teste estão propensos a erros. Para garantir qualidade dos casos de teste, há o critério de teste chamado teste de mutação, que possui a finalidade de adequar os casos de teste para alcançar uma boa cobertura do sistema de software (DELAMARO et al., 2016).

Escrever testes de software eficazes é uma tarefa desafiadora, especialmente quando há ausência de metas de teste bem definidas (ALVIN et al., 2019). Para garantir qualidade dos casos de teste, o objetivo da técnica do teste de mutação é adequar os testes de unidade para alcançar boa cobertura do sistema de software (DELAMARO et al., 2016). Considerando as três técnicas utilizadas por critérios de teste mencionadas anteriormente, o critério de teste de mutação é baseado em erros, visto que esse critério é o foco principal desta pesquisa. A seguir, são descritos os conceitos fundamentais de tal critério.

2.3 Teste de Mutação

O teste de mutação tem sido estudado desde a década de 1970 e foi proposto por DeMillo et al. em 1978 (DEMILLO et al., 1978). Desde então, vários esforços de pesquisa têm sido conduzidos nessa área (JIA; HARMAN, 2011; PAPADAKIS et al., 2015; LIMA et al., 2016; PAPADAKIS et al., 2019; DURELLI et al., 2017). DeMillo et al. descrevem as ideias principais dessa técnica em duas hipóteses: *i)* do programador competente; e *ii)* do efeito acoplamento. A primeira hipótese implica em desenvolver sistemas de software mais próximos da versão correta. Embora possa haver falhas no sistema de software entregue por um programador competente, assume-se que tais falhas são relativamente simples e que podem ser corrigidas com pequenas mudanças. A segunda hipótese propõe que os testes, capazes de revelar erros simples, também são capazes de evidenciar erros mais complexos. Em outras palavras, assume-se que erros complexos, produzidos durante a execução do sistema de software normalmente são acoplados a erros simples (DEMILLO et al., 1978).

Em 1989, Offutt (OFFUTT, 1989) estendeu a hipótese do efeito acoplamento, realizando um estudo empírico para fornecer evidências experimentais sobre essa hipótese. O experimento

utilizou falhas simples e falhas complexas para averiguar se os mesmos casos de teste pudessem identificar essas falhas. As falhas simples são aquelas que apenas uma alteração é realizada (mutantes de primeira ordem). Falhas mais complexas são modeladas pela indução de múltiplas mutações no sistema de software simultaneamente (mutantes de segunda ordem). Os resultados obtidos por meio do experimento mostraram que o efeito acoplamento é válido, uma vez que o mesmo conjunto de casos de teste construídos para detectar mutantes de primeira ordem, na verdade identificou maior porcentagem de mutantes quando aplicado a mutantes de segunda ordem. Dessa forma, os autores concluíram que, ao testar explicitamente as falhas simples, também é possível testar implicitamente falhas mais complexas. Consequentemente, sugere-se que as estratégias de teste baseadas em falhas podem fornecer formas eficazes de realizar os testes de software.

O teste de mutação é uma técnica baseada em defeitos, que consiste em mudar o sistema de software em teste por meio da aplicação de operadores de mutação. Como resultado, aplicando as mudanças feitas pelos operadores de mutação no sistema de software original, obtém-se várias versões diferentes do sistema de software sendo testado que, nesse contexto, são denominadas mutantes (DELAMARO et al., 2016).

Mutantes são gerados alterando a sintaxe do sistema de software sendo testado, como se defeitos estivessem sendo *"semeados"* no código original. Assim, temos regras de transformação sintática, chamadas de "operadores de mutação", que definem como introduzir mudanças sintáticas no sistema de software. Após a geração da mutação, o conjunto de casos de teste construídos inicialmente são executados nos mutantes gerados, com o objetivo de verificar se o resultado obtido em cada teste de unidade corresponde ao resultado esperado. Em geral, enfatiza a criação de casos de teste mais eficazes por meio do refinamento do conjunto de casos de teste inicial (PAPADAKIS et al., 2019).

O teste de mutação utiliza a ideia de que, elaborando casos de teste, falhas semeadas no sistema de software original serão descobertas (DURELLI, 2013). Essas falhas são mudanças sintáticas semeadas por meio de operadores de mutação, que são implementados nas ferramentas de mutação (JUST, 2014; PAPADAKIS et al., 2019) (na Seção 2.3.4 estão descritas algumas ferramentas existentes). Esses operadores são aplicados ao código-fonte e modificam as expressões substituindo, inserindo ou excluindo operações, variáveis, dentre outras modificações. Assim, esses operadores realizam tipos diferentes de mutação, como por exemplo, alteração de

uma condição booleana de *true* para *false*; outros operadores substituem incrementos por decrementos, substituem valores para *null*, dentre outras alterações que são realizadas nos códigos.

Os operadores de mutação são definidos para cada linguagem de programação e cada ferramenta desenvolvida possui seus operadores específicos. Dessa forma, não existe uma maneira única para definir um operador de mutação. Um mesmo operador de mutação pode gerar vários mutantes para uma mesma classe no sistema de software desenvolvido (DELAMARO et al., 2016). Com base na ferramenta de mutação escolhida, que contém os operadores de mutação, um conjunto de instâncias mutantes é gerado e as análises são realizadas. Dessa forma, o objetivo dos testes de unidade são matar os mutantes, consequentemente, os desenvolvedores tendem a projetar casos de teste com o intuito de matar todos os mutantes, o que melhora a qualidade do conjunto de testes (PAPADAKIS et al., 2019). Na Tabela 2.1, é possível observar alguns exemplos de mudanças impostas pelos operadores de mutação (COLES, 2018).

Tabela 2.1 – Exemplos de alterações realizadas.

Condição original	Condição modificada
+	-
-	+
*	/
/	*
<	>
>	<
&	|
|	&
==	!=
!=	==
<=	<
>=	>
<	<=
>	>=
<=	>
>=	<
<	>=
>	<=

Fonte: Elaborado pela autora (2019).

Para exemplificar uma alteração no código de um sistema de software, é possível observar como a mutação é realizada por meio dos Códigos 2.1 e 2.2, escritos em Java. Na função contida no Código 2.1, números positivos são transformados em números negativos. Logo, tendo como entrada um número maior que zero (positivo), ele será alterado para negativo na linha 3; caso contrário, retorna o próprio número informado. Para que a mutação seja realizada, o operador de mutação é aplicado e o código é alterado, conforme alteração no Código 2.2, na linha 2. O mutante substitui o operador relacional *“>”* (maior) por *“<”* (menor). Desse modo, o sistema de software terá uma saída diferente do esperado. Cogitando que um número nega-

tivo seja informado para o código mutante, ele será modificado para positivo com a instrução contida na linha 3, fazendo com que se comporte de forma diferente do especificado.

Código 2.1 – Código original

```
1 int negativo (int x) {
2     if (x > 0) {
3         return (x * -1);
4     } else {
5         return x;
6     }
7 }
```

Código 2.2 – Código mutante

```
1 int negativo (int x) {
2     if (x < 0) {
3         return (x * -1);
4     } else {
5         return x;
6     }
7 }
```

2.3.1 Mutantes Equivalentes

Um mutante é equivalente quando ele possui a mesma semântica do programa original, porém são sintaticamente diferentes, o que leva a uma possível mudança não observável no comportamento (JIA; HARMAN, 2011). Durante a análise dos mutantes equivalentes, a intervenção humana é essencial e é um dos principais obstáculos do teste de mutação. Se a análise dos mutantes sobreviventes não é realizada, não é possível indicar se novos casos de testes são necessários para matar esses mutantes e, consequentemente, indica que a eficácia do conjunto de teste não é confiável, pois não conseguiu matar todos os mutantes (PAPADAKIS et al., 2019). Ou seja, sem a análise, o analista não terá a informação se os mutantes sobreviventes são equivalentes ou se o conjunto de teste é insuficiente.

Os mutantes equivalentes atuam como falsos positivos e podem indicar uma fraqueza no conjunto de testes, mas de fato não deve ser considerada uma fraqueza, pois nenhum teste pode detectá-los (GRÜN et al., 2009). Usando o teste de mutação, esses falsos positivos consomem tempo de um analista, o que contribui para elevar o custo do teste de mutação. Do ponto de vista de avaliar a qualidade do conjunto de teste, esses mutantes não contribuem para sua eficácia.

Conforme mencionado e provado por Budd e Angluin (1982), a identificação de mutantes equivalentes é um problema indecidível. Portanto, mutantes que podem ser equivalentes ao programa original precisam ser analisados manualmente por um testador. A questão é que os

testadores nunca têm certeza se um mutante vivo é simplesmente um mutante “teimoso” (do termo em inglês *stubborn mutant*) - um mutante difícil de matar, porém não equivalente - ou é realmente um mutante equivalente (DURELLI et al., 2017). Yao et al. relataram a relação entre mutantes equivalentes e mutantes teimosos, sugerindo que os desenvolvedores de ferramentas de teste de mutação devem priorizar operadores que geram mutantes teimosos, mas poucos mutantes equivalentes (YAO et al., 2014).

Para exemplificar um mutante equivalente, foi mantido o mesmo exemplo relatado anteriormente, onde o código transforma números positivos em números negativos (Código 2.3). Um mutante equivalente pode ser observado no Código 2.4, linha 3, no qual o operador de mutação substitui o operador de multiplicação “*” pelo operador de divisão “/”, o que produz o mesmo resultado entre os dois códigos sendo testados, uma vez que as saídas são iguais. Dessa forma, o teste de unidade é incapaz de identificar a alteração realizada no código.

Código 2.3 – Código original

```
1 int negativo (int x) {
2     if (x > 0) {
3         return (x * -1);
4     } else {
5         return x;
6     }
7 }
```

Código 2.4 – Exemplo de mutante equivalente

```
1 int negativo (int x) {
2     if (x > 0) {
3         return (x / -1);
4     } else {
5         return x;
6     }
7 }
```

O teste de mutação é caro devido ao número normalmente alto de mutantes que precisam ser executados e analisados. Além da grande quantidade de mutantes gerados para o sistema de software, no qual o suporte de ferramentas automatizadas exige alto custo computacional, determinar manualmente a equivalência entre os códigos original e mutante é uma tarefa dispendiosa (PAPADAKIS et al., 2019; FERRARI et al., 2018). Por exemplo, a realização de testes de mutação, mesmo considerando uma única classe, aplicando os operadores de mutação, resulta em centenas de mutantes (USAOLA; MATEO, 2010). Embora não seja possível encontrar soluções para esses dois obstáculos ao teste de mutação, a literatura está repleta de estudos

que exploram como contorná-los (JIA; HARMAN, 2011; FERRARI et al., 2018; ZHU et al., 2018; PAPADAKIS et al., 2019).

2.3.2 Escore de Mutação

Há uma medida para avaliar a qualidade da atividade de teste de mutação, essa medida é denominada *"escore de mutação"*. Pode-se definir como 'escore de mutação' ou cobertura de mutação a proporção de mutantes que os casos de teste conseguem matar. Em essência, o escore de mutação denota o grau de cobertura dos casos de teste na realização dos objetivos do teste. Assim, o escore de mutação pode ser usado como uma medida de adequação (OFFUTT et al., 1996).

O escore de mutação é dado pela razão dos mutantes mortos, dentre os mutantes gerados pelos operadores de mutação, e dos mutantes não equivalentes (i. e., pela quantidade total de mutantes gerados subtraído da quantidade de mutantes equivalentes). O escore de mutação varia entre 0 (zero) e 1 (um). Nesse caso, depois de aplicada a mutação e o resultado do escore de mutação atingir o valor 1 (100%), obtém-se dois resultados (USAOLA; MATEO, 2010): *i)* um conjunto satisfatório de casos de teste; e *ii)* um programa original confiável, dado que o conjunto de casos de teste não encontrou falhas no sistema de software.

Na maioria dos casos, alcançar um escore de mutação 100% é impraticável. Um dos fatores que contribui para que esse percentual não seja viável é a análise manual dos mutantes, portanto um valor limiar pode ser estabelecido, representando o valor mínimo para o escore de mutação. Dado o programa **P** e o conjunto de casos de teste **T**, o escore de mutação pode ser calculado utilizando a função MS(P, T) (DEMILLO, 1980):

$MS(P,T) = \frac{DM(P,T)}{M(P)-EM(P)}$ onde:

DM(P, T): quantidade de mutantes mortos pelo conjunto de casos de teste T;

M(P): quantidade total de mutantes gerados a partir do programa P;

EM(P): quantidade de mutantes gerados equivalentes a P.

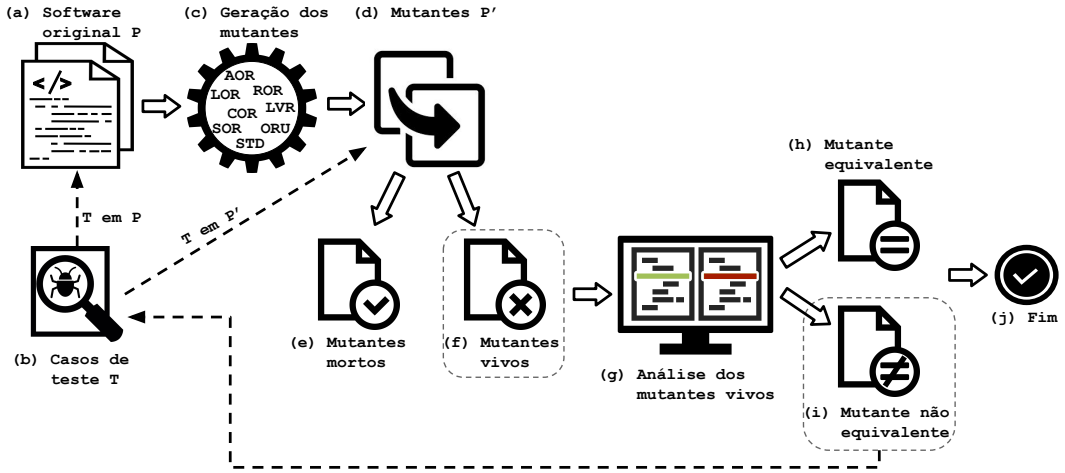
Para que a medida seja calculada de maneira exata, é necessário identificar a quantidade de mutantes equivalentes gerados. Dessa forma, os mutantes não detectados pelos casos de teste (i. e. permaneceram vivos) necessitam ser analisados, para garantir se são equivalentes ao programa original ou se novos testes de unidade devem ser construídos. Mesmo que os mutantes equivalentes não ajudem a melhorar os casos de teste, pois nenhum teste pode detectá-los (GRÜN et al., 2009), é necessário identificá-los para não afetar o cálculo do escore de

mutação e, assim, manter a precisão da métrica. Na prática, usando o escore de mutação como medida de adequação, assume implicitamente que todos os mutantes são de igual valor. Porém, há os mutantes equivalentes que não agregam valor aos casos de teste, ou seja, não afetam a geração de novos casos de teste, mas afeta o cálculo do escore de mutação (PAPADAKIS et al., 2019). Uma vez adotado o teste de mutação, é importante considerar a pontuação da mutação como uma métrica importante para monitorar as atividades de teste.

2.3.3 Processo para realização da Mutação

Tendo em vista complementar as etapas do teste de mutação e elucidar todo o processo do teste, os passos seguidos para aplicação dessa técnica é apresentado na Figura 2.2. Dado um sistema de software **P** em desenvolvimento (Fig. 2.2 (a)), testes de unidade **T** serão aplicados para **P** (Fig. 2.2 (b)), se o resultado é diferente do esperado, o sistema de software possui um defeito e **P** precisa ser corrigido. As correções em **P** serão realizadas e, o passo anterior, é refeito até que erros no código original não sejam detectados pelo conjunto de casos de teste **T**. Nesse ponto, o sistema de software pode conter defeitos, mas não foram revelados pelos casos de teste construídos inicialmente.

Figura 2.2 – Processo de teste de mutação.



Fonte: Adaptado de Ammann e Offutt (2008).

Com o sistema de software original corrigido, o próximo passo consiste em gerar os mutantes **P'**. Dessa forma, conforme pode ser observado na Figura 2.2, o sistema de software original **P** sofre alterações por meio de operadores de mutação (Fig. 2.2 (c)), dando origem a um conjunto de mutantes **P'** (*P'1, P'2, P'3... P'n*) (Fig. 2.2 (d)). Os mesmos casos de teste **T** são executados em cada mutante gerado **P'**. Após a execução dos testes de unidade, há duas

instâncias dos mutantes, ou seja, os resultados apresentam os mutantes **P'** mortos (Fig. 2.2 (e)) pelos casos de teste e os mutantes que permaneceram vivos (Fig. 2.2 (f)).

No caso de algum mutante **P'** ter sobrevivido aos testes de unidade, uma análise é necessária para determinar se há equivalência desse mutante **P'** com o código original **P** (Fig. 2.2 (g)). Por meio da análise dos mutantes sobreviventes, é possível indicar o status desses mutantes. Um dos status é estabelecer que os mutantes sobreviventes **P'** são equivalentes ao código original (Fig. 2.2 (h)); caso todos os mutantes analisados sejam equivalentes ao sistema de software original, o processo é finalizado (Fig. 2.2 (j)). Caso contrário, ou seja, em decorrência da existência de mutantes **P'** não equivalentes a **P** (Fig. 2.2 (i)), os casos de teste **T** devem ser incrementados com novos testes de unidade para matar os mutantes não equivalentes ao código original **P** e permaneceram vivos. Com os novos casos de teste construídos, o processo retorna a fase inicial e refaz os passos novamente. O novo conjunto de testes **T** é executado para tentar detectar os mutantes **P'** que permaneceram vivos, até que todos os mutantes sejam mortos ou quando o analista determina que os casos de teste **T** são satisfatórios (Fig. 2.2 (j)) e o processo é finalizado.

As condições necessárias para um teste **T** matar um mutante **P'** podem ser descritas usando o modelo de alcance, infecção e propagação (RIP - ***R**eachability, **I**nfection e **P**ropagation*) (AMMANN; OFFUTT, 2008; KUSHIGIAN et al., 2019):

- **Alcançabilidade** (*Reachability*): a localização alterada no código mutante deve ser executada;
- **Infecção** (*Infection*): o estado do programa mutante deve estar incorreto após o local com defeito ser executado, ou seja, seja diferente do estado original;
- **Propagação** (*Propagation*): o estado de execução infectado deve ser propagado para alguma saída observável.

2.3.4 Ferramentas de apoio ao teste de mutação

É importante salientar que, tradicionalmente, não existe uma maneira direta de definir os operadores de mutação para uma dada linguagem. Usualmente, os operadores de mutação são projetados tendo como base a experiência no uso de cada linguagem (DELAMARO et al., 2016). Assim, diversas ferramentas para auxiliar e apoiar o teste de mutação foram desenvolvidas, cada uma implementando um conjunto diferente de operadores de mutação. De acordo

com Jia e Hamman (JIA; HARMAN, 2011) e Kintis et al. (KINTIS et al., 2016), as principais ferramentas utilizadas para o teste de mutação são:

- **Mothra**[2]: ferramenta desenvolvida por pesquisadores da Purdue University e do Georgia Institute of Technology para programas em Fortran;
- **Proteum**[3]: ferramenta desenvolvida para dar suporte ao critério Análise de Mutantes para programas desenvolvidos em C;
- **JesTer**[4]: ferramenta desenvolvida para dar suporte à linguagem Java com integração com casos de teste utilizando o JUnit;
- **Jumble**[5]: ferramenta desenvolvida para a linguagem Java;
- **muJava**[6]: ferramenta desenvolvida para Java com a colaboração entre duas universidades, Korea Advanced Institute of Science e Technology (KAIST) e George Mason University;
- **MuClipse**[7]: desenvolvida a partir da MuJava, foi exportada para o IDE Eclipse como um *plugin*;
- **Pit**[8]: ferramenta desenvolvida para a linguagem Java;
- **Major**[9]: ferramenta desenvolvida para prover o suporte de teste de mutação para Java.

As ferramentas de mutação citadas anteriormente, são utilizadas para realizar a mutação do código do sistema de software a ser testado. Essas ferramentas não possuem um mecanismo de análise dos mutantes para verificação da equivalência. Diante da ausência de um ambiente que contemple desde a geração da mutação até à análise, a DiffMutAnalyze foi desenvolvida para esse fim.

Esta pesquisa objetiva focar apenas na linguagem de programação Java. Assim, apenas ferramentas desenvolvidas para teste de mutação em programas Java foram consideradas. Kintins et al. (KINTIS et al., 2016) identificaram as ferramentas de teste de mutação mais utilizadas para a linguagem Java e fizeram um experimento comparando-as. A análise foi realizada nas ferramentas: muJava, Major e Pit. Na Tabela 2.2, são mostrados alguns resultados do experimento

[2] Mothra. Disponível em: https://cs.gmu.edu/ offutt/rsrch/mut.htmlMOTHRA
[3] Proteum. Disponível em: http://napsol.icmc.usp.br/pt-br/node/122
[4] JesTer. Disponível em: http://jester.sourceforge.net/
[5] Jumble. Disponível em: http://jumble.sourceforge.net/
[6] muJava. Disponível em: https://cs.gmu.edu/ offutt/mujava/
[7] MuClipse. Disponível em: http://muclipse.sourceforge.net/
[8] Pit. Disponível em: http://pitest.org/
[9] Major. Disponível em: http://mutation-testing.org/

realizado por Kintins et al. A ferramenta Major gerou 808 mutantes, enquanto as ferramentas Pit e muJava geraram 662 e 1.854, respectivamente. Considerando a geração de mutantes equivalentes, as ferramentas Major, muJava e Pit geraram, respectivamente, 12%, 11% e 6% de mutantes equivalentes. Quanto à quantidade de casos de teste, essas três ferramentas requerem 97, 138 e 80 novos casos de teste, respectivamente (KINTIS et al., 2016).

Tabela 2.2 – Resultado da análise manual dos mutantes realizada por Kintis et al.

Descrição	**Major**	**Pit**	**muJava**
Mutantes gerados	808	662	1854
Mutantes equivalentes	94	43	203
Testes necessários	97	80	138

Fonte: Kintis et al. (2016).

Para este trabalho, a ferramenta Major foi escolhida para realizar a mutação do projeto inserido na ferramenta DiffMutAnalyze. Como este trabalho objetiva analisar os mutantes sobreviventes, é preciso obter os arquivos *.java* de cada mutante para que os mesmos possam ser comparados com o código original, para verificação da equivalência entre os códigos. Na ferramenta Pit, a execução da mutação dos códigos é rapida, porém, ela não salva em disco o código dos mutantes gerados. Dessa forma, não foi possível utilizar a Pit. Como observado na Tabela 2.2, na ferramenta muJava, foi gerada a maior quantidade de mutantes, o que aumenta o custo computacional da geração da mutação. Considerando que nesta pesquisa, o objetivo é verificar o custo da análise manual, a ferramenta Major atende aos requisitos da pesquisa, uma vez que os mutantes são salvos em disco, gerando uma menor quantidade de mutantes, quando comparada à quantidade de mutantes gerados pela muJava, o que contribui para a redução do custo computacional do teste de mutação.

Major é uma ferramenta de mutação integrada ao compilador Java e não requer uma estrutura de análise de mutação específica. Por isso, ela pode ser usada em qualquer ambiente baseado em Java. Major manipula a árvore sintática abstrata (*AST - Abstract Syntax Tree*) analisando o código-fonte do programa em teste. Como Major opera na AST, que representa o código desenvolvido, evita gerar mutantes que não podem ser mapeados para uma localização específica no código original (JUST et al., 2011; JUST, 2014).

Como o teste de mutação faz alterações no programa a ser testado, há algumas regras para tais modificações. Essas regras são implementadas nos operadores de mutação, esses operadores são projetados para realizar a transformação do código, onde são aplicados para modificar variáveis, expressões, operadores entre outras alterações. Um ponto importante da ferra-

menta de mutação é possuir um conjunto de operadores de mutação adequados. Um conjunto de operadores bem projetados resulta em testes potentes; caso contrário, pode resultar em testes ineficazes, por exemplo, uma ferramenta que possui operadores que alteram os predicados para *true* ou *false* (AMMANN; OFFUTT, 2008). Os operadores que compõem a ferramenta Major e os exemplos das mudanças impostas por esses operadores estão descritos na Tabela 2.3 (JUST et al., 2011; JUST, 2014; JUST, 2017). Além disso, exemplos de mutantes gerados por alguns desses operadores de mutação, podem ser visualizados no Apêndice A.1.

Tabela 2.3 – Operadores que a ferramenta Major implementa.

Operador	Exemplo
AOR (*Arithmetic Operator Replacement*)	a + b $\rightarrow a - b$
LOR (*Logical Operator Replacement*)	a ^ b $\rightarrow a \vert b$
COR (*Conditional Operator Replacement*)	a || b $\rightarrow a \&\& b$
ROR (*Relational Operator Replacement*)	a == b $\rightarrow a >= b$
SOR (*Shift Operator Replacement*)	a » b $\rightarrow a << b$
ORU (*Operator Replacement Unary*)	-a $\rightarrow \sim a$
EVR *(Expression Value Replacement)*	return a $\rightarrow return0$
	int a = b $\rightarrow inta = 0$
LVR (*Literal Value Replacement*)	$0 \rightarrow 1$
	$1 \rightarrow -1$
	$1 \rightarrow 0$
	true $\rightarrow false$
	false $\rightarrow true$
	"Hello"$\rightarrow$" "
STD (*Statement Deletion Operator*)	return a $\rightarrow < no - op >$
	break $\rightarrow < no - op >$
	continue $\rightarrow < no - op >$
	foo(a,b) $\rightarrow < no - op >$
	a = b $\rightarrow < no - op >$
	++a $\rightarrow < no - op >$
	- -a $\rightarrow < no - op >$

Fonte: Just (2017)

2.4 Considerações finais

Neste capítulo, foi apresentada breve introdução ao teste de software, mais especificamente sobre o teste de mutação. Desde a origem desse critério de teste, a pesquisa em teste de mutação vem gerando muitos esforços de pesquisa. A análise dos mutantes equivalentes é o passo que requer mais intervenção humana e um dos maiores obstáculos para a aplicação do teste de mutação. Desse modo, o problema dos mutantes equivalentes é uma área de pesquisa

recorrente, no qual os pesquisadores buscam alternativas de mitigar o esforço gasto na análise desses mutantes.

