Avaliação da qualidade de oráculos de teste utilizando mutação

Ana Claudia Maciel

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito:

Assinatura: ______________________

Ana Claudia Maciel

Avaliação da qualidade de oráculos de teste utilizando mutação

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, como parte dos requisitos para obtenção do título de Mestre em Ciências – Ciências de Computação e Matemática Computacional. *VERSÃO REVISADA*

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientador: Prof. Dr. Márcio Eduardo Delamaro

USP – São Carlos
Junho de 2017

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados fornecidos pelo(a) autor(a)

M634a

Maciel, Ana Claudia
Avaliação da qualidade de oráculos de teste utilizando mutação / Ana Claudia Maciel; orientador Márcio Eduardo Delamaro. - São Carlos - SP, 2017.
98 p.

Dissertação (Mestrado - Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional) - Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, 2017.

1. oráculos de teste; teste de mutação; operadores de mutação; teste de software. I. Delamaro, Márcio Eduardo, orient. II. Título.

Ana Claudia Maciel

Quality evaluation of test oracles using mutation

Master dissertation submitted to the Instituto de Ciências Matemáticas e de Computação – ICMC-USP, in partial fulfillment of the requirements for the degree of the Master Program in Computer Science and Computational Mathematics. *FINAL VERSION*

Concentration Area: Computer Science and Computational Mathematics

Advisor: Prof. Dr. Márcio Eduardo Delamaro

USP – São Carlos
June 2017

Aos meus pais, o cuidado e dedicação de vocês foi que deram, em alguns momentos, a esperança para seguir. Ao meu irmão, Andre, sua presença significou segurança e certeza de que não estou sozinha nessa caminhada.

AGRADECIMENTOS

Aos meus pais, Zeni Maciel e Luiz Maciel, agradeço pelo amor incondicional e pela paciência. Por terem feito o possível e o impossível para me oferecerem a oportunidade de estudar em São Carlos, longe deles, acreditando e respeitando minhas decisões e nunca deixando que as dificuldades acabassem com os meus sonhos, serei eternamente grata.

Agradeço ao meu irmão, Andre Maciel, que por mais difícil que fossem as circunstâncias, sempre me ajudava com suas "piadas" e sua alegria.

A todos os meus familiares, primos e tios. Não citarei nomes, para não me esquecer de ninguém. Mas há aquelas pessoas especiais que diretamente me incentivaram. Aos modelos em que procuro me espelhar sempre: aos meus avós Genésio (*in memoriam*), amor incondicional eterno, e Délia pelo exemplo de vida e por não esquecer de mim nenhum dia.

Aos colegas do Laboratório de Engenharia de Software (LabES-ICMC) pela paciência e companheirismo nas infinitas horas que permanecemos juntos no laboratório. Principalmente àqueles que se tornaram grandes amigos.

Ao Rafael Oliveira por me auxiliar com o projeto do início ao fim, passando a mim, confiança e forças para continuar.

Aos meus amigos de São Carlos, com certeza, sem vocês tudo teria sido muito mais difícil. Obrigada pelas risadas, pelas festas, pela amizade, alegrias, tristezas e dores compartilhadas. Com vocês, as pausas entre um parágrafo e outro de produção melhora tudo o que tenho produzido na vida. Vocês foram um presente que ganhei durante esses anos em São Carlos. Amo vocês.

Ao Prof. Dr. Márcio Delamaro, meu orientador, pela amizade, pelas orientações e transferência de conhecimento.

A FAPESP (Fundação de Amparo à Pesquisa do Estado de São Paulo) pelo financiamento do projeto por meio do processo de número 2014/09629-1.

E finalmente agradeço a Deus, por proporcionar estes agradecimentos à todos que tornaram minha vida mais afetuosa, além de ter me dado uma família maravilhosa e amigos sinceros. Deus, que a mim atribuiu alma e missões pelas quais já sabia que eu iria batalhar e vencer. Agradecer é pouco!

"O êxito da vida não se mede pelo caminho
que você conquistou, mas sim pelas
dificuldades que superou no caminho."
(Abraham Lincoln)

RESUMO

MACIEL, A. C.. **Avaliação da qualidade de oráculos de teste utilizando mutação**. 2017. 98 f. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação (ICMC/USP), São Carlos – SP.

No desenvolvimento de software, a qualidade do produto está diretamente relacionada à qualidade do processo de desenvolvimento. Diante disso, atividades de Verificação, Validação & Teste (VV&T) realizadas por meio de métodos, técnicas e ferramentas são de extrema necessidade para o aumento da produtividade, qualidade e diminuição de custos no desenvolvimento de software. Do mesmo modo, técnicas e critérios contribuem para a produtividade das atividades de teste. Um ponto crucial para o teste de software é sua automatização, tornando as atividades mais confiáveis e diminuindo significativamente os custos de desenvolvimento. Na automatização dos testes, os oráculos são essenciais, representando um mecanismo (programa, processo ou dados) que indica se a saída obtida para um caso de teste está correta. Este trabalho de mestrado utiliza a ideia de mutação para criar implementações alternativas de oráculos de teste e, assim, avaliar a sua qualidade. O teste de mutação se refere à criação de versões do sistema em desenvolvimento com pequenas alterações sintáticas de código. A mutação possui alta eficácia na detecção de defeitos e é bastante flexível na sua aplicação, podendo ser utilizada em diversos tipos de artefatos. Adicionalmente, este trabalho propõe operadores de mutação específicos para oráculos, implementa uma ferramenta de apoio à utilização desses operadores para oráculos e também descreve um estudo empírico dos operadores, destacando benefícios e desafios associados ao seu uso.

Palavras-chave: oráculos de teste; teste de mutação; operadores de mutação; teste de software.

ABSTRACT

MACIEL, A. C.. **Avaliação da qualidade de oráculos de teste utilizando mutação**. 2017. 98 f. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação (ICMC/USP), São Carlos – SP.

In software development, product quality is directly related to the quality of the development process. Therefore, activities of Verification, Validation & Testing (VV&T) performed by methods, techniques and tools are urgently required to increase productivity, quality and cost reduction in software development. Similarly, testing technique and criteria contribute to the productivity of test activities. A crucial point for the software testing automation is making the most reliable activities and significantly reducing development costs. Regarding software testing automation, test oracles are essential, representing an mechanism (program, process or data) to indicate whether the actual output for a given test case is correct. This master's thesis aims to explore concepts of mutation testing to create alternative implementations of the oracle procedure and thus assess their quality. Mutation testing refers to the creation of system development versions with minor syntactic code changes. It has high efficiency on defects detecting and it is very flexible in its application and it is being used in various types of artifacts. This work also proposes specific mutation operators for oracles, implements an useful support tool for using these oracle mutation operators and conducts an empirical study of operators, highlighting benefits and challenges associated with their use.

Keywords: test oracles; mutation testing; mutation operators; software testing.

LISTA DE ILUSTRAÇÕES

Figura 1 – Mapa de conceitos da atividade de teste. 34
Figura 2 – Modelo genérico de um processo de teste de software envolvendo oráculo. . 35
Figura 3 – Estrutura arquitetural da MuJava. 44
Figura 4 – Gerador de mutantes da MuJava. 45
Figura 5 – Visualizador de mutantes da MuJava. 46
Figura 6 – Executor de Mutantes da MuJava. 46
Figura 7 – Análise de estudos publicados sobre oráculos de teste. 54
Figura 8 – Estrutura de diretórios da “MuJava 4 JUnit”. 70
Figura 9 – Interface da ferramenta “MuJava 4 JUnit”. 71
Figura 10 – Processo para adicionar as anotações nos arquivos dos oráculos mutantes. . 74
Figura 11 – Exemplo de exibição do relatório sobre mutantes vivos e mortos. 77
Figura 12 – Passos do experimento. 80

LISTA DE CÓDIGOS-FONTE

Código-fonte 1 – Programa original *P*. 39
Código-fonte 2 – Programa mutante *P’* gerado com o uso do operador AOR. 39
Código-fonte 3 – Exemplo de classe de teste do JUnit. 43
Código-fonte 4 – Exemplo do uso do operador de nível de assinatura (original). 58
Código-fonte 5 – Exemplo do uso do operador de nível de assinatura (mutante). 59
Código-fonte 6 – Exemplo do uso do operador de nível de anotação (original). 60
Código-fonte 7 – Exemplo do uso do operador de nível de anotação (mutante). 60
Código-fonte 8 – Exemplo do uso do operador ATV. 62
Código-fonte 9 – Exemplo operador DCfTV. 62
Código-fonte 10 – Exemplo do uso do operador ICfTV. 63
Código-fonte 11 – Exemplo do uso do operador RBA. 64
Código-fonte 12 – Exemplo do uso do operador RTV. 65
Código-fonte 13 – Exemplo operador AEC. 65
Código-fonte 14 – Exemplo de uso do *timeout*. 67
Código-fonte 15 – Exemplo operador DCfT. 67
Código-fonte 16 – Exemplo operador ICfT. 68
Código-fonte 17 – Exemplo operador RTA. 69
Código-fonte 18 – Implementação do operador ATV. 72
Código-fonte 19 – Método `outputToFile()`. 73
Código-fonte 20 – Implementação da classe `JUnitExecutionListener`. 75
Código-fonte 21 – Exemplo do operador ATV no experimento (original). 85
Código-fonte 22 – Exemplo do operador ATV no experimento (mutante). 85

LISTA DE TABELAS

Tabela 1 – Diferentes tipos de assertivas do JUnit. 43
Tabela 2 – Estudos relacionados à avaliação de oráculos de teste. 48
Tabela 3 – Métodos da classe `Assert` utilizados na pesquisa. 59
Tabela 4 – Tabela de operadores (Nível de assinatura). 59
Tabela 5 – Anotações do JUnit exploradas para a criação de oráculos mutantes. 60
Tabela 6 – Tabela de operadores (Nível de anotação). 60
Tabela 7 – Aplicações utilizadas no experimento. 82
Tabela 8 – Casos de teste utilizados no experimento. 82
Tabela 9 – Resumo número de mutantes vivos e mutantes mortos coletados no experimento. 84
Tabela 10 – Mutantes vivos e mortos por operador da “MuJava 4 JUnit”. 86
Tabela 11 – Quantidade de mutantes gerados em cada programa separados por operador. 86

SUMÁRIO

1 INTRODUÇÃO 23

1.1 Contexto 23

1.2 Motivação 25

1.3 Objetivos 25

1.4 Contribuições e limitações 26

1.5 Organização do Trabalho 27

2 FUNDAMENTAÇÃO TEÓRICA 29

2.1 Considerações iniciais 29

2.2 Teste de Software 29

2.2.1 Técnicas e critérios 31

2.2.2 Automatização do teste de software 32

2.3 Oráculos de teste 33

2.3.1 Taxonomia 34

2.3.2 Problema de oráculo: falsos positivos e falsos negativos 38

2.4 Teste de mutação 38

2.4.1 Ferramentas de apoio 41

2.5 *Framework* JUnit 42

2.6 MuJava 44

2.7 Considerações finais 45

3 AVALIAÇÃO DA QUALIDADE DE ORÁCULOS DE TESTE 47

3.1 Considerações iniciais 47

3.2 Critérios de avaliação para oráculos 47

3.3 Visão geral dos estudos primários 53

3.4 Considerações finais 55

4 MUJAVA 4 JUNIT 57

4.1 Considerações iniciais 57

4.2 Operadores de mutação para oráculos de teste baseados em assertivas 57

4.2.1 Operadores nível de assinatura 58

4.2.2 Operadores nível de anotação 59

4.3 Discussão individual sobre os operadores de mutação 61

4.3.1 ATV (Adicionar valor constante) 61

4.3.2 DCfTV (Decrementar valor constante) 62
4.3.3 ICfTV (Incrementar valor constante) 63
4.3.4 RBA (Substituir assertiva booleana) 64
4.3.5 RTV (Remover valor constante) 64
4.3.6 AEC (Adicionar classe esperada) 65
4.3.7 DCfT (Decrementar constante no timeout) 66
4.3.8 ICtT (Incrementar constante no timeout) 68
4.3.9 RTA (Remover constante timeout) 68
4.4 Estrutura da “MuJava 4 JUnit” 69
4.5 Implementação 70
4.5.1 Adicionando anotações JUnit 74
4.5.2 Geração de relatório sobre os oráculos mutantes 75
4.5.3 Biblioteca `OpenJava` 77
4.6 Considerações finais 77

5 ESTUDO EMPÍRICO 79
5.1 Considerações iniciais 79
5.2 Um estudo empírico com a “MuJava 4 JUnit” 79
5.2.1 Design do experimento 79
5.2.2 Questões de pesquisa 81
5.2.3 Seleção de programas 82
5.2.4 Execução do experimento 82
5.3 Resultados 83
5.4 Respostas das questões de pesquisa 83
5.5 Vantagens e desvantagens 86
5.6 Limitações e ameaças à validade 87
5.7 Considerações finais 88

6 CONCLUSÕES 89
6.1 Contribuições 89
6.2 Discussões 90
6.3 Trabalhos futuros 90

REFERÊNCIAS 93

CAPÍTULO

1

INTRODUÇÃO

1.1 Contexto

Teste de software é o processo de execução de um produto de software para determinar se ele atinge suas especificações e funciona corretamente no ambiente para o qual foi projetado (MYERS *et al.*, 2004). O objetivo do teste é revelar falhas em um produto, para que as causas dessas falhas sejam posteriormente identificadas e possam ser corrigidas pela equipe de desenvolvimento antes da entrega final (DELAMARO; MALDONADO; JINO, 2007). Ammann e Offutt (2008) definem a atividade de teste como uma verificação dinâmica do comportamento de um determinado programa para um conjunto finito de casos de teste, selecionados adequadamente para um domínio de execução específico, contra um comportamento esperado, ou seja, com a intenção de encontrar defeitos.

No processo de desenvolvimento de software, defeitos são inseridos por humanos durante o processo de codificação e, apesar do uso dos melhores processos de desenvolvimento, ferramentas ou profissionais, tais defeitos permanecem nos produtos finais, o que torna a aplicação das atividades de Validação, Verificação e Teste (VV&T) fundamentais durante o desenvolvimento de um software. Desse modo, pode-se afirmar que em consequência da crescente utilização de sistemas baseados em computação no mercado atual, houve uma crescente demanda por qualidade e produtividade no processo de produção de software, tornando a atividade de teste indispensável (AMMANN; OFFUTT, 2008).

O planejamento dos testes deve ocorrer em diferentes níveis e em paralelo ao desenvolvimento do software, sendo que os principais níveis de execução dos testes, segundo Rocha, Maldonado e Weber (2001) são: (i) teste de unidade, (ii) teste de integração, (iii) teste de sistema, e (iv) teste de regressão.

O teste de unidade busca identificar falhas relacionadas a algoritmos incorretos, estruturas de dados incorretas, ou simples defeitos de programação (MYERS *et al.*, 2004). Tem como foco

as menores unidades de um programa. No teste de integração, a ênfase é na construção da estrutura do sistema, testa a integração entre as unidades do software, verifica se a união das unidades não leva a falhas e funciona de maneira adequada. O teste de sistema tem objetivo de verificar se as funcionalidades especificadas nos documentos de requisitos estão implementadas corretamente. Por fim, o teste de regressão, que acontece na fase de manutenção do software, verificando se as alterações ou implementações de novos requisitos funcionam corretamente, assim como se os requisitos já testados anteriormente continuam válidos (DELAMARO; MALDONADO; JINO, 2007).

Em relação à complexidade, eficiência, eficácia, produtividade e custo, o teste automatizado permite que os testes sejam executados rapidamente, podendo ser repetidos diversas vezes sem aumento de custo, sendo considerado mais produtivo, sistemático e eficaz. A automatização de teste possui vantagens como reexecução simples, suporte à regressão, tempo de projeto reduzido e maior produtividade (DUSTIN; GARRETT; GAUF, 2009). A capacidade do teste automatizado em realizar as atividades para as quais foi programado, quantas vezes forem necessárias, contribui para o aumento da produtividade e representa uma forma de reduzir os custos, pois todos os casos de teste podem ser facilmente repetidos a qualquer momento e com pouco esforço humano (OLIVEIRA; KANEWALA; NARDI, 2014).

Dentro do contexto da automatização do processo de teste, a grande quantidade de casos de teste gerada durante todo o processo, e a execução repetida de grandes quantidades de casos de teste exigem a utilização de oráculos, os quais representam um método para verificar se o programa em teste tem um comportamento correto durante uma execução específica (BARESI; YOUNG, 2001). Embora os oráculos sejam estudados desde o final da década de 1970, nos últimos anos, estratégias para automatização de oráculos de teste têm atraído muita atenção dos pesquisadores e obtido um grande crescimento (HARMAN *et al.*, 2013).

Tecnicamente, os oráculos de teste representam um mecanismo para verificar as saídas dos casos de teste. Em cenários de teste em que os oráculos de teste não são automatizados, os testadores aplicam várias técnicas para obter alto índice de cobertura e gerar dados de teste. No entanto, as saídas ou comportamentos do SUT (do inglês, *System Under Test*), devem ser verificados ou avaliados por um oráculo humano. Os oráculos humanos são comuns em várias situações, como avaliar a eficácia de novas técnicas de teste, detectar erros inesperados, etc. Para obter atividades testes de software produtivas, os testadores devem automatizar seus oráculos de teste para mitigar os defeitos manuais (OLIVEIRA; KANEWALA; NARDI, 2014).

Este trabalho insere-se no contexto dos esforços de pesquisa que visam a avaliar a qualidade dos oráculos de teste, contribuindo para o teste automatizado. O teste automatizado pode significar uma grande variedade de aspectos relacionados aos processos de teste, tais como, desenvolvimento orientado a testes (TDD), testes de unidade, scripts personalizados, etc. (DUSTIN; GARRETT; GAUF, 2009). A automatização de teste é uma questão de ênfase na área de Engenharia de Software (ES) porque promove abordagens de teste mais sistemáticas.

1.2 Motivação

É senso comum entre os desenvolvedores que, no mercado competitivo atual, a qualidade do produto final é fator essencial em qualquer processo de desenvolvimento. Também é de domínio dos desenvolvedores que a implantação de atividades de avaliação e testes durante todo o processo de desenvolvimento, embora favoreça a qualidade, eleva consideravelmente o tempo e o custo do projeto (DELAMARO; MALDONADO; JINO, 2007). Isso se dá em função das dificuldades encontradas quando se busca uma definição precisa do modo de avaliar a qualidade de determinado processamento. Dificuldade similar é encontrada na procura de um conjunto ideal de testes que seja completo o bastante para revelar as mais diferentes falhas (GONÇALVES *et al.*, 2011).

A automatização de teste gera diminuição nos custos mas é essencial que se tenha conhecimento acerca dos resultados esperados. Dentro do contexto de teste, um oráculo automatizado é a fonte mais precisa sobre os resultados esperados de uma determinada execução (BEIZER, 1990). Oráculos automatizados de teste são componentes essenciais na atividade de teste de software. Definir um oráculo implica em sintetizar uma estrutura formal, ou até mesmo informal, automatizada, que seja capaz de oferecer ao testador um veredito indicativo da exatidão de uma execução do sistema ao final das aplicações do teste. Sendo assim, pode ser dito que oráculo é o mecanismo que define e dá um veredito acerca da correção de uma execução de um programa em teste (OLIVEIRA; DELAMARO; NUNES, 2008).

Apesar da importância dos mecanismos dos oráculos, não são encontradas na literatura formas sistematizadas de se avaliar sua qualidade. Em outras palavras, não há algo análogo aos critérios de teste para oráculos, que indique se o oráculo se comporta como o testador espera. As avaliações de oráculos são sempre feitas em domínios particulares e não podem ser generalizadas para diferentes tipo de oráculos. A implementação de oráculos de teste é tão importante quanto a seleção das entradas de teste e, portanto, deve ser implementada de forma sistemática de acordo com requisitos bem definidos. Mesmo que a implementação e avaliação de oráculos ainda estejam longe de ser sistemáticas (SHRESTHA; RUTHERFORD, 2011), o número de publicações sobre oráculos tem aumentado e os oráculos vêm sendo altamente discutidos em congressos de teste (HARMAN *et al.*, 2013).

1.3 Objetivos

O objetivo desta dissertação de mestrado é contribuir para a definição de mecanismos de avaliação da qualidade de oráculos especificamente no contexto de testes de unidade, descritos no formato do *framework* JUnit, que é um *framework* utilizado para programar e automatizar a execução de testes unitários. Para tanto, utiliza-se uma abordagem baseada em conceitos de mutação, semelhante à que se utiliza para programas convencionais. Desse modo, por meio de implementações e ferramentas, este estudo promove a criação de versões alternativas de

implementações de oráculos e utiliza tais versões como fonte de informação para avaliar o oráculo original. Assim, seguindo os conceitos de mutação, oráculos de teste podem ser avaliados automaticamente. Os objetivos específicos deste trabalho, são:

- Definição de novos operadores de mutação para oráculos de teste baseados em assertivas;
- Inserção dos novos operadores na ferramenta de mutação já existente, MuJava (MA; OFFUTT; KWON, 2005); e
- Validação da proposta.

1.4 Contribuições e limitações

As principais contribuições deste trabalho de mestrado estão associadas ao contexto de automatização de processos associados a Engenharia de Software. Pontualmente, quatro contribuições são providas por meio deste trabalho:

- Proposição e validação de operadores de mutação para oráculos de teste baseados em assertivas;
- Adaptação de uma ferramenta tradicional de mutação para linguagem Java de modo a automatizar a geração de mutantes com os operadores propostos;
- Utilização da abordagem e da ferramenta com programas reais de diferentes funções, mostrando as principais características dos operadores e as limitações da estratégia proposta; e
- Discussão sobre avaliação automatizada de qualidade de oráculos automatizados e sua importância para a melhora nos testes automatizados.

Infelizmente pôde-se notar, durante o desenvolvimento do trabalho, que, ao contrário do que acontece com programas tradicionais e linguagens de programação utilizadas em aplicações “normais”, não existe na literatura uma taxonomia que considere defeitos ou falhas em oráculos de teste.

Esse fato prejudicou sobremaneira a execução deste trabalho que, em alguns momentos baseou-se mais na intuição e experiência da autora do que em artigos publicados. Sem uma taxonomia de defeitos, a definição de operadores de mutação foi baseada principalmente no conhecimento de operadores utilizados para outras linguagens e para domínios (artefatos) diferentes. Portanto, podem não refletir precisamente as necessidades no contexto de oráculos baseados em asserções.

A segunda limitação deste trabalho diz respeito à sua validação empírica. Se por um lado, ao tratarmos de programas convencionais, existe vasto registro sobre objetos de experimentação (programas) e defeitos neles encontrados durante seu desenvolvimento, por outro lado o mesmo não ocorre em relação a oráculos de teste. É fato que em alguns casos podem-se encontrar repositórios de programas que possuem conjuntos de teste no formato JUnit. Porém, mesmo para esses não existe qualquer registro sobre defeitos que foram produzidos no processo de desenvolvimento do oráculo.

Isso não significa que os defeitos não existiram e que não seja necessária uma abordagem para validação desses oráculos. Significa apenas que esses defeitos não foram registrados e que tem-se disponível apenas a "versão final" do oráculo em questão. Em resumo, torna-se muito difícil selecionarem-se objetos de experimentação que possuam defeitos e que possam ser utilizados para validar o presente trabalho.

1.5 Organização do Trabalho

Neste capítulo foram apresentados contexto, motivação e objetivos da pesquisa. Além deste capítulo introdutório, este documento inclui os seguintes capítulos:

- Capítulo 2: são apresentados os principais conceitos relacionados a teste de software, oráculos de teste, teste de mutação, *framework* JUnit e ferramenta de mutação para Java, MuJava;
- Capítulo 3: é apresentada uma revisão bibliográfica acerca de estudos que apresentem estratégias de avaliação de oráculos de teste, destacando as necessidades de estudos nessa área de pesquisa;
- Capítulo 4: é apresentada a ferramenta "MuJava 4 JUnit", detalhes de sua implementação e os operadores de mutação específicos para oráculos de teste projetados durante a realização do mestrado;
- Capítulo 5: é apresentado um estudo empírico aplicando os operadores específicos para oráculos de teste baseados em assertivas da ferramenta "MuJava 4 JUnit"; e
- Capítulo 6: são apresentados trabalhos futuros a serem realizados e as conclusões obtidas após a realização do trabalho.

CAPÍTULO

2

FUNDAMENTAÇÃO TEÓRICA

2.1 Considerações iniciais

Neste capítulo são apresentados os conceitos relevantes para o entendimento do trabalho realizado. O capítulo aborda conceitos relacionados à área de Engenharia de Software e Teste de Software. Adicionalmente, assuntos específicos relacionados a oráculos de teste, teste de mutação, *framework* JUnit e à MuJava são esmiuçados.

2.2 Teste de Software

O teste de software é o processo de execução de um produto de software visando verificar se ele atinge suas especificações considerando o ambiente no qual foi projetado (BERTOLINO, 2003). Ou seja, a atividade de teste é dinâmica, com o objetivo de verificar a corretude do sistema.

Independentemente do tipo de software desenvolvido é necessário elaborar estratégias para planejamento sistemático de teste, execução e controle iniciando de pequenas unidade do software até abranger o programa como um todo. O teste de software absorve a maior parte do esforço técnico em um processo de software. A atividade de teste varia de acordo com a abordagem de desenvolvimento utilizada (PRESSMAN, 2005).

Não se pode garantir que todo software funcione corretamente, sem a presença de erros, haja vista que os sistemas de software, muitas vezes, possuem um grande número de estados com fórmulas, atividades e algoritmos complexos. O tamanho do projeto a ser desenvolvido e a quantidade de profissionais envolvidos no processo aumentam a complexidade das atividades de teste. Dentro da ES, o teste de software é inserido em um contexto mais amplo, conhecido como verificação[1], validação[2] e teste. Idealmente, um sistema de software deveria ser testado

[1] Verificação: garante que o software implementa corretamente uma função específica.
[2] Validação: assegura que o software foi criado de acordo com os requisitos definidos.

com todos os possíveis valores do domínio de entrada. Entretanto, na prática, isso é impossível, devido à grande quantidade de dados possíveis para tal domínio de entrada do SUT (MYERS *et al.*, 2004), ou seja, a partição das entradas em classes equivalentes.

O teste de software é uma atividade dinâmica, seu intuito é executar o SUT utilizando algumas entradas em particular e verificar se seu comportamento está de acordo com o esperado (DELAMARO; MALDONADO; JINO, 2007). O teste de software pode ser visto como uma parcela do processo de qualidade de software e a qualidade da aplicação varia significativamente de sistema para sistema. As informações oriundas da atividade de teste são significativas para as atividades de depuração, manutenção e estimativa de confiabilidade de software. No entanto, a atividade de teste ainda possui altos custos no desenvolvimento de software, podendo assumir até 40% do esforço despendido no processo de desenvolvimento, já que, dependendo do projeto o domínio de entradas e saídas é extenso, gerando vários cenários a serem testados (WAZLAWICK, 2013).

O teste de produtos de software envolve basicamente quatro etapas: (i) planejamento de testes; (ii) projeto de casos de teste; (iii) execução; e (iv) avaliação dos resultados dos testes (BARBOSA *et al.*, 2000). Inicialmente os testes são focados individualmente, garantindo o funcionamento como uma unidade. Em seguida, as unidades devem ser integradas construindo o software como um todo. Após o software estar integrado, são realizados testes utilizando critérios de validação, estabelecidos durante a análise de requisitos, garantindo que o software satisfaz todos os requisitos pré-definidos.

Um ponto crucial na atividade de teste, independentemente da etapa, é o projeto e/ou a avaliação da qualidade de um determinado conjunto de casos de teste **T** utilizado para o teste de um produto **P**, dado que, em geral, é impraticável utilizar todo o domínio de dados de entrada para avaliar os aspectos funcionais e operacionais do SUT (DELAMARO; MALDONADO; JINO, 2007).

Atividades de VV&T estão diretamente relacionadas com garantia de qualidade e visam à revelação de falhas e detecção de defeitos. Nesse contexto, é importante destacar que a IEEE 610.12 (IEEE, 1990) padroniza os termos utilizados na área de teste de software: (i) **Engano (*mistake*)**: um resultado incorreto é causado por engano humano (programador); (ii) **Defeito (*fault*)**: passo, processo ou definição de dados incorretos, podendo configurar diversas falhas; (iii) **Erros (*errors*)**: diferença entre o valor obtido e o esperado; e (iv) **Falha (*failure*)**: parte do código é executada levando o sistema a um estado inesperado, diferente da saída esperada.

Além das etapas descritas previamente, a atividade de teste é comumente dividida em fases que possuem objetivos distintos. Neste contexto, cada etapa da fase de teste apresenta características únicas e serve para guiar o processo de teste de software. De acordo com Delamaro, Maldonado e Jino (2007), as fases podem ser definidas em:

- **Teste de unidade:** tem como objetivo testar as menores unidades que compõem o sis-

tema (métodos, funções, procedimentos, classes, etc.). Espera-se encontrar defeitos de programação, algoritmos incorretos ou mal implementados, estruturas de dados incorretas, etc;

- **Teste de integração:** o foco principal é verificar as estruturas de comunicação entre as unidades que compõem o sistema. As técnicas de projeto de casos de teste que exploram entradas e saídas são as mais utilizadas durante a fase de integração; e

- **Teste de sistema:** o objetivo é verificar se os requisitos satisfazem as especificações e se as funcionalidades do sistema foram implementadas corretamente, ou seja, o sistema é testado como um todo procurando simular um ambiente de execução real.

2.2.1 Técnicas e critérios

O alto custo do teste fez com que surgissem estudos teóricos e empíricos de critérios de teste fornecendo subsídios para o estabelecimento de estratégias de baixo custo e alta eficácia. Uma escolha de critérios de teste adequados faz com que as vantagens de cada um desses critérios sejam combinadas objetivando uma atividade de teste de maior qualidade e mais complexa (DELAMARO; MALDONADO; JINO, 2007).