Nesse contexto, neste capítulo, foi mostrada uma visão geral sobre a aplicação do teste de mutação e como o esforço humano é necessário para realizar as análises dos mutantes sobreviventes, além de ser, em alguns casos, um esforço gasto sem agregar valor ao conjunto de casos de teste, em função dos mutantes equivalentes, pois esses mutantes não requerem melhoria nos casos de teste. Para tentar mitigar esses e outros problemas, pesquisadores buscam reduzir o custo do teste de mutação, seja utilizando técnicas para reduzir a quantidade de mutantes gerados e, consequentemente, reduzir a quantidade de mutantes equivalentes, seja utilizando técnicas para tentar determinar a equivalência entre o código original e o mutante (JIA; HARMAN, 2011; PAPADAKIS et al., 2019).

Considerando os conceitos apresentados, é possível compreender o contexto da análise dos mutantes sobreviventes. Em particular, nesta pesquisa, onde o objetivo é identificar o custo da análise dos mutantes equivalentes, comparando a análise manual com a análise guiada por meio de uma ferramenta de apoio computacional para auxiliar os analistas a identificar a equivalência entre os códigos analisados. Nos próximos capítulos serão descritos a ferramenta DiffMutAnalyze (Capítulo 3) e a metodologia (Capítulo 4) adotada para as análises.

3 DIFFMUTANALYZE

3.1 Considerações iniciais

O teste de mutação refere-se ao processo de análise de mutação para suportar o conjunto de casos de teste, a fim de melhorar a qualidade dos testes de unidade produzidos. Assim, casos de teste capazes de distinguir os comportamentos dos mutantes e do código original são indicados a identificar possíveis falhas no sistema de software (PAPADAKIS et al., 2019). Há os mutantes equivalentes, nos quais não podem ser identificados pelos casos de teste (GRÜN et al., 2009), dessa forma, eles não contribuem para melhorar o conjunto de casos de teste. A medida que os mutantes equivalentes não são descobertos pelos casos de teste, eles necessitam ser identificados manualmente. Pois esses mutantes, além de não contribuir para aumentar a qualidade do conjunto de casos de teste, afetam o cálculo do escore de mutação.

Analisar os mutantes não é uma tarefa trivial, pois demanda tempo e trabalho de um analista. Assim, nesta pesquisa, a ferramenta DiffMutAnalyze (BOTELHO et al., 2018) foi desenvolvida para facilitar e auxiliar o analista de teste a identificar os mutantes equivalentes. Dessa forma, a utilização da ferramenta DiffMutAnalyze auxilia no teste de mutação com as seguintes contribuições:

- Realizar todo o processo da mutação em um ambiente único;
- Diminuir o tempo gasto em todo o processo;
- Utilizar projetos diretamente do GitHub ou um arquivo .zip de um projeto;
- Realizar a mutação e aplicar os casos de teste nos mutantes gerados;
- Analisar a possível equivalência dos mutantes sobreviventes;
- Disponibilizar a visualização do local onde a mutação foi realizada;
- Definir a equivalência entre os códigos original e mutante;
- Indicar o grau de dificuldade em analisar cada mutante individualmente;
- Contabilizar o tempo gasto com a análise;
- Calcular o escore de mutação; e
- Visualizar relatórios com informações da mutação e da análise.

Considerando as ferramentas de mutação existentes, elas realizam a mutação no código do sistema e indica quais mutantes foram mortos pelos casos de teste, porém não conta com um ambiente para análise da equivalência dos mutantes sobreviventes, dessa forma, a DiffMutAnalyze diferencia no fato de que a análise dos mutantes é disponibilizada logo após a realização da mutação e execução dos casos de teste. Além disso, por meio da indicação do grau de dificuldade da análise, é possível perceber os operadores que geram os mutantes mais difíceis de serem analisados e quais os que demandam maior tempo de um analista, uma vez que a DiffMutAnalyze contabiliza o tempo de cada análise.

3.2 Proposta da DiffMutAnalyze

O objetivo da ferramenta de apoio computacional denominada DiffMutAnalyze é auxiliar os analistas na identificação dos mutantes equivalentes. A principal motivação para o desenvolvimento dessa ferramenta é o alto custo da análise dos mutantes para verificação da equivalência (FERRARI et al., 2018). Assim, a ferramenta possui o propósito de localizar a alteração realizada pelo operador de mutação e apresentar ao analista a comparação, lado a lado, da diferença entre o código original e o mutante.

DiffMutAnalyze é uma ferramenta que gerencia e controla os mutantes analisados. Uma tabela é formada com todos os mutantes gerados pela ferramenta. Nessa tabela é possível visualizar os mutantes a serem analisados de acordo com seu *status*, ou seja, aqueles que foram inspecionados e a equivalência foi determinada, assim como os mutantes não visualizados pelo usuário. Assim, o analista consegue gerenciar e controlar a análise desses mutantes. Após a inspeção dos códigos (original e mutante) o analista determina a equivalência de ambos e avalia a análise de acordo com seu grau de dificuldade. Finalizada a avaliação, o analista procede com a análise do próximo mutante, uma vez que todos os mutantes foram inseridos previamente.

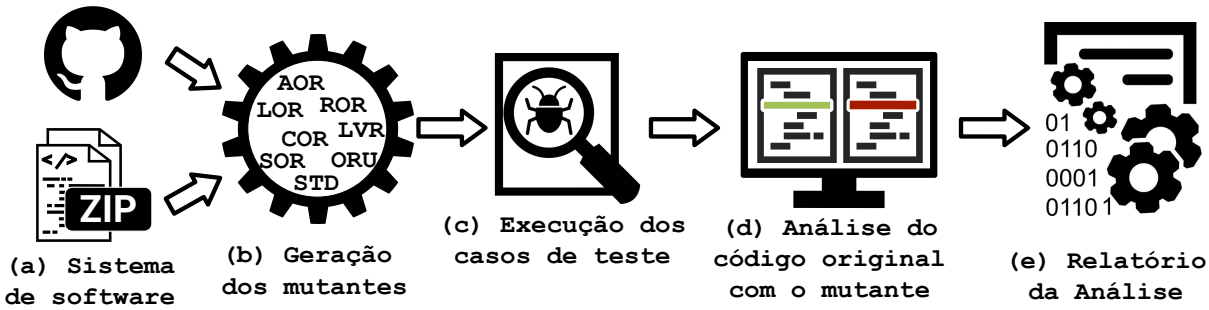
Caso haja dúvida da equivalência de um mutante com seu código original correspondente, é permitido prosseguir para a próxima análise e voltar posteriormente para esse mutante. Desse modo, na visualização da tabela dos mutantes, é possível identificar os mutantes analisados e determinada sua equivalência, os inspecionados, porém sua equivalência não foi determinada e os mutantes não visualizados. Assim, é possível escolher qual mutante será o próximo a ser inspecionado.

O tempo gasto pelo analista em analisar os mutantes é contabilizado automaticamente em cada análise e, ao final, é possível obter o tempo total gasto em todas as análises. Um

relatório é exibido ao analista contendo todos os dados da análise, como quantidade de mutantes equivalentes, avaliação do grau de dificuldade e tempo gasto com a análise. Mais detalhes do funcionamento da ferramenta pode ser visto na Seção 3.4.

DiffMutAnalyze [1] é uma ferramenta que possui um design web para ser executada em um servidor local, permitindo que os projetos inseridos sejam salvos localmente, garantindo o sigilo e a segurança do código do sistema de software a ser testado. Na Figura 3.1, é possível observar os procedimentos realizados na ferramenta para a realização da mutação. Inicialmente, o sistema de software deve ser inserido na DiffMutAnalyze. A DiffMutAnalyze suporta somente sistemas desenvolvidos em Java. É possível inserir o projeto a ser testado por duas maneiras distintas (Figura 3.1(a)): *i)* por meio de um link do GitHub do projeto do sistema; ou *ii)* por meio da inserção de uma pasta compactada do projeto do sistema de software, em formato ZIP (.zip). Para verificar se os casos de teste conseguem identificar os mutantes gerados, é necessário que o projeto inserido na ferramenta contenha os casos de teste criados inicialmente, para que os mesmos sejam executados quando os mutantes forem gerados. Os casos de teste são reconhecidos pela ferramenta por meio da anotação "`@Test`". Com o projeto inserido na ferramenta, a mutação é realizada no sistema de software (Figura 3.1(b)), aplicando os operadores contidos na ferramenta de mutação Major, uma vez que essa ferramenta foi integrada à DiffMutAnalyze.

Figura 3.1 – Visão geral da DiffMutAnalyze.



Fonte: Elaborado pela autora (2018).

Como o principal objetivo do teste de mutação é avaliar a qualidade dos casos de teste, o próximo passo é executar os casos de teste nos mutantes gerados (Figura 3.1(c)). Ao executar os casos de teste, os mutantes são classificados em mortos e vivos, caso todos os mutantes sejam mortos pelos casos de teste, não é necessário realizar nenhuma análise dos códigos, pois o conjunto de testes de unidades criados é suficiente para detectar todos os mutantes gerados; caso contrário, é preciso verificar se os mutantes sobreviventes são equivalentes ao sistema de

[1] Disponível em: https://github.com/PqES/DiffMutAnalyze

software original ou se é necessária a criação de novos testes de unidade, capazes de identificar e “matar” os mutantes que permaneceram vivos. Desse modo, novos casos de teste são inseridos no projeto sendo testado e a DiffMutAnalyze realiza o processo da mutação novamente considerando os novos casos de teste.

Para os mutantes sobreviventes aos casos de teste, uma análise é necessária, esses mutantes são disponibilizados para análise em conjunto com o código original do sistema de software correspondente a esses mutantes (Figura 3.1(d)). A análise é realizada por meio da visualização da comparação dos códigos original e mutante, de maneira que a alteração realizada, no momento da mutação, seja identificada e evidenciada para o analista, facilitando a análise da possível equivalência. Por meio dessa análise, no qual identifica-se a necessidade de criar novos casos de teste, é possível melhorar a qualidade do conjunto de testes do projeto, fazendo com que seja capaz de identificar prováveis falhas durante o desenvolvimento do software, sem que essas falhas se propaguem até o final do desenvolvimento e sejam descobertas somente quando o projeto estiver finalizado.

Conforme os mutantes são analisados e comparados com o código original, o analista deverá registrar algumas informações dessa análise, indicando se existe equivalência entre os códigos, além de informar o grau de dificuldade em inspecionar o mutante, outra informação registrada é o tempo gasto em cada análise, no qual a ferramenta DiffMutAnalyze contabiliza automaticamente. Durante a análise, caso seja necessário o analista realizar alguma pausa na análise, ele pode pausar o tempo na DiffMutAnalyze, há um botão disponível para esse fim. O botão pode ser acionado novamente para continuação da análise, assim, o tempo é medido conforme o tempo real gasto pelo analista. Ao informar o grau de dificuldade em analisar os mutantes, é possível identificar a complexidade da mutação. Ao realizar a análise de todos os mutantes sobreviventes aos casos de teste, a ferramenta DiffMutAnalyze disponibiliza relatórios com gráficos das análises (Figura 3.1(e)), como gráficos da quantidade de mutantes equivalentes identificados, tempo gasto com as análises, grau de dificuldade em analisar os mutantes, dentre outras informações disponíveis para consulta em relação a cada sistema de software inserido na ferramenta. A seguir, está descrito o fluxo de execução da ferramenta.

3.3 *Overview* da Ferramenta

DiffMutAnalyze foi desenvolvida na linguagem de programação Java [2] e suporta somente projetos implementados na linguagem Java para realização do teste de mutação. Para que a mutação seja realizada, a ferramenta Major foi integrada à DiffMutAnalyze, uma vez que os mutantes gerados por essa ferramenta são armazenados em disco e, para a análise dos mutantes, é necessário que os mutantes sejam acessados pela DiffMutAnalyze. Para auxiliar no desenvolvimento da ferramenta, o *framework* Spring Boot (WALLS, 2011) foi utilizado. Ao utilizar o Spring Boot, é possível gerenciar as dependências do projeto por meio das dependências do Maven (WALLS, 2011). Desse modo, para utilizar a ferramenta DiffMutAnalyze, é necessário que o Maven esteja configurado na máquina local. Com a aplicação configurada, a ferramenta DiffMutAnalyze é inicializada através do comando "`mvn spring-boot:run`" a partir do diretório raiz da ferramenta. Além disso, a ferramenta utiliza o banco de dados MySQL para armazenar os dados da aplicação.

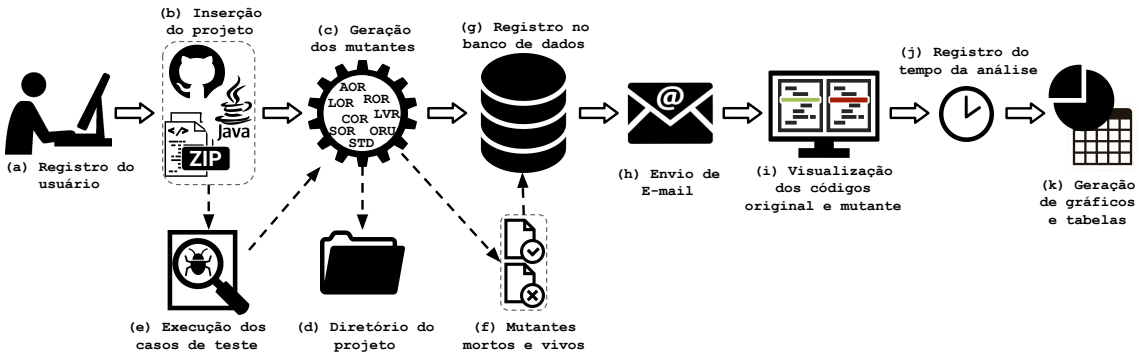
Na Figura 3.2, está representada a arquitetura da DiffMutAnalyze. O primeiro passo é a realização do registro do usuário (Figura 3.2(a)). Ao inserir o projeto Java do sistema de software (Figura 3.2(b)), contendo o código do sistema de software e os casos de teste construídos inicialmente, os passos seguintes são realizados automaticamente pela ferramenta DiffMutAnalyze. Uma vez inserido o projeto a ser testado, a ferramenta de mutação Major é executada pela DiffMutAnalyze e a mutação é gerada no código inserido (Figura 3.2(c)). Ao iniciar o processo da mutação, uma pasta é criada (Figura 3.2(d)) localmente no computador em que a DiffMutAnalyze está sendo executada. Nessa pasta estão contidos o código do sistema de software original e os mutantes gerados. É necessário salvar esses códigos em disco para que os códigos possam ser obtidos e disponibilizados para a análise.

Ao gerar os mutantes, os casos de teste são executados nesses mutantes (Figura 3.2(e)) e os mutantes são registrados como mortos ou vivos (Figura 3.2(f)) no banco de dados (Figura 3.2(g)). Finalizada a mutação, um e-mail é enviado (Figura 3.2(h)) ao usuário cadastrado, informando a finalização da mutação e disponibilizando um link para a análise. Além disso, o usuário cadastrado, pode acessar a análise por meio da DiffMutAnalyze. Ao entrar no módulo de análise dos mutantes, os mutantes que necessitam ser analisados são disponibilizados na DiffMutAnalyze (Figura 3.2(i)). Os códigos mutante e original são lidos pela ferramenta e o local da alteração realizada é evidenciado. Durante a análise, o tempo é contabilizado auto-

[2] Disponível em: https://www.java.com/pt_BR

maticamente (Figura 3.2(j)). As informações da mutação e da análise são exibidas por meio de gráficos e tabelas no módulo de relatórios (Figura 3.2(k)).

Figura 3.2 – Fluxo de execução da ferramenta DiffMutAnalyze.



Fonte: Elaborado pela autora (2019).

3.4 Ferramenta DiffMutAnalyze

Para iniciar o teste de mutação por meio da DiffMutAnalyze, o primeiro passo é efetuar o cadastro na DiffMutAnalyze. Conforme pode ser observado na Figura 3.3, é preciso informar o nome, e-mail e senha para realizar o cadastro do usuário na ferramenta.

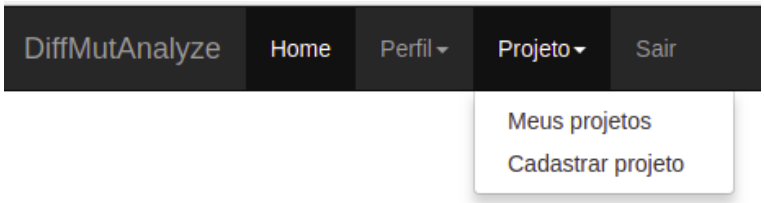
Figura 3.3 – Cadastro de Usuário.

Fonte: Imagem da ferramenta DiffMutAnalyze (2018).

Com as informações de acesso cadastradas, o passo seguinte para realização da mutação, é inserir o projeto na DiffMutAnalyze. O usuário navegará pelo menu na barra superior da ferramenta, acessando a opção “Cadastrar projeto” (Figura 3.4).

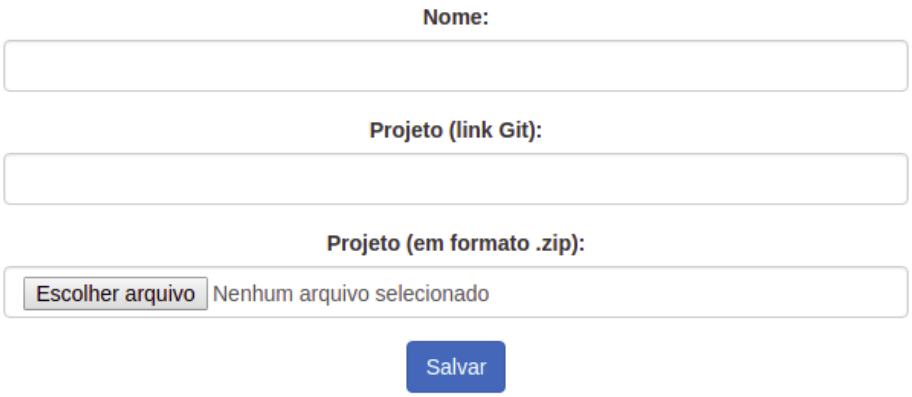
Para inserir o projeto do sistema de software na DiffMutAnalyze, é preciso informar o nome para o mesmo e inserir o projeto na ferramenta por duas formas distintas: *i)* por meio do link do projeto do GitHub; ou *ii)* por meio da pasta compactada do projeto (em formato .zip) (Figura 3.5), ou seja, basta inserir o projeto a ser testado em alguma das duas formas disponíveis.

Figura 3.4 – Menu de acesso.



Fonte: Imagem da ferramenta DiffMutAnalyze (2018).

Figura 3.5 – Cadastro do Projeto a ser testado.



Fonte: Imagem da ferramenta DiffMutAnalyze (2018).

Após inserido o projeto a ser testado, os operadores da Major são executados e os mutantes são gerados e armazenadas dentro do diretório criado para o projeto em teste. Após gerar os mutantes, o próximo passo é executar os casos de teste existentes e identificar quais mutantes podem ser mortos pelos casos de testes, examinando assim, a qualidade dos testes de unidade construídos previamente. Ao final da execução dos casos de teste, os mutantes são classificados entre mortos e vivos. Os mutantes vivos necessitam ser analisados, comparando-os com o código original, para verificar se ambos são equivalentes ou se novos testes de unidade são necessários para encontrar a alteração realizada. Dessa forma, um e-mail é enviado ao usuário cadastrado (Figura 3.6) informando que o projeto foi cadastrado e disponibiliza um link para a análise dos mutantes. O link de acesso à análise do projeto tem a função de disponibilizar a análise para terceiros com acesso ao servidor que se encontra a ferramenta e sua base de dados.

Além do link enviado por e-mail, para verificar os mutantes sobreviventes e analisá-los sobre sua equivalência, o usuário deverá acessar o menu “Meus projetos” (Figura 3.4) e navegar até o projeto a ser analisado, conforme Figura 3.7. Para acessar o módulo de análise dos mutantes, selecione a opção “Análise” (Figura 3.7) e a ferramenta será direcionada para a tela de análise dos mutantes.

Figura 3.6 – E-mail enviado ao final da execução da mutação.

Fonte: Imagem da ferramenta DiffMutAnalyze (2018).

Figura 3.7 – Acesso à análise dos mutantes.

Fonte: Imagem da ferramenta DiffMutAnalyze (2018).

Os mutantes destinados à análise são transpostos em uma tabela (Figura 3.8) para verificação da possível equivalência entre tais mutantes e o seu código original correspondente. Nessa tabela é possível visualizar o *status* que os mutantes se encontram. As classes do projeto são inseridas em cada linha dessa tabela e os mutantes são numerados e inseridos nas colunas na tabela, conforme a ferramenta Major gera os mutantes e os inserem em pastas numeradas, assim, cada número representa um mutante. Por meio dessa tabela, é possível verificar a situação da análise e controlar a quantidade de mutantes analisados e os que devem ser verificados, uma vez que a ferramenta define o *status* dos mutantes através de cores que indicam a situação de cada mutante. Assim, é possível controlar a quantidade de mutantes analisados e os que ainda deverão ser analisados.

Figura 3.8 – Tabela dos mutantes.

Fonte: Imagem da ferramenta DiffMutAnalyze (2018).

As cores que representam o *status* dos mutantes são: *i)* azul; *ii)* verde; e *iii)* vermelho. A cor azul representa todos os mutantes não analisados; a cor verde representa os mutantes sem pendências em sua análise, ou seja, os códigos foram comparados e indicado se existe a equivalência entre eles, além de determinar o grau de dificuldade em inspecionar o dado mutante; a cor vermelha representa os mutantes somente visualizados, sem que a equivalência tenha sido determinada e o grau de dificuldade definido, dado que é permitido navegar entre os mutantes sem que seja obrigatório definir a situação do mutante. Como exemplo dessa visualização, como pode ser observado na Figura 3.8, a classe "Merge.java" possui sete mutantes, sendo três mutantes analisados e sem pendências (mutantes 1, 2 e 4), dois mutantes com alguma pendência (mutantes 3 e 5) e dois mutantes não analisados (mutantes 6 e 7).

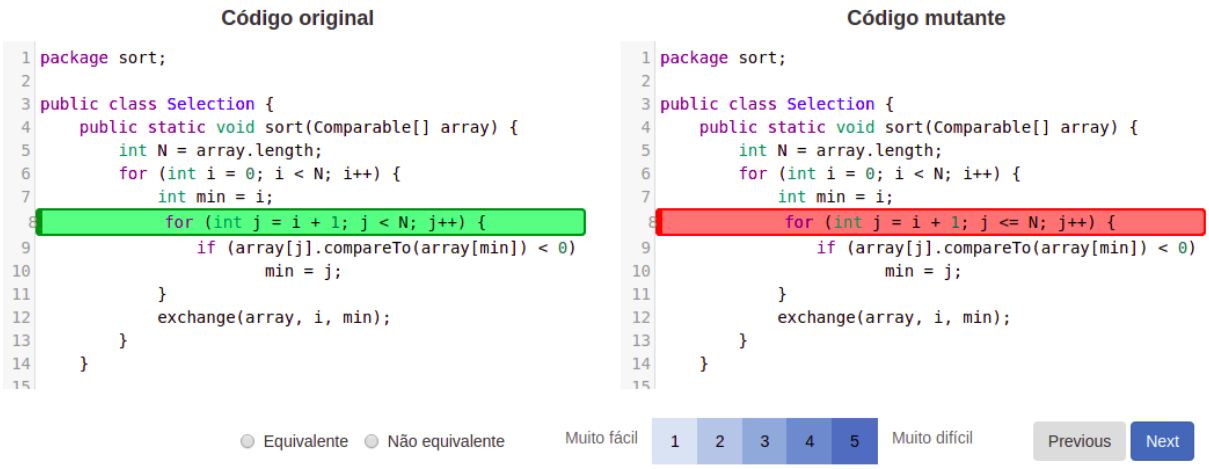
Para inspecionar um mutante específico, o usuário poderá clicar diretamente sobre o retângulo que o representa na tabela. Além disso, é possível buscar pelo mutante a ser analisado ou que possui alguma pendência da análise, essa ação é realizada por meio do botão "Buscar mutante não analisado", no qual, a cada vez que o botão é selecionado, esses mutantes são disponibilizados para inspeção.

Uma vez que os mutantes são disponibilizados para análise, os códigos (original e mutante) podem ser visualizados e comparados lado a lado e a alteração que gerou o código mutante é evidenciada (Figura 3.9). A cor verde representa a linha do código original que possui a instrução original antes da mutação e a cor vermelha é representada no código mutante, evidenciando o local da alteração. Como o local exato da mutação é indicado, é possível considerar ganho no tempo da análise, pois facilita a inspeção do código, sem a necessidade de procurar pela alteração, visto que em sistemas de softwares que possuem grande quantidade de linhas de código, a busca pela alteração pode se tornar dispendiosa.

Após a inspeção dos códigos, é necessário indicar se existe a equivalência entre os códigos original e mutante e indicar o grau de dificuldade da análise. Abaixo da tela de visualização dos códigos original e mutante, há uma barra para inserir as informações da análise (Figura 3.9). Sendo assim, o usuário informa a possível equivalência do mutante e indica o grau de dificuldade da análise. Além disso, nessa barra, é possível navegar entre os mutantes da tabela por meio dos botões "*Previous*" e "*Next*", seguindo para o próximo mutante ou retornando ao mutante anterior.

O tempo gasto com a análise dos mutantes é contabilizado automaticamente pela Diff-MutAnalyze e pode ser controlado por meio de um botão disponível na ferramenta. Como pode

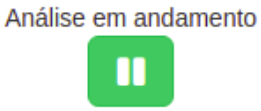
Figura 3.9 – Visualização dos códigos original e mutante.



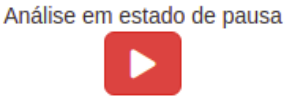
Fonte: Imagem da ferramenta DiffMutAnalyze (2018).

ser observado na Figura 3.10(a), quando a análise está em andamento, o tempo está sendo cronometrado; quando a análise necessita ser pausada, o botão é acionado e a contagem do tempo é pausada (Figura 3.10(b)).

Figura 3.10 – Marcação do tempo da análise dos mutantes.



(a) Tempo da análise em andamento.



(b) Pausa do tempo da análise.

Fonte: Imagem da ferramenta DiffMutAnalyze (2018).

Finalizada a análise dos mutantes, é possível consultar os dados da análise por meio de relatórios. Para visualizar o relatório, o usuário deverá navegar pelo menu através da opção “Meus Projetos” (Figura 3.4) e selecionar a opção “Relatórios” (Figura 3.7) do projeto analisado. Na DiffMutAnalyze, as informações são disponibilizadas por meio de gráficos (Figura 3.11) e tabelas com os dados da análise, assim, é possível obter informações como: escore de mutação; quantidade de mutantes mortos pelos casos de teste e os que permaneceram vivos; quantidade de mutantes determinados como equivalentes e de mutantes não equivalentes; tempo total gasto com a análise de todos os mutantes e o tempo médio por mutante; grau de dificuldade da análise, dentre outras informações.

Figura 3.11 – Relação entre a quantidade de mutantes mortos e vivos e dos mutantes equivalentes e não equivalentes.

(a) Mutantes mortos vs. vivos.

(b) Mutantes equivalentes vs. não equivalentes.

Fonte: Imagem da ferramenta DiffMutAnalyze (2018).

3.5 Considerações finais

Neste capítulo, foi apresentada a Ferramenta DiffMutAnalyze para o auxílio na análise e na identificação dos mutantes equivalentes, realizando a inspeção dos códigos original e mutante. Nessa ferramenta, é possível realizar todo o processo do teste de mutação, uma vez que a ferramenta de mutação Major está integrada à DiffMutAnalyze. Dessa forma, os mutantes são gerados e disponibilizados para a análise da possível equivalência entre eles e o código original correspondente ao sistema de software em teste. Além disso, é possível inserir informações da análise, como grau de dificuldade em analisar os mutantes e visualizar o tempo gasto com a análise. Diante do objetivo proposto que é investigar o custo da análise dos mutantes proposto neste projeto de pesquisa, a DiffMutAnalyze foi desenvolvida com o propósito de ser utilizada no experimento a fim de comparar o uso dela com a análise manual. Dessa forma, no próximo capítulo, estão descritos os procedimentos dos experimentos realizados.

4 METODOLOGIA

4.1 Considerações iniciais

Muitos estudos buscam soluções para automatizar todo o processo de mutação (USAOLA; MATEO, 2010; JIA; HARMAN, 2011; ZHU et al., 2018; PAPADAKIS et al., 2019) e alavancar resultados que reduzam o custo do teste de mutação (FERRARI et al., 2018), além de viabilizar a aplicação desse critério de teste, devido ao alto custo computacional e esforço humano utilizado. Neste capítulo, o objetivo é descrever como foi realizada a investigação do custo humano da análise para identificar se há equivalência entre um mutante e seu código original. Além disso, está descrito como a ferramenta DiffMutAnalyze pretende auxiliar os analistas na identificação dos mutantes equivalentes. A seguir, está descrita a metodologia aplicada para realizar o experimento para obter os resultados desta pesquisa.

4.2 Metodologia

4.2.1 Tipo de Pesquisa

Os estudos baseados em experimentos na Engenharia de Software consideram um conjunto de princípios para a análise dos resultados. No experimento, há elementos considerados principais, sendo eles as variáveis, os objetos, os participantes, o contexto do experimento, as hipóteses e o tipo de projeto do experimento. Em um paradigma experimental, as soluções existentes são observadas e novas soluções são propostas para contribuir com a melhoraria do objeto estudado (WOHLIN et al., 2012).

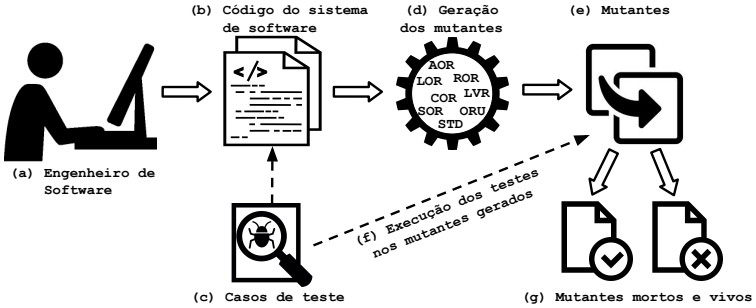
Assim, neste projeto de pesquisa, foram realizados dois experimentos com diferentes participantes. No primeiro experimento, foi realizado um projeto piloto para verificar o uso da ferramenta DiffMutAnalyze para auxiliar a análise dos mutantes. No segundo experimento, o objetivo foi avaliar o custo da análise da identificação dos mutantes equivalentes. Para a realização dos experimentos, inicialmente os participantes obtiveram uma aula sobre os principais conceitos do teste de mutação, para posteriormente realizar a mutação durante o experimento realizado em laboratório. Uma característica do experimento realizado em laboratório é a obtenção do total domínio do processo realizado. Para alcançar bons resultados, deve-se considerar parâmetros como caracterização dos participantes (TRAVASSOS et al., 2002). Assim, um formulário que caracteriza os participantes foi elaborado (Apêndice A.2), a fim de conhecer a experiência deles.

Para apoiar a realização do experimento, a ferramenta DiffMutAnalyze foi desenvolvida como apoio computacional para a análise dos mutantes e contabilização do tempo dessa análise. Na área da tecnologia, o desenvolvimento de novos produtos é caracterizado como pesquisa aplicada, no qual esse tipo de pesquisa visa obter conhecimentos para otimizar produtos ou processos e produzir conhecimentos tecnológicos (JUNG, 2004). Por meio da realização de estudos empíricos, é possível comparar os critérios dos estudos e obter uma estratégia eficaz para revelar a presença de erros no programa, com baixo custo de aplicação (MALDONADO et al., 1998). Dessa forma, por meio da obtenção de evidências dos experimentos, a análise de mutantes tem contribuído para o planejamento dos testes de software.

4.2.2 Procedimentos Gerais dos Experimentos

Alguns procedimentos são necessários para identificar o custo em análise dos mutantes. Conforme mencionado anteriormente (Capítulo 1), há uma sequência de passos a serem seguidos para obter o resultado deste estudo (Figura 1.1). Inicialmente é preciso selecionar o sistema de software a ser testado para realizar a mutação. Como pode ser observado na Figura 4.1, o engenheiro de software (4.1(a)) seleciona o sistema de software (4.1(b)). Como o intuito do teste de mutação é avaliar a qualidade dos casos de teste, o ideal é que o projeto tenha testes de unidade (4.1(c)) construídos inicialmente para o sistema selecionado. O sistema de software é inserido na DiffMutAnalyze e a geração dos mutantes é realizada pela ferramenta Major (4.1(d)), contida na DiffMutAnalyze, assim, são obtidos os mutantes (4.1(e)). Em seguida, a DiffMutAnalyze executa os casos de teste, em cada mutante gerado (4.1(f)). Dessa forma, os mutantes identificados pelos casos de teste são considerados mortos; caso algum mutante não seja descoberto pelos casos de teste, ele é considerado vivo (4.1(g)) [1].

Figura 4.1 – Geração dos mutantes.

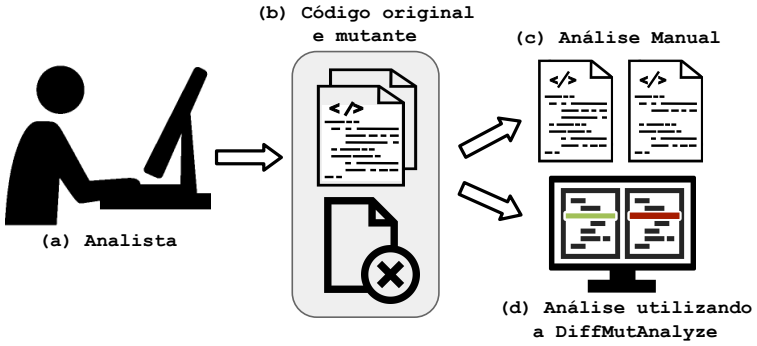


Fonte: Elaborado pela autora (2018).

[1] Mutantes mortos são simbolizados pelo "✓" (detectados) e os vivos pelo "x" (não detectados).

Na Figura 4.2, é ilustrado como o experimento foi conduzido para auxiliar e guiar o analista na análise dos mutantes. Nesse passo, há duas formas para inspecionar os mutantes, seja de maneira manual ou utilizando a DiffMutAnalyze. Dessa forma, caso o analista (4.2(a)) opte pela análise manual (4.2(c)) dos mutantes, a DiffMutAnalyze salva os mutantes na pasta do projeto, assim, o analista poderá abrir o código original do sistema de software e abrir o código do mutante (4.2(b)), para identificar onde a alteração do mutante foi realizada e verificar se há equivalência entre os dois códigos; caso a análise seja realizada pela DiffMutAnalyze (4.2(d)), os códigos (i.e. original e mutante) são automaticamente disponibilizados para visualização e a alteração realizada pela mutação é evidenciada, facilitando a comparação dos códigos original e mutante efetuada pelo analista.

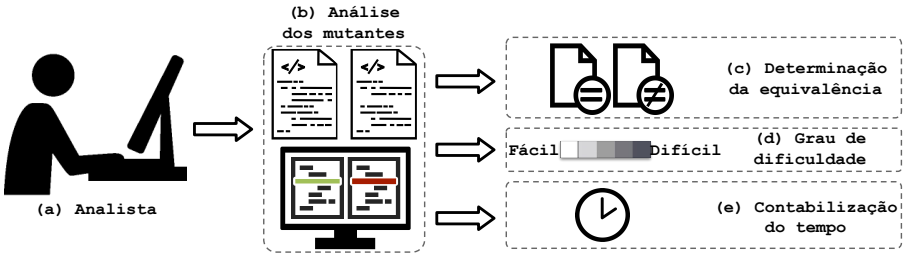
Figura 4.2 – Análise dos mutantes.



Fonte: Elaborado pela autora (2018).

A cada mutante inspecionado, o analista deve informar alguns dados da análise (Figura 4.3), assim, ao final de cada análise (4.3(b)), o analista deve indicar se o mutante é equivalente ao código original (se geram saídas iguais) ou se o mutante não é equivalente ao seu código correspondente (se suas saídas são diferentes) (4.3(c)). Outra informação que o analista precisa indicar é o grau de dificuldade da análise do mutante (4.3(d)), uma vez que a dificuldade de analisar os mutantes pode contribuir para o custo da análise, pois conforme a dificuldade da análise aumenta, pode ser gasto mais tempo para analisar o determinado mutante. Para contabilizar o tempo de cada análise (4.3(e)), a ferramenta DiffMutAnalyze cronometra o tempo gasto com a análise do mutante. Na análise manual, o tempo gasto com a análise é contabilizado utilizando um cronômetro, visto que essa análise não possui o auxílio de uma ferramenta exclusiva para esse tipo de análise. Dessa forma, o analista deve contar com recursos que o possibilite iniciar a contagem do tempo no início da inspeção dos códigos e finalizar ao término da análise. Assim, os passos que correspondem à análise (Figura 4.2) e à avaliação (Figura 4.3) dos mutantes, são executados até que todos os mutantes sejam analisados.

Figura 4.3 – Avaliação dos mutantes.



Fonte: Elaborado pela autora (2018).

Por fim, as análises realizadas por meio da DiffMutAnalyse, são processadas e os resultados são gerados, assim, o engenheiro de software poderá visualizar as informações da realização do teste de mutação no sistema de software testado. Na ferramenta DiffMutAnalyze é possível visualizar os relatórios da análise com informações como: *i)* quantidade de mutantes mortos e vivos; *ii)* quantidade de mutantes definidos como equivalentes; *iii)* cálculo do escore de mutação; *iv)* tempo gasto com a análise dos mutantes; e *(v)* grau de dificuldade em analisar os mutantes. Na análise manual, os dados das análises são documentados e o engenheiro de software necessita interpretar os dados informados para obter as informações desejadas.

4.2.3 Questões de Pesquisa Secundárias (QPS)

Para atingir o objetivo proposto nesta pesquisa, um experimento foi realizado a fim de responder as Questões de Pesquisa 1 e 2 (QP1 e QP2) deste estudo:

QP1: *"Qual é o custo para determinar manualmente a equivalência de mutantes com o sistema de software original?"*

QP2: *"A ferramenta DiffMutAnalyze pode contribuir com a redução do custo da análise, auxiliando os analistas na identificação dos mutantes equivalentes?"*

No contexto deste estudo, o custo humano, para identificar os mutantes equivalentes, é medido utilizando algumas questões de pesquisa secundárias. Dessa forma para auxiliar a responder a QP1, as questões secundárias 1, 2, 3 e 4 foram criadas e a questão 5 foi criada para auxiliar a responder a QP2. As questões secundárias são:

1. Qual é o tempo gasto com a análise manual dos mutantes?
2. O operador de mutação possui influência no custo da análise dos mutantes?
3. Os erros da definição dos mutantes equivalentes possuem influência no custo?
4. Qual é o grau de dificuldade em analisar os mutantes?
5. Utilizando a ferramenta DiffMutAnalyze é possível reduzir o tempo da análise?

Para responder as QP's secundárias listadas anteriormente, alguns procedimentos foram executados:

QPS 1 - Qual é o tempo gasto com a análise manual dos mutantes?

O tempo para analisar os mutantes foi medido de acordo com o tipo de análise. Para a análise utilizando a DiffMutAnalyze, o tempo foi contabilizado pela ferramenta. A contabilização do tempo inicia quando o analista seleciona o mutante a ser analisado e é finalizada ao término da análise, quando o analista segue para o próximo mutante a ser analisado. O tempo é medido a cada mutante disponível para análise. A DiffMutAnalyze disponibiliza um botão para pausar o tempo, caso seja necessário o analista parar com a análise; ao retomar a análise, o analista retorna a contagem do tempo e finaliza-a. Na análise manual, sem a utilização da ferramenta DiffMutAnalyze, o tempo foi contabilizado utilizando um cronômetro, no qual cada análise foi monitorada e o tempo anotado.

QPS 2 - O operador de mutação possui influência no custo da análise dos mutantes?

Os operadores de mutação são responsáveis por realizar as alterações nos códigos do sistema de software original, assim, cada tipo de mutação pode ter sua dificuldade durante a análise da verificação da possível equivalência entre os códigos. Dessa forma, foi observado o tempo gasto com a análise de cada mutante gerado por cada operador de mutação.

QPS 3 - Os erros da definição dos mutantes equivalentes possuem influência no custo?

Os mutantes sobreviventes serão analisados previamente pela autora desta pesquisa e identificados os mutantes equivalentes, dessa forma, é possível comparar com as análises dos participantes. Portanto, após a inspeção de todos os mutantes sobreviventes, será verificada a corretude de cada análise. Um erro cometido pelo participante é definir equivocadamente um mutante como equivalente ou determinar que um mutante não seja equivalente e, na realidade, ele possui equivalência com o código original.

QPS 4 - Qual é o grau de dificuldade em analisar os mutantes?

Cada mutante possui sua particularidade em ser analisado, sendo dada pelo modo que a alteração foi realizada, ou seja, depende do tipo de alteração realizada no código original. Além disso, o código analisado pode contribuir com esse fator, pois há códigos mais complexos de serem entendidos. Assim, o analista determina a complexidade da análise de cada mutante

examinado. Em relação ao grau de dificuldade, foi utilizado o conceito de escala *Likert* [2], considerando cinco níveis: *i)* muito fácil; *ii)* fácil; *iii)* médio; *iv)* difícil; e *v)* muito difícil.

QPS 5 - Utilizando a ferramenta DiffMutAnalyze é possível reduzir o tempo da análise? Essa questão de pesquisa secundária está relacionada com a questão secundária QP1. Uma vez que, por meio da contabilização do tempo dos dois tipos de análises (utilizando a DiffMutAnalyze e sem a utilização da ferramenta), é possível comparar o tempo gasto da análise realizada com a DiffMutAnalyze com o tempo da análise sem o uso de uma ferramenta de apoio computacional da análise. Dessa forma, obtém-se evidências se a ferramenta auxilia na redução do tempo gasto com a análise dos mutantes.

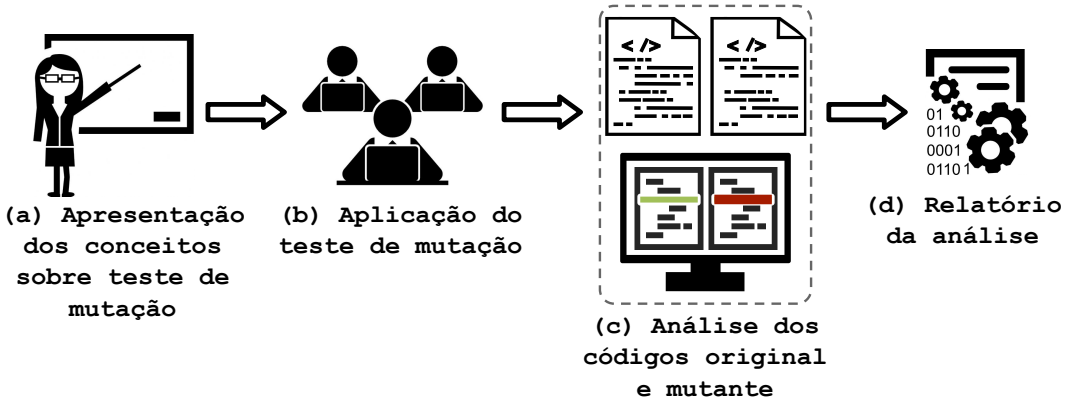
4.3 Descrição dos Experimentos

Dois experimentos foram realizados durante esta pesquisa. Inicialmente um experimento piloto foi aplicado para verificar o uso da ferramenta DiffMutAnalyze no contexto acadêmico. O segundo experimento teve o objetivo de analisar o custo da aplicação do teste de mutação no que tange a análise para verificar se os mutantes sobreviventes são equivalentes ao código original.

Para ambos experimentos, os procedimentos iniciais foram semelhantes. Dessa forma, conforme ilustrado na Figura 4.4, inicialmente foi explicada a teoria com os principais conceitos sobre o teste de mutação (Figura 4.4(a)), alguns exemplos de alterações realizadas pelos operadores de mutação foram apresentados para que os alunos / participantes pudessem compreender como a mutação é realizada. Em seguida, a ferramenta DiffMutAnalyze foi apresentada aos alunos e demonstrado seu funcionamento. Diante do conhecimento sobre esse critério de o teste, os alunos puderam utilizar a ferramenta e aplicar a mutação (Figura 4.4(b)). Para a realização dos experimentos, foram utilizados algoritmos de ordenação (Seção 4.3.1) desenvolvidos em Java. Dessa forma, os alunos aplicaram a mutação nos códigos e analisaram os mutantes (Figura 4.4(c)), de maneira manual (i. e. sem a utilização da DiffMutAnalyze) e utilizando a ferramenta, para identificar se são equivalentes ao sistema de software original. Ao final da análise, os dados das análises foram coletados e utilizados para obter informações sobre o custo da análise (Figura 4.4(d)).

[2] Escala *Likert*: mede construtos como atitudes, percepções, interesses, no contexto das ciências comportamentais. Essa escala é usada para medir concordância de pessoas a determinadas afirmações relacionadas a construtos de interesse (JUNIOR; COSTA, 2014).

Figura 4.4 – Apresentação dos conceitos sobre Teste de Mutação e utilização da DiffMutAnalyze.



Fonte: Elaborado pela autora (2018).

No experimento piloto, um minicurso foi realizado na Semana de Tecnologia da Informação (SE7I-2018) da Universidade Federal de Lavras, os alunos do minicurso conheceram os conceitos sobre Teste de Mutação e utilizaram a ferramenta DiffMutAnalyze para realizar o processo da mutação no mesmo dia, o minicurso teve duração de 4 horas. Os mutantes foram gerados e foram analisados para verificar a possível equivalência entre eles e o sistema de software original somente por meio da DiffMutAnalyze. Dessa forma, foi possível observar a utilização da ferramenta pelos alunos.

No experimento para verificação do custo da análise dos mutantes foram gastos dois dias para aplicação do mesmo. No primeiro dia, os alunos tiveram a explicação sobre os conceitos do teste de mutação e conheceram o ambiente da ferramenta DiffMutAnalyze; no segundo dia, os participantes realizaram a análise dos códigos original e mutante para extrair os dados do experimento. Para a análise manual, os participantes utilizaram editores de texto para abrir os códigos e compará-los, a fim de identificar a mutação realizada e verificar se os códigos possuem equivalência. Essa análise manual foi realizada em computadores com o sistema operacional macOS Mojave, versão 10.14.6. Foi utilizado o editor de texto incluído no Mac, denominado Notes, versão 4.6. O motivo por usar um editor de texto simples, ou seja, que não possua recursos para auxiliar a análise do mutantes, foi com intuito de comparar as análises e verificar o custo real da análise manual, sem a utilização de recursos que possam mascarar o tempo real gasto. Quando a mutação é realizada pela ferramenta Major (contida na DiffMutAnalyze), os mutantes são salvos em pastas numeradas de acordo com a quantidade de mutantes gerados, assim sendo, cada mutante é salvo em uma pasta e possui sua numeração. Dessa forma, os mutantes disponibilizados para análise, foram catalogados para que o sujeito anotasse a equivalência e o grau de dificuldade referente a cada mutante. Além disso, o tempo gasto com a inspeção dos códigos

foi cronometrado e registrado de acordo com cada análise. Assim, o tempo foi contabilizado, os participantes analisaram os códigos e responderam a duas perguntas:

1. *O mutante é equivalente ao programa original?*
2. *Qual é o grau de dificuldade em analisar se o mutante é equivalente ao programa original?*

Para a análise por meio da ferramenta DiffMutAnalyze, o tempo é registrado automaticamente, a ferramenta possui os campos de marcação para informar se os mutantes são equivalentes ao seu código original correspondente e para informar o grau de dificuldade em analisar o determinado mutante.

4.3.1 Algoritmos de Ordenação

Para que a mutação seja gerada na ferramenta DiffMutAnalyze, é necessário utilizar um projeto implementado na linguagem Java. Dessa forma, foram considerados dois algoritmos de ordenação escritos em Java: *i) InsertionSort* (Código 4.1); e *ii) SelectionSort* (Código 4.2). A escolha desses algoritmos foi baseada em tentar atender ao nível de conhecimento acadêmico dos alunos, uma vez que esses algoritmos são conhecidos no âmbito acadêmico por grande parte dos alunos que cursam Ciência da Computação e áreas afins, facilitando assim a análise dos códigos, uma vez que possivelmente os alunos possuem conhecimento prévio desses códigos.

Código 4.1 – Algoritmo de ordenação *InsertionSort*

```
1  public class Insertion {
2          public static void sort(Comparable[] array) {
3                  int N = array.length;
4                  for (int i = 1; i < N; i++) {
5                          int j = i;
6                          while (j > 0 && less(array[j], array[j - 1])) {
7                                  exchange(array, j, j - 1);
8                                  j--;
9                          }
10                 }
11         }
12
13         private static boolean less(Comparable x, Comparable y) {
14                 return x.compareTo(y) < 0;
15         }
16
17         private static void exchange(Comparable[] a, int i, int j) {
18                 Comparable t = a[i];
19                 a[i] = a[j];
20                 a[j] = t;
21         }
22 }
```

Código 4.2 – Algoritmo de ordenação *SelectionSort*

```
1  public class Selection {
2          public static void sort(Comparable[] array) {
3                  int N = array.length;
4                  for (int i = 0; i < N; i++) {
5                          int min = i;
6                          for (int j = i + 1; j < N; j++) {
7                                  if (array[j].compareTo(array[min]) < 0)
8                                          min = j;
9                          }
10                         exchange(array, i, min);
11                 }
12         }
13
14         private static void exchange(Comparable[] array, int i, int j) {
15                 Comparable t = array[i];
16                 array[i] = array[j];
17                 array[j] = t;
18         }
19 }
```

Embora esses algoritmos sejam pequenos algoritmos de livros didáticos, eles possuem instruções com operadores lógicos, condicionais e aritméticos, o que gera grande quantidade de mutantes, uma vez que parte dos operadores de mutação realizam alterações nesses tipos de operadores contidos nos códigos. O algoritmo *InsertionSort* possui 22 linhas de código (Código 4.1) e 38 mutantes foram gerados; o algoritmo *SelectionSort* possui 19 linhas de código (Código 4.2) e 26 mutantes foram gerados.

4.3.2 Procedimento do Experimento Piloto

Antes de aplicar o experimento principal desta pesquisa, foi realizado um experimento piloto a fim de verificar a utilização da DiffMutAnalyze, além disso, foi verificada a possibilidade de utilizar a DiffMutAnalyze em ambientes acadêmicos. Dessa forma, um minicurso foi realizado na V Semana de Tecnologia da Informação (SETI [3]) na Universidade Federal de Lavras (UFLA). Os alunos inscritos no minicurso eram alunos de graduação do curso de Ciência da Computação e de Sistemas de Informação da UFLA, além de dois participantes da indústria.

No minicurso, o conceito sobre teste de mutação foi apresentado aos alunos e, posteriormente, a DiffMutAnalyze foi utilizada na realização da aplicação do teste de mutação pelos alunos. Os alunos realizaram todo o processo do teste de mutação por meio da ferramenta, ou seja, todos os passos foram conduzidos pelos alunos, desde o cadastramento na ferramenta até a análise dos mutantes. Dessa forma, os alunos puderam realizar uma adaptação prática da realização do teste de mutação antes de realizar a análise dos mutantes, obtendo conhecimento prático do funcionamento da DiffMutAnalyze. Para a realização da mutação, os algoritmos de ordenação *SelectionSort* e *InsertionSort* foram utilizados em conjunto na DiffMutAnalyze.