As técnicas de teste de software existentes procuram estabelecer regras para definir subdomínios, a fim de criar conjunto de casos de teste que satisfaçam os requisitos de teste pertencentes ao subdomínio abordado. Técnicas de teste devem ser aplicadas de forma complementar, de modo que as vantagens individuais possam ser melhor exploradas, chegando a uma melhor abordagem. Estudos teóricos e empíricos de critérios de teste têm chamado a atenção na comunidade científica e possuem alta relevância para a área, fornecendo subsídios para o estabelecimento de estratégias de baixo custo e alta eficácia (SOUZA *et al.*, 2015; SHIN; JEE; BAE, 2012). Considerando somente as técnicas de teste, pode-se destacar, principalmente, teste funcional, teste estrutural e teste baseado em defeitos.

O teste funcional, também conhecido na literatura como "caixa preta", não utiliza o código-fonte para a geração dos casos de teste. Assim, os conjuntos de testes são elaborados apenas com base nos requisitos do software e nas especificações das suas funcionalidades (MYERS *et al.*, 2004). Um dos critérios da técnica funcional é o **Particionamento em Classes de Equivalência** que define um conjunto de condições válidas ou inválidas para um subconjunto de entradas, denominadas classes de equivalência. Uma vez identificadas as classes de equivalência, devem-se determinar os casos de teste que enderecem a pelo menos um elemento de cada classe (COPELAND, 2004). Outro critério da técnica funcional é a **Análise do valor limite**, utilizado em conjunto com o particionamento em classes de equivalência, no qual os dados de teste devem ser selecionados de forma que valores no limite de cada classe de equivalência sejam usados.

No teste estrutural os critérios e requisitos de teste são derivados exclusivamente a partir das características do código-fonte do software em teste, utilizam diferentes estruturas internas dos sistemas em avaliação, possibilitando que haja uma maior atenção aos pontos mais importantes do código. O teste estrutural também é conhecido como teste “caixa branca” (KOSCIANSKI; SOARES, 2007). A maioria dos critérios desta técnica utiliza uma representação do programa conhecida como grafo de fluxo de controle (GFC) (MALDONADO, 1991). Os critérios baseados em fluxo de controle utilizam características de controle da execução do programa, como comandos ou desvios. Os critérios mais conhecidos dessa classe são: (i) Todos-Nós; (ii) Todas-Arestas; e (iii) Todos-Caminhos. Outros critérios da técnica estrutural são os baseados em fluxo de dados. A idéia básica destes critérios consiste em relacionar a atribuição de valores às variáveis e o seu uso posterior, estabelecendo que tais estruturas devem ser exercitadas. Assim, a ocorrência de uma variável em um programa pode ser de dois tipos: definição e uso. Os critérios que fazem parte dessa classe são, por exemplo: (i) Todas-Defs; (ii) Todos-Usos; e (iii) Todos-Du-Caminhos (MATHUR, 2008).

Por fim, o *teste baseado em defeitos* utiliza informações sobre defeitos frequentes no processo de desenvolvimento de software e é o foco principal deste trabalho, por isso é detalhado na Seção 2.4. Na seção seguinte é apresentada uma discussão sobre automatização das atividades de teste.

2.2.2 Automatização do teste de software

O teste automatizado tem o objetivo de aplicar estratégias e ferramentas para a execução dos casos de teste, tendo em vista a redução do esforço humano em atividades manuais repetitivas. A automatização permite que técnicas e critérios sejam implementados por meio de ferramentas que produzem relatórios que, então, podem ser analisados por humanos. A automatização possibilita a execução do teste de regressão com maior amplitude e profundidade (KANER; BACH; PETTICHORD, 2008).

O teste manual é improdutivo, pois a cognição humana faz com que o testador se engane. A execução e repetição de um vasto conjunto de testes manualmente é uma tarefa cara e cansativa, levando os testadores a não verificar todos os casos a cada mudança significativa do código. É diante desse cenário que são entregues sistemas de software, diminuindo sua qualidade.

O desenvolvimento de ferramentas de teste para o suporte à automatização das atividades de teste é fundamental. Isso é devido ao fato de que a atividade de teste é muito propensa a defeitos, além de improdutiva, se aplicada manualmente.

Ferramentas são importantes também para dar apoio a estudos empíricos que visem avaliar e comparar os diversos critérios de teste. Assim, a disponibilidade de ferramentas de teste propicia maior qualidade e produtividade para as atividades de teste (BARBOSA *et al.*, 2000). Automatizar atividades de teste é uma importante vertente da Engenharia de Software pelo fato

de promover abordagens de teste mais sistemáticas, produtivas e confiáveis (IVORY; HEARST, 2001).

O teste automatizado deve compor algumas características básicas (HOFFMAN, 2001):

- Executar automaticamente um ou mais casos de teste específicos ou um conjunto de casos de teste;
- Não necessitar de intervenção humana após a execução dos testes;
- Configurar automaticamente o ambiente de teste;
- Capturar os resultados relevantes;
- Comparar os resultados obtidos com os resultados esperados, gerando estatísticas; e
- Analisar os resultados.

A automatização das atividades de teste apresenta diversas vantagens, pois os casos de teste podem ser facilmente executados a qualquer momento exigindo pouco esforço humano. Quando executada corretamente, a automatização de teste é uma das mais eficientes formas de reduzir o tempo de teste no ciclo de vida do software, diminuindo o custo e aumentando a produtividade, além de, consequentemente, aumentar a qualidade do produto final (FANTINATO *et al.*, 2005). Por isso, diversos pesquisadores tem dedicado esforços de pesquisa para a criação de recursos para o teste automatizado.

2.3 Oráculos de teste

Um oráculo de teste é um mecanismo capaz de decidir se uma execução está correta ou não, ou seja, se apresenta ou não os resultados esperados (WEYUKER, 1982). Oráculos representam um método para verificar se o SUT obtém um comportamento correto em uma determinada execução (BARESI; YOUNG, 2001). A automatização do teste requer a incorporação de mecanismos de oráculo para que as saídas possam ser avaliadas. Diante disso, em abordagens de teste automatizado, a presença de um oráculo é assumida explícita ou tacitamente (OLIVEIRA, 2010).

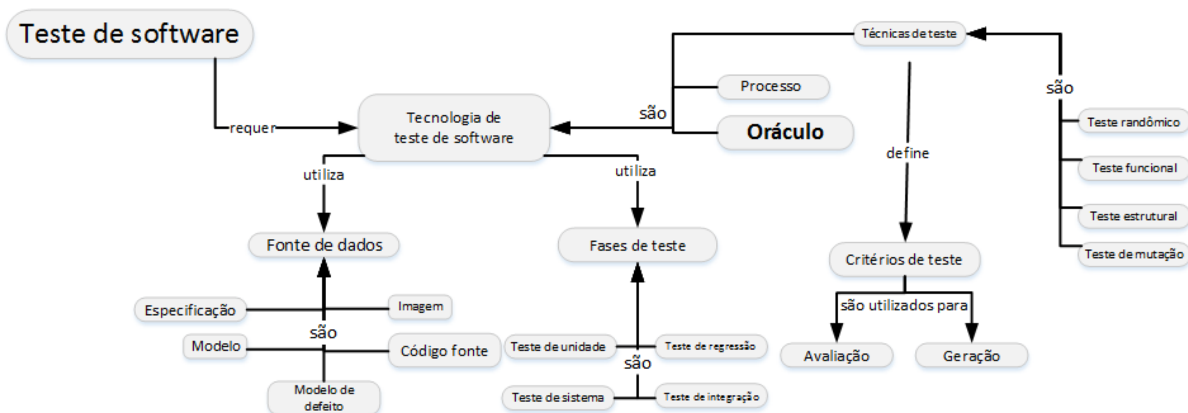
Aspectos dirigidos aos oráculos tratam de uma questão básica na atividade de teste: se o comportamento de um programa **P** com um determinado dado de teste é correto ou não. Entretanto, não é uma tarefa trivial determinar para **P** um conjunto **S** de saídas esperadas (NARDI, 2013). Em alguns casos o procedimento de oráculo **O** e sua avaliação podem ser triviais. Se a saída esperada, **S**, e o conjunto de entradas, **E**, são simples valores numéricos, basta compará-los para decidir se o resultado produzido condiz com o esperado. Em outras situações, a complexidade da saída de **P** pode representar um problema para a automatização fazendo com que **S** seja bastante

complexo. Isso ocorre, por exemplo, quando a saída é uma imagem, processada por **P**. Nesse caso, para avaliar a correção da execução pode ser necessário extrair características como cor, forma ou textura para que se decida sobre sua correção (DELAMARO; NUNES; OLIVEIRA, 2013).

Muitas vezes o papel de oráculo é desempenhado pelo próprio testador que deve consultar a especificação e dar o veredito sobre a correção da execução. Para diminuir esforços, os testadores devem automatizar os oráculos pois abordagens manuais podem gerar resultados insatisfatórios ou incorretos devido a enganos humanos (NUNES; DELAMARO; OLIVEIRA, 2009).

Visando estabelecer uma definição da função do oráculo dentro do contexto de teste, a Figura 1 representa um mapa de conceitos da atividade de teste de software contendo informações sobre as fases de teste, critérios de teste, processos e oráculos de teste. De acordo com a figura apresentada originalmente em Durelli *et al.* (2013), o oráculo de teste é uma tecnologia de teste de software que pode ser associada com diferentes processos e técnicas de teste de software. A Figura 1 ilustra um conceito no qual pode-se inferir que os oráculos de teste são independentes de qualquer outro aspecto de teste de software. Em um ambiente específico de teste, os oráculos podem ser resultado de uma série de processos. A Figura 2 mostra um exemplo prático de uma associação genérica entre todos esses processos de teste de software.

Figura 1 – Mapa de conceitos da atividade de teste.

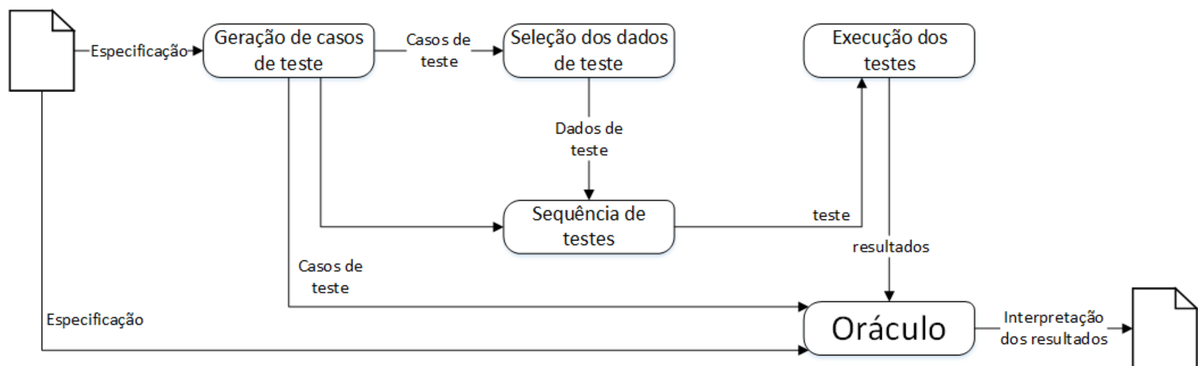


Fonte: Adaptada de Durelli *et al.* (2013).

2.3.1 Taxonomia

O termo "oráculos de teste" apareceu pela primeira vez em um trabalho de William Howden no ano de 1978 (BUDD *et al.*, 1980). Propostas de novas técnicas e conceitos para a especificação formal de oráculos de teste atingiram o ápice em 1990, mas foram gradualmente diminuindo na última década (PEZZÈ; ZHANG, 2014). No entanto, área de pesquisa relacionada

Figura 2 – Modelo genérico de um processo de teste de software envolvendo oráculo.



Fonte: Adaptada de Oliveira, Kanewala e Nardi (2014).

a oráculos de teste obteve ascensão, aumentando a quantidade de publicações por ano (HARMAN *et al.*, 2013).

Devido à diversidade dos sistemas atuais, existem diferentes tipos de oráculos de teste. Um oráculo desenvolvido para um propósito específico é capaz de predizer ou julgar o comportamento específico e particular de determinada aplicação. Durante a implementação de um sistema de teste, uma prática comum é a incorporação de diferentes oráculos para verificar diferentes condições e domínios para checar comportamentos e resultados (BARESI; YOUNG, 2001).

Nesse contexto, os oráculos são classificados de diferentes formas na literatura. McDonald e Strooper (1998) apresentam uma divisão genérica que é adotada entre os pesquisadores:

- Ativos: implementam o comportamento esperado do software em teste, são diretamente responsáveis por dirigir ou coordenar as atividades de testes. Oráculos ativos produzem um resultado esperado para uma entrada e exploram uma comparação para verificar os resultados reais diante dos resultados esperados; e

- Passivos: oráculos que recebem como entrada o par, comportamento desejado, comportamento real produzido, e julgam a correção da execução. Um oráculo passivo deve apenas verificar o comportamento de um componente, mas não reproduzi-lo.

Beizer (1990), por sua vez, identifica cinco categorias de oráculos:

- Oráculo "Kiddie": baseado na observação das saídas de um sistema em teste. Quando o teste é executado, se a saída foi classificada como correta e coerente, a saída será julgada como correta;

- Conjunto de Teste de Regressão: as saídas são derivadas de uma versão anterior e correta do sistema que está sendo testado;

- Validação de Dados: as saídas são oriundas de uma tabela de valores, fórmula ou qualquer outra saída válida;
- Conjunto de Teste Padrão: as decisões são baseadas em um conjunto de teste padrão, previamente criado e validado; e
- Programa Existente: o oráculo é baseado em resultados de entradas executadas em um sistema já existente.

Shrestha e Rutherford (2011), por sua vez, categorizam quatro tipo de oráculos:

- *Ideal Oracle:* um oráculo ideal fornece um veredito infalível sobre qualquer possível execução do programa em teste. É o oráculo mais preciso dentre os outros tipos existentes;
- *Null Oracle:* é fornecido em tempo de execução do sistema. Por exemplo, a Máquina Virtual do Java que monitora a execução dos programas e sinaliza os erros por meio de exceções fornecidas pela linguagem;
- *Test-Harness Oracle:* oráculos implementados como expressões *booleanas* que comparam os valores retornados com os casos de teste. A implementação deste tipo de oráculo é *ad hoc*, tornando difícil a realização de uma avaliação empírica do mesmo; e
- *Runtime Assertion Checking:* utiliza assertivas (do inglês, *assertions*), ou seja, comandos formais sobre o estado do programa e são tipicamente implementadas como expressões *booleanas* que são avaliadas no local do código onde elas aparecem.

Recentemente, Pezzè e Zhang (2014) dividiram os oráculos em três categorias:

- Taxonomias genéricas: relacionadas ao propósito geral do oráculo de teste, os oráculos desta categoria não estão associados a aspectos técnicos aplicados para implementar a função de oráculo como fonte de informação ou automatização:
    1. Pseudo-Oráculos: são programas escritos em paralelo com o SUT, e seguindo as mesmas especificações. Oráculo e SUT executam com os mesmos dados de entrada e saída e são comparados;
    2. Oráculos parciais: têm objetivo de identificar onde os resultados dos testes estão incorretos sem saber qual a saída correta;
    3. Oráculos Ativos: copia o comportamento do SUT e um oráculo passivo verifica o comportamento do SUT, mas não o reproduz; e
    4. Oráculos Passivos: age como comparador simples entre os resultados dos testes atuais e dos resultados esperados.