[3] SE7I - http://seti.ufla.br/

Para verificar o uso da DiffMutAnalyze, os alunos receberam o projeto Java, com os algoritmos de ordenação, para ser inserido na ferramenta e a mutação ser realizada. Após certificar que todos os alunos possuíam os mutantes gerados e o projeto estava pronto para ser analisado, foi autorizado o início da análise dos códigos original e mutante. Os sujeitos indicaram se possuía alguma equivalência entre os códigos e indicaram o grau de dificuldade em analisar cada mutante, além de contabilizar o tempo automaticamente pela própria ferramenta. Dessa forma, foi possível verificar a utilização da ferramenta.

4.3.3 Procedimento do Experimento para Investigação do Custo da Análise dos Mutantes Equivalentes

A investigação para verificar o custo humano em analisar manualmente a possibilidade de haver equivalência entre os códigos mutantes e o original foi realizada por meio de um experimento com onze alunos de pós-graduação em Ciência da Computação da Universidade Federal de Lavras. É importante elucidar que os participantes deste experimento são diferentes dos participantes do experimento piloto. Embora os dois experimentos foram conduzidos com onze sujeitos, nenhum sujeito participou dos dois experimentos. Lembrando que os sujeitos do experimento anterior, eram alunos de graduação e participantes da indústria. Ambos experimentos possuem a mesma quantidade de sujeitos participantes, porém, esse fato ocorreu ocasionalmente. Nesse experimento, os alunos analisaram os códigos de duas maneiras distintas: *(i)* utilizando a DiffMutAnalyze; e *(ii)* sem a utilização da ferramenta.

Como mencionado anteriormente, para realizar o experimento, o mesmo foi dividido em duas etapas, em dois dias distintos. No primeiro dia, uma aula foi realizada para apresentar os conceitos e exemplos sobre teste de mutação, foi demonstrado exemplo de mutante equivalente para que os alunos pudessem compreender a maneira como um mutante se comporta, quando comparada a saída do sistema de software original e desse mutante. Passado o conhecimento necessário para que os alunos compreendessem o que é o teste de mutação e como ele é realizado, a ferramenta DiffMutAnalyze foi apresentada e os sujeitos puderam verificar seu funcionamento na prática. No segundo dia, a análise dos mutantes foi realizada pelos alunos. Antes de iniciar a análise dos mutantes, dois laboratórios foram preparados para o experimento, um para a realização da análise utilizando a DiffMutAnalyze e o outro para a realização da análise manual (sem utilizar a DiffMutAnalyze).

Para a análise manual, foi utilizado o algoritmo de ordenação *SelectionSort*, os mutantes desse algoritmo foram gerados previamente pela ferramenta Major na DiffMutAnalyze e separados para análise. Uma vez que os mutantes são salvos em disco, foi possível extrair a pasta com os mutantes designados para a análise manual, juntamente com seu código original correspondente. Para assim, os alunos compararem os códigos (i. e. original e mutante) e analisar se havia equivalência entre eles. Para a análise utilizando a ferramenta DiffMutAnalyze, o laboratório foi preparado com a ferramenta em execução e com os mutantes do algoritmo *InsertionSort* gerados. O motivo pelo qual os mutantes foram gerados previamente, é dado ao fato que o objetivo do experimento é investigar o custo da análise dos mutantes, dessa forma, o tempo foi contabilizado considerando somente a análise. O tempo da execução dos mutantes foi desconsiderado, pois esse custo está relacionado com o custo computacional.

Dessa maneira, cada aluno analisou todos os mutantes gerados pela ferramenta Major (inserida na DiffMutAnalyze), dos algoritmos *SelectionSort* e *InsertionSort*. Foram gerados 26 mutantes do algoritmo *SelectionSort* e 38 mutantes do algoritmo *InsertionSort*. Os sujeitos do experimento analisaram todos os mutantes, independente se eles poderiam ser mortos pelos casos de teste. Essa análise completa dos mutantes, foi realizada para verificar os erros e acertos dos sujeitos. Um erro cometido pelo sujeito, é definir o mutante como equivalente ao código original e o mesmo não ser equivalente, ou vice-versa, ou seja, o mutante é equivalente ao código original e não foi definido, pelo sujeito, como equivalente. Além disso, por meio da análise de todos os mutantes, é possível verificar o grau de dificuldade em analisar diferentes tipos de mutantes, ou seja, analisar a mutação realizada por diferentes operadores de mutação.

Em relação aos operadores de mutação da ferramenta Major, os mutantes gerados nos algoritmos utilizados no experimento, estão descritos na Tabela 4.1. Como pode ser observado na tabela, no algoritmo *InsertionSort*, os mutantes foram gerados pelos operadores AOR, COR, EVR, LVR, ROR e STD, sendo a maior quantidade de mutantes gerados pelo operador LVR (10 mutantes). Nesse algoritmo, alguns mutantes equivalentes foram gerados, sendo três do operador LVR e três do ROR, totalizando seis mutantes equivalentes, dentre os 38 mutantes gerados, o que representa 15,8% dos mutantes. Os mutantes gerados no algoritmo *SelectionSort*, foram dos operadores AOR, EVR, LVR, ROR e STD, sendo a maior quantidade de mutantes gerados pelo operador ROR (9 mutantes). Em relação aos mutantes equivalentes, dois foram gerados pelo operador AOR, dois pelo LVR e três pelo ROR, somando sete mutantes equivalentes, representado 26,9% dos mutantes desse algoritmo (total de 26 mutantes gerados). Dessa

forma, o operador ROR foi o que gerou maior quantidade de mutantes equivalentes, considerando os dois algoritmos de ordenação, sendo gerados seis mutantes no total, o operador LVR gerou cinco mutantes equivalentes no total e o operador AOR dois mutantes equivalentes, os demais operadores não geraram mutações equivalentes. É possível perceber que o operador COR não foi utilizado para realizar a mutação no algoritmo *SelectionSort*, mas foi utilizado no algoritmo *InsertionSort*. A mutação é gerada automaticamente pela Major e os operadores são aplicados no código original de acordo com as instruções contidas em cada código, uma vez que cada operador realiza uma modificação diferente. Dessa forma, as mudanças são aplicadas no código conforme a possibilidade de mudança contida no mesmo.

Tabela 4.1 – Quantidade de mutantes gerados por cada operador de mutação.

InsertionSort			***SelectionSort***		
Operador	**Mutantes gerados**	**Mutantes equivalentes**	**Operador**	**Mutantes gerados**	**Mutantes equivalentes**
AOR	8	0	**AOR**	4	2
COR	4	0	**EVR**	3	0
EVR	3	0	**LVR**	6	2
LVR	10	3	**ROR**	9	3
ROR	9	3	**STD**	4	0
STD	4	0			
Total	**38**	**6**	**Total**	**26**	**7**

Fonte: Elaborado pela autora (2019).

Para a realização das análises, os alunos foram divididos em dois grupos, cada grupo foi disposto em laboratórios diferentes, de acordo com o tipo de análise efetuada inicialmente. Essa divisão dos grupos, foi realizada com o objetivo de intercalar e inverter as formas de análise, ou seja, um grupo realizou primeiramente a análise dos mutantes utilizando a ferramenta DiffMutAnalyze e, posteriormente, efetuou a análise manual (sem o auxilio da ferramenta), utilizando uma ferramenta de texto para visualização dos códigos original e mutante. Consequentemente, o outro grupo iniciou a inspeção dos códigos pela análise manual, para depois realizar a análise utilizando a DiffMutAnalyze. Essa divisão dos grupos foi realizada a fim de verificar se havia influência nos resultados do experimento, considerando a maneira que a análise foi realizada inicialmente. A divisão dos participantes em grupos diferentes foi realizada aleatoriamente, sem utilizar um critério de agrupamento entre os sujeitos, pois o intuito foi comparar a análise realizada por cada participante independente do seu perfil.

O primeiro grupo, composto pelos sujeitos S5, S6, S9, S10 e S11, analisou os mutantes utilizando a DiffMutAnalyze. O segundo grupo, composto pelos sujeitos S1, S2, S3, S4,

S7 e S8, analisou os mutantes sem a utilização da ferramenta DiffMutAnalyze. Dessa forma, os sujeitos fizeram as análises e algumas informações foram coletadas, como: informar se os mutantes são equivalentes, definir o grau de dificuldade e registrar o tempo gasto na análise.

Para registrar essas informações de cada análise, os mutantes analisados na DiffMutAnalyze, tiveram o tempo contabilizado automaticamente pela ferramenta e as informações da equivalência e da dificuldade, foram informadas pelo sujeito e registradas na DiffMutAnalyze. Na análise manual, os sujeitos receberam um formulário para preencher as informações da análise manual. O tempo foi contabilizado utilizando um cronômetro, sendo iniciado e pausado a cada análise. Além disso, foi registrada a hora inicial da primeira análise e a hora final da última análise, para contabilizar o tempo total gasto com a análise de todos os mutantes, uma vez que esse tempo inclui a abertura dos códigos de cada mutante e a localização do local da alteração. Portanto, o tempo total é a soma das inspeções do códigos original e mutante e do tempo utilizado para abrir o código e localizar a mutação. Uma vez que a DiffMutAnalyze faz esse processo, de abertura do mutante e localização da alteração realizada, de forma automática.

4.4 Considerações finais

O uso do teste de mutação, como critério de teste, possui a característica de melhorar a qualidade do conjunto de testes. Uma vez que, caso haja mutantes sobreviventes e não equivalentes, novos casos de teste são produzidos para revelar defeitos não identificados pelos testes construídos inicialmente. Para este fim, os mutantes vivos precisam ser analisados, para verificar a necessidade de novos testes de unidade. Dessa forma, neste capítulo, foram fornecidos os procedimentos adotados para aplicar o experimento para a investigação do custo em analisar esses mutantes.

Em suma, os resultados obtidos com o experimento contribuem para determinar o custo da análise dos mutantes. Além disso, as informações extraídas podem auxiliar a outros pesquisadores em relação ao tempo gasto com a análise e o grau de dificuldade em analisar os mutantes. Como mencionado anteriormente, os mutantes equivalentes não contribuem para a melhoria dos casos de teste, assim, os mutantes vivos precisam ser analisados para verificar se realmente são equivalentes ou se novos testes são necessários. No próximo capítulo, estão descritos os resultados dos experimentos realizados.

5 RESULTADOS DOS EXPERIMENTOS

Neste capítulo, estão apresentados os dois experimentos realizados nesta pesquisa. Inicialmente, são apresentados os resultados do experimento piloto realizado para verificar a experiência do uso da DiffMutAnalyze para a aplicação do teste de mutação no ambiente acadêmico, verificando se o uso da ferramenta auxilia na análise dos mutantes. No segundo experimento, estão descritos os resultados da verificação do custo em analisar se os mutantes sobreviventes aos casos de teste, possuem equivalência com o sistema de software original. As análises dos códigos original e mutante foram realizadas por diferentes maneiras (manual e utilizando a DiffMutAnalyze). Dessa forma, foi possível medir e comparar o tempo gasto com a análise dos mutantes entre as duas formas distintas.

5.1 Experimento Piloto

Um experimento piloto foi aplicado para testar e avaliar o uso da DiffMutAnalyze com alunos participantes da SETI. O objetivo deste experimento é apresentar os conceitos do teste de mutação na Semana de Tecnologia da Informação da UFLA e, consequentemente, verificar o uso da DiffMutAnalyze junto aos participantes, analisando se durante o uso da ferramenta os alunos possuíam alguma dificuldade em utilizá-la.

5.1.1 Perfil dos Participantes

Um formulário de caracterização dos participantes foi aplicado para cada aluno, assim, com o preenchimento desse formulário, o perfil dos alunos participantes deste experimento está descrito na Tabela 5.1. Como pode ser observado, onze sujeitos participaram do estudo, sendo sete estudantes do curso de Ciência da Computação, dois do curso de Sistemas de Informação e dois participantes da indústria. Embora, apenas dois sujeitos deste estudo sejam da indústria – de acordo com um estudo recente - existe uma baixa proporção de profissionais como participantes de estudos acadêmicos (FELDT et al., 2018). Em uma revisão de literatura de Sjøberg et al. (SJØBERG et al., 2005), foi identificado que apenas 9% dos sujeitos eram profissionais. No contexto deste estudo, pode ser impraticável e dispendioso obter amostras apropriadas de profissionais. Assim, a utilização de estudantes é uma boa alternativa, embora o número e a magnitude de potenciais ameaças à validade aumentam.

Tabela 5.1 – Perfil dos participantes do estudo.

Sujeito	Conhecimento sobre Java	Utilização de Teste de Unidade	Conhecimento sobre Teste de Mutação	Conhecimento sobre algoritmos de ordenação
S1	Nenhum	Trabalha (trabalhou) com testes de unidade	Nenhum	Básico
S2	Nenhum	Trabalha (trabalhou) com testes de unidade	Nenhum	Nenhum
S3	Intermediário	Nunca utilizou	Já ouviu falar, mas nunca utilizou	Intermediário
S4	Intermediário	Nunca utilizou	Já ouviu falar, mas nunca utilizou	Intermediário
S5	Intermediário	Nunca utilizou	Nenhum	Intermediário
S6	Intermediário	Já utilizou em disciplinas do curso	Nenhum	Intermediário
S7	Básico	Nunca utilizou	Nenhum	Intermediário
S8	Intermediário	Já utilizou em disciplinas do curso	Já ouviu falar, mas nunca utilizou	Intermediário
S9	Básico	Já utilizou em disciplinas do curso	Nenhum	Básico
S10	Básico	Já utilizou em disciplinas do curso	Nenhum	Intermediário
S11	Básico	Já utilizou em disciplinas do curso	Nenhum	Básico

Fonte: Elaborado pela autora (2019).

Dentre os 11 sujeitos, dois não possuem conhecimento sobre a linguagem Java, porém esses dois possuem maior conhecimento sobre teste de unidade e trabalham com essa técnica, cinco sujeitos já utilizaram testes de unidade apenas em disciplinas dos cursos em que estão matriculados e, os outros quatro sujeitos, nunca o utilizaram. Especificamente sobre o teste de mutação, três sujeitos já ouviram falar, mas nunca utilizaram, os demais sujeitos nunca ouviram falar sobre esse critério de teste.

Para confirmar a hipótese sobre o conhecimento prévio dos alunos em relação aos algoritmos de ordenação, os alunos foram questionados sobre o nível de conhecimento de tais algoritmos, sendo apenas um sujeito afirmando não possuir conhecimento desses algoritmos, três possuem conhecimento básico e sete possuem conhecimento intermediário. Dessa forma, pode-se concluir que a utilização de algoritmos de ordenação é uma boa escolha para aplicar a mutação em ambientes acadêmicos devido a dois fatores: *i)* esses algoritmos possuem grande quantidade de operadores lógicos e operadores condicionais, o que leva a geração de muitos

mutantes, sem a necessidade de ter um projeto com muitas classes; e *ii)* os alunos possivelmente já conhecem esses algoritmos. O que facilita no momento do aprendizado sobre o teste de mutação, uma vez que o foco principal do aprendizado é fazer com que os alunos conheçam a técnica do teste de mutação e aprendam a aplicá-la.

5.1.2 Análise e Interpretação dos Dados

Os dados das análises dos sujeitos foram extraídos e uma base de dados foi montada considerando apenas os mutantes analisados por completo, ou seja, foram considerados os mutantes cujo sua equivalência foi definida e o grau de dificuldade foi indicado. Os mutantes analisados, mas sem classificação foram excluídos da base de dados, dessa maneira, a quantidade de mutantes analisados varia para cada sujeito.

Na Tabela 5.2, está representada a quantidade de mutantes analisados por cada sujeito, o tempo total gasto com a análise e o tempo médio por mutante analisado. O maior tempo de análise foi do sujeito S1, sendo analisados 36 mutantes em aproximadamente 30 minutos, foi gasto, em média, 50,6 segundos para analisar cada mutante. Já o menor tempo de análise, foi para o sujeito S11, no qual foram analisados 37 mutantes em 9 minutos e 30 segundos, tendo em média, 15 segundos por mutante.

Tabela 5.2 – Quantidade de mutantes analisados e tempo total gasto com a análise de cada sujeito.

Sujeito	**Quantidade de mutantes analisados**	**Tempo total**	**Tempo médio por mutante**
S1	36	00:30:40	00:00:51
S2	38	00:26:30	00:00:42
S3	47	00:20:00	00:00:26
S4	47	00:22:10	00:00:28
S5	36	00:13:50	00:00:22
S6	38	00:23:20	00:00:37
S7	48	00:23:30	00:00:29
S8	38	00:16:20	00:00:26
S9	39	00:25:00	00:00:38
S10	42	00:23:40	00:00:33
S11	37	00:09:30	00:00:15
Tempo médio		00:21:10	00:00:31

Fonte: Elaborado pela autora (2019).

Na Tabela 5.3, estão relacionados os resultados da definição da equivalência entre os códigos original e mutante, os sujeitos S1, S2 e S9 definiram maior quantidade de mutantes equivalentes analisados. Os demais sujeitos indicaram entre dois e quatro mutantes equivalentes em suas análises.

Tabela 5.3 – Quantidade de mutantes Equivalentes definidos pelos sujeitos.

Sujeito	Mutantes Equivalentes	Mutantes não-Equivalentes
S1	11	25
S2	19	19
S3	2	45
S4	4	43
S5	2	34
S6	3	35
S7	4	44
S8	2	36
S9	13	26
S10	2	40
S11	2	35

Fonte: Elaborado pela autora (2019).

Os sujeitos definiram o grau de dificuldade em analisar cada mutante (Tabela 5.4). Grande parte dos sujeitos indicou o grau de dificuldade variando entre muito fácil e fácil, somando a quantidade de mutantes definidos com um desses dois níveis de dificuldade, os sujeitos S3, S4, S5, S6, S7, S8, S10 e S11 tiveram uma diferença maior entre o nível de dificuldade muito fácil ou fácil, em relação aos níveis médio, difícil ou muito difícil. O sujeito S11 definiu dificuldade muito fácil ou fácil para todos os mutantes analisados. Por outro lado, os sujeitos S1, S2 e S9 tiveram uma diferença menor entre os níveis muito fácil e fácil, em relação aos níveis médio, difícil e muito difícil. Como pode ser observado na tabela, o sujeito S2 não definiu algum mutante com os níveis de dificuldade muito fácil ou fácil, a maioria dos mutantes foi definido como difícil, com 34 indicações nesse nível de dificuldade.

Tabela 5.4 – Grau de dificuldade em analisar os mutantes.

Sujeito	Muito Fácil	Fácil	Médio	Difícil	Muito Difícil
S1	5	16	14	1	0
S2	0	0	2	34	2
S3	41	5	1	0	0
S4	29	13	3	2	0
S5	11	15	5	5	0
S6	26	2	3	3	4
S7	35	6	5	2	0
S8	23	8	5	2	0
S9	2	18	14	3	2
S10	16	25	0	1	0
S11	33	4	0	0	0

Fonte: Elaborado pela autora (2019).

Considerando todas as análises, no geral os sujeitos analisaram os mutantes em uma média de 31 segundos por mutante (Tabela 5.2). Considerando o maior tempo médio de análise dos mutantes, o sujeito S1 gastou em média 51 segundos em cada mutante (Tabela 5.2), esse fato pode ter ocorrido devido ao seu baixo conhecimento sobre algoritmos de ordenação e por não possuir conhecimentos sobre a linguagem de programação Java (Tabela 5.1). O mesmo ocorre com o sujeito S2, no qual foi o segundo que obteve maior tempo médio de análise, além disso, esse sujeito não possui conhecimentos sobre Java e algoritmos de ordenação (Tabela 5.1). Esse mesmo sujeito (S2) teve maior dificuldade em analisar os códigos original e mutante (Tabela5.4), o que pode influenciar no tempo da análise, uma vez que ele não possui conhecimento sobre a linguagem Java e nem sobre os algoritmos de ordenação (Tabela 5.1). Ao final do experimento, foi perguntado aos alunos sobre a utilização da ferramenta DiffMutAnalyze, todos os sujeitos disseram que não tiveram dificuldades em utilizar a ferramenta e que conseguiram visualizar a maneira que a mutação é realizada de forma rápida e prática.

Além de realizar a análise dos códigos para verificação da equivalência entre eles, os alunos conseguiram verificar na prática como os operadores de mutação realizam as mutações e puderem ter conhecimento sobre os tipos de alterações realizadas. Considerando a utilização da DiffMutAnalyze em sala de aula, os alunos, além de realizar a mutação no sistema de software a ser testado e analisar os códigos original e mutantes, podem criar novos casos de teste para os algoritmos e realizar o procedimento do teste de mutação novamente. Assim, é possível verificar a importância de melhorar a qualidade dos testes de unidade em um projeto de software.

5.2 Experimento para Investigação do Custo da Análise dos Mutantes Equivalentes

Nesta Seção, está descrito o resultado do experimento aplicado nesta pesquisa, que tem o objetivo de investigar o custo envolvido em analisar a possível equivalência entre os mutantes e o código original.

5.2.1 *Perfil dos Participantes do Experimento*

O experimento foi realizado com onze alunos de pós-graduação em Ciência da Computação da Universidade Federal de Lavras. Para identificar o perfil dos sujeitos do experimento, um formulário de caracterização de sujeitos foi enviado aos alunos. Esse formulário está disponível no Apêndice A.2. Como pode ser observado na Tabela 5.5 foi identificado o nível de conhecimento dos sujeitos referente à linguagem Java, ao teste de unidade, ao teste de mutação

e aos algoritmos de ordenação. Dentre os onze sujeitos, três possuem conhecimento básico sobre a linguagem Java, cinco possuem conhecimento intermediário e três possuem conhecimento avançado. Com relação ao conhecimento sobre teste de unidade, dois sujeitos não possuem conhecimento, sete possuem conhecimento básico e dois intermediário. Sobre o conhecimento relacionado ao teste de mutação, a maioria não possui conhecimento, somando sete sujeitos e quatro possuem os conhecimentos básico, adquiridos previamente ao experimento.

Tabela 5.5 – Perfil dos sujeitos.

Sujeito	**Conhecimento sobre Java**	**Conhecimento sobre Teste de Unidade**	**Conhecimento sobre Teste de Mutação**	**Conhecimento sobre Algoritmos de ordenação**
S1	Avançado	Básico	Básico	Avançado
S2	Intermediário	Nenhum	Nenhum	Básico
S3	Intermediário	Básico	Nenhum	Intermediário
S4	Intermediário	Básico	Básico	Intermediário
S5	Básico	Nenhum	Nenhum	Básico
S6	Avançado	Intermediário	Nenhum	Avançado
S7	Básico	Básico	Nenhum	Básico
S8	Intermediário	Básico	Nenhum	Intermediário
S9	Avançado	Básico	Básico	Avançado
S10	Básico	Básico	Nenhum	Intermediário
S11	Intermediário	Intermediário	Básico	Intermediário

Fonte: Elaborado pela autora (2019).

Para confirmar a hipótese sobre o conhecimento prévio dos alunos em relação aos algoritmos de ordenação, os alunos foram questionados sobre o nível de conhecimento de tais algoritmos, sendo que todos os sujeitos possuem algum nível de conhecimento prévio sobre esses algoritmos. Dentre eles, três alunos possuem conhecimento básico, cinco intermediário e três avançado. Dessa forma, podemos concluir que a utilização de algoritmos de ordenação é uma boa escolha para aplicar a mutação em ambientes acadêmicos.

Quando os sujeitos foram questionados se possuem o hábito de atualizar os casos de teste, para tentar melhorá-los durante o desenvolvimento de software. Dentre os onze sujeitos, nove sujeitos responderam que não possuem esse hábito, um sujeito afirmou que sempre atualiza e outro sujeito atualiza os casos de teste raramente. Conhecendo esse perfil dos participantes, é possível perceber que o teste de mutação é promissor para melhorar a qualidade dos casos de teste, uma vez que o conjunto de casos de teste construídos inicialmente não são atualizados pela maioria dos sujeitos.

5.2.2 Resultados Descritivos do Experimento

Nesta seção, estão descritos os resultados do experimento realizado para investigar o custo da análise manual dos mutantes e verificação da equivalência entre eles e o código original. Assim, foram comparados os resultados entre a análise utilizando a DiffMutAnalyze e a análise manual (sem o uso de uma ferramenta de apoio). A seguir, os resultados estão organizados de acordo com as questões de pesquisa secundárias (QPS) propostas neste trabalho.

QPS 1 - Qual é o tempo gasto com a análise manual dos mutantes?

Para responder essa questão de pesquisa, o tempo da análise foi comparado entre as duas formas de análise do experimento. Considerando a análise realizada pela DiffMutAnalyze, os onze sujeitos analisaram 38 mutantes do algoritmo *InsertionSort*, conforme mencionado anteriormente. Como pode ser observado na Tabela 5.6, os sujeitos gastaram em média 14 minutos para analisar os mutantes utilizando a DiffMutAnalyze e, considerando o tempo médio para analisar cada mutante, os sujeitos analisaram em 21 segundos cada mutante. O maior tempo de análise gasto foi do sujeito S11, gastando 19 minutos para analisar todos os mutantes, obtendo o tempo médio de 30 segundos por mutante. Considerando o menor de tempo de análise, o sujeito S3 gastou 10 minutos e 1 segundo para analisar os 38 mutantes, atingindo o tempo médio de 16 segundos por mutante. O mesmo ocorre com o sujeito S7, considerando o menor tempo médio obtido em cada mutante, atingindo o mesmo tempo médio por mutante, 16 segundos, porém o tempo total da análise desse sujeito é de 10 minutos e 6 segundos. Dessa forma, esses dois sujeitos, foram os que obtiveram o menor tempo de análise dos mutantes.

Tabela 5.6 – Tempo gasto com a análise dos mutantes utilizando a DiffMutAnalyze.

Sujeito	**Quantidade de mutantes analisados**	**Tempo total gasto**	**Tempo médio por mutante**
S1	38	00:12:07	00:00:19
S2	38	00:12:00	00:00:19
S3	38	00:10:01	00:00:16
S4	38	00:15:05	00:00:24
S5	38	00:16:05	00:00:25
S6	38	00:11:08	00:00:17
S7	38	00:10:06	00:00:16
S8	38	00:14:08	00:00:22
S9	38	00:14:03	00:00:22
S10	38	00:16:07	00:00:25
S11	38	00:19:00	00:00:30
Tempo médio		00:14:00	00:00:21

Fonte: Elaborado pela autora (2019).

Em relação ao tempo gasto com a análise manual, sem a utilização da ferramenta de apoio computacional DiffMutAnalyze, os sujeitos analisaram 26 mutantes gerados do algoritmo *SelectionSort*, com exceção do sujeito S7, no qual ele analisou somente 20 mutantes (conforme Tabela 5.7). Vale ressaltar que o próprio participante (S7) decidiu analisar somente essa quantidade de mutantes, não sendo instruído a realizar essa ação.

Para medir o tempo da análise manual, foi considerado o tempo inspecionando cada mutante individualmente e o tempo total gasto com a análise, ou seja, o tempo total é atribuído o tempo para abrir o código do mutante, uma vez que cada mutante é gerado em uma classe diferente. Esse tempo foi contabilizado nessa análise, pois na DiffMutAnalyze, os mutantes são selecionados automaticamente e disponibilizados para o usuário conforme o usuário solicita o próximo mutante a ser analisado (navegando pelo botão "Próximo"). Além disso, o tempo para encontrar onde a alteração foi realizada na classe mutante também é considerado, pois a DiffMutAnalyze coloca em evidência o local da mutação. Nesse caso, da avaliação sem o uso da ferramenta, é necessário que o sujeito identifique também onde foi realizada a mutação.

Na Tabela 5.7, está descrito o tempo gasto com a análise manual, há duas colunas de tempo, a coluna "Tempo de análise" considera somente o tempo que o mutante foi inspecionado, ou seja, o tempo em que os códigos foram comparados para verificação da equivalência. A coluna "Tempo total gasto" soma o tempo de análise e o tempo utilizado para abrir a pasta com o mutante a ser analisado, abrir o código do devido mutante, realizar a inspeção do código e anotar as informações da possível equivalência entre os códigos e do grau de dificuldade da análise. Lembrando que esses procedimentos são realizados manualmente a cada mutante.

Tabela 5.7 – Tempo gasto com a análise dos mutantes sem a utilização da DiffMutAnalyze.

Sujeito	**Quantidade de mutantes analisados**	**Tempo de análise**	**Tempo total gasto**	**Tempo total médio por mutante**
S1	26	00:28:40	00:50:00	00:01:58
S2	26	00:09:36	00:40:00	00:01:31
S3	26	00:11:14	00:26:00	00:01:00
S4	26	00:26:27	00:53:00	00:02:03
S5	26	00:10:20	00:21:00	00:00:48
S6	26	00:04:48	00:22:00	00:00:51
S7	20	00:16:52	00:25:00	00:01:25
S8	26	00:14:32	00:26:00	00:01:00
S9	26	00:10:52	00:29:00	00:01:06
S10	26	00:06:22	00:21:00	00:00:48
S11	26	00:11:15	00:26:00	00:01:00
Tempo médio		00:12:50	00:31:30	00:01:06

Fonte: Elaborado pela autora (2019).

Os sujeitos gastaram em média aproximadamente 13 minutos para comparar os códigos dos mutantes com o código original e 31 minutos para realizar todo o procedimento da análise. O tempo médio gasto com a análise manual de cada mutante, considerando o tempo total, é de 1 minuto e 6 segundos. Para analisar todos os mutantes e considerando o tempo total gasto, o sujeito S4 gastou 53 minutos para efetuar todo o processo de análise, obtendo o tempo médio por mutante de aproximadamente dois minutos.

O menor tempo total de análise é dos sujeitos S5 e S10, contabilizando 21 minutos para analisar os mutantes, com o tempo médio de 48 segundos por mutante. Conforme mencionado anteriormente, o sujeito S7 analisou 20 mutantes, diferente dos outros sujeitos que analisaram todos os 26 mutantes gerados. Assim, o tempo total de análise desse sujeito é de 25 minutos, porém o tempo médio de cada mutante é de 1 minuto e 25 segundos, uma vez que ele analisou menor quantidade de mutantes.

QPS 2 - O operador de mutação possui influência no custo da análise dos mutantes?

Como o principal fator relacionado ao custo da análise dos mutantes é o tempo gasto com essa análise, foi verificada a influência do operador de mutação no custo da análise dos mutantes por meio do tempo gasto no mutante gerado por cada operador.

Relacionando o tempo gasto com a análise ao operador de mutação que originou a mudança imposta, na Tabela 5.8, estão descritos os operadores que originaram os mutantes com os três maiores tempos de cada sujeito do experimento utilizando a DiffMutAnalyze. Considerando o tempo gasto com a análise de todos os sujeitos, o operador que ocupou a primeira posição em maior quantidade de sujeitos foi o EVR e foi o operador que gerou a mutação com maior demanda de tempo de análise, onde o sujeito S11 gastou 2 minutos e 59 segundos para analisar esse mutante. Como mencionando anteriormente, o sujeito S11 obteve a maior média de tempo de análise (Tabela 5.6). Dessa forma, a média do sujeito S11 foi de 30 segundos analisando cada mutante e, nesse mutante específico, o tempo gasto foi de 2 minutos e 29 segundo a mais que a média desse sujeito. Na segunda posição, o operador ROR foi o segundo maior tempo na realização da análise e, na terceira posição, o operador LVR obteve o terceiro maior tempo para a maioria dos sujeitos. Como observado na tabela, os operadores ROR e LVR apareceram como primeiro maior tempo em alguns sujeitos. Assim, os operadores de mutação EVR, ROR e LVR foram considerados como os operadores que geram as mutações que demandam mais tempo de análise, aumentando o custo da mutação.

Tabela 5.8 – Operadores de mutação que geraram os mutantes com os maiores tempos de análise de cada sujeito utilizando a DiffMutAnalyze.

Sujeito	1º maior tempo	2º maior tempo	3º maior tempo
S1	ROR	LVR	ROR
S2	LVR	ROR	LVR
S3	ROR	EVR	ROR
S4	EVR	LVR	LVR
S5	EVR	ROR	LVR
S6	EVR	LVR	AOR
S7	ROR	AOR	LVR
S8	ROR	EVR	AOR
S9	EVR	ROR	COR
S10	EVR	ROR	LVR
S11	EVR	ROR	ROR

Fonte: Elaborado pela autora (2019).

Em relação aos operadores de mutação que geraram os mutantes com os maiores tempos de análise durante a análise manual, na Tabela 5.9, o operador EVR foi o operador com o maior tempo gasto na análise pela maior quantidade de sujeitos, assim como ocorreu na análise realizada pela DiffMutAnalyze. Na análise manual, o maior tempo utilizado para verificar o mutante gerado por esse operador foi de 5 minutos e 29 segundos, realizado pelo sujeito S4, que obteve o maior tempo médio de análise (Tabela 5.7), dentre os demais sujeitos. O segundo maior tempo da maioria dos sujeitos, também foi da análise de um mutantes gerado pelo operador EVR e o terceiro maior tempo, foi da maioria gerado pelo operador LVR.

Tabela 5.9 – Operadores de mutação que gerou os mutantes com os maiores tempos de análise de cada sujeito na análise manual.

Sujeito	1º maior tempo	2º maior tempo	3º maior tempo
S1	LVR	EVR	EVR
S2	LVR	LVR	LVR
S3	EVR	EVR	LVR
S4	EVR	ROR	AOR
S5	EVR	ROR	LVR
S6	LVR	EVR	LVR
S7	EVR	LVR	LVR
S8	EVR	ROR	ROR
S9	LVR	EVR	ROR
S10	EVR	LVR	LVR
S11	EVR	EVR	ROR

Fonte: Elaborado pela autora (2019).

QPS 3 - Os erros da definição dos mutantes equivalentes possuem influência no custo?

Como o objetivo da análise era definir se os mutantes são equivalentes ao código original, a seguir, estão descritas as quantidades de mutantes definidos pelos sujeitos como equivalentes e não equivalentes ao código original. Considerando a análise realizada pela DiffMutAnalyze (Tabela 5.10), dentre os onze sujeitos participantes do experimento, dez definiram algum mutante como equivalente e o sujeito S1 não definiu algum mutante como sendo equivalente ao código original. Em contrapartida, o sujeito S11 definiu 19 mutantes como sendo equivalentes, dentre os 38 mutantes analisados. Uma vez que seis mutantes do algoritmo *InsertionSort* são equivalentes ao código original (Tabela 4.1). Os demais sujeitos definiram entre 1 a 7 mutantes como equivalentes ao original.

Tabela 5.10 – Quantidade de mutantes definidos como equivalentes na análise na DiffMutAnalyze.

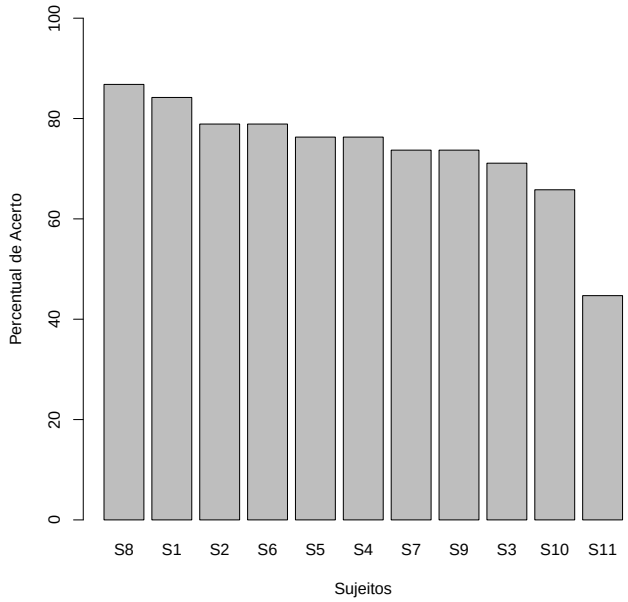
Sujeito	**Quantidade de mutantes equivalentes**	**Quantidade de mutantes não-equivalentes**
S1	0	38
S2	2	36
S3	5	33
S4	5	33
S5	3	35
S6	2	36
S7	3	35
S8	1	37
S9	4	34
S10	7	31
S11	19	19

Fonte: Elaborado pela autora (2019).

Por meio das definições da equivalência dos mutantes, foi possível verificar o percentual de erros e acertos dos sujeitos. Nesse contexto, um erro cometido pelo sujeito, é definir um mutante como equivalente e o mesmo não possuir equivalência relacionada com o código original e o mesmo ocorre com o inverso. Dessa forma, na Figura 5.1, está representado o percentual de acerto dos sujeitos por meio de gráfico, o sujeito S8 obteve o maior percentual de acerto, com 86,8%. Em seguida, o sujeito S1 obteve 84,2% de acerto. O sujeito S11 obteve o menor percentual de acerto, acertando 44,7% dos mutantes. Os demais sujeitos acertaram entre 60% e 80% dos mutantes.

Em relação à definição da equivalência na análise manual, sem a utilização da DiffMutAnalyze, alguns mutantes foram definidos como equivalentes pelos sujeitos, com exceção do sujeito S8, que definiu todos os 26 mutantes gerados como não equivalentes, conforme descrito

Figura 5.1 – Quantidade de acertos na análise da equivalência dos códigos por meio da DiffMutAnalyze de cada sujeito.



Fonte: Elaborado pela autora (2019).

na Tabela 5.11. Para os mutantes definidos como equivalentes, o sujeito S11 definiu 13 mutantes equivalentes, dentre os 26 mutantes gerados do algoritmo *SelectionSort*. Lembrando que sete mutantes desse algoritmo são equivalentes ao código original (Tabela 4.1).

Tabela 5.11 – Quantidade de mutantes definidos como equivalentes durante a análise manual.

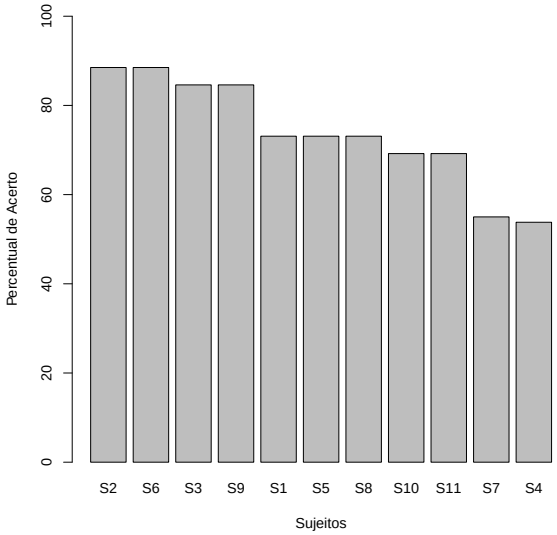
Sujeito	**Quantidade de mutantes equivalentes**	**Quantidade de mutantes não-equivalentes**
S1	2	24
S2	6	20
S3	7	19
S4	5	21
S5	4	22
S6	6	20
S7	2	18
S8	0	26
S9	3	23
S10	5	21
S11	13	13

Fonte: Elaborado pela autora (2019).

Em relação à quantidade de erros e acertos cometidos pelos sujeitos, quando analisaram o algoritmo *SelectionSort*, está representado no gráfico da Figura 5.2, nesse gráfico estão contidos os percentuais de acerto de cada sujeito. Os sujeitos com o maior percentual de acertos durante a análise foram o sujeito S2 e o S6, atingindo 88,5% das asserções cada um deles. Os

sujeitos S3 e S9 obtiveram 84,6% de acertos cada. Os sujeitos S1, S5 e S8 acertaram 73,1% dos mutantes, os sujeitos S10 e S11 acertaram 69,2% e o sujeito S7 acertou 55,0%. O menor percentual de acerto foi do sujeito S4, no qual atingiu o percentual de 53,8% de mutantes definidos corretamente.

Figura 5.2 – Quantidade de acertos na análise manual da equivalência dos códigos de cada sujeito.

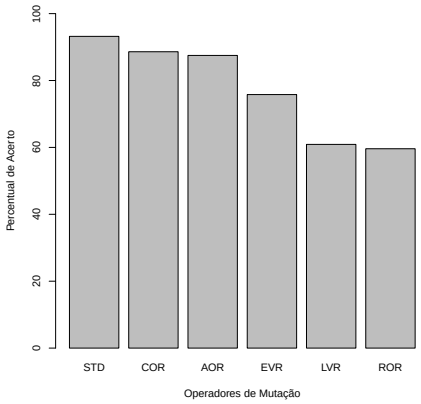


Fonte: Elaborado pela autora (2019).

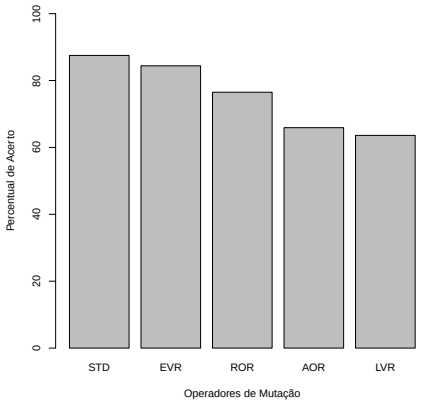
Além de verificar as asserções dos mutantes equivalentes e não equivalentes dos sujeitos, foi verificado o percentual de acerto dos mutantes originados pelos operadores de mutação, ou seja, foi obtido o percentual de asserções dos mutantes gerados em cada operador, considerando a análise de todos os sujeitos. Na Figura 5.3, estão relacionadas os percentuais de acertos da análise dos algoritmos *IsertionSort* e *SelectionSort*. Na análise do algoritmo *IsertionSort* (Figura 5.3(a)), realizada na DiffMutAnalyze, os sujeitos acertaram 93,2% dos mutantes gerados pelo operador STD. O menor percentual de acertos, é dos mutantes gerados pelo operador ROR, uma vez que os sujeitos obtiveram 59,6% das asserções dos mutantes desse operador. Em relação aos operadores de mutação aplicados no algoritmo *SelectionSort* (Figura 5.3(b)), o maior percentual de acerto foi para os mutantes gerados pelo operador STD, sendo 87,5% dos mutantes gerados por esse operador definidos corretamente. O menor percentual foi dos mutantes gerados pelo operador LVR, sendo 63,6% de acertos.

Com relação à influência dos erros cometidos pelos sujeitos no custo da análise, é possível perceber que há influência quando o sujeito define um mutante como não-equivalente, sendo que o mesmo possui equivalência com o código original, pois o analista gastará tempo tentando criar um caso de teste para tentar matar o determinado mutante, uma vez que não há teste de

Figura 5.3 – Percentual de acertos durante a análise dos algoritmos *IsertionSort* e *SelectionSort* em relação aos operadores de mutação aplicados.



(a) Percentual de acerto dos operadores aplicados no algoritmo *IsertionSort*.



(b) Percentual de acerto dos operadores aplicados no algoritmo *SelectionSort*.

Fonte: Elaborado pela autora (2019).

unidade que consiga identificar a alteração de um mutante equivalente. Em contrapartida, se o contrário ocorre, ou seja, quando o sujeito define um mutante como equivalente e o mesmo não é equivalente, o analista não criará um novo caso de teste para tentar matar esse mutante, assim, estará comprometendo a qualidade do conjunto de testes, pois era necessário criar testes de unidade para tentar identificar essa alteração. Dessa forma, quando o sujeito comete erros durante a análise dos mutantes, tem como consequência o tempo e a qualidade, uma vez que podem ser gastos esforços sem a garantia de sucesso e a qualidade pode ser comprometida quando não se produz novos casos de testes que seriam necessários.

QPS 4 - Qual é o grau de dificuldade em analisar os mutantes?

Para o grau de dificuldade da análise dos mutantes na DiffMutAnalyze, na Tabela 5.12, está descrita a quantidade de definições realizadas pelos sujeitos em cada nível estabelecido para o grau de dificuldade da análise. A maioria dos sujeitos definiu o grau de dificuldade em analisar os mutantes do algoritmo *InsertionSort* como fáceis de serem analisados no geral. Os sujeitos S2, S3, S5, S6, S7, S8, S9 e S10 definiram maior quantidade de mutantes como muito fáceis ou fáceis de serem analisados, quando comparado com os outros níveis de dificuldade: médio, difícil ou muito difícil. O sujeito S1 definiu a maioria dos mutantes como dificuldade média, sendo 36 mutantes considerados nesse nível de dificuldade. O sujeito S4 definiu metade dos mutantes como fáceis de analisar, 15 mutantes como médio e quatro como difícil de serem analisados. O sujeito S11 definiu maior parte dos mutantes com o grau de dificuldade médio,

difícil ou muito difícil de serem analisados, somando 24 mutantes definidos nesses níveis, os outros 14 mutantes foram definidos como muito fácil ou fácil de serem analisados. Uma obervação a ser mencionada, é sobre o sujeito S7, o mesmo analisou os 38 mutantes gerados e definiu sua equivalência, porém, em um mutante a definição da dificuldade em analisá-lo não foi atribuída. Dessa forma, a soma total dos níveis da tabela, desse sujeito, corresponde a 37.

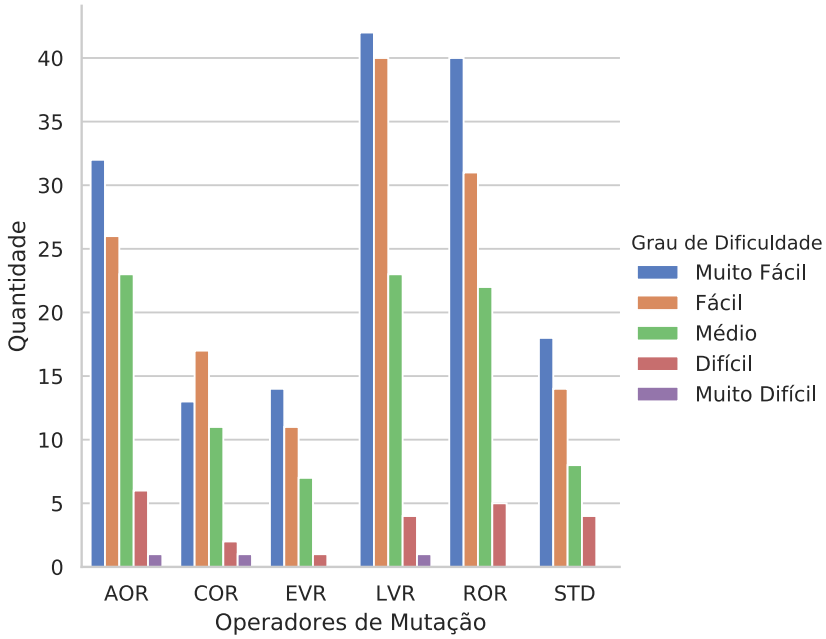
Tabela 5.12 – Grau de dificuldade em analisar os mutantes utilizando a DiffMutAnalyze.

Sujeito	Muito fácil	Fácil	Médio	Difícil	Muito difícil
S1	0	1	36	1	0
S2	30	7	1	0	0
S3	28	7	3	0	0
S4	0	19	15	4	0
S5	11	14	7	6	0
S6	26	8	3	1	0
S7	16	13	8	0	0
S8	13	22	2	1	0
S9	30	5	1	2	0
S10	0	34	4	0	0
S11	5	9	14	7	3

Fonte: Elaborado pela autora (2019).

Considerando o grau de dificuldade em analisar os mutantes gerados pelos operadores de mutação aplicados no algoritmo *InsertionSort*, na Figura 5.4, está relacionada a quantidade de mutantes definidos pelos sujeitos em cada nível de dificuldade de acordo com o operador de mutação. Todos os operadores tiveram a maioria dos seus mutantes definidos como muito fáceis e fáceis de serem analisados. As maiores quantidades de dificuldade média durante a análise dos mutantes estão relacionadas com os operadores de mutação AOR, LVR e ROR, sendo 23, 23 e 22 definições nesse nível de dificuldade respectivamente. Dentre as 22 análises definidas com o grau de dificuldade alto, os operadores AOR (6), ROR (5), LVR (4) e STD (4) apresentam as maiores quantidades de análises definidas nesse nível de dificuldade.

Figura 5.4 – Quantidade de mutantes definidos em cada nível do grau de dificuldade de acordo com o operador de mutação aplicado no algoritmo *InsertionSort*.



Fonte: Elaborado pela autora (2019).

Na Tabela 5.13, está descrito o grau de dificuldade das análises realizadas manualmente de cada sujeito, distribuído de acordo com o nível de dificuldade.

Tabela 5.13 – Grau de dificuldade em inspecionar os mutantes durante a análise manual.

Sujeito	Muito fácil	Fácil	Médio	Difícil	Muito difícil
S1	0	5	16	3	2
S2	17	7	1	1	0
S3	15	6	5	0	0
S4	2	10	10	4	0
S5	9	11	3	2	1
S6	20	4	1	1	0
S7	6	13	1	0	0
S8	10	16	0	0	0
S9	18	5	3	0	0
S10	0	34	4	0	0
S11	0	8	12	5	1

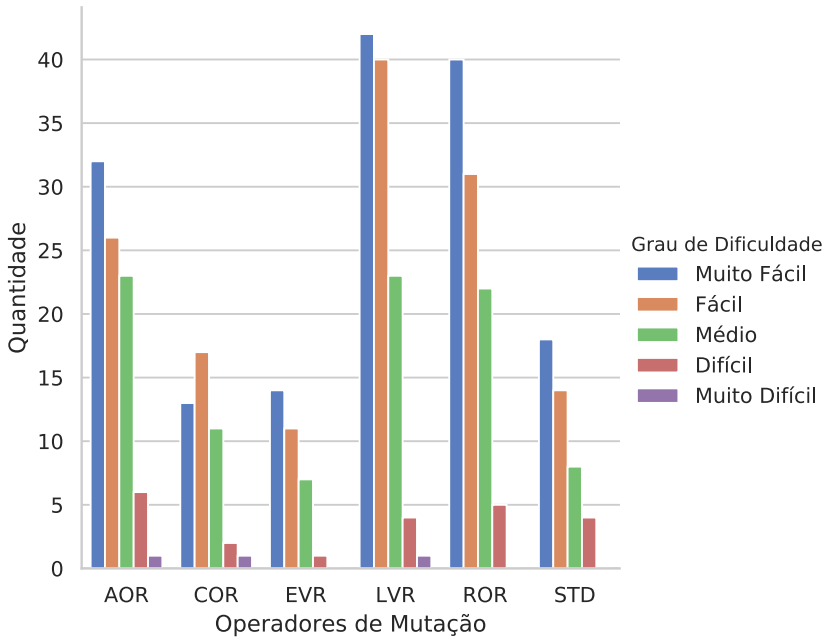
Fonte: Elaborado pela autora (2019).

O mesmo cenário se repete nessa análise, ou seja, a maioria dos sujeitos consideram os mutantes do algoritmo *SelectionSort* fáceis de serem analisados. Dessa forma, os mesmos sujeitos que definiram maior quantidade de mutantes como sendo fáceis de serem analisados na ferramenta DiffMutAnalyze, também definiram na análise manual, sendo os sujeitos S2, S3, S5, S6, S7, S8, S9 e S10. O sujeito S1 definiu o grau de dificuldade fácil para cinco

mutantes e os outros 21 mutantes como médio, difícil ou muito difícil de serem analisados. O sujeito S4 definiu 12 mutantes como muito fácil ou fácil e os outros 14 mutantes como médio ou difícil de serem analisados. Por fim, o sujeito S11 definiu 8 mutantes como fáceis e os demais mutantes como médio, difícil ou muito difícil de verificar sua equivalência com o código original, somando 18 mutantes.

Em relação ao grau de dificuldade definido nas análises do mutantes originados pelos operadores de mutação, na Figura 5.5, estão contidas as quantidades de dificuldades em cada operador de mutação. Os operadores ROR e LVR foram os que tiveram maior quantidade de mutantes analisados pelos sujeitos no total, dessa forma, o operador ROR obteve 80 definições do grau de dificuldade muito fácil e fácil, enquanto que 18 definições foram dos níveis de dificuldade médio e alto. Para o operador LVR, somando as definições de dificuldade de todos os sujeitos, 41 análises foram definidas como muito fáceis e fáceis de serem realizadas, para as análises definidas como média e alta, somam 25 registros nesses níveis de dificuldades.