- Taxonomias específicas: definidas analisando diferentes aspectos como fonte de informação, automação de processos e características de informação. Nessa categoria, Harman *et al.* (2013) classificaram os oráculos em quatro categorias:

  1. Oráculos Específicos: todos os oráculos de teste baseados em especificações a fim de julgar os comportamentos e execuções dos testes;
  2. Oráculos Derivados: oráculos que utilizam qualquer tipo de artefatos ou recursos que foram criados por outros oráculos de teste;
  3. Oráculos Implícitos: têm a função de identificar situações em que a presença de um defeito no SUT é óbvio; e
  4. Ausência de Oráculos: existe uma ausência de oráculos automatizados, porém o objetivo principal é reduzir os esforços humanos no julgamento de saídas de teste.

- Taxonomias baseadas nas características dos oráculos: representa o comportamento esperado do SUT. Este tipo de oráculo pode ser ser obtido a partir da especificação, os resultados armazenados, a execução paralela do programa, máquinas de aprendizagem, e outras fontes.

Outro estudo recente que dedica esforços de pesquisa em direção à categorização e classificação dos diversos tipos de oráculos de teste consiste do *survey* apresentado por Oliveira, Kanewala e Nardi (2014). Tal estudo define uma taxonomia acerca de oráculos de teste similar à taxonomia apresentada por Harman *et al.* (2013). Entretanto, analisando características básicas de oráculos para diferentes domínios, os autores propõem a seguinte taxonomia: (1) oráculos baseados em especificação; (2) oráculos baseados em relações metamórficas; (3) oráculos baseados em aprendizado de máquina; e (4) oráculos baseados em diferentes versões do SUT.

Além da categorização dos oráculos, o estudo apresenta diversas análises quantitativas incluindo mais de 300 estudos cujos temas centrais são oráculos de teste. Mapa de colaboração entre pesquisadores, número de estudos para cada tipo de oráculo, número de estudos conduzidos em ambiente industrial, etc. são algumas das informações apresentadas no estudo. Além de mostrar que as pesquisas acerca de oráculos de teste vêm aumentando, o *survey* revela o fato que o uso de relações metamórficas, apesar de estar intimamente ligado às especificações do SUT, merecem ser considerados em uma categoria de oráculos específica, compondo uma área de pesquisa particular. Relações metamórficas são estudos de como alterações em entradas do SUT, sejam essas aleatórias ou definidas automaticamente, podem refletir nas saídas (OLIVEIRA; KANEWALA; NARDI, 2014). Sendo assim, relações metamórficas podem ser definidas como um conjunto de propriedades que associam entradas e saídas do SUT.

Os oráculos podem assumir diferentes papéis e diferentes funções, dependendo das aplicações e do SUT. Entretanto, oráculos têm sempre a mesma meta: identificar e revelar ao testador se determinada execução é válida ou não (BARESI; YOUNG, 2001).

2.3.2 Problema de oráculo: falsos positivos e falsos negativos

O problema de oráculo está definido nos casos em que, utilizando meios práticos, é impossível ou muito difícil decidir sobre a correção de casos de teste e saídas de teste (WEYUKER, 1982). A decisão sobre a correção de uma execução e, consequentemente, revelações de falhas é o aspecto mais essencial relacionado a qualquer atividade de teste, mesmo quando realizada de forma manual (STAATS; WHALEN; HEIMDAHL, 2011).

Apesar das diversas taxonomias, não existe um oráculo de teste ideal (BARESI; YOUNG, 2001). Nesse contexto, dependendo do tipo de oráculo, podem ocorrer os seguintes problemas (WEYUKER, 1982):

- Falsos positivos: o resultado do teste não apresenta inconsistência. No entanto, algum estado inconsistente pode não ter sido verificado no processo de teste; e
- Falsos negativos: casos em que o resultado do oráculo apresenta falha enquanto o SUT está funcionando adequadamente.

Casos de falsos positivos/negativos são extremamente indesejados, entretanto eles devem ser considerados e aferidos pelos testadores. Isso é devido ao fato que dependendo do SUT, é extremamente difícil predizer comportamentos esperados comparados aos comportamentos reais. Falhas podem ocorrer em diversas situações tornando a verificação do oráculo complexa ou até impossível de ser executada (BARESI; YOUNG, 2001).

2.4 Teste de mutação

O Teste de Mutação (DEMILLO; LIPTON; SAYWARD, 1978) é um critério de teste da técnica baseada em defeitos. O programa que está sendo testado é alterado diversas vezes, criando-se um conjunto de versões alternativas com alterações sintáticas, chamados mutantes. A ideia básica é gerar várias versões modificadas do programa original, com o objetivo de revelar os defeitos mais comuns introduzidos nos programas pelos programadores. O trabalho do testador é selecionar casos de teste que identifiquem a diferença de comportamento entre o programa original e os programas mutantes (DELAMARO; MALDONADO; JINO, 2007), "matando" os mutantes. Tais diferenças são consolidadas ao se notar diferentes saídas do programa real quando comparados aos programas mutantes.

O Teste de Mutação tem por finalidade medir o grau de adequação de um conjunto de casos de teste (DEMILLO; LIPTON; SAYWARD, 1978; DEMILLO, 1980). O conjunto de casos de teste capaz de identificar os mutantes também deve ser capaz de revelar falhas reais.

Segundo DeMillo, Lipton e Sayward (1978), a definição do critério de mutação é baseada em duas hipóteses:

- *Hipótese do Programador Competente:* assume que programas desenvolvidos por programadores competentes estão corretos ou bem próximos do correto; e

- *Efeito de Acoplamento:* assume que um conjunto de dados de teste capaz de evidenciar erros simples em um programa, também é capaz de detectar erros complexos, devido ao fato de estudos empíricos terem evidenciado que erros complexos normalmente estão relacionados a erros simples.

Considerando tais hipóteses, o testador deve aplicar um conjunto de teste T contra um programa P com o objetivo de verificar se o resultado obtido é igual ao resultado esperado. A partir do programa P é gerado um conjunto de programas $(P_1, P_2, P_3, P_4, ..., P_n)$ denominados mutantes. Tais programas mutantes são gerados com objetivo de simular os enganos comuns que podem acontecer durante o processo de desenvolvimento de software (AMMANN; OFFUTT, 2008). Para modelar os desvios sintáticos e gerar os mutantes, os **operadores de mutação** (do inglês, *mutant operators*) são aplicados a um programa **P**, transformando-o em programas similares: os mutantes de **P**. Entende-se por operador de mutação as regras que definem as alterações sintáticas que devem ser aplicadas no programa original **P** (DELAMARO; MALDONADO; JINO, 2007).

O Código-fonte 1 apresenta o trecho de código da versão original de um programa, e o Código-fonte 2 apresenta um mutante do Código-fonte 1 gerado a partir da aplicação do operador de mutação *Arithmetic Operator Replacement* (AOR) .

Código-fonte 1 – Programa original *P*.

```
1: int a = 0;
2: int b = 0;
3: int soma = 0;
4: soma = a - b;
```

Código-fonte 2 – Programa mutante *P'* gerado com o uso do operador AOR.

```
1: int a = 0;
2: int b = 0;
3: int soma = 0;
4: soma = a + b;
```

Há mutantes que têm o mesmo comportamento que o programa original, produzindo a mesma saída, e não podem ser detectados por nenhum caso de teste. Esses mutantes são chamados de mutantes equivalentes. A equivalência entre programas é uma questão indecidível e requer a intervenção manual do testador. Nas ferramentas de mutação, geralmente, existem funcionalidades específicas para identificar e marcar os mutantes equivalentes. Contudo, existem

estudos que trabalham no sentido de minimizar este problema. Orzeszyna (2011) mostra por meio de uma revisão sistemática que existem diversos trabalhos que investigam alternativas para solucionar esse problema. Entre as alternativas propostas estão: (i) técnicas para detecção de mutantes equivalentes, (ii) técnicas para evitar a geração de mutantes equivalentes, (iii) técnicas para a sugestão de mutantes equivalentes.

O teste de mutação fornece uma medida objetiva do nível de confiança da adequação dos casos de teste analisados por meio da definição de um escore de mutação (MS, do inglês, *Mutation Score*). Para tanto, é necessário a realização do cálculo que relaciona o número de mutantes gerados, com o número de mutantes mortos. O escore de mutação de um conjunto de casos de teste **T** para um programa **P** é obtido com a seguinte fórmula:

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

Onde:

DM(P,T): número de mutantes mortos do programa **P** pelo conjunto de casos de teste **T**.
M(P): número total de mutantes gerados para um programa **P**.
EM(P): número de mutantes equivalentes a um programa **P**.

O MS varia entre 0 e 1 e, quanto maior o escore de mutação, mais completo é o conjunto de casos de teste. Contudo, atingir 100% do escore de mutação exige lidar com o problema da equivalência de mutantes. Isso muitas vezes se torna um grande desafio devido ao número elevado de mutantes gerados, mesmo para programas relativamente simples. Os mutantes equivalentes são aqueles que apesar de serem sintaticamente diferentes do programa original, apresentam a mesma funcionalidade daquele, ou seja, não causa efeito nas saídas. Por meio do escore de mutação é possível quantificar e avaliar a qualidade da atividade de teste (BARBOSA, 1998).

Um problema que impede o teste de mutação de tornar-se uma técnica de teste prática é o alto custo computacional de executar um enorme número de mutantes contra um conjunto de casos de testes (BARR *et al.*, 2014). Os outros problemas estão relacionados à quantidade de esforço humano envolvido na utilização do teste de mutação, por exemplo no oráculo humano. O problema oráculo humano refere-se ao processo de verificação de saída do programa original para cada caso de teste. Particularmente, isso não é um problema exclusivo do teste de mutação (OFFUTT, 1992). Em diversas técnicas de testes, uma vez que um conjunto de entradas foram obtidas, ainda há o problema de verificação de saída. O teste de mutação é eficaz e exigente e isso pode levar a um aumento no número de casos de teste. Além disso, por causa da impossibilidade de decidir a equivalência de mutantes, a detecção de mutantes equivalentes envolve esforço humano adicional (JIA; HARMAN, 2011).

Uma das vantagens do teste de mutação é que sua aplicação pode ser feita não apenas em programas mas também em outros artefatos que sejam executáveis (DELAMARO; OFFUTT; AMMANN, 2014). Por exemplo, a validação de alguns tipos de modelos já foi explorada utilizando-se o teste de mutação. São os casos de Máquinas de Estados Finitos (FABBRI *et al.*, 1999a), Máquinas de Petri (FABBRI *et al.*, 1995), Estelle (SOUZA *et al.*, 1999) e Statecharts (TRAKHTENBROT, 2007).

2.4.1 Ferramentas de apoio

Diversas ferramentas foram desenvolvidas para dar apoio a utilização do Teste de Mutação. Sem o auxílio das ferramentas a aplicação do critério pode ser comprometida, devido à série de passos envolvidos durante o processo.

Na década de 1990, a ferramenta *PRogram TEsting Using Mutants* (PROTEUM) foi desenvolvida (DELAMARO *et al.*, 1993), iniciando um novo período na utilização do teste de mutação, dando suporte a programas C, que mais tarde foi estendida para outros modelos como Máquinas de Estados Finitos, Redes de Petri e Statecharts (FABBRI *et al.*, 1995; FABBRI *et al.*, 1999a; FABBRI *et al.*, 1999b).

A atividade de teste na PROTEUM é conduzida por meio de sessões de teste. Cada sessão de testes é caracterizada por uma base de dados composta por um conjunto de casos de testes e um conjunto de mutantes. A ferramenta PROTEUM é composta por uma série de módulos que atuam sobre a base de dados. Cada modulo é responsável por realizar uma atividade ligada ao teste de mutação (DELAMARO; MALDONADO; VINCENZI, 1997).

Dando suporte a programas Java, têm-se a MuJava[3] (MA; OFFUTT; KWON, 2006) que implementa operadores de mutação a nível de método e de classe. A ferramenta possui uma interface gráfica para facilitar aos testadores a aplicação do critério de mutação em que os mutantes são gerados automaticamente, e executados contra um conjunto de testes. Adicionalmente, a ferramenta informa o escore de mutação do conjunto de testes. A MuJava é detalhada na Seção 2.6, pois ela foi adaptada para implementar os operadores específicos para oráculos de teste projetados neste trabalho.

A ferramenta Mothra (CHOI *et al.*, 1989) era um ambiente de teste para programas em Fortran 77[4] e foi desenvolvida por pesquisadores da *Purdue University* e do *Georgia Institute of Technology*. A ferramenta possibilita ao testador selecionar o conjunto de operadores de mutação a serem aplicados e o valor da porcentagem de mutantes que deve ser utilizado durante o teste de um determinado programa. A criação dos mutantes é feita gradativamente à medida que o teste é desenvolvido. A execução dos mutantes e a comparação com os resultados originais é feita de forma automática.

[3] http://cs.gmu.edu/ offutt/mujava/
[4] http://web.stanford.edu/class/me200c/tutorial_77/

Uma das principais características da ferramenta Mothra é o gerador automático de casos de teste, Godzilla (DEMILLI; OFFUTT, 1991) em que cada mutante é associado a uma restrição, o gerador procura criar casos de teste que satisfaçam as restrições e levem à distinção dos mutantes correspondentes (DELAMARO; MALDONADO; JINO, 2007).

A ferramenta *Pascal Mutants* (OLIVEIRA *et al.*, 2014) foi projetada para dar suporte à mutação na linguagem de programação Pascal. Possui sete operadores de mutação, e injeta defeitos artificiais no programa original, gerando diferentes versões mutantes do programa original. Os usuários podem selecionar operadores de mutação específicos, criar projetos de testes, criar e executar os dados de teste, verificar o escore de mutação e comparar códigos originais e mutantes. Tecnicamente, a *Pascal Mutants* integra seis módulos principais: (1) um analisador de sintaxe da linguagem e gerador de árvore de sintaxe; (2) um compilador e um gerador de mutantes; (3) um gerenciador de casos de teste; (4) um oráculo de teste; (5) um gerenciador de mutantes; e (6) um gerador de relatórios.

2.5 *Framework* JUnit

Dentro da família xUnit, o *Framework* JUnit permite a automação de testes de unidade para programas Java, implementação de classes e métodos de testes, execução dos testes de forma visual, criação de testes individuais para os métodos pertencentes a uma mesma classe, definição e execução de um conjunto de testes individuais e relato de problemas ocorridos após a realização dos testes com identificação específica da localização dos defeitos (CHEON; LEAVENS, 2002). Desse modo, o JUnit é uma ferramenta de apoio ao teste de unidade que auxilia desenvolvedores na automação dos testes e verificação dos resultados e consequentemente na implementação de oráculos de teste. No JUnit, uma classe de teste consiste em um conjunto de métodos de teste. Tecnicamente, o *framework* usa a introspecção para encontrar todos esses métodos.