Figura 5.5 – Quantidade de mutantes definidos em cada nível do grau de dificuldade de acordo com o operador de mutação aplicado no algoritmo *SelectionSort*.



Fonte: Elaborado pela autora (2019).

5.2.3 Discussão

Nesta seção serão discutidos os resultados obtidos pelo experimento. Dessa forma, algumas comparações foram realizadas e analisadas para verificar o custo em analisar os mutantes. Inicialmente, está descrito o tempo gasto com a análise, comparando a análise manual com a

análise realizada pela DiffMutAnalyze. Além disso, os mutantes equivalentes definidos pelos sujeitos participantes do experimento foram relacionados com o tempo e o grau de dificuldade em analisá-los.

A - TEMPO DE ANÁLISE

O principal fator a ser explorado para indicar o custo da análise é o tempo gasto para analisar os mutantes, sendo um importante fator a ser reduzido, fazendo com que a análise seja rápida e seja possível diminuir gastos com horas de um analista. Por meio da comparação dos dados do tempo gasto com a análise dos mutantes, é possível responder a questão de pesquisa secundária: ***QPS 5 - Utilizando a ferramenta DiffMutAnalyze é possível reduzir o tempo da análise?***. Portanto, o tempo da análise foi contabilizado durante o experimento e foi observado que a ferramenta DiffMutAnalyze contribuiu para a redução do tempo de análise dos mutantes. Na Tabela 5.14, consta uma comparação do tempo médio que cada sujeito gastou para inspecionar cada mutante, tanto na análise manual quanto na análise realizada por meio da DiffMutAnalyze.

O tempo médio gasto para analisar os mutantes utilizando a ferramenta DiffMutAnalyze está contido na coluna "Tempo médio - DiffMutAnalyze" e o tempo gasto para realizar a análise manual, está descrito na coluna "Tempo médio - Manual". Além disso, está descrito na tabela a forma com que cada sujeito iniciou a análise, ou seja, se a análise foi realizada inicialmente pela DiffMutAnalyze (inspecionando o algoritmo *InsertionSort*) ou se foi iniciada manualmente (inspecionando o algoritmo *SelectionSort*), assim, essa informação está contida na coluna "Análise inicial".

Tabela 5.14 – Comparação do tempo médio da análise por mutante.

Sujeito	Tempo médio - DiffMutAnalyze	Tempo médio - Manual	Análise inicial
S1	00:00:19	00:01:58	Manual
S2	00:00:19	00:01:31	Manual
S3	00:00:16	00:01:00	Manual
S4	00:00:24	00:02:03	Manual
S5	00:00:25	00:00:48	DiffMutAnalyze
S6	00:00:17	00:00:51	DiffMutAnalyze
S7	00:00:16	00:01:25	Manual
S8	00:00:22	00:01:00	Manual
S9	00:00:22	00:01:06	DiffMutAnalyze
S10	00:00:25	00:00:48	DiffMutAnalyze
S11	00:00:30	00:01:00	DiffMutAnalyze

Fonte: Elaborado pela autora (2019).

Como pode ser observado na Tabela 5.14, todos os sujeitos obtiveram menor tempo médio utilizando a ferramenta DiffMutAnalyze. Considerando a maneira que o sujeito iniciou a análise, é possível perceber que a maioria dos sujeitos obteve maior tempo de análise quando comparado ao segundo grupo, ou seja, que fez a mesma análise depois que inverteu a forma de analisar os mutantes (da análise manual para DiffMutAnalyze e vice-versa). Analisando o tempo médio da coluna "Tempo médio - Manual", os sujeitos S1, S2, S3, S4, S7 e S8 iniciaram suas análises manualmente, assim é possível perceber que quase todos esses sujeitos obtiveram maior tempo de análise, quando comparado com os sujeitos que iniciaram pela DiffMutAnalyze (S5, S6, S9, S10 e S11) e depois realizaram a análise manual. Houve uma exceção, o sujeito S9 iniciou a análise pela DiffMutAnalyze e quando realizou a análise manual, obteve 6 segundos a mais que o menor tempo (1 minuto) dos sujeitos que iniciaram a análise manualmente, ou seja, o sujeito S9 obteve o tempo médio de 1 minuto e 6 segundos, obtendo 6 segundos a mais que os sujeitos S3 e S8, que possuem o tempo médio de 1 minuto.

O mesmo ocorre quando comparados os sujeitos que iniciaram a análise pela DiffMutAnalyze (coluna "Tempo médio - DiffMutAnalyze"), os maiores tempos de análise são dos sujeitos que iniciaram pela ferramenta, sendo os sujeitos S5, S10 e S11. Os sujeitos S1, S2, S3, e S7 que analisaram inicialmente sem a ferramenta, contabilizaram os menores tempos de análise. Nesses dois cenários, é possível indicar uma exceção em cada um deles, o sujeito S4 obteve o tempo médio de 24 segundos utilizando a DiffMutAnalyze, porém sua análise inicial foi manual. E o inverso ocorre com o sujeito S6, que gastou em média 17 segundos para analisar utilizando a DiffMutAnalyze, sendo que essa foi sua primeira forma de análise.

Ao inverter os grupos para analisar os mutantes, utilizando a DiffMutAnalyze ou manualmente, faz com que os sujeitos adquiram conhecimento sobre as mutações realizadas e utilizem esse conhecimento para realizar a próxima análise. Como os algoritmos analisado são parecidos, os sujeitos conseguem compreender com maior rapidez o funcionamento do algoritmo e as mutações realizadas na segunda análise.

B - MUTANTES EQUIVALENTES

Na seção resultados (Seção 5.2.2), foi mencionado que os sujeitos definiram alguns mutantes como equivalentes ao código original, esses mutantes estão descritos nas Tabelas 5.15 e 5.16. Inicialmente, será descrito sobre as definições de equivalência durante as análises realizadas na DiffMutAnalyze, contidas na Tabela 5.15. Nessa tabela não consta o sujeito S1, pois o mesmo não definiu algum mutante como equivalente. Dentre os sujeitos que definiram

algum mutantes como equivalente, relacionando esses mutantes com o tempo médio de análise e o grau de dificuldade indicado para cada mutante, é possível perceber que as análises desses mutantes obtiveram tempo maior que a média de alguns sujeitos. O sujeito S2 definiu os mutantes 4 e 10 como equivalentes, com o tempo de análise de 35 e 42 segundos respectivamente. Ambos foram definidos como fáceis de serem analisados. Porém, comparando com o tempo médio de análise desse sujeito, esses mutantes possuem tempo acima da média, uma vez que o sujeito S2 obteve tempo médio de 19 segundos por mutante.

Tabela 5.15 – Mutantes definidos pelos sujeitos como equivalentes na DiffMutAnalyze.

Sujeito	Mutante	Dificuldade	Tempo
S2	4	Fácil	00:00:35
	10	Fácil	00:00:42
S3	4	Muito fácil	00:00:21
	31	Média	00:00:34
	10	Fácil	00:00:15
	19	Média	00:00:25
	34	Média	00:00:26
S4	4	Média	00:02:06
	10	Fácil	00:00:16
	12	Fácil	00:00:11
	3	Média	00:00:18
	33	Média	00:00:10
S5	10	Muito fácil	00:00:26
	5	Média	00:00:47
	34	Difícil	00:00:25
S6	4	Fácil	00:01:56
	10	Fácil	00:00:22
S7	26	Média	00:00:23
	21	Média	00:00:14
	8	Média	00:00:14
S8	28	Fácil	00:00:39
S9	4	Fácil	00:02:00
	31	Média	00:01:07
	10	Fácil	00:00:20
	34	Fácil	00:00:38
S10	4	Fácil	00:02:34
	10	Média	00:00:11
	22	Média	00:00:23
	12	Fácil	00:00:15
	5	Fácil	00:00:19
	11	Fácil	00:00:11
	9	Fácil	00:00:14
S11	4	Fácil	00:02:59
	17	Muito fácil	00:00:16
	31	Difícil	00:01:24
	37	Média	00:01:03
	38	Muito fácil	00:00:16
	14	Fácil	00:00:39
	23	Média	00:00:14
	10	Fácil	00:00:41
	27	Muito fácil	00:00:10
	29	Média	00:00:11
	26	Média	00:00:24
	21	Média	00:00:27
	24	Fácil	00:00:11
	8	Média	00:00:20
	2	Fácil	00:00:33
	13	Difícil	00:00:16
	3	Média	00:00:16
	16	Média	00:00:07
	35	Fácil	00:00:09

Fonte: Elaborado pela autora (2019)

O sujeito S3 afirmou que cinco mutantes são equivalentes, quatro desses mutantes estão com o tempo acima da média desse sujeito, sendo a média geral do sujeito S3 de 16 segundos e os mutantes 4, 31, 19 e 34 gastaram 21, 34, 25 e 26 segundos respectivamente. Apenas o mutante 10 foi definido em menor tempo (15 segundos) que a média desse sujeito. Para esses mutantes, o grau de dificuldade foi definido como muito fácil para o mutante 4, a dificuldade média foi definida para os mutantes 31, 19 e 34 e o mutante 10 foi de fácil análise. O sujeito S4 difere um pouco dos outros sujeitos, uma vez que somente o mutante 4 está muito acima da média geral (24 segundos), sendo que esse mutante gastou um pouco mais de dois minutos para ser analisado, porém, os demais mutantes ficaram abaixo do tempo médio geral. Além disso, o

mutante 4 foi definido como dificuldade média, assim como os mutantes 3 e 33. Os mutantes 10 e 12 foram definidos como fáceis.

O sujeito S5 obteve o tempo médio de 25 segundos e os mutantes definidos como equivalentes possuem tempo maior ou igual a essa média. Sendo o mutante 10 analisado em 26 segundos e definido como muito fácil de ser analisado, o mutante 5 gastou 47 segundos para ser analisado e foi definido como dificuldade média, o mutante 34 obteve o tempo igual a média e foi definido como difícil de ser analisado. O sujeito S6 definiu dois mutantes como equivalentes e os dois estão com o tempo acima da média geral (17 segundos), sendo os mutantes 4 e 10, com o tempo de análise de 1 minuto e 56 segundos para o mutante 4 e 22 segundos para o 10, ambos com grau de dificuldade fácil.

O sujeito S7 definiu três mutantes como equivalentes, sendo eles o 26, 21 e 8, todos foram considerados como dificuldade média. A média geral desse sujeito é de 16 segundos, assim o mutante 26 gastou 23 segundos e os mutantes 21 e 8 gastaram 14 segundos cada, ficando com o tempo abaixo da média. O sujeito S8 definiu apenas um mutante como equivalente, sendo gasto 39 segundos de análise e a média geral desse sujeito é de 22 segundos. O mutante definido como equivalente foi o 28 e indicado como fácil de ser analisado.

O sujeito S9 obteve o tempo médio geral de 22 segundos, porém ao analisar os mutantes 4 e 31 foi gasto um tempo muito superior a média, sendo gastos 2 minutos para analisar o mutante 4 e 1 minuto e 7 segundos para o 31. O mutante 34 também ficou acima da média, com 38 segundos de duração da análise. O mutante 10 teve um tempo um pouco abaixo da média, igual a 20 segundos. Para o grau de dificuldade, o mutante 31 foi definido como médio e os demais como fáceis de serem analisados. O sujeito S10 possui um cenário semelhante ao sujeito S4, onde os mutantes definidos como equivalentes obtiveram tempo menor que a sua média geral (25 segundos). Dentre os sete mutantes determinados como equivalentes, somente o mutante 4 está acima da média, com 2 minutos e 34 segundos de análise. Os mutantes 4, 12, 5, 11 e 9 foram definidos como fáceis de serem analisados e os mutantes 10 e 22 como médio.

O sujeito S11 foi o que definiu maior quantidade de mutantes equivalentes, sendo 19 mutantes, metade dos mutantes analisados. O tempo médio desse sujeito é de 30 segundos e, diante desses 19 mutantes, seis mutantes estão com o tempo acima da média e os outros 13, abaixo da média geral. O maior tempo de análise desses mutantes foi de 2 minutos e 59 segundos, para o mutante 4, o menor tempo gasto analisando o mutante foi de 7 segundos do

mutante 16. Dois mutantes foram indicados como difíceis de serem analisados, oito mutantes foram definidos como médio e os outros nove foram definidos como muito fácil ou fácil.

Na análise manual, sem a utilização da DiffMutAnalyze, alguns mutantes do algoritmo *SelectionSort* foram definidos como equivalentes. Na Tabela 5.16, estão representados os mutantes equivalentes de acordo com a definição de cada sujeito. Para analisar esses mutantes, o tempo de análise individual de cada mutante não poderá ser comparado com o tempo médio dos sujeitos, uma vez que o tempo médio contempla a análise e o tempo gasto com os outros procedimentos realizados. Porém, como o tempo descrito nessa tabela é o tempo de análise do mutante, está desconsiderando o processo de abertura e identificação da alteração. Nesse caso, a análise do tempo será realizada em função da soma do tempo dos mutantes por cada sujeito.

Tabela 5.16 – Mutantes definidos pelos sujeitos como equivalentes na análise manual.

Sujeito	Mutante	Dificuldade	Tempo
S1	14	Difícil	00:01:18
	24	Muito difícil	00:02:01
S2	4	Muito fácil	00:00:15
	8	Difícil	00:02:15
	10	Fácil	00:00:32
	11	Fácil	00:00:14
	13	Muito fácil	00:00:03
	14	Muito fácil	00:00:11
S3	8	Média	00:00:49
	11	Muito fácil	00:00:25
	13	Fácil	00:00:18
	14	Muito fácil	00:00:17
	20	Fácil	00:00:26
	22	Fácil	00:00:46
	24	Média	00:01:01
S4	2	Fácil	00:01:01
	3	Média	00:01:17
	7	Muito fácil	00:00:35
	16	Difícil	00:00:52
	18	Difícil	00:00:38
S5	5	Fácil	00:00:31
	8	Média	00:00:28
	15	Difícil	00:00:41
	20	Fácil	00:00:13
S6	4	Muito fácil	00:00:06
	8	Difícil	00:00:54
	9	Média	00:00:11
	11	Muito fácil	00:00:04
	13	Muito fácil	00:00:08
	14	Muito fácil	00:00:07
S7	2	Fácil	00:01:14
	3	Fácil	00:01:17
S9	4	Muito fácil	00:00:43
	14	Muito fácil	00:00:35
	20	Fácil	00:00:11
S10	4	Fácil	00:00:19
	7	Fácil	00:00:13
	14	Fácil	00:00:11
	18	Fácil	00:00:11
	25	Fácil	00:00:15
S11	4	Fácil	00:00:11
	5	Média	00:00:36
	8	Média	00:00:37
	11	Média	00:00:26
	13	Fácil	00:00:12
	14	Difícil	00:00:28
	15	Fácil	00:00:23
	19	Média	00:00:39
	20	Média	00:00:23
	22	Média	00:00:18
	23	Média	00:00:11
	24	Difícil	00:00:25
	25	Fácil	00:00:22

Fonte: Elaborado pela autora (2019)

Para o sujeito S1 foram gastos 3 minutos e 19 segundos para analisar os dois mutantes definidos como equivalentes, esse tempo de análise demandou um esforço de 11,2% do tempo total de análise desse sujeito. Uma vez que o percentual médio de cada mutante, corresponde a 3,85% do tempo total dos 26 mutantes analisados. Assim, a soma do percentual médio de dois mutantes é de 7,7% do total. Dessa forma, a análise desses dois mutantes está 3,5% acima

do percentual médio. Além disso, esses dois mutantes (14 e 24) foram considerados difícil e muito difícil de serem analisados respectivamente. Comparado ao tempo médio de análise do sujeito S1, quando utilizada a DiffMutAnalyze, que foi de 19 segundos, na análise manual, o tempo desses dois mutantes está muito acima dessa média. É interessante mencionar que o tempo contabilizado na análise manual, não está incluso o tempo de abertura do mutante, somente o tempo de comparação dos códigos. Como a DiffMutAnalyze disponibiliza o próximo mutante instantaneamente, o tempo de "abertura" do mutante está incluso na análise realizada na ferramenta.

Para o sujeito S2, a soma do tempo de análise dos seis mutantes definidos como equivalente é de 3 minutos e 30 segundos, o que corresponde a 35,3% do tempo total. O percentual médio para analisar seis mutantes é de 23,1%, dessa forma, o sujeito S2 obteve maior percentual de tempo analisando esses mutantes definidos por ele como equivalentes. Se comparado com o tempo médio obtido por meio da DiffMutAnalyze (19 segundos) desse sujeito, alguns mutantes da análise manual estão abaixo dessa média, sendo os mutantes 4, 11, 13 e 14. Os mutante 8 e 10 estão acima dessa média, com uma atenção ao mutante 8 que consumiu o maior tempo dentre eles e sendo considerado difícil de ser analisado. Os outros mutantes foram considerados como muito fáceis ou fáceis de serem analisados.

Na análise do sujeito S3, sete mutantes foram considerados equivalentes ao código original, somando 4 minutos e 2 segundos de análise, o que equivale a 36,1% do tempo. Sete mutantes equivale, em média, 26,9% do tempo, assim, o tempo gasto com esses mutantes está acima da média. Na análise utilizando a DiffMutAnalyze, o sujeito S3 obteve o tempo médio de 16 segundos, o que significa que todos os mutantes definidos como equivalentes na análise manual, possui o tempo de análise superior a média obtida na DiffMutAnalyze. Os mutantes 8 e 24 foram definidos como dificuldade média e são os que possuem maior tempo de análise dentre os demais, gastando 49 segundos no mutante 8 e pouco mais de 1 minuto no mutante 24. Os outros mutantes foram fáceis ou muito fáceis de serem analisados.

O sujeito S4 estabeleceu cinco mutantes como equivalentes. O tempo de análise desses mutantes somam 4 minutos e 23 segundos, equivalente a 16,1% do tempo total. Para analisar cinco mutantes, o percentual médio do tempo é de 19,2%, assim, esses mutantes definidos como equivalentes gastaram um percentual menor que o do tempo médio. Quando comparado à análise realizada na DiffMutAnalyze, o tempo médio desse sujeito foi de 24 segundos, dessa forma, os mutantes analisados de maneira manual possuem maior tempo de análise que a média obtida

por meio da ferramenta. Os maiores tempos de análise dos mutantes considerados equivalentes são para os mutantes 2 e 3, sendo gastos 1 minuto e 1 segundo e 1 minuto e 17 segundos respectivamente, esses mutantes foram considerados como fácil e médio de serem analisados. Os mutantes definidos como difíceis de serem analisados são: 16 e 18.

O sujeito S5 gastou 1 minuto e 53 segundos para analisar os quatro mutantes determinados como equivalentes, obtendo 15,0% do tempo de análise, sendo o tempo médio de 15,4%, dessa forma, o percentual gasto está próximo da média, mas está um pouco abaixo. O tempo médio obtido na DiffMutAnalyze desse sujeito foi de 25 segundos e, quando comparado com a análise manual, o tempo gasto com os mutantes 5, 8 e 15 foi superior à essa média, somente o mutante 20 ficou abaixo da média, com 13 segundos de análise. O mutante 15 obteve o maior tempo gasto e foi considerado como difícil de ser analisado, o mutante 8 foi considerado como médio e os mutantes 5 e 20 como fáceis.

Seis mutantes foram registrados como equivalentes pelo sujeito S6 e foram gastos 1 minuto e 30 segundos para analisá-los, o que corresponde a 29% do tempo total. O percentual médio do tempo para analisar seis mutantes é de 23,1%, dessa forma, o percentual gasto é maior que o percentual médio, gastando mais tempo com a análise desses mutantes. Na DiffMutAnalyze, o tempo médio desse sujeito é de 17 segundos, quase todos os mutantes definidos como equivalentes na análise manual (4, 9, 11, 13 e 14) estão abaixo dessa média, com exceção do mutante 8, que possui o tempo de análise de 54 segundos, além disso, foi considerado como difícil de ser analisado.

O sujeito S7 determinou dois mutantes como equivalentes e foram gastos 13,9% do tempo analisando-os. Esse sujeito analisou 20 mutantes, com isso, cada mutante analisado equivale a 5% do tempo e não 3,85% como na análise dos outros sujeitos. Portanto, para analisar dois mutantes, o percentual médio é de 10,0% do tempo total. Assim, o percentual do tempo gasto com esses mutantes é superior ao percentual médio. Comparando com o tempo médio obtido por esse sujeito na DiffMutAnalyze, sendo 16 segundos, o tempo analisando os mutantes 2 e 3 definidos como equivalentes na análise manual é de 1 minuto e 14 segundos e 1 minuto e 17 segundos respectivamente, sendo esses tempos superiores ao tempo médio da DiffMutAnalyze. Para o grau de dificuldade, os dois mutantes foram fáceis de serem analisados.

O sujeito S8 não considerou algum mutante como equivalente. O sujeito S9 definiu três mutantes como sendo equivalentes ao código original. Foram gastos 1 minuto e 29 segundos com a análise dos três mutantes, equivalente a 12,3% do tempo total de análise, sendo esse per-

centual maior que o percentual do tempo médio, que equivale a 11,5%. O sujeito S9 obteve o tempo médio de 22 segundos quando utilizou a DiffMutAnalyze, sendo o mutante 4 e 14 sendo analisado em mais tempo (43 e 35 segundos respectivamente) que essa média, em contrapartida, o mutante 20 foi analisado em 11 segundos, ou seja, em tempo menor que a média da ferramenta. Esses mutantes foram fáceis ou muito fáceis de serem analisados.

Na análise do sujeito S10, cinco mutantes foram considerados como equivalentes e foram gastos 1 minuto e 9 segundos em suas análises. Como mencionado anteriormente, o percentual médio do tempo para analisar cinco mutantes é de 19,2%, e para a análise desses mutantes foram gastos 17,5% do tempo, assim, o percentual do tempo de análise desses mutantes foi inferior ao percentual médio. Durante a análise utilizando a DiffMutAnalyze, o sujeito S10 gastou em média 25 segundos para analisar os mutantes. Nesse caso, todos os mutantes considerados equivalentes na análise manual gastaram menos tempo que na análise com a ferramenta. Além disso, os cinco mutantes foram considerados fáceis de serem analisados.

O último sujeito (S11) definiu treze mutantes como equivalentes e analisou-os em 5 minutos e 11 segundos, correspondente a 45,8% do tempo total. Considerando que essa quantidade de mutantes equivalentes é a metade dos mutantes gerados, o percentual médio do tempo é de 50%, dessa forma, esses mutantes gastaram aproximadamente 4% a menos do tempo médio. O tempo médio desse sujeito durante a análise com a ferramenta DiffMutAnalyze é de 30 segundos, comparando com a análise manual, três mutantes estão acima dessa média (mutantes 5, 8 e 19) e os outros dez mutantes estão abaixo da média (mutantes 4, 11, 13, 14, 15, 20, 22, 23, 24 e 25), sendo que três mutantes estão próximos dessa média (mutantes 11, 14 e 24). Com relação ao grau de dificuldade, a maioria dos mutantes foram definidos como médio de serem analisados, quatro foram definidos como fáceis e dois como difíceis.

Quando comparado com o tempo médio de análise utilizando a DiffMutAnalyze, os mutantes definidos como equivalentes durante a análise manual gastaram mais tempo. É importante salientar que na análise manual o tempo de análise não é o tempo total de todo o procedimento realizado para efetuar a identificação se há equivalência entre os códigos. Assim, é possível observar que a DiffMutAnalyze auxilia na redução do tempo de análise dos mutantes.

C - OPERADORES DE MUTAÇÃO

Os operadores de mutação possuem influência no custo da análise, uma vez que são responsáveis por realizar a mutação, alterando o código do sistema de software a ser testado. Considerando os maiores tempos de análise dos sujeitos participantes do experimento, o ope-

rador com maior demanda de tempo, tanto na análise manual quanto na DiffMutAnalyze, foi dos mutantes gerados pelo operador EVR. Sendo o maior tempo gasto na análise manual de um mutante, correspondente a 5 minutos e 29 segundos. Esse mutante foi gerado pelo operador EVR. Na análise realizada por meio da DiffMutAnalyze, o maior tempo foi de 2 minutos e 59 segundos, o mutante que demandou esse tempo de análise foi gerado pelo operador EVR. Nas duas formas de análise, esse operador foi o que demandou maior tempo para verificar se o mutante gerado por ele era equivalente ao seu código original correspondente. Além disso, os outros operadores que demandaram maiores tempos de análise foram LVR e o ROR, conforme Tabelas 5.8 e 5.9 em ambas as análises.

Ao verificar as definições do grau de dificuldade em analisar os mutantes, conforme mencionado anteriormente nos gráficos 5.4 (referente à análise na DiffMutAnalyze) e 5.5 (referente à análise manual), as definições do grau de dificuldade "Alto" e "Muito Alto" estão relacionadas em maior parte nos operadores AOR, LVR e ROR na análise pela DiffMutAnalyze e nos operadores EVR, LVR e ROR na análise manual. Relacionando esses operadores com o tempo gasto com a análise, é possível perceber que os operadores EVR, LVR e ROR geram os mutantes mais difíceis de serem analisados e, consequentemente, demanda maior tempo de análise, sendo assim, esses operadores de mutação contribuem para o aumento do custo da análise dos mutantes.

Além disso, os mutantes equivalentes foram gerados pelos operadores LVR e ROR no algoritmo *InsertionSort* e pelos operadores LVR, ROR e AOR no algoritmo *SelectionSort*, conforme descrito na Tabela 4.1. É importante elucidar que os mutantes equivalentes não contribuem para a melhoria da qualidade do conjunto de casos de teste do sistema de software (JIA; HARMAN, 2011). Dessa forma, além de gerar mutantes difíceis de serem analisados e com maior tempo gasto durante a análise, esses operadores geraram os mutantes equivalentes.