As duas principais classes do JUnit são: `TestCase` e `TestSuite`. Essas classes implementam a interface `Test` que contém o método *run* responsável pela execução dos testes. A classe `TestCase` é responsável por executar um caso de teste, que corresponde a um método testado no JUnit. A classe `TestSuite` é responsável por executar um conjunto de casos de teste ou classes de teste (HUNT; THOMAS, 2015). Ao executar um caso de teste usando o JUnit, todas as suas entradas são validadas por meio dos métodos da classe `Assert`. O JUnit registra todo o caminho percorrido durante as falhas obtidas nos métodos da classe e relata os resultados após a execução de todos os testes.

O Código-fonte 3 apresenta uma classe de teste do JUnit, em que um caso de teste é executado para verificar se uma lista está vazia ou não e um outro caso de teste para assegurar que ao tentar extrair um elemento da lista vazia, há o lançamento de determinada exceção. Na Linha 7 da classe apresentada no Código-fonte 3 o oráculo é definido por meio do método `assertEquals` do *framework* JUnit. O caso de teste iniciado na linha 10 verifica se o tratamento

de exceções está adequadamente ajustado.

Código-fonte 3 – Exemplo de classe de teste do JUnit.

```
1: import org.junit.Assert;
2: import org.junit.Test;
3: public class ExampleTest{
4:   private java.util.List<Object> emptyList;
5:   @Test
6:   public void executeTest(){
7:     Assert.assertEquals ("A lista deve ter 0 elementos",
8:     0,emptyList.size());
9:   }
10:  @Test(expected=IndexOutOfBoundsException.class)
11:  public void testException(){
12:    emptyList.get(0);
13:  }
14: }
```

A utilização do JUnit facilita a criação de casos de teste, pelo fato de automatizar os testes de software, evitando escrever testes duplicados e permite escrever testes que retêm seu valor ao longo do tempo, ou seja, podem ser reutilizados. O *framework* fornece uma completa API (do inglês, *Application Program Interface*) para construir e executar os testes. Adicionalmente, as anotações `@Before` e `@After` permitem especificar pré e pós condições comuns a todos os casos de teste.

Tecnicamente, o JUnit implementa a base para a automatização dos oráculos por meio das suas classes de teste e os comandos `Assert`. JUnit fornece sobrecarga de métodos das assertivas para todos os tipos primitivos, objetos e matrizes. Os métodos `Assert` normalmente começam com `assert` e permitem que se especifique a mensagem de erro, o resultado esperado e o resultado obtido. Um método `assert` compara o resultado retornado diante da execução de um caso de teste com o resultado esperado, e lança uma `AssertionException` se a comparação falhar. A Tabela 1 apresenta alguns dos diferentes tipos de assertivas existentes no JUnit.

Tabela 1 – Diferentes tipos de assertivas do JUnit.

Método	**Descrição**
fail(String)	Faz o método falhar.
assertTrue([message], boolean condition)	Verifica se a condição booleana é verdadeira.
assertFalse([message], boolean condition)	Verifica se a condição booleana é falsa.
assertEquals([String message], expected, actual)	Testa se dois valores são os iguais.
assertNull([message], object)	Verifica se determinado objeto é nulo.
assertNotNull([message], object)	Verifica se determinado objeto não é nulo.
assertSame([String], expected, actual)	Verifica de duas variáveis fazem referência a mesma instância de um objeto.
assertNotSame([String], expected, actual)	Verifica de duas variáveis fazem referência a objetos diferentes.

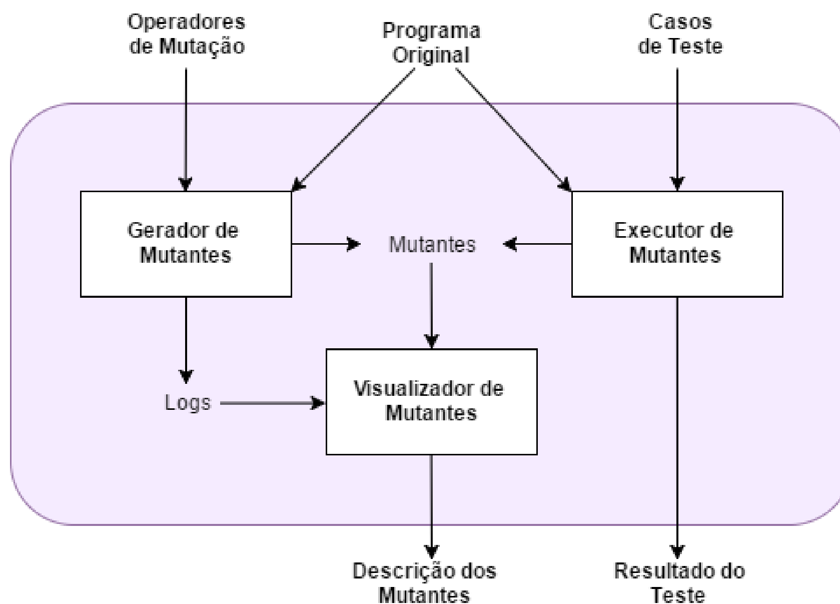
2.6 MuJava

Desenvolvida em ambiente acadêmico, a MuJava (do inglês, ***Mu****tation System for* ***Java***) é uma ferramenta de mutação para programas desenvolvidos na linguagem Java (MA; OFFUTT; KWON, 2006). A ferramenta gera mutantes automaticamente, executa os mutantes contra um conjunto de casos de testes, e reporta o escore de mutação após a execução do caso de teste. Devido a sua interface amigável e sua facilidade de uso, a MuJava é amplamente utilizada em estudos acadêmicos e ambientes industriais (SMITH; WILLIAMS, 2007).

A ferramenta MuJava classifica os seus operadores em dois níveis: (i) operadores tradicionais; e (ii) operadores de classe. Os operadores tradicionais realizam alterações substituindo, deletando ou inserindo expressões existentes no código. Os operadores de classe foram projetados para aplicar possíveis mudanças sintáticas relacionadas a orientação a objetos (OO).

MuJava consiste de 3 componentes principais: (i) gerador de mutantes; (ii) visualizador de mutantes; e (iii) executor de mutantes. Os três componentes possuem interface gráfica. A estrutura arquitetural da ferramenta está ilustrada na Figura 3.

Figura 3 – Estrutura arquitetural da MuJava.

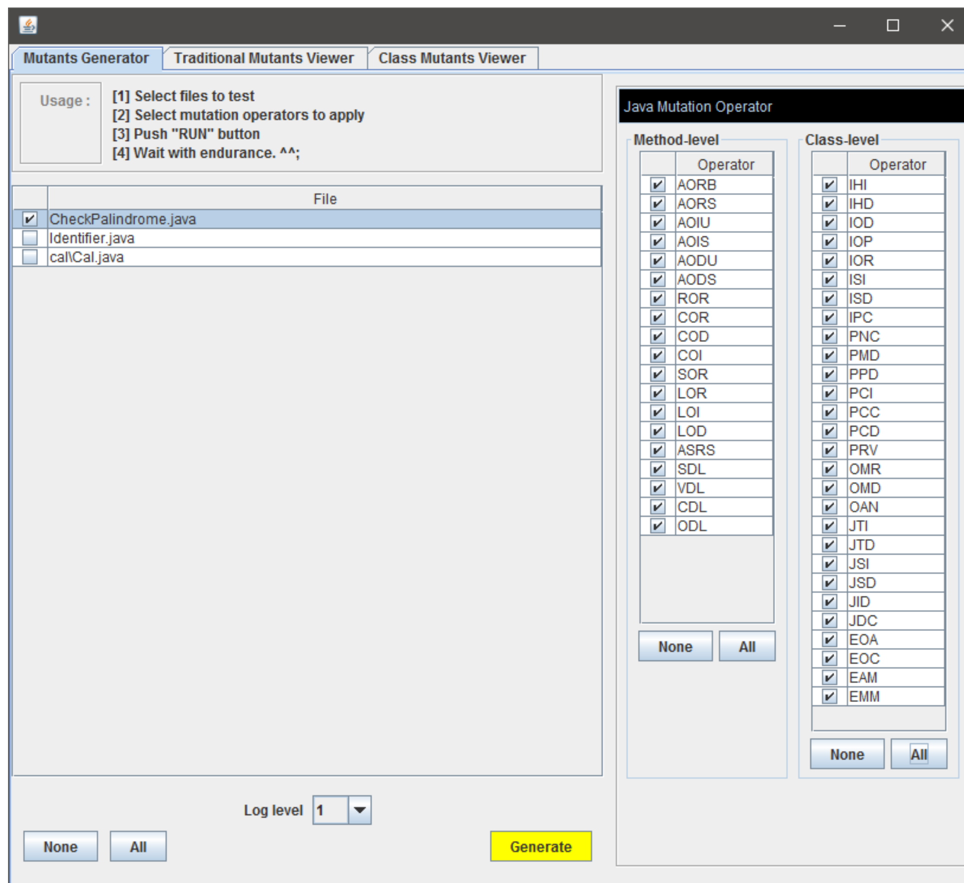


Fonte: Adaptada de Ma, Offutt e Kwon (2006).

O gerador de mutantes cria mutantes em ambos os níveis de operadores disponíveis na ferramenta, mutantes tradicionais e mutantes de classe. A interface do gerador de mutantes pode ser visualizado na Figura 4. O testador pode escolher qual programa irá ser mutado e quais operadores serão aplicados. Os mutantes são gerados após clicar no botão “Generate” (Figura 4). Após a geração dos mutantes, os mesmos poder ser vistos no visualizador de mutantes.

O visualizador de mutantes mostra quantos mutantes foram gerados, separados por nível

Figura 4 – Gerador de mutantes da MuJava.



Fonte: Elaborada pelo autor.

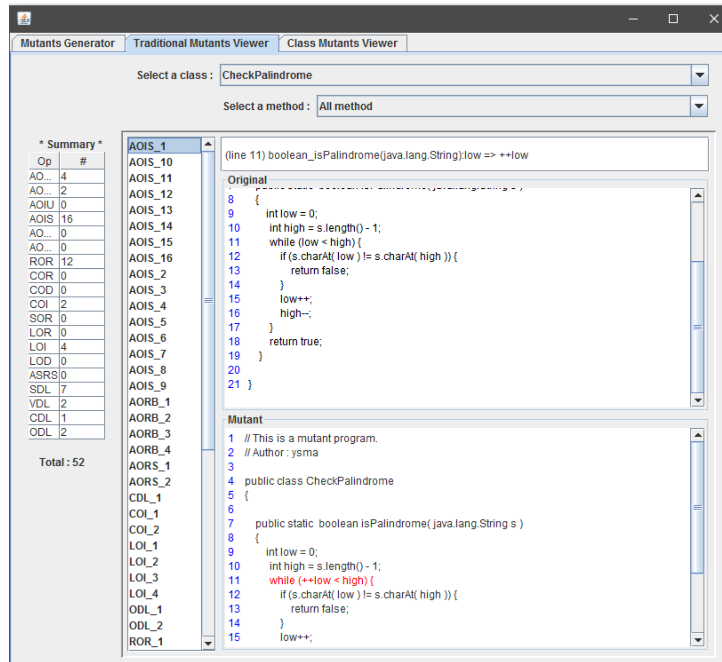
de operador. É possível visualizar também, qual foi a parte do código alterada, ajudando os testadores a identificar os mutantes equivalentes ou gerar novos casos de teste para mutantes considerados difíceis de matar. A Figura 5 mostra a interface do visualizador de mutantes.

O executor de mutantes, executa os mutantes gerados contra um conjunto de casos de teste e mostra o resultado do escore de mutação. Os casos de teste devem estar escritos na forma de classe JUnit.z A interface do executor de mutantes pode ser visualizado na Figura 6.

2.7 Considerações finais

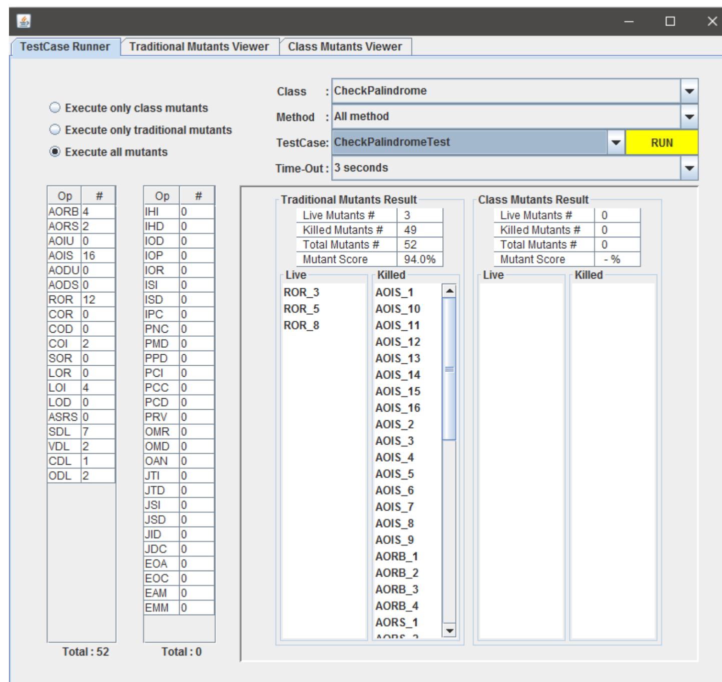
Neste capítulo apresentou-se uma visão geral sobre os principais conceitos que envolvem detalhes e paradigmas sobre os temas teste de software, oráculos de teste, teste de mutação, *framework* JUnit e ferramenta MuJava. No capítulo seguinte, sob a perspectiva de enriquecer as motivações do trabalho a ser realizado relacionado ao problema de oráculo, é apresentada uma revisão bibliográfica considerando o tema avaliação de oráculos de teste. Dados quantitativos e qualitativos extraídos de estudos publicados relatam a necessidade de alternativas sistematizadas de definição dos oráculos de teste e suas respectivas formas de avaliação.

Figura 5 – Visualizador de mutantes da MuJava.



Fonte: Elaborada pelo autor.

Figura 6 – Executor de Mutantes da MuJava.



Fonte: Elaborada pelo autor.

CAPÍTULO

3

AVALIAÇÃO DA QUALIDADE DE ORÁCULOS DE TESTE

3.1 Considerações iniciais

Esta seção descreve estudos diretamente relacionados à realização desse trabalho de mestrado. Em linhas gerais, poucos trabalhos científicos trazem avaliação de oráculos de teste. Na literatura, encontram-se trabalhos, em sua maioria, relacionados à automatização de oráculos de teste. Não obstante, estudos revelam que esta área de pesquisa está sofrendo ascensão desde 2008, aumentando o número de publicações em eventos da área devido a sua relevância (OLIVEIRA; KANEWALA; NARDI, 2014).

3.2 Critérios de avaliação para oráculos

A qualidade dos oráculos de teste pode ser medida levando em conta diversas características, de acordo com a precisão dos resultados, ou seja, qual a porcentagem de resultados verdadeiros foram gerados pelo oráculo. Outro fator é o montante de falsos positivos o oráculo produz, ou quantas falhas são identificadas.

Os critérios de avaliação para oráculos são particulares dentro do contexto de cada trabalho, prejudicando o compartilhamento do conhecimento. Não foram encontradas na literatura formas sistemáticas de selecionar os critérios de avaliação de oráculos de teste.