5.2.4 Percepção dos Sujeitos em Relação ao Experimento

Um questionário de caracterização dos participantes do experimento foi enviado a cada sujeito, para que os mesmos pudessem se identificar e descrever seu perfil, conforme descrito na Seção 5.2.1. Além dessa caracterização, os sujeitos responderam algumas perguntas relacionadas ao experimento realizado. Inicialmente, os sujeitos responderam sobre o grau de dificuldade em utilizar a DiffMutAnalyze, sendo que seis sujeitos classificaram-na como fácil de ser utilizada e os outros cinco sujeitos, a consideraram muito fácil.

Ao avaliar a utilização da DiffMutAnalyze, os sujeitos foram questionados sobre o que acharam da ferramenta em geral. De maneira geral, a ferramenta foi considerada uma boa opção para apoiar na análise dos mutantes. Os sujeitos citaram que a ferramenta é bem intuitiva e fácil de ser utilizada: *"Bastante intuitiva."*; *"Muito interessante."*; *"Legal e intuitiva."*; *"Achei bem indutiva e simples."*. Alguns sujeitos mencionaram sobre a visualização dos códigos lado a lado e sobre a indicação do local da mutação, o que facilita a análise. Os comentários relacionados a essa funcionalidade da ferramenta foram: *"Achei interessante, pois mostra exatamente o local onde o código sofreu mutação, facilitando a análise."*; *"Achei muito prático a indicação dos mutantes destacados em cores."*; *"Muito fácil e intuitiva para analisar o código, pois é marcado somente onde foi feita a modificação do código."*; *"Facilita a encontrar os mutantes gerados. Com a análise manual, se perde muito tempo para encontrar o mutante e a ferramenta auxilia de forma objetiva neste ponto."*. Um dos sujeitos citou que a ferramenta pode ser utilizada para o processo de aprendizagem sobre testes: *"Achei uma ótima proposta para detecção de equivalências. E pelo uso nos testes, me pareceu ser uma opção para apoiar até mesmo em questões relacionadas ao processo de aprendizagem de testes, além do de mutação."*.

Como o experimento foi realizado em duas etapas, separando os participantes em dois grupos, sendo um grupo analisando primeiro pela ferramenta e o outro manualmente, os sujeitos foram questionados sobre qual dessas duas formas foi a melhor para analisar os códigos. Assim, considerando a forma de analisar os mutantes, 100% dos sujeitos responderam que foi melhor realizar a análise utilizando a DiffMutAnalyze. Para ambas as respostas da pergunta anterior, foi perguntado o motivo da escolha da maneira em analisar os mutantes, sendo a maioria das respostas indicando a facilidade de encontrar a alteração e, com isso, a análise foi mais rápida, diminuindo o tempo total da análise. Além disso, foi mencionado sobre a abertura dos códigos na análise manual, uma vez que na análise manual, era necessário que os sujeitos abrissem cada mutante para inspecionar o código, encontrar a mutação e verificar se possuía equivalência com o código original, gastando assim, esse tempo a mais que na análise realizada pela DiffMutAnalyze, devido a ela realizar esses procedimentos automaticamente.

Para a avaliação final da ferramenta, foi perguntado sobre os pontos positivos e negativos. Os pontos positivos da ferramenta citados, foram, em sua maioria, as mesmas citações das perguntas anteriores, onde os sujeitos consideram os principais pontos sendo: facilidade, rapidez, praticidade e organização. Com relação aos pontos negativos, os sujeitos citaram sobre a mudança de um código mutante para o outro, ou seja, quando uma análise finaliza e se inicia a

próxima. Dessa forma, os sujeitos sugerem que seja informado que os mutantes de um mesmo código original foi finalizado e um novo código será inspecionado. Por exemplo, se a análise estiver sendo realizada por classes de um sistema de software, é interessante informar ao usuário a mudança de uma classe para a outra. Na DiffMutAnalyze não é exibida essa informação, pois ao mudar de um mutante para o outro, o código mutante e o original são carregados na tela e o usuário consegue identificar a classe (ou código) a ser analisado. Porém, essa funcionalidade pode ser implementada na ferramenta para auxiliar o analista a identificar o que está sendo analisado de forma rápida e eficiente. Outro ponto de melhoria é deixar mais visível os mutantes que estão sendo analisados. Além disso, alguns sujeitos disseram que não haviam pontos negativos. Por fim, os sujeitos foram questionados se a DiffMutAnalyze pode auxiliar na análise dos mutantes e identificação do mutantes equivalentes, todos os sujeitos afirmaram que a ferramenta pode auxiliar na análise.

5.3 Ameaças à Validade

As ameaças à validade de um experimento concentra-se na relação entre a teoria relacionada ao experimento e a observação realizada. Uma ameaça à validade estabelece uma maneira específica que indique que possa haver algum erro no experimento. Assim, há a perspectiva de identificar possíveis ameaças e identificar ações para mitigar essas ameaças (WOHLIN et al., 2012). Dessa forma, algumas ameaças à validade que podem comprometer os resultados do experimento desta pesquisa, podem ser identificadas a seguir.

Validade interna: Nível de Experiência dos Participantes: o conhecimento dos participantes em relação ao projeto utilizado para realizar a mutação pode afetar os dados coletados durante a análise. Para mitigar essa ameaça, foram utilizados algoritmos de ordenação para tentar abranger o conhecimento prévio dos participantes para a análise dos códigos original e mutante, uma vez que os participantes são alunos do curso da área da ciência da computação. Além disso, durante o treinamento, os participantes tiveram uma formação sobre os principais conceitos do teste de mutação e foram treinados sobre como utilizar a ferramenta DiffMutAnalyze. Produtividade sob avaliação: existe a possibilidade de que isso possa influenciar os resultados do experimento, porque os alunos tendem a pensar que estão sendo avaliados pelos resultados das experiências. Para mitigar isso, explicamos aos alunos que ninguém estava sendo avaliado e sua participação era considerada anônima.

Validade da Construção: Marcação do tempo da análise: existe a possibilidade da marcação do tempo da análise não ser real, pois os participantes podem realizar alguma ação que não seja relacionada com a análise dos códigos, fazendo com que o tempo da análise daquele determinado mutante seja corrente até que o participante prossiga para a próxima análise. Para mitigar essa ameaça, foi colocado na ferramenta um botão para pausar o tempo, para que os participantes pudessem parar a contagem do tempo em caso de realizar outra ação que não esteja relacionada com a análise dos mutantes. Dessa forma, foi instruído aos participantes que utilizassem esse recurso caso necessário.

Validade externa: A amostra de sujeitos pode não ser representativa: Como mencionado, o experimento foi realizado por estudantes de pós-graduação, assim, eles possuem conhecimentos acadêmicos. Não se pode excluir a ameaça de que os resultados poderiam ter sido diferentes se os sujeitos fossem selecionados proporcionalmente entre participantes do âmbito acadêmico e participantes que tivessem alguma experiência no âmbito industrial.

6 TRABALHOS RELACIONADOS

O objetivo desta pesquisa é semelhante a outros esforços que tentam identificar o custo da análise de mutação. Estudos empíricos revelam quais os tipos de mutantes são difíceis de serem mortos, onde muitos estão relacionados ao operador de mutação aplicado (PRAPHAMONTRIPONG et al., 2016). Na pesquisa em questão, o intuito foi identificar o custo da análise manual dos mutantes equivalentes e quais operadores de mutação geram mutantes mais complexos de serem analisados.

Durelli et al. (DURELLI et al., 2017) comparam o custo humano em detectar a equivalência de mutantes gerados pelos operadores de deleção com o custo de analisar os mutantes criados pelos operadores tradicionais. Os resultados do experimento realizado mostram que não há diferença significativa entre os custos de avaliar manualmente se há equivalência entre os mutantes gerados pelos operadores tradicionais com os mutantes gerados pelos operadores de deleção. Porém, o tipo de operador afeta significativamente a quantidade de erros e o tempo gasto analisando mutantes manualmente. Ou seja, o tipo de operador afeta o custo humano de realizar a análise manual dos mutantes. Delamaro et al (DELAMARO et al., 2014). também abordam o uso de apenas um tipo de mutante, o operador de mutação de exclusão de declaração. O uso de operadores de mutação de exclusão geram relativamente poucos mutantes, mas produz testes que são quase tão eficazes quanto o uso de todos os mutantes, com o benefício que menos mutantes equivalentes são gerados. Porém esses trabalhos analisaram somente um tipo de operador de mutação; dessa forma, no estudo em questão foram avaliados vários operadores de mutação, identificando quais são mais complexos de analisar, além de elucidar os operadores com maior demanda de tempo de análise, identificando quais operadores contribuem com o custo da análise dos mutantes.

Papadakis et al. (PAPADAKIS et al., 2014) apresentam uma abordagem com o intuito de isolar os mutantes equivalentes automaticamente, utilizando a classificação de mutantes. Conforme o conjunto de teste evolui, os mutantes com maior impacto são mortos, uma vez que o objetivo é evoluir os testes para tentar matar os mutantes (vivos) com maior impacto, ou seja, aqueles mutantes mais difíceis de serem mortos. Assim, é possível isolar os mutantes equivalentes que possuem menor impacto. O experimento revela que a classificação de mutantes isola os mutantes equivalentes efetivamente quando são utilizados conjuntos de teste de baixa qualidade, ou seja, testes que matam mutantes fáceis de serem detectados. No entanto, de acordo com a evolução dos testes, é possível verificar que o benefício dessa prática é reduzido. Com

base nesses resultados, a classificação de mutantes pode melhorar o escore de mutação quando se pretende analisar uma quantidade pequena de mutantes equivalentes. Quando é necessário analisar maior quantidade de mutantes equivalentes para atingir alto valor do escore de mutação, a abordagem de mutação tradicional proporciona uma solução melhor (PAPADAKIS et al., 2014). Dessa forma, a DiffMutAnalyze auxilia na identificação dos mutantes equivalentes, assim, na ferramenta é possível obter um ambiente de geração da mutação e de análise dos mutantes não detectados pelos casos de teste.

Papadakis et al. (PAPADAKIS et al., 2015) propõem uma técnica que explora o uso da tecnologia de compilação, denominada *Trivial Compiler Equivalence* (TCE). Essa técnica determina a equivalência quando o mutante e o original possuem o mesmo código de máquina, ou seja, os códigos são compilados e comparados. TCE também pode detectar mutantes duplicados, comparando cada mutante com outros da mesma função. Os resultados elucidam a possibilidade de minimizar o esforço humano gasto com a análise dos mutantes equivalentes, uma vez que a abordagem TCE pode detectar cerca de 30% de todos os mutantes equivalentes. Além disso, os resultados mostram que pelo menos 21% dos mutantes são duplicados e podem ser descartados.

Um obstáculo para que as ferramentas de detecção dos mutantes equivalentes sejam efetivas é o acesso ao código-fonte do mutante gerado. No trabalho de Sun et al. (SUN et al., 2017) foi necessário identificar os mutantes equivalentes manualmente, pois a ferramenta de mutação utilizada no experimento não permite que o código do mutante seja acessado. Assim, os autores relatam que a indisponibilidade do código restringiu significativamente a aplicabilidade da técnica TCE (PAPADAKIS et al., 2015) para identificar os mutantes equivalentes gerados de forma automática. Assim, é preciso ter acesso ao código-fonte dos mutantes e a mutação realizada na ferramenta DiffMutAnalyze, é possível acessar o código do mutantes, uma vez que um diretório do projeto contendo os códigos dos mutantes e o código original do sistema de software, tornando possível o acesso ao código-fonte.

Diferentes técnicas são propostas na literatura para abordar o problema do mutante equivalente. No entanto, por causa da natureza indecidível desse problema, não é possível propor um método automatizado que forneça a resposta correta para todas as instâncias desse problema. Assim, uma abordagem de gamificação para identificar os mutantes equivalentes de maneira interativa foi proposta por Houshmand e Paydar (HOUSHMAND; PAYDAR, 2017), denominada EMVille. Essa abordagem propõe uma estrutura que procura reduzir as limitações

das técnicas automatizadas existentes EMVille é um sistema baseado na *web* onde os analistas verificam os mutantes sobreviventes utilizando um jogo. Cada instância do problema é dada ao analista como uma missão que ele deve resolver, então ele recebe uma pontuação caso resolva com sucesso uma missão ou reduz a incerteza do sistema sobre a equivalência de um mutante. EMVille é composto por três componentes principais. O primeiro componente implementa uma técnica de detecção de mutantes equivalente: TCE (PAPADAKIS et al., 2015). O segundo componente fornece uma abordagem baseada em gamificação para utilizar o envolvimento de humanos na análise de mutantes. Por fim, o terceiro componente usa a técnica de aprendizado de máquina para reutilizar o esforço de especialista humano em instâncias de problemas adicionais. Os resultados do experimento mostram que o envolvimento dos participantes é visivelmente aumentado pela abordagem de gamificação. Além disso, EMVille possui o *Diff Visualizer*, um visualizador que mostra as diferenças do código original e do mutante de uma forma que o jogador pode compará-los. Assim, os participantes do experimento informaram que os componentes que contemplam essa abordagem auxiliam na analise das instâncias dos mutantes. Dessa forma, é possível perceber que a existência de uma ferramenta para auxiliar a análise dos mutantes sobreviventes é útil durante o processo de análise deles.

Seguindo a mesma linha de pesquisa, relacionada à gameficação, Code Defenders (ROJAS et al., 2017) é um uma ferramenta de gamificação para teste de mutação, em que o próprio jogador realiza a mutação, assumindo o papel de atacante, em defesa, outro jogador assume o papel de defensor, criando casos de teste para matar os mutantes gerados. O atacante consegue uma pontuação se criar mutantes que não possam ser detectados pelos casos de teste e o defensor pontua se seus casos de testes conseguem matar os mutantes. Os autores possuem a hipótese de que os jogadores tendem a criar mutantes e testes mais fortes para alcançar maior pontuação, dessa forma, eles terão um melhor desempenho nas tarefas que envolvam o teste de software. Porém a análise dos mutantes é necessária para verificar se os mutantes possuem equivalência e, na DiffMutAnalyze, os mutantes são gerados automaticamente e as mudanças impostas pelos operadores de mutação são evidenciadas. Além disso, o tempo de análise dos mutantes é contabilizado e o usuário define o grau de dificuldade da análise, o que acarreta em informações que podem ser extraídas para trabalhos de pesquisa e estudos relacionados com o custo da análise.

Apesar de muitas pesquisas na área do teste de mutação, há ainda questões que precisam de soluções, como a análise dos mutantes equivalentes, uma vez que a análise manual desses

mutantes ainda é necessária. Papadakis e Malevris (PAPADAKIS; MALEVRIS, 2010) elucidam que a identificação manual dos mutantes equivalentes se torna uma atividade mais difícil do que produzir e melhorar o conjunto de casos de teste. Nesse contexto, os resultados mostram que o custo da análise manual demanda tempo do testador. Assim, os resultados trazem evidências para a necessidade de ferramentas que auxiliem na análise desses mutantes.

Os pesquisadores estão buscando reduzir o custo da aplicação do teste de mutação (FERRARI et al., 2018). Assim, o custo associado à identificação de mutantes equivalentes vem sendo discutido como um desafio na adoção do teste de mutação (ZHU et al., 2018). Dessa forma, nesta pesquisa foi possível verificar que por meio da ferramenta DiffMutAnalyze, é possível reduzir o tempo de análise dos mutantes, uma vez que na investigação do custo da análise, foi possível observar a redução do tempo quando comparado com a análise manual.

7 CONCLUSÃO

No contexto geral do teste de mutação, os autores buscam por soluções que possam auxiliar os analistas durante a análise dos mutantes sobreviventes, para verificar a existência de mutantes equivalentes (PAPADAKIS et al., 2019; JIA; HARMAN, 2011; USAOLA; MATEO, 2010; PAPADAKIS et al., 2015; LIMA et al., 2016), visto que os mutantes equivalentes são um problema para o teste de mutação (PAPADAKIS et al., 2014; ARCAINI et al., 2017). Dessa forma, esta pesquisa pesquisa objetivou realizar um experimento para verificar o custo da análise dos mutantes. Assim, a DiffMutAnalyze foi desenvolvida para auxiliar os analistas durante a análise. Por meio do experimento realizado, foi possível verificar se o uso da ferramenta pode contribuir com a redução do custo da análise. A seguir, estão relacionadas as considerações finais desta pesquisa.

7.1 Considerações Finais do Projeto de Mestrado

Uma das fases do desenvolvimento de software é realizar os testes do sistemas desenvolvidos. Dentre os critérios de teste existentes, o teste de mutação é promissor para verificar a qualidade dos casos de teste existentes, assim, os mutantes formam os objetivos do processo de teste. Os casos de teste capazes de distinguir os comportamentos dos mutantes e do código original são indicados a identificar possíveis falhas no sistema de software (PAPADAKIS et al., 2019). Na prática, alguns desses mutantes são equivalentes, ou seja, eles formam versões funcionalmente equivalentes ao sistema de software original. Dessa forma, eles não contribuem para melhorar o conjunto de casos de teste e precisam ser analisados para verificar se são realmente equivalentes ao código original ou se novos casos de teste são necessários para identificar os mutantes sobreviventes (JIA; HARMAN, 2011).

Para esta pesquisa de mestrado, duas questões foram propostas. Respondendo a **QP1:** *"Qual é o custo para determinar manualmente a equivalência de mutantes com o sistema de software original?"*, foi observado que o custo em analisar os mutantes está relacionado em sua maior parte com o tempo gasto com a análise em geral. Em consequência desse tempo, o operador de mutação tem influência, pois, de acordo com a mutação realizada, os sujeitos gastaram mais tempo analisando determinados mutantes.

O primeiro experimento (Seção 5.2) está relacionado com o custo da análise. Os resultados foram obtidos por meio da comparação da análise realizada de forma manual (sem a utilização da DiffMutAnalyze) e por meio do auxílio da DiffMutAnalyze. Considerando o

tempo médio total da análise por mutante, todos os participantes do experimento gastaram menor tempo quando utilizada a DiffMutAnalyze. O tempo médio da análise manual, entre todos os sujeitos, é de 1 minuto e 6 segundos; o tempo médio utilizando a DiffMutAnalyze, dos 11 sujeitos, é de 21 segundos. Assim, é possível perceber que utilizando a DiffMutAnalyze, o tempo médio da análise por mutante, teve redução de aproximadamente 60% do tempo da análise manual.

Considerando o tempo médio total da análise de todos os mutantes, correspondente ao tempo de abertura dos códigos dos mutantes e de seu código original correspondente do sistema de software, o tempo médio total da análise manual é de 31 minutos e 30 segundos e o tempo médio utilizando a DiffMutAnalyze é de 14 minutos. O que reduz o tempo médio gasto para analisar todos os mutantes em aproximadamente 50% do tempo. Diante disso, é possível observar que o custo médio em analisar os mutantes e verificar se são equivalentes ao sistema de software original, é 50% maior quando não é utilizada uma ferramenta de apoio computacional.

Em relação ao operador de mutação, a alteração realizada pelo operador EVR foi a que demandou maior tempo de análise nas duas formas (i. e. manual e pela DiffMutAnalyze). Diante dos três maiores tempos gastos com análise, considerando todos os sujeitos, os operadores EVR, ROR e LVR estão presentes em maior parte dos maiores tempos de análise de cada sujeito. Dessa forma, esses operadores geram os mutantes com maior demanda de tempo, o que aumenta o custo da análise.

Por meio dos dados obtidos anteriormente, é possível elucidar evidências para responder a segunda questão de pesquisa: **QP2:** *"A ferramenta DiffMutAnalyze pode contribuir com a redução do custo da análise, auxiliando os analistas na identificação dos mutantes equivalentes?"*. Considerando o tempo contabilizado durante o experimento, foi observado que a ferramenta DiffMutAnalyze ajudou a reduzir o tempo da análise dos mutantes.

No segundo experimento (Seção 5.1), os alunos avaliaram o uso da DiffMutAnalyze para verificar o uso da ferramenta no que tange a identificação dos mutantes equivalentes. Além disso, verificar o uso da ferramenta para aplicação do teste de mutação no ambiente acadêmico. Dessa forma, por meio da realização do experimento para avaliar a utilização da ferramenta DiffMutAnalyze foi possível observar que os alunos tiveram interesse em conhecer sobre o teste de mutação, e também se manifestaram entusiasmados em poder ver como o teste de mutação funciona na prática, uma vez que este torna o aprendizado mais dinâmico.

Como foi demonstrado nos resultados das análises, tanto no primeiro experimento quanto no segundo, a escolha da utilização dos algoritmos de ordenação para aplicar a mutação facilitou o processo da análise, uma vez que esses algoritmos são conhecidos por grande parte dos participantes de ambos os experimentos. Desse modo, é possível observar que o teste de mutação pode ser aplicado no contexto acadêmico e utilizar algum projeto ou algoritmos já conhecidos pelos alunos, facilita no processo da análise dos mutantes sobreviventes e diminui o tempo gasto com essa análise. Além disso, os professores podem utilizar algoritmos ou projetos que os alunos estão utilizando em outras disciplinas para aplicar a mutação, facilitando a inspeção dos códigos e diminuindo o tempo de análise. Assim, é possível incluir aulas práticas em disciplinas sobre testes e incluir o teste de mutação em sala de aula, além de fazer com que os alunos também construam casos de testes mais eficazes e aumente a qualidade do conjunto de testes.

7.2 Contribuições desta Dissertação

Nesta pesquisa, são oferecidas algumas contribuições para estudos em relação ao teste de mutação. A primeira contribuição, está relacionada com o tempo gasto com a análise dos mutantes. A análise manual dos mutantes aumenta o custo do teste de mutação, uma vez que o analista necessita verificar se os mutantes são equivalentes ao programa original. Dessa forma, pesquisas para minimizar esse custo, são necessárias.

Os operadores de mutação possuem grande influência no custo, uma vez que determinado operador pode gerar maiores quantidades de mutantes equivalentes. Dessa forma, os mutantes equivalentes gerados foram identificados e relacionados com o operador de mutação que o gerou. Além disso, foram identificados os operadores de mutação que geraram os mutantes com maior demanda de tempo de análise e os operadores que realizaram alterações mais difíceis de serem analisadas. Essa é uma importante contribuição, pois alguns autores buscam diminuir a quantidade de mutantes gerados, para diminuir o custo computacional da geração dos mutantes e, por meio dessa informação dos operadores de mutação com maior tempo de análise e grau de dificuldade elevado, os pesquisadores podem considerar a não utilização desses operadores e realizar a mutação utilizando alguns operadores específicos, verificando se podem contribuir com a qualidade dos casos de teste e diminuindo os custos da aplicação do teste de mutação.

Outra contribuição está relacionada com a ferramenta de apoio computacional para a análise dos mutantes sobreviventes. A DiffMutAnalyze auxilia na verificação dos mutantes

equivalentes, contemplando um ambiente amplo para a realização do teste de mutação. Por meio da ferramenta, as mutações são geradas no sistema de software a ser testado, os mutantes que permaneceram vivos são disponibilizados para inspeção dos códigos e comparação do código original e do mutante, lado a lado, evidenciando a alteração realizada, o que facilita a análise, sem que o analista procure pela alteração, antes de verificar se os códigos são equivalentes. Dessa forma, a DiffMutAnalyze pode ser promissora para estudos empíricos futuros.

7.3 Limitações

Esta pesquisa apresenta algumas limitações. A principal limitação está relacionada com a amostra dos sujeitos do experimento, uma vez que a amostra foi não-probabilística. O experimento para avaliar o custo da análise dos mutantes foi realizado com onze alunos da pós-graduação. Dessa forma, a definição da amostra pode ser considerada um fator limitante, o que resulta em resultados obtidos apenas para a população em questão. Para futuras investigações, sugere-se amostras amplas e com participantes da indústria. Com relação às limitações da DiffMutAnalyze, é possível inserir somente projetos Java com estrutura Maven para que a mutação seja realizada.

7.4 Sugestões de Trabalhos Futuros

Como trabalho futuro, pretende-se prosseguir com a realização dos experimentos em âmbito industrial, verificando a possibilidade da ferramenta DiffMutAnalyze poder contribuir com a aplicação do teste de mutação em sistemas de software industriais e ajudar a reduzir o custo da aplicação e da análise dos mutantes sobreviventes.

Diante das pesquisas realizadas no contexto do teste de mutação, é possível compreender a necessidade de buscar alternativas para auxiliar a análise dos mutantes equivalentes. Assim, pode-se integrar outras técnicas de análise de mutantes na ferramenta DiffMutAnalyze. Além disso, como mencionado anteriormente, alguns autores buscam diminuir o custo computacional do teste de mutação, dessa forma, pode ser disponibilizado na DiffMutAnalyze a possibilidade de escolha do operador de mutação a ser utilizado para realizar as alterações no código do sistema de software a ser testado.

REFERÊNCIAS

ALVIN, J.; KURTZ, B.; AMMANN, P.; RANGWALA, H.; JUST, R. Guiding testing effort using mutant utility. In: **41st International Conference on Software Engineering (ICSE)**. [S.l.: s.n.], 2019.

AMBLER, S. W. **Introduction to test-driven development (TDD)**. 2002. <http://www.agiledata.org/essays/tdd.html>. [ONLINE]. Acessado em 08 de janeiro de 2019.

AMMANN, P.; OFFUTT, J. **Introduction to Software Testing**. [S.l.]: Cambridge University Press, 2008.

ANICHE, M.; HERMANS, F.; DEURSEN, A. van. Pragmatic software testing education. In: **50th ACM Technical Symposium on Computer Science Education**. [S.l.: s.n.], 2019. p. 414–420.

ARCAINI, P.; GARGANTINI, A.; RICCOBENE, E.; VAVASSORI, P. A novel use of equivalent mutants for static anomaly detection in software artifacts. **Information and Software Technology**, v. 81, p. 52–64, 2017.

ARCURI, A.; CAMPOS, J.; FRASER, G. Unit test generation during software development: Evosuite plugins for maven, intellij and jenkins. In: **International Conference on Software Testing, Verification and Validation (ICST)**. [S.l.: s.n.], 2016. p. 401–408.