Os estudos que realizaram algum tipo de avaliação de oráculos de teste estão sumarizados na Tabela 2. Pode-se observar que a maioria dos trabalhos aplicam mutação nos programas em teste. A proposta deste trabalho é aplicar a mutação diretamente nos oráculos de teste. Alguns trabalhos também apresentam formas de automatização de oráculos de teste e fazem a avaliação dentro do contexto específico de cada trabalho. Um dos trabalhos compara vários tipos de

oráculos e classifica o melhor dentro do contexto do mesmo.

Tabela 2 – Estudos relacionados à avaliação de oráculos de teste.

Estudo	Forma de avaliação
Shrestha e Rutherford (2011)	Aplica mutação nos programas em teste
Tan e Edwards (2004)	Aplica mutação nos programas em teste
Cheon (2007)	Aplica mutação nos programas em teste
Coppit e Haddox-Schatz (2005)	Aplica mutação nos programas em teste
Schuler e Zeller (2011)	Aplica mutação nos programas em teste
Kim-Park, Riva e Tuya (2010)	Automatização de oráculos de teste
Shahamiri *et al.* (2011)	Automatização de oráculos de teste
Shahamiri, Kadir e Mohd-Hashim (2009)	Comparação entre diversos tipos de oráculos

O estudo proposto por Shrestha e Rutherford (2011) avalia dois tipos de oráculos de teste por meio de inserção de defeitos nos códigos dos programas utilizados no experimento. Foram utilizadas sete classes principais da linguagem de programação Java, extraídas da versão 1.4.2 do JDK, em inglês *Java Development Kit*, e oito classes de dois programas simples. Para aplicar mutação, foi utilizada a ferramenta MuJava, os mutantes foram gerados utilizando todos os operadores de mutação tradicionais da ferramenta. Os casos de teste foram executados utilizando *null oracle* e *embedded runtime assertions*, previamente apresentados na Seção 2.3.1.

A distribuição da efetividade do *null oracle* com as assertivas JML em relação à captura de defeitos por método, segundo o estudo de Shrestha e Rutherford (2011), mostra que o *null oracle* não é efetivo para localizar defeitos além do limite de 50% em 48,8% do tempo, ou seja, em 50 de um total de 104 métodos analisados, ele consegue capturar entre 0 e 5% dos defeitos. Por outro lado, em 17,31% do tempo (18 instâncias), o *null oracle* consegue identificar entre 95 e 100% dos defeitos, porém esses números são obtidos quando há uma pequena quantidade de mutantes gerados por método. Logo, chega-se a conclusão de que a efetividade do oráculo do tipo *null oracle* é baixa.

Shrestha e Rutherford (2011) apresentam uma mineração de dados com o objetivo de prever situações em que diferentes oráculos são efetivos. Para o *null oracle*, o resultado obtido revela que os casos de teste e a complexidade do método são os indicadores mais importantes da efetividade do oráculo. Já em *embedded runtime assertions*, se o método tiver a quantidade de instruções menor ou igual a dois, terá 100% de eficiência, caso contrário, este oráculo não será efetivo. O artigo apresenta um estudo satisfatório em relação a avaliação de oráculos de teste, mas por outro lado, deixa claro que ainda são necessários muitos estudos posteriores para que haja uma forma sistematizada de implementação e avaliação de oráculos.

Tan e Edwards (2004) apresentam um estudo parecido com o de Shrestha e Rutherford (2011) no qual foi avaliada a efetividade do JUnit e das anotações JML. Assim como em Shrestha e Rutherford (2011), esse estudo também utilizou análise de mutação para realizar os estudos empíricos. Em Tan e Edwards (2004) foram escolhidas seis classes do pacote `java.util` e a

ferramenta de mutação foi a Jester[1].

O estudo apresentado por Shrestha e Rutherford (2011) e o trabalho de Tan e Edwards (2004) analisam a efetividade dos oráculos por meio de assertivas, porém em Tan e Edwards (2004) o escopo das classes é menor e a ferramenta de mutação escolhida não fornece operadores de mutação tradicionais. Para a realização do experimento em Tan e Edwards (2004) foram realizados quatro passos: (i) escolha das classes para realização dos testes; (ii) geração dos mutantes, aplicando mutação diretamente no código das classes utilizando a ferramenta Jester; (iii) geração das anotações JML e das classes JUnit; e (iv) execução das anotações JML e das classes JUnit, armazenando o número de mutantes para os quais os testes falharam.

Utilizando a ferramenta Jester, cada mutante gerado modifica o código fonte da classe correta para ter uma das seguintes mudanças: (i) alterar constantes numéricas, alterando o dígito 0 para 1; 5 para 6; e 9 para 0; (ii) trocar variáveis booleanas, alterando de *true* para *false* e vice-versa; (iii) mudar as condições de instruções *if* para sempre avaliar como *true*, ou *false*; (iv) alterar $++$ para $--$ e vice-versa; e (v) alterar != para == e vice-versa.

Adicionalmente, os defeitos de especificação foram separados em três categorias gerais no estudo de Tan e Edwards (2004): (i) especificação incorreta ao lidar com valores nulos; (ii) falta da especificação do comportamento ao lançar exceções; e (ii) não diferenciar entre identidade de objeto (==) e o método de igualdade.

O número de mutantes gerados variaram entre 13 e 650. Tan e Edwards (2004) afirmam que a eficácia do JML-JUnit não foi eficiente em relação à detecção de defeitos. Todavia, a execução do JML-JUnit nos componentes corretamente implementados revelou vários defeitos nas especificações de quase todas as classes testadas.

Shahamiri *et al.* (2011) propõem um *framework* para automatização de oráculos de teste e também utilizam como forma de avaliação do oráculo o teste de mutação gerando duas versões dos programas utilizados para o estudo de caso: *Golden Version*, sem aplicar mutação e *Mutated Version*, com mutação. Shahamiri *et al.* (2011) ressaltam alguns desafios existentes na automatização dos oráculos, sendo eles: (i) geração do domínio de saída; (ii) mapeamento entre o domínio de entrada e de saída; e (iii) um comparador para decidir a eficácia dos resultados obtidos. No entanto, apesar desses desafios o estudo concluiu que a automatização dos oráculos pode ser alcançada.

Para enfrentar esses desafios levantados em relação a automatização de oráculos de teste, Shahamiri *et al.* (2011) desenvolveram um *framework*. Foi utilizada uma análise de relacionamento para a geração do domínio de saída e vários oráculos gerados com base em redes neurais artificiais resolveram o problema do mapeamento entre o domínio de entrada e de saída. A eficácia dos oráculos foi avaliada por meio do teste de mutação, ou seja, inserção de defeitos nos programas utilizados no experimento. Os resultados mostraram que a precisão dos

[1] http://jester.sourceforge.net/

oráculos ficaram entre 91,17% e 98,26%. Nesse caso, portanto, o teste de mutação não faz parte da solução apresentada pelos autores para melhorar a qualidade dos oráculos, mas foi utilizada como forma de validação da sua abordagem.

Shahamiri, Kadir e Mohd-Hashim (2009) apresentam uma análise comparativa entre seis categorias de oráculos de teste. Os autores ressaltam que todos os métodos apresentados no estudo têm vantagens e desvantagens e, em seguida, apresentam uma análise com o objetivo de comparar as abordagens considerando vários aspectos: o custo do oráculo, a confiabilidade do oráculo e o tipo de teste que pode ser automatizado pelo oráculo. Um dos principais focos do trabalho é ressaltar as atividades realizadas pelos oráculos e quais os desafios na sua automatização. Dentre os tipos de oráculos analisados no experimento, estão as ANNs, do inglês, *Artificial Neural Network*.

Chegou-se à conclusão de que o principal problema desse tipo de oráculo automatizado é não conseguir automatizar a geração das saídas dos testes, com exceção do teste de regressão. No entanto, foi constatado que oráculos de teste baseados em ANN estão chamando atenção de testadores para criação de *frameworks* de automatização completa dos oráculos.

O estudo de Shahamiri, Kadir e Mohd-Hashim (2009) conclui, principalmente, dois aspectos fundamentais a serem considerados sobre o problema de oráculo. Em primeiro lugar, não é possível automatizar completamente todo o processo de oráculo. Em segundo lugar, não existe uma abordagem única de automatização para todas as atividades de oráculos de teste em diferentes circunstâncias.

Cheon (2007), em seu trabalho, lista alguns problemas em utilizar assertivas JML: (i) expõe aos clientes os detalhes de implementação, como estruturas de dados que são irrelevantes para os clientes; (ii) as assertivas tendem a ser longas e complicadas, portanto, difíceis de ler e entender; (iii) as assertivas não são reutilizáveis e são de difícil manutenção, uma vez que estão ligadas a decisões de implementação específicas e uma pequena mudança na implementação pode exigir mudanças ao longo das assertivas.

Diante das constatações dos problemas em utilizar assertivas, Cheon (2007) propôs uma abordagem para avaliar a efetividade e eficiência na escrita de assertivas utilizando *model variable* como oráculos de teste, ou seja, permite ao testador escrever assertivas sem fazer referência direta aos estados concretos do programa. A abordagem *model variable* permite especificar as propriedades do programa de uma forma não apenas abstrata, concisa e independente de detalhes de representação, mas também pode ser verificada em tempo de execução.

Para realizar a avaliação da abordagem proposta por Cheon (2007), foi desenvolvido um experimento utilizando o teste de mutação. Os códigos dos programas do experimento foram mutados utilizando três operadores de mutação: (i) substituir um operador por outro do mesmo tipo; (ii) substituir um operador por outro de tipo diferente; e (iii) substituir uma variável por outra. Foram gerados 10 programas mutantes manualmente e os testes foram gerados utilizando

a ferramenta JET. JET é uma ferramenta de automatização de teste unitário para classes da linguagem Java, e tem a funcionalidade de exportar os casos de teste para uma classe JUnit.

De acordo com o experimento realizado por Cheon (2007), os resultados foram classificados como promissores. Assertivas escritas em termos do *model variable* são tão eficazes quanto as assertivas escritas sem o uso do *model variable* na detecção de defeitos. Espera-se que os estudos realizados por meio do *model variable* facilitem a utilização dos oráculos de teste baseados em assertivas.

Coppit e Haddox-Schatz (2005) investigaram a viabilidade de revelar falhas por meio do aumento de código utilizando assertivas. O objetivo principal foi determinar se oráculos de teste baseados em assertivas são viáveis em revelar defeitos e iniciar uma avaliação de custo e efetividade dos mesmos.

A abordagem de avaliação utilizada em Coppit e Haddox-Schatz (2005) foi: (i) desenvolver uma especificação formal do software, em que foi especificado o comportamento do software em detalhes suficientes para fornecer uma definição satisfatória de correção; (ii) traduzir manualmente a especificação para as assertivas do software; (iii) utilizar injeção de defeitos manual e automatizada para criar um conjunto de programas mutantes; e (iv) executar, para cada mutante, o código de auto-verificação para um conjunto de entradas de teste, para determinar se a falha é revelada pelas assertivas.

Os defeitos foram injetadas no código por meio da ferramenta JavaWrap (HADDOX; KAPFHAMMER, 2002), que possui um subsistema de injeção de defeitos que permite que um usuário especifique quais tipos de dados devem ser corrompidos, bem como posições no código onde o defeito deve ocorrer. Além disso, o usuário pode escolher entre vários tipos de defeitos para injetar no código.

Para avaliar a eficácia das assertivas baseadas em especificações, foi gerado um mutante do sistema para cada defeito. Os casos de teste foram executados e verificado se alguma assertiva foi violada. Caso as assertivas estejam corretas, então qualquer entrada de teste que cause uma falha para afetar negativamente o estado do programa, deve fazer com que ocorra falha na assertiva.

A abordagem utilizada no estudo de Coppit e Haddox-Schatz (2005) foi de injetar defeitos na implementação, depois executar o código para um número de casos de teste. Se as assertivas forem suficientes para revelar os defeitos a qualquer momento, elas não modificam o estado do programa em análise, então elas servem como uma alternativa para separar a implementação do oráculo.

Coppit e Haddox-Schatz (2005) fizeram a avaliação da abordagem proposta baseada em dois estudos de caso sobre sistemas industriais, com foco na viabilidade, custos e benefícios. Também foram feitas análise a respeito do processo de conversão de especificações em assertivas. Os resultados mostraram que as assertivas são efetivas em revelar defeitos. Em relação ao custo,

o uso de assertivas aumenta o custo dos oráculos, pois não são triviais de implementar, assim como a especificação formal aumenta a dificuldade do processo.

Kim-Park, Riva e Tuya (2010) definiram um oráculo de teste automatizado para programas de processamento XML (do inglês, *eXtensible Markup Language*) que opera com níveis diferenciados de especificação. A automatização do oráculo é alcançada ao transformar esses níveis de especificação no código do programa, e a implementação oráculo desenvolvida é avaliada por meio de um estudo experimental que revelou resultados promissores.

O teste de programas de processamento XML é uma tarefa desafiadora, uma vez que os dados de entrada e saída envolvidos nas execuções de teste podem ser complexos e de grande volume, o que torna difícil determinar a corretude dos resultados de execução.

Para o cenário de teste, Kim-Park, Riva e Tuya (2010) consideraram o processamento dos programas XML, incluindo a execução do oráculo para apoiar o julgamento das execuções dos testes. O oráculo desenvolvido nesse estudo nunca emite falsos negativos. A eficácia do oráculo foi medida em função dos falsos negativos que ele emite quando ele é executado em um conjunto de resultados dos casos de teste com falhas.

Os casos de teste no trabalho de Kim-Park, Riva e Tuya (2010) foram gerados pela ferramenta *XML Query Use Cases*[2], que é um conjunto de 12 casos de teste, cada um incluindo uma entrada de teste e uma saída esperada. O processo para obter os casos de teste do oráculo consistiu nas seguintes etapas: (i) definir um modelo de um caso de teste do oráculo para cada programa do *XML Use Case*; (ii) gerar um conjunto de saídas de teste incorretas para cada programa; e (iii) instanciar os modelos dos casos de teste do oráculo em cada uma dessas saídas de teste incorretas para criar casos de teste para oráculo. Como resultado, Kim-Park, Riva e Tuya (2010) obtiveram casos de teste para oráculo com requisitos de entrada e de comportamento exclusivos para cada programa do *XML Use Cases*, mas diferenciados por suas saídas incorretas.

Durante o processo de avaliação em Kim-Park, Riva e Tuya (2010), não foi considerado um critério de seleção de dados de entrada dos testes, prejudicando a qualidade do oráculo, uma vez que a maioria das informações derivadas da seleção de entrada de teste pode ser útil para categorizar as saídas de teste esperadas. Os resultados experimentais mostraram que o oráculo é altamente eficaz na detecção de falhas, com resultados promissores de mais de 90% de eficácia.

Em Schuler e Zeller (2011) é apresentada uma forma alternativa de custo e eficiência para avaliar a qualidade de oráculos de teste. Usando *dynamic slicing*, técnica que determina o conjunto de comandos que potencialmente influenciam as variáveis utilizadas em um determinado local, para determinar a *checked coverage*, ou seja a cobertura é baseada nas características do código que realmente contribuem para os resultados verificados por oráculos de teste.

Schuler e Zeller (2011) alegam que utilizar *checked coverage* tem as seguintes vantagens:

[2] http://www.w3.org/TR/xmlquery-use-cases/

- poucos ou insuficientes oráculos resultam imediatamente em uma baixa *checked coverage*, proporcionando uma avaliação mais realista da qualidade do teste;
- declarações que são executadas, mas cujos resultados nunca são verificados seriam consideradas *uncovered*; e
- ao invés de se concentrar na execução de código, os desenvolvedores podem verificar uma maior quantidade de resultados, capturando mais defeitos no processo;

Para a avaliação do estudo proposto por Schuler e Zeller (2011), apenas os conjuntos de programas que ajudavam nos casos de teste para a *checked coverage* foram utilizados. Sete programas reais *open source* que tinham vários anos de desenvolvimento e com um conjunto de classes JUnit foram utilizados para computar a *checked coverage*.

Durante o experimento realizado em Schuler e Zeller (2011), para cada um dos sete programas do estudo foram armazenados os valores de *checked coverage*, e escore de mutação para um conjunto de casos de testes com todas as assertivas habilitadas, ou seja, 0% das assertivas removidas do código, com 25%, 50%, 75% e 100% das assertivas removidas, respectivamente. O estudo conclui que a *checked coverage* é mais sensível quando as assertivas são removidas, em relação à remoção de comandos do código, ou seja, ao aplicar-se a mutação tradicional.

3.3 Visão geral dos estudos primários

A busca por estudos que realizam a avaliação de oráculos de teste foi realizada de maneira sistemática, em que o método de busca utilizado foi definido por meio da definição de uma string de busca aplicada em bibliotecas digitais. As bases de busca utilizadas foram: (i) ACM Digital Library[3]; (ii) IEEExplore[4]; (iii) ScienceDirect[5]; e (iv) Scopus[6].

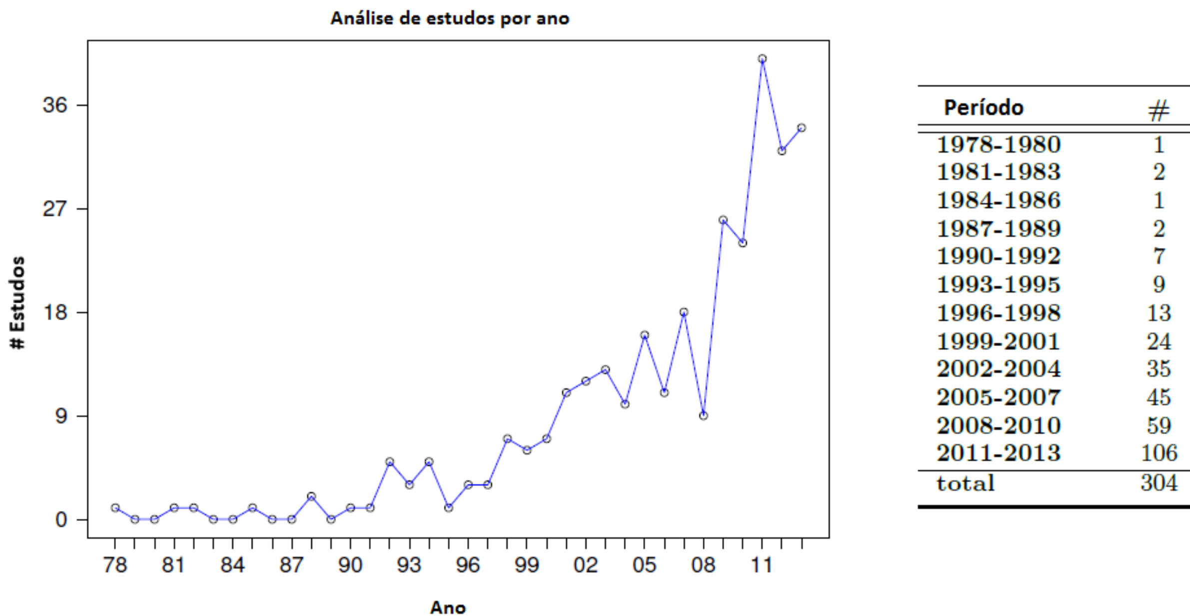
Com os artigos obtidos na busca nas bibliotecas digitais citadas, os estudos primários foram analisados baseando-se no resumo e introdução dos artigos inicialmente levantados em busca de estudos que apresentassem avaliação de oráculos de teste, criando assim uma segunda lista de artigos, contendo apenas os artigos relevantes para o tema. Em seguida os artigos selecionados foram lidos e analisados buscando formas de avaliação da qualidade de oráculos de teste. No entanto, foram poucos estudos que apresentaram métodos de avaliação de oráculos de teste, fazendo com que a busca fosse estendida a análises de repositórios separadamente.

Desde 2008 é possível observar que houve um crescimento expressivo na área de pesquisa relacionada a oráculos de teste (OLIVEIRA; KANEWALA; NARDI, 2014)(Figura 7). Não obstante, os oráculos são desenvolvidos de forma *ad hoc*, ainda não há forma sistemática

[3] http://dl.acm.org/
[4] http://ieeexplore.ieee.org/
[5] http://www.sciencedirect.com/
[6] http://www.scopus.com/

de implementá-los. De modo consequente, estratégias de avaliação de oráculos de teste para domínios específicos são pouco exploradas.

Figura 7 – Análise de estudos publicados sobre oráculos de teste.



Período	#
1978-1980	1
1981-1983	2
1984-1986	1
1987-1989	2
1990-1992	7
1993-1995	9
1996-1998	13
1999-2001	24
2002-2004	35
2005-2007	45
2008-2010	59
2011-2013	106
total	304

Fonte: Adaptada de Oliveira, Kanewala e Nardi (2014).

Diante das buscas de trabalhos relacionados, chegou-se à conclusão de que além de não existirem relatos na literatura de forma sistematizada de implementação de oráculos, também não há forma sistematizada de avaliá-los.

Todos os trabalhos descritos na Seção 3.2 apresentam alguma forma de avaliação da qualidade de oráculos de teste ou automatização de oráculos de teste. Entretanto, não é possível identificar nenhuma forma sistematizada de realizar a avaliação da qualidade dos oráculos, sendo que cada trabalho realiza a avaliação da qualidade dentro do escopo do estudo e suas necessidades específicas. Dentro do escopo do trabalho realizado durante o mestrado, foram propostos operadores de mutação para oráculos de teste como forma de avaliação sistemática da qualidade dos oráculos de teste no formato do *framework* JUnit.

Em outras palavras, os trabalhos mencionados neste capítulo abordam aspectos relevantes na criação de oráculos de teste como, por exemplo, custo e eficácia. Então, para avaliar a abordagem proposta, os autores definem diferentes formas de validação, que em muitos casos, envolve a utilização de programas mutantes. Diferente do que se propõe nesta dissertação, os trabalhos não buscam estabelecer um método de "teste" para oráculos que possa ser utilizado pelos desenvolvedores de software. Trabalhos nessa linha não foram identificados na literatura.

3.4 Considerações finais

O presente capítulo apresentou os principais trabalhos relacionados à avaliação de oráculos de teste. Foi possível observar que ainda existe uma grande lacuna nessa área de pesquisa, pois os oráculos ainda não possuem uma forma sistemática de implementação e avaliação. É importante ressaltar que não foram encontrados trabalhos similares ao realizado neste trabalho de mestrado, em que a mutação é aplicada diretamente no oráculo de teste, fornecendo uma forma sistemática de avaliar a qualidade de oráculos de teste baseados em assertivas.

No próximo capítulo a ferramenta "MuJava 4 JUnit" será descrita detalhadamente. A ferramenta utilizou a estrutura da ferramenta MuJava, adicionando os novos operadores de mutação para oráculos de teste escritos no formato do *framework* JUnit.

CAPÍTULO

4

MUJAVA 4 JUNIT

4.1 Considerações iniciais

Este capítulo apresenta um novo conjunto de operadores de mutação que foram propostos especificamente para oráculos de teste escritos no formato de classes JUnit e detalhes de implementação da ferramenta de mutação, "MuJava 4 JUnit".

4.2 Operadores de mutação para oráculos de teste baseados em assertivas

Foram propostos operadores de mutação para introduzir pequenas modificações nas assertivas das classes de teste JUnit. O fundamento por trás desta abordagem é que pretende-se, com modificações sintáticas, alterar-se o comportamento do oráculo de teste apenas e não as entradas fornecidas nos casos de teste. Por isso, apenas parte de um método JUnit (um caso de teste) é endereçado pelos operadores de mutação. No caso, a parte relativa ao oráculo concretiza-se por meio das assertivas e alguns tipos de anotações, que definem o resultado esperado da execução.

Por tratar-se de um novo contexto de utilização de mutantes, não existem referências que pudessem guiar com segurança o tipo de mutação mais apropriada. Assim, as alterações que caracterizam os operadores de mutação baseiam-se principalmente em experiências anteriores, com outras linguagens de programação e outros contextos de aplicação (DELAMARO *et al.*, 1993; MA; OFFUTT; KWON, 2005; CHOI *et al.*, 1989; DELAMARO; OFFUTT; AMMANN, 2014). Podem, portanto, não ser as mais adequadas.

Foram definidas quatro classes de operadores:

- Adição: parâmetros são adicionados ao método `assert` ou à anotação;

- Remoção: parâmetros são removidos do método `assert` ou da anotação;
- Alteração: parâmetros do método `assert` ou da anotação são alterados; e
- Substituição: o método `assert` é substituído por outro método `assert`.

Adicionalmente às quatro classes, os operadores propostos podem ser classificados em dois níveis:

- Nível de assinatura: as alterações são feitas no tipo do método `assert` em execução, ou nos parâmetros recebidos pelo método `assert`; e
- Nível de anotação: as alterações são aplicadas substituindo as anotações, removendo, ou alterando seus parâmetros.

4.2.1 Operadores nível de assinatura

Os operadores de mutação no nível de assinatura foram definidos combinando as assinaturas do método `assert`, adicionando ou removendo parâmetros, ou substituindo um método `assert` por outro método `assert`.

No Código-fonte 4 uma classe `MyMath` está sendo testada por meio do método de teste `testSquareRoot001`. O caso de teste apresentado verifica se o resultado de $\sqrt{9} = 3$, com um possível erro de 10^{-8}, chamando o método `MyMath.SquareRoot`. O método `assertEquals` faz o papel do procedimento de oráculo, especificando que o resultado obtido deve ser igual a 3.0. O mutante, apresentado no Código-fonte 5 remove o terceiro parâmetro do método, passando a desconsiderar o possível erro. As duas implementações podem ter resultados iguais ou diferentes dependendo do valor do terceiro parâmetro. Caso a diferença entre os oráculos nunca seja revelada, isso indica a fragilidade do oráculo de teste. Se o mutante sempre se comporta como o oráculo original, isso indica que o terceiro parâmetro não é necessário, ou que não foi utilizado um caso de teste que mostre que ele é necessário, ou seja, um caso de teste que mostre que o resultado fornecido pode apresentar um pequeno erro.

Código-fonte 4 – Exemplo do uso do operador de nível de assinatura (original).

```
1: @Test
2: void testSquareRoot001(){
3:   double x = 9.0;
4:   MyMath m = new MyMath();
5:   x = m.squareRoot(x);
6:   assertEquals(3.0, x, 0.00000001);
7: }
```

Código-fonte 5 – Exemplo do uso do operador de nível de assinatura (mutante).

```
1: @Test
2: void testSquareRoot001(){
3:   double x = 9.0;
4:   MyMath m = new MyMath();
5:   x = m.squareRoot(x);
6:   assertEquals(3.0, x);
7: }
```

Todas as assertivas pertencem à classe `Assert`. Esta classe fornece um conjunto de métodos `assert` úteis para descrever os resultados esperados dos casos de teste. Os métodos da classe `Assert` utilizados nesse trabalho estão sumarizadas na Tabela 3.

Tabela 3 – Métodos da classe `Assert` utilizados na pesquisa.

Método	**Descrição**
`assertEquals`	Verifica se dois tipos primitivos/Object são iguais
`assertTrue`	Verifica se a condição é verdadeira
`assertFalse`	Verifica se a condição é false

Os operadores do nível de assinatura são descritos na Tabela 4. Esses operadores foram desenvolvidos de acordo com a especificação do JUnit e podem simular problemas, cometidos pelo testador, ao codificar os oráculos de teste.

Tabela 4 – Tabela de operadores (Nível de assinatura).

Nível de Assinatura			
#	Classe	Descrição	Sigla
1	Adição	Adicionar valor constante	ATV
2	Alteração	Decrementar valor constante	DCfTV
3	Alteração	Incrementar valor constante	ICtTV
4	Substituição	Substituir assertiva booleana	RBA
5	Remoção	Remover valor constante	RTV

4.2.2 Operadores nível de anotação

As anotações foram introduzidas a partir da versão 4 do JUnit. Anotações são como meta-tags em que códigos podem ser adicionados e aplicados em métodos ou classes. As anotações no JUnit levam informações sobre os métodos: quais métodos são executados antes ou depois dos métodos de teste, quais métodos são executados antes ou depois de todos os casos de teste, e quais métodos são ignorados durante a execução. Algumas anotações podem, também, definir o comportamento do caso de teste, em termos do resultado esperado. As anotações utilizadas nos operadores desenvolvidos nessa pesquisa são apresentadas na Tabela 5.

Tabela 5 – Anotações do JUnit exploradas para a criação de oráculos mutantes.

Anotação	**Descrição**
@Test	Indica ao JUnit para executar o método como um caso de teste.
@Test(expected=Exception.class)	Declara que um método de teste deve lançar uma exceção.
@Test(timeout=100)	Faz com que um teste falhe se ele demorar mais do que o tempo de *clock* especificado.

Diante disso, os operadores no nível de anotação foram desenvolvidos com base em três alterações distintas: (1) modificando ou removendo o *timeout* na execução do oráculo; e (2) adicionando possíveis exceções que podem ocorrer na execução dos oráculos e que não foram previamente pensadas pelo testador.

Se o testador deseja determinar um tempo máximo para a execução do caso de teste, deve-se utilizar o *timeout* na anotação. A execução falha se o método levar mais tempo do que o estipulado no caso de teste. Se um caso de teste espera a ocorrência de alguma exceção, o parâmetro *"expected"* é utilizado. A execução irá falhar caso o método não lance a exceção. Os operadores do nível de anotação são sumarizados na Tabela 6.

Tabela 6 – Tabela de operadores (Nível de anotação).

Nível de Anotação			
#	Classe	Descrição	Sigla
1	Adição	Adicionar classe esperada	AEC
2	Alteração	Decrementar constante no *timeout*	DCfT
3	Alteração	Incrementar constante no *timeout*	ICtT
4	Remoção	Remover constante *timeout*	RTA

Os Códigos-fonte 6 e 7 mostram um exemplo em que o operador de mutação AEC adiciona uma exceção esperada na execução do oráculo. No primeiro caso, o oráculo está realizando uma divisão por zero, configurando uma operação não permitida que deve gerar o lançamento de uma exceção. Adicionando a exceção aritmética, a execução do oráculo ocorre normalmente e o resultado da execução será verdadeira.