BARESI, L.; PEZZE, M. An introduction to software testing. **Electronic Notes in Theoretical Computer Science**, v. 148, n. 1, p. 89–111, 2006.

BERTOLINO, A. Software testing research: Achievements, challenges, dreams. In: **Future of Software Engineering**. [S.l.: s.n.], 2007. p. 85–103.

BOTELHO, J.; DURELLI, V. H.; BORGES, S. S.; ENDO, A. T.; ELER, M. M.; DELAMARO, M. E.; DURELLI, R. S. On the costs of applying logic-based criteria to mobile applications: An empirical analysis of predicates in real-world objective-c and swift applications. In: **2nd Brazilian Symposium on Systematic and Automated Software Testing (SAST)**. [S.l.: s.n.], 2017. p. 4.

BOTELHO, J.; PEREIRA, C. H.; DURELLI, V. H. S.; DURELLI, R. S. Diffmutanalyze: Uma abordagem para auxiliar a identificação de mutantes equivalentes. In: **VI Workshop on Software Visualization, Evolution and Maintenance (VEM)**. [S.l.: s.n.], 2018.

BUDD, T. A.; ANGLUIN, D. Two notions of correctness and their relation to testing. **Acta Informatica**, v. 18, n. 1, p. 31–45, 1982.

CHRISTENSEN, H. B. Systematic testing should not be a topic in the computer science curriculum! In: **ACM SIGCSE Bulletin**. [S.l.: s.n.], 2003. p. 7–10.

COLES, H. **Pit Mutation Testing tool**. 2018. <http://pitest.org/>. [ONLINE]. Acessado em 10 de janeiro de 2018.

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

DELAMARO, M. E.; OFFUTT, J.; AMMANN, P. Designing deletion mutation operators. In: **Seventh International Conference on Software Testing, Verification and Validation (ICST)**. [S.l.: s.n.], 2014. p. 11–20.

DEMILLO, R. A. **Mutation Analysis as a Tool for Software Quality Assurance.** [S.l.], 1980.

DEMILLO, R. A.; LIPTON, R. J.; SAYWARD, F. G. Hints on test data selection: Help for the practicing programmer. **Computer**, v. 11, n. 4, p. 34–41, 1978.

DURELLI, V. H.; SOUZA, N. M. D.; DELAMARO, M. E. Are deletion mutants easier to identify manually? In: **10th International Conference on Software Testing, Verification and Validation Workshops (ICSTW)**. [S.l.: s.n.], 2017. p. 149–158.

DURELLI, V. H. S. **Toward harnessing a Java high-level language virtual machine for supporting software testing**. Tese (Doutorado) — Universidade de São Paulo, 2013.

FELDT, R.; ZIMMERMANN, T.; BERGERSEN, G. R.; FALESSI, D.; JEDLITSCHKA, A.; JURISTO, N.; MÜNCH, J.; OIVO, M.; RUNESON, P.; SHEPPERD, M. et al. Four commentaries on the use of students and professionals in empirical software engineering experiments. **Empirical Software Engineering**, v. 23, n. 6, p. 3801–3820, 2018.

FERRARI, F. C.; PIZZOLETO, A. V.; OFFUTT, J. A systematic review of cost reduction techniques for mutation testing: Preliminary results. In: **International Conference on Software Testing, Verification and Validation Workshops (ICSTW)**. [S.l.: s.n.], 2018. p. 1–10.

GRÜN, B. J.; SCHULER, D.; ZELLER, A. The impact of equivalent mutants. In: **2nd International Conference on Software Testing, Verification and Validation Workshops (ICSTW)**. [S.l.: s.n.], 2009. p. 192–199.

HOUSHMAND, M.; PAYDAR, S. Emville: A gamification-based approach to address the equivalent mutant problem. In: **7th International Conference on Computer and Knowledge Engineering (ICCKE)**. [S.l.: s.n.], 2017. p. 326–332.

HOWDEN, W. E. **Functional program testing and analysis**. [S.l.]: McGraw-Hill, Inc., 1986.

HU, J.; LI, N.; OFFUTT, J. An analysis of oo mutation operators. In: **Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)**. [S.l.: s.n.], 2011. p. 334–341.

IEEE Standard Glossary of Software Engineering Terminology. **IEEE Std 610.12-1990**, p. 1–84, 1990.

JANZEN, D. S.; SAIEDIAN, H. Test-driven learning: intrinsic integration of testing into the cs/se curriculum. In: **ACM SIGCSE Bulletin**. [S.l.: s.n.], 2006. p. 254–258.

JIA, Y.; HARMAN, M. An analysis and survey of the development of mutation testing. **IEEE transactions on Software Engineering**, v. 37, n. 5, p. 649–678, 2011.

JUNG, C. F. **Metodologia para pesquisa e desenvolvimento: aplicada a novas tecnologias, produtos e processos**. [S.l.]: Axcel Books, 2004.

JUNIOR, S.; COSTA, F. J. Mensuração e escalas de verificação: uma análise comparativa das escalas de likert e phrase completion. **PMKT–Revista Brasileira de Pesquisas de Marketing, Opinião e Mídia**, v. 15, p. 1–16, 2014.

JUST, R. The major mutation framework: Efficient and scalable mutation analysis for Java. In: **International Symposium on Software Testing and Analysis (ISSTA)**. [S.l.: s.n.], 2014. p. 433–436.

JUST, R. **The Major Mutation Framework**. 2017. <http://mutation-testing.org/doc/>. [ONLINE]. Acessado em 19 de janeiro de 2018.

JUST, R.; JALALI, D.; INOZEMTSEVA, L.; ERNST, M. D.; HOLMES, R.; FRASER, G. Are mutants a valid substitute for real faults in software testing? In: **22nd International Symposium on Foundations of Software Engineering**. [S.l.: s.n.], 2014. p. 654–665.

JUST, R.; SCHWEIGGERT, F.; KAPFHAMMER, G. M. Major: An efficient and extensible tool for mutation analysis in a Java compiler. In: **26th International Conference on Automated Software Engineering (ASE)**. [S.l.: s.n.], 2011. p. 612–615.

KINTIS, M.; MALEVRIS, N. Medic: A static analysis framework for equivalent mutant identification. **Information and Software Technology**, v. 68, p. 1–17, 2015.

KINTIS, M.; PAPADAKIS, M.; PAPADOPOULOS, A.; VALVIS, E.; MALEVRIS, N. Analysing and comparing the effectiveness of mutation testing tools: A manual study. In: **16th International Working Conference on Source Code Analysis and Manipulation (SCAM)**. [S.l.: s.n.], 2016. p. 147–156.

KUSHIGIAN, B.; RAWAT, A.; JUST, R. Medusa: Mutant equivalence detection using satisfiability analysis. In: **International Workshop on Mutation Analysis (Mutation)**. [S.l.: s.n.], 2019.

LEMOS, O. A. L.; SILVEIRA, F. F.; FERRARI, F. C.; GARCIA, A. The impact of software testing education on code reliability: An empirical assessment. **Journal of Systems and Software**, v. 137, p. 497–511, 2018.

LEWIS, W. E. **Software testing and continuous quality improvement**. [S.l.]: Auerbach publications, 2017.

LIMA, J. A.; GUIZZO, G.; VERGILIO, S. R.; SILVA, A. P.; EHRENFRIED, H. V. et al. Evaluating different strategies for reduction of mutation testing costs. In: **1st Brazilian Symposium on Systematic and Automated Software Testing (SAST)**. [S.l.: s.n.], 2016. p. 4.

MA, Y.-S.; KIM, S.-W. Mutation testing cost reduction by clustering overlapped mutants. **Journal of Systems and Software**, v. 115, p. 18–30, 2016.

MA, Y.-S.; OFFUTT, J.; KWON, Y.-R. Mujava: a mutation system for Java. In: **28th International Conference on Software Engineering**. [S.l.: s.n.], 2006. p. 827–830.

MALDONADO, J. C.; VINCENZI, A. M. R.; BARBOSA, E. F.; SOUZA, S. do Rocio Senger de; DELAMARO, M. E. **Aspectos teóricos e empíricos de teste de cobertura de software**. [S.l.]: ICMSC-USP, 1998.

MYERS, G. J.; SANDLER, C.; BADGETT, T. **The art of software testing**. [S.l.]: John Wiley & Sons, 2011.

NATELLA, R.; COTRONEO, D.; DURAES, J. A.; MADEIRA, H. S. On fault representativeness of software fault injection. **IEEE Transactions on Software Engineering**, v. 39, n. 1, p. 80–96, 2013.

NICA, S.; WOTAWA, F. Eqmutdetect—a tool for equivalent mutant detection in embedded systems. In: **Tenth Workshop on Intelligent Solutions in Embedded Systems (WISES)**. [S.l.: s.n.], 2012. p. 57–62.

NOONAN, R. E.; PROSL, R. H. Unit testing frameworks. In: **ACM SIGCSE Bulletin**. [S.l.: s.n.], 2002. p. 232–236.

OFFUTT, A. The coupling effect: fact or fiction. In: **ACM SIGSOFT Software Engineering Notes**. [S.l.: s.n.], 1989. p. 131–140.

OFFUTT, A. J.; LEE, A.; ROTHERMEL, G.; UNTCH, R. H.; ZAPF, C. An experimental determination of sufficient mutant operators. **ACM Transactions on Software Engineering and Methodology (TOSEM)**, v. 5, n. 2, p. 99–118, 1996.

OFFUTT, A. J.; PAN, J. Automatically detecting equivalent mutants and infeasible paths. **Software Testing, Verification and Reliability**, v. 7, n. 3, p. 165–192, 1997.

PAPADAKIS, M.; DELAMARO, M.; TRAON, Y. L. Mitigating the effects of equivalent mutants with mutant classification strategies. **Science of Computer Programming**, v. 95, p. 298–319, 2014.

PAPADAKIS, M.; JIA, Y.; HARMAN, M.; TRAON, Y. L. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In: **37th International Conference on Software Engineering (ICSE)**. [S.l.: s.n.], 2015. p. 936–946.

PAPADAKIS, M.; KINTIS, M.; ZHANG, J.; JIA, Y.; TRAON, Y. L.; HARMAN, M. Mutation testing advances: an analysis and survey. In: **Advances in Computers**. [S.l.: s.n.], 2019. p. 275–378.

PAPADAKIS, M.; MALEVRIS, N. An empirical evaluation of the first and second order mutation testing strategies. In: **3th international conference on Software testing, verification, and validation workshops (ICSTW)**. [S.l.: s.n.], 2010. p. 90–99.

PAPADAKIS, M.; TRAON, Y. L. Using mutants to locate "unknown" faults. In: **Fifth International Conference on Software Testing, Verification and Validation (ICST)**. [S.l.: s.n.], 2012. p. 691–700.

PETERS, J. F.; PEDRYCZ, W. Engenharia de software: teoria e prática. **Rio de Janeiro: Campus**, v. 681, n. 519.683, p. 2, 2001.

PRAPHAMONTRIPONG, U.; OFFUTT, J.; DENG, L.; GU, J. An experimental evaluation of web mutation operators. In: **Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)**. [S.l.: s.n.], 2016. p. 102–111.

PRESSMAN, R. S. M. B. R. **Engenharia de Software: uma abordagem profissional**. [S.l.]: AMGH, 2016.

RADHAKRISHNAN, P.; KANMANI, S.; NANDHINI, M. Xsoft: A generic software teaching and learning model. **Computer Applications in Engineering Education**, v. 23, n. 3, p. 432–442, 2015.

ROJAS, J. M.; WHITE, T. D.; CLEGG, B. S.; FRASER, G. Code defenders: Crowdsourcing effective tests and subtle mutants with a mutation testing game. In: **39th International Conference on Software Engineering (ICSE)**. [S.l.: s.n.], 2017. p. 677–688.

SCHULER, D.; ZELLER, A. Covering and uncovering equivalent mutants. **Software Testing, Verification and Reliability**, v. 23, n. 5, p. 353–374, 2013.

SJØBERG, D. I.; HANNAY, J. E.; HANSEN, O.; KAMPENES, V. B.; KARAHASANOVIC, A.; LIBORG, N.-K.; REKDAL, A. C. A survey of controlled experiments in software engineering. **IEEE Transactions on Software Engineering**, v. 31, n. 9, p. 733–753, 2005.

SOMMERVILLE, I. **Engenharia de Software**. [S.l.]: Pearson, 2011.

SPACCO, J.; PUGH, W. Helping students appreciate test-driven development (tdd). In: **21st Symposium on Object-oriented Programming Systems, Languages, and Applications (ACM SIGPLAN)**. [S.l.: s.n.], 2006. p. 907–913.

STEFAN, P.; HORKY, V.; BULEJ, L.; TUMA, P. Unit testing performance in Java projects: Are we there yet? In: **8th International Conference on Performance Engineering**. [S.l.: s.n.], 2017. p. 401–412.

SUN, C. ai; XUE, F.; LIU, H.; ZHANG, X. A path-aware approach to mutant reduction in mutation testing. **Information and Software Technology**, v. 81, p. 65–81, 2017.

TRAVASSOS, G. H.; GUROV, D.; AMARAL, E. **Introdução à engenharia de software experimental**. [S.l.]: UFRJ, 2002.

USAOLA, M. P.; MATEO, P. R. Mutation testing cost reduction techniques: a survey. **IEEE Software**, v. 27, n. 3, 2010.

WALLS, C. **Spring in Action**. [S.l.]: Manning Publications Co., 2011.

WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLÉN, A. **Experimentation in software engineering**. [S.l.]: Springer Science & Business Media, 2012.

YAO, X.; HARMAN, M.; JIA, Y. A study of equivalent and stubborn mutation operators using human analysis of equivalence. In: **36th International Conference on Software Engineering (ICSE)**. [S.l.: s.n.], 2014. p. 919–930.

ZEROUALI, A.; MENS, T. Analyzing the evolution of testing library usage in open source Java projects. In: **24th International Conference on Software Analysis, Evolution and Reengineering (SANER)**. [S.l.: s.n.], 2017. p. 417–421.

ZHENG, W.; BAI, Y.; CHE, H. A computer-assisted instructional method based on machine learning in software testing class. **Computer Applications in Engineering Education**, v. 26, n. 5, p. 1150–1158, 2018.

ZHU, Q.; PANICHELLA, A.; ZAIDMAN, A. A systematic literature review of how mutation testing supports quality assurance processes. **Software Testing, Verification and Reliability**, v. 28, n. 6, p. 1675, 2018.

A APÊNDICE

A.1 Exemplos de Mutantes Gerados pelos Operadores de Mutação da Major

Alguns mutantes gerados no código dos algoritmos de ordenação utilizados nos experimentos deste projeto de pesquisa, estão relacionados a seguir. O código original dos algoritmos *SelectionSort* e *InsertionSort* estão descritos na Seção 4.3.1. Cada código contém a alteração realizada por algum operador de mutação da ferramenta Major. Os mutantes do algoritmo *SelectionSort* A.3 e A.4 são equivalentes ao código original e os mutantes equivalentes do algoritmo *InsertionSort* são A.9 e A.10.

Código A.1 – Mutação realizada pelo operador AOR no algoritmo *SelectionSort*

```
1  public class Selection {
2          public static void sort(Comparable[] array) {
3                  int N = array.length;
4                  for (int i = 0; i < N; i++) {
5                          int min = i;
6                          for (int j = i * 1; j < N; j++) {
7                                  if (array[j].compareTo(array[min]) < 0)
8                                          min = j;
9                          }
10                         exchange(array, i, min);
11                 }
12         }
13
14         private static void exchange(Comparable[] array, int i, int j) {
15                 Comparable t = array[i];
16                 array[i] = array[j];
17                 array[j] = t;
18         }
19 }
```

Código A.2 – Mutação realizada pelo operador EVR no algoritmo *SelectionSort*

```
1  public class Selection {
2          public static void sort(Comparable[] array) {
3                  int N = 0;
4                  for (int i = 0; i < N; i++) {
5                          int min = i;
6                          for (int j = i + 1; j < N; j++) {
7                                  if (array[j].compareTo(array[min]) < 0)
8                                          min = j;
9                          }
10                         exchange(array, i, min);
11                 }
12         }
13
14         private static void exchange(Comparable[] array, int i, int j) {
15                 Comparable t = array[i];
16                 array[i] = array[j];
17                 array[j] = t;
18         }
19 }
```

Código A.3 – Mutação realizada pelo operador LVR no algoritmo *SelectionSort*

```
1  public class Selection {
2          public static void sort(Comparable[] array) {
3                  int N = array.length;
4                  for (int i = 0; i < N; i++) {
5                          int min = i;
6                          for (int j = i + 0; j < N; j++) {
7                                  if (array[j].compareTo(array[min]) < 0)
8                                          min = j;
9                          }
10                         exchange(array, i, min);
11                 }
12         }
13
14         private static void exchange(Comparable[] array, int i, int j) {
15                 Comparable t = array[i];
16                 array[i] = array[j];
17                 array[j] = t;
18         }
19 }
```

Código A.4 – Mutação realizada pelo operador ROR no algoritmo *SelectionSort*

```
1  public class Selection {
2          public static void sort(Comparable[] array) {
3                  int N = array.length;
4                  for (int i = 0; i < N; i++) {
5                          int min = i;
6                          for (int j = i + 1; j != N; j++) {
7                                  if (array[j].compareTo(array[min]) < 0)
8                                          min = j;
9                          }
10                         exchange(array, i, min);
11                 }
12         }
13
14         private static void exchange(Comparable[] array, int i, int j) {
15                 Comparable t = array[i];
16                 array[i] = array[j];
17                 array[j] = t;
18         }
19 }
```

Código A.5 – Mutação realizada pelo operador STD no algoritmo *SelectionSort*

```
1  public class Selection {
2          public static void sort(Comparable[] array) {
3                  int N = array.length;
4                  for (int i = 0; i < N; i++) {
5                          int min = i;
6                          for (int j = i + 1; j < N; j++) {
7                                  if (array[j].compareTo(array[min]) < 0)
8                                          min = j;
9                          }
10                         exchange(array, i, min);
11                 }
12         }
13
14         private static void exchange(Comparable[] array, int i, int j) {
15                 Comparable t = array[i];
16
17                 array[j] = t;
18         }
19 }
```

Código A.6 – Mutação realizada pelo operador AOR no algoritmo *InsertionSort*

```
1  public class Insertion {
2          public static void sort(Comparable[] array) {
3                  int N = array.length;
4                  for (int i = 1; i < N; i++) {
5                          int j = i;
6                          while (j > 0 && less(array[j], array[j - 1])) {
7                                  exchange(array, j, j + 1);
8                                  j--;
9                          }
10                 }
11         }
12
13         private static boolean less(Comparable x, Comparable y) {
14                 return x.compareTo(y) < 0;
15         }
16
17         private static void exchange(Comparable[] a, int i, int j) {
18                 Comparable t = a[i];
19                 a[i] = a[j];
20                 a[j] = t;
21         }
22 }
```

Código A.7 – Mutação realizada pelo operador COR no algoritmo *InsertionSort*

```
1  public class Insertion {
2          public static void sort(Comparable[] array) {
3                  int N = array.length;
4                  for (int i = 1; i < N; i++) {
5                          int j = i;
6                          while (j > 0) {
7                                  exchange(array, j, j - 1);
8                                  j--;
9                          }
10                 }
11         }
12
13         private static boolean less(Comparable x, Comparable y) {
14                 return x.compareTo(y) < 0;
15         }
16
17         private static void exchange(Comparable[] a, int i, int j) {
18                 Comparable t = a[i];
19                 a[i] = a[j];
20                 a[j] = t;
21         }
22 }
```

Código A.8 – Mutação realizada pelo operador EVR no algoritmo *InsertionSort*

```
1  public class Insertion {
2          public static void sort(Comparable[] array) {
3                  int N = array.length;
4                  for (int i = 1; i < N; i++) {
5                          int j = i;
6                          while (j > 0 && less(array[j], array[j - 1])) {
7                                  exchange(array, j, j - 1);
8                                  j--;
9                          }
10                 }
11         }
12
13         private static boolean less(Comparable x, Comparable y) {
14                 return x.compareTo(y) < 0;
15         }
16
17         private static void exchange(Comparable[] a, int i, int j) {
18                 Comparable t = null;
19                 a[i] = a[j];
20                 a[j] = t;
21         }
22 }
```

Código A.9 – Mutação realizada pelo operador LVR no algoritmo *InsertionSort*

```
1  public class Insertion {
2          public static void sort(Comparable[] array) {
3                  int N = array.length;
4                  for (int i = 0; i < N; i++) {
5                          int j = i;
6                          while (j > 0 && less(array[j], array[j - 1])) {
7                                  exchange(array, j, j - 1);
8                                  j--;
9                          }
10                 }
11         }
12
13         private static boolean less(Comparable x, Comparable y) {
14                 return x.compareTo(y) < 0;
15         }
16
17         private static void exchange(Comparable[] a, int i, int j) {
18                 Comparable t = a[i];
19                 a[i] = a[j];
20                 a[j] = t;
21         }
22 }
```

Código A.10 – Mutação realizada pelo operador ROR no algoritmo *InsertionSort*

```
1  public class Insertion {
2          public static void sort(Comparable[] array) {
3                  int N = array.length;
4                  for (int i = 1; i < N; i++) {
5                          int j = i;
6                          while (j > 0 && less(array[j], array[j - 1])) {
7                                  exchange(array, j, j - 1);
8                                  j--;
9                          }
10                 }
11         }
12
13         private static boolean less(Comparable x, Comparable y) {
14                 return x.compareTo(y) <= 0;
15         }
16
17         private static void exchange(Comparable[] a, int i, int j) {
18                 Comparable t = a[i];
19                 a[i] = a[j];
20                 a[j] = t;
21         }
22 }
```

Código A.11 – Mutação realizada pelo operador STD no algoritmo *InsertionSort*

```
1  public class Insertion {
2          public static void sort(Comparable[] array) {
3                  int N = array.length;
4                  for (int i = 1; i < N; i++) {
5                          int j = i;
6                          while (j > 0 && less(array[j], array[j - 1])) {
7                                  exchange(array, j, j - 1);
8                                  j--;
9                          }
10                 }
11         }
12
13         private static boolean less(Comparable x, Comparable y) {
14                 return x.compareTo(y) < 0;
15         }
16
17         private static void exchange(Comparable[] a, int i, int j) {
18                 Comparable t = a[i];
19                 a[i] = a[j];
20
21         }
22 }
```

A.2 Formulário de caracterização de participantes

Este formulário contém questões para coleta de dados e informações sobre a caracterização dos participantes do experimento, uma vez que, a experiência e conhecimento dos participantes, podem interferir nos resultados obtidos. Além disso, contém a avaliação da ferramenta DiffMutAnalyze utilizada no experimento.

As informações contidas neste formulário serão de uso exclusivo para coleta de dados da proposta de mestrado de Juliana Botelho de Carvalho, aluna do programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Lavras - UFLA. As informações pessoais dos participantes não serão divulgadas ou expostas na dissertação do mestrado. Dessa forma, você concorda em participar do experimento para análise de mutantes?

INFORMAÇÕES PESSOAIS

Nome:

Curso:

CONHECIMENTO BÁSICO

Você trabalha ou trabalhou com atividades que envolvem testes de software?

1. Trabalho atualmente com testes de software na empresa
2. Trabalhei com testes de software em empresas
3. Trabalho(ei) com testes de software somente em disciplinas do curso
4. Conheço somente a teoria sobre testes de software, mas nunca utilizei na prática
5. Não conheço sobre testes de software e nunca trabalhei

Nível de conhecimento sobre testes de software:

1. Básico
2. Intermediário
3. Avançado
4. Nenhum

Você trabalha ou trabalhou com desenvolvimento em Java?

1. Trabalho atualmente com desenvolvimento em Java na empresa
2. Trabalhei com desenvolvimento em Java em empresas
3. Trabalho(ei) com desenvolvimento em Java somente em disciplinas do curso
4. Conheço a linguagem de programação Java, mas nunca utilizei no desenvolvimento
5. Não conheço e nunca utilizei a linguagem Java

Nível de conhecimento sobre o desenvolvimento em Java:

1. Básico
2. Intermediário
3. Avançado
4. Nenhum

Você trabalha ou trabalhou com Testes de Unidade?

1. Trabalho atualmente com Testes de Unidade na empresa
2. Trabalhei com Testes de Unidade em empresas
3. Trabalho(ei) com Testes de Unidade somente em disciplinas do curso
4. Conheço sobre Testes de Unidade, mas nunca utilizei em projetos de desenvolvimento
5. Não conheço e nunca utilizei Testes de Unidade

Nível de conhecimento sobre Testes de Unidade:

1. Básico
2. Intermediário
3. Avançado
4. Nenhum

Você tem o hábito de atualizar os casos de teste para tentar melhorá-los durante o desenvolvimento de software?

1. Sim, sempre
2. Sim, às vezes
3. Sim, raramente
4. Não

Nível de conhecimento sobre Teste de Mutação:

1. Básico
2. Intermediário
3. Avançado
4. Nenhum

Você conhecia a lógica dos algoritmos de ordenação?

1. Sim, utilizei algoritmos de ordenação em disciplinas do curso
2. Sim, conheço os algoritmos de ordenação, mas nunca utilizei
3. Não possuo conhecimento sobre algoritmos de ordenação

Nível de conhecimento sobre Algoritmos de Ordenação:

1. Básico
2. Intermediário
3. Avançado
4. Nenhum

FERRAMENTA DIFFMUTANALYZE

Qual o nível de dificuldade em utilizar a Ferramenta DiffMutAnalyze?

1. Muito fácil
2. Fácil
3. Moderado
4. Difícil
5. Muito Difícil

Qual maneira foi melhor de analisar os códigos?

1. De maneira manual
2. Utilizando a ferramenta DiffMutAnalyze

Por que essa maneira foi melhor? *(Obs: explicar o motivo que achou melhor analisar manualmente ou por meio da ferramenta)*

Pontos positivos da DiffMutAnalyze.

Pontos negativos da DiffMutAnalyze.

Faça uma breve descrição do que você achou do experimento, considerando os pontos importantes.