Código-fonte 6 – Exemplo do uso do operador de nível de anotação (original).

```
1: @Test
2: public void testDiv(){
3:   assertEquals(c1.Divide(5,0),0,0);
4: }
```

Código-fonte 7 – Exemplo do uso do operador de nível de anotação (mutante).

```
1: @Test(expected=ArithmeticException.class)
2: public void testDiv(){
3:   assertEquals(c1.Divide(5,0),0,0);
```

```
4: }
```

4.3 Discussão individual sobre os operadores de mutação

Em seguida, é apresentada uma análise de cada um dos operadores apresentados. Procura-se definir com precisão as alterações processadas por cada o operador e identificar o comportamento dos mutantes gerados, bem como sua utilização na avaliação dos oráculos.

4.3.1 ATV (Adicionar valor constante)

O método `assertEquals` é utilizado para verificar se o valor produzido pelo programa em teste é igual ao valor esperado. A assinatura desse método é:

```
assertEquals(expected, actual)
```

e o caso de teste falha se os valores `expected` e `actual` não forem iguais. O primeiro parâmetro é o valor esperado e o segundo o valor produzido pelo programa em teste.

Existe também a versão com três parâmetros:

```
assertEquals(expected, actual, delta)
```

O propósito do parâmetro *delta* é determinar o valor limiar máximo da diferença entre os números *expected* e *actual* para que eles sejam considerados similares e aceitáveis em um dado teste. O valor é definido pela seguinte desigualdade:

$$actual - delta \leq expected \leq actual + delta$$

Existem casos em que pequenas diferenças numéricas podem ser desconsideradas, em que casas decimais não são relevantes para o resultado final. No entanto, determinar quantas casas decimais importam é essencial no contexto de sistemas financeiros. Por exemplo, um sistema de mercado pode tolerar falhas de cálculo a partir da terceira casa decimal, pois no real a precisão é dada em centavos. Entretanto, sistemas de postos de gasolina precisam de maior precisão, usando 3 ou 4 casas dependendo do cálculo. Em geral, para sistemas financeiros, o valor de precisão (delta) deve ser especificado pois é essencial nas regras de negócio. Não obstante, não é recomendável depender do valor de precisão.

O operador ATV é um operador de nível de assinatura e classe de adição. Adiciona o parâmetro delta. Com isso, tem-se uma versão mutada do oráculo original, no qual o resultado é aceito como correto, considerando um certo erro de precisão. Porém, como mencionado acima, nem sempre é fácil saber qual o valor aceitável para uma determinada aplicação. Atualmente, apenas o valor 0.001 é utilizado como delta, mas outros valores poderiam ser considerados, levando-

se em conta o próprio valor esperado. Por exemplo: $expected/2$, $expected/10$, $expected/100$, $expected/1000$, etc.

O Código-fonte 8 realiza o cálculo de uma função do segundo grau por meio da fórmula de Bhaskara em que os coeficientes são 1, 2 e 1. Dependendo do valor dos coeficientes, as raízes podem gerar valores com várias casas decimais, portanto, é importante adicionar o delta. O mutante encontra-se na linha 6 do Código-fonte 8. As implementações podem ter resultados iguais ou diferentes dependendo do valor do delta e dos coeficientes em questão. Caso a diferença entre os oráculos nunca seja revelada, isso pode indicar que a fragilidade está no caso de teste ou o erro pode estar diretamente no programa que está sendo executado pelo oráculo.

Código-fonte 8 – Exemplo do uso do operador ATV.

```
1: @Test
2: public void testBhaskara() {
3:      Bhaskara B = new Bhaskara();
4:      double raiz = B.raiz(1,2,1);
5:      assertEquals(-1.0, raiz);
6: //-> assertEquals(-1.0, raiz , 0.001);
7: }
```

4.3.2 DCfTV (Decrementar valor constante)

O operador DCfTV é um operador de nível de assinatura e classe de alteração. Decrementa o parâmetro do delta, terceiro parâmetro do método `assertEquals(expected, actual, delta)`. O oráculo mutante é morto dependendo do valor decrementado e do valor esperado. Se o oráculo foi projetado com um caso que, alterando-se o valor de precisão altera-se o seu resultado, aplicando este operador, o oráculo mutante passará a ter o resultado diferente do oráculo original.

O Código-fonte 9 utiliza, na linha 8, o método `assertEquals(expected, actual, delta)`, e está sendo testado o cálculo de uma taxa de 10% sobre o valor de determinado produto. O operador DCfTV permite ao testador ajustar o valor do delta de acordo com as suas necessidades. Na linha 9, vê-se o mutante que seria gerado, decrementando-se o valor de delta.

Código-fonte 9 – Exemplo operador DCfTV.

```
1:     @Test
2:     public void impostoSobreValor() {
3:         Produto produto = new Produto("Televisor", 600, Produto.
       ELETRONICO);
4:         CalculaTaxas calculadorDeTaxas = new CalculaTaxas();
5:
6:         double taxa = calculadorDeTaxas.getImposto(produto);
```

```
 7:          double precoFinal = produto.getPreco() * (1 + taxa);
 8:          assertEquals(660, precoFinal, 0.001);
 9: // -->   assertEquals(660, precoFinal, 0.0001);
10:      }
```

O comportamento do oráculo mutante difere do original à medida que o primeiro exige um erro menor que o segundo. Para diferenciar os dois, o testador precisa analisar seus casos de teste e seu oráculo e verificar o limite de erro estabelecido. No exemplo, o testador deveria prover um caso de teste que tivesse um erro inferior ao erro inicial, mas "próximo" deste, ou seja: $0.0001 < erro \leq 0.001$. Para um caso assim, o oráculo original indica que o teste passa mas o oráculo mutante indica uma falha. Assim, o mutante auxilia o testador a verificar seu oráculo ou planejar novos casos de teste que exercitem seu oráculo.

Da mesma forma que no caso do operador ATV, é difícil "acertar" o quanto decrementar do valor de delta. Assim, pode-se pensar em estender o operador DCfTV utilizando-se valores como $erro/2$, $erro/10$, $erro/100$, $erro/1000$, etc.

4.3.3 ICfTV (Incrementar valor constante)

O operador ICfTV é um operador de nível de assinatura e classe de alteração. Incrementa o parâmetro do valor de delta, terceiro parâmetro do método `assertEquals(expected, actual, delta)`. O mutante é morto dependendo do valor incrementado. Se o oráculo foi projetado com um caso que, alterando-se o valor de precisão altera-se o seu resultado, aplicando este operador, o oráculo mutante passará a ter o resultado diferente do oráculo original.

No Código-fonte 10 o oráculo está na linha 4, onde o método `assertEquals` verifica o resultado de uma multiplicação com o valor de delta 0.001.

Código-fonte 10 – Exemplo do uso do operador ICfTV.

```
1:      @Test
2:      public void assertSoma(){
3:          Calculadora c = new Calculadora();
4:          assertEquals(4.0, c.multiplicacao(2, 2), 0.001);
5: // -->   assertEquals(4.0, c.multiplicacao(2, 2), 0.01);
6:      }
```

O oráculo mutante pode resultar em um comportamento diferente do oráculo original, ao passo que o oráculo mutante considera um erro maior que o original. Ao contrário do que acontece com o operador DCfTV, o oráculo mutante neste caso é mais permissivo que o oráculo original. Então, todo caso de teste que passe pelo oráculo original vai também passar pelo mutante. Então, para que o mutante seja morto deve-se ter um caso de teste que falhe com o oráculo original e que passe com o mutante. No exemplo apresentado no Código-fonte 10, o

testador deveria projetar um caso de teste com um valor de erro maior que o delta inicial, porém que seja próximo do valor mutado, ou seja: $0.0001 > erro \geq 0.001$. Portanto, a implementação do oráculo mutante auxilia o testador em gerar novos casos de teste.

Da mesma forma que no caso dos operadores ATV e DCfTV, é difícil “acertar” o quanto incrementar do valor de delta. Assim, pode-se pensar em estender o operador ICfTV utilizando-se valores como $erro/2$, $erro/10$, $erro/100$, $erro/1000$, etc.

4.3.4 RBA (Substituir assertiva booleana)

O operador RBA é um operador de nível de assinatura e classe de alteração. Substitui as assertivas booleanas (`assertTrue, assertFalse`), sendo útil para revelar defeitos em oráculos projetados para casos booleanos. Com a substituição das assertivas o oráculo mutante pode melhorar a qualidade do oráculo original revelando erros no programa em teste, ou no caso de teste executado pelo oráculo de teste.

O `assertTrue` falhará se o parâmetro for avaliado como *false*. Em outras palavras, essa assertiva garante que o valor seja *true*. O `assertFalse` faz o contrário. Os métodos `assertTrue` e `assertFalse` são baseados em uma única condição booleana.

O operador RBA substitui o método `assertTrue` pelo método `assertFalse` e vice-versa. Os mutantes gerados por esse operador podem indicar defeitos no programa que está sendo testado caso os oráculos mutantes continuem com o mesmo resultado que o oráculo original.

No Código-fonte 11, o oráculo apresentado na linha 4 com o método `assertTrue` verifica se a String “Ame a Ema” é um palíndromo, aplicando o operador RBA, o método `assertFalse` passará a ser executado (Linha 5). Caso o resultado do oráculo mutante seja diferente do oráculo original, o mutante será considerado morto. Se os oráculos mutante e original apresentarem o mesmo resultado, o testador deverá verificar o caso de teste e/ou o programa em teste.

Código-fonte 11 – Exemplo do uso do operador RBA.

```
1: @Test
2: public void testPalindrome3(){
3:     CheckPalindrome cp = new CheckPalindrome();
4:     assertTrue(cp.isPalindrome("Ame a Ema"));
5: //-> assertFalse(cp.isPalindrome("Ame a Ema"));
6: }
```

4.3.5 RTV (Remover valor constante)

RTV é um operador de nível de assinatura e classe de remoção. O operador RTV remove o valor de delta. O oráculo mutante é morto dependendo do oráculo em teste. Se o oráculo foi

projetado para um caso de teste em que removendo o valor de precisão ele tenha o seu resultado alterado, aplicando este operador, o oráculo mutante passará a ter o resultado diferente do oráculo original.

O Código-fonte 12 apresenta a média aritmética entre dois números. O oráculo, na linha 5, tem o valor de delta definido em 0,001. Aplicando o operador RTV, o valor de delta é removido, gerando duas implementações: com o valor de delta (original) e sem o valor de delta (mutante). As duas implementações (Linha 5 e Linha 6), podem ter resultados iguais ou diferentes dependendo do valor incrementado, que é definido pelo testador. Neste caso, podem-se ter duas implementações corretas, em que caberá ao testador fazer a análise da corretude do oráculo mutante.

Código-fonte 12 – Exemplo do uso do operador RTV.

```
1: @Test
2: public void testAverage() {
3:        int x=10;
4:        int y=7;
5:        assertEquals(8.5, calcAverage(10,7), 0.001);
6: //-> assertEquals(8.5, calcAverage(10,7));
7: }
```

Não é recomendável que um oráculo seja projetado dependendo do valor de delta. Portanto, caso a remoção deste valor altere o resultado do oráculo, pode sugerir ao testador projetar novos casos de teste que não dependam desse valor de erro. Na prática, este operador corresponde a trocar o valor de delta por zero.

4.3.6 AEC (Adicionar classe esperada)

AEC é um operador de nível de anotação e classe de adição. O operador AEC adiciona uma classe esperada na anotação `@Test`. O mutante é morto dependendo da exceção executada e do oráculo em execução.

Uma das mais importantes anotações para testes unitários é a `@Test(expected=...)`. Nesta anotação pode-se verificar se o método está retornando uma exceção pré determinada. Essa funcionalidade é útil para verificar se o sistema está realmente tratando exceções e/ou validando erros internos.

O operador AEC ajuda o testador a tratar as exceções que podem ocorrer durante a execução do oráculo. No Código-fonte 13, por exemplo, a exceção `NullPointerException` evita que algum mês que não exista seja chamado no método `getAllDays`.

Código-fonte 13 – Exemplo operador AEC.

```
1: @Test(expected=NullPointerException.class)
2: public void testException() {
3:     int month = 4;
4:     Assert.assertNotNull(calendar.getAllDays(month));
5: }
```

O operador AEC pode adicionar as exceções `IOException`, `NullPointerException`, `IllegalArgumentException`, `ClassNotFoundException`, `ArrayIndexOutOfBoundsException`, `ArithmeticException` e `Exception`. As exceções e suas funções são:

- `IOException`: indica que uma exceção de I/O ocorreu. Essa classe é a classe geral de exceções produzidas por operações de I/O com falhas ou interrompidas.
- `NullPointerException`: indica uso ilegal do objeto *null*.
- `IllegalArgumentException`: indica que um método passou um argumento ilegal ou impróprio.
- `ClassNotFoundException`: indica que foi realizada uma chamada para a classe, mas a classe não existe ou não foi encontrada.
- `ArrayIndexOutOfBoundsException`: indica que um *array* foi acessado com um índice negativo ou maior ou igual ao tamanho do *array*.
- `ArithmeticException`: indica que uma exceção aritmética ocorreu.
- `Exception`: como essa é uma superclasse de todas as exceções, ao adicionar-se essa classe na anotação, permite que o oráculo trate qualquer tipo de exceção, fazendo com que o operador de mutação seja mais abrangente e menos específico.

O testador deve adicionar a exceção de acordo com a operação que está sendo realizada, assim como realizado no Código-fonte 13, em que é possível evitar a chamada de um valor nulo. As funções de cada exceção descrita acima auxiliam o testador na hora de definir qual mutante será útil para o seu oráculo de teste.

4.3.7 DCfT (Decrementar constante no timeout)

DCfT é um operador de nível de anotação e classe de alteração. O operador DCfT decrementa um valor constante do *timeout* pré estabelecido, mata o mutante dependendo do valor decrementado e do valor do *timeout*. Se o oráculo depende do valor de *timeout* previamente estabelecido, aplicando este operador, o oráculo mutante passará a ter o resultado diferente do oráculo original.

O *timeout* é útil para capturar e encerrar *loops* infinitos, mas não deve ser considerado determinístico. Por exemplo, o teste apresentado no Código-fonte 14 pode, ou não, falhar dependendo de como o sistema operacional cronometra *threads*.

Código-fonte 14 – Exemplo de uso do *timeout*.

```
1: @Test (timeout=100)
2: public void sleep100(){
3:     Thread.sleep(100);
4: }
```

O Código-fonte 15 está definido com um valor de *timeout* em 10 segundos. O operador DCfT, pode reduzir esse valor, dependendo da quantidade de registros que estão cadastrados no banco de dados. Reduzir esse *timeout* é uma boa solução, pois diminui o tempo de espera pelo resultado.

Código-fonte 15 – Exemplo operador DCfT.

```
1: \\ -> @Test(timeout=1000)
2: @Test(timeout=10000)
3: public void pesquisarFuncionario(){
4:     Funcionario funci = funcionarioDAO.buscarFuncionario("12345");
5:     Assert.assertEquals("JOHN", funci.getNome());
6:     verify(transacao, atMostOnce()).executar("12345");
7: }
```

O oráculo mutante (Código-fonte 15, Linha 1) pode resultar em um comportamento diferente do oráculo original (Código-fonte 15, Linha 2), visto que o oráculo mutante considera um valor de *timeout* menor do que o oráculo original. No exemplo apresentado no Código-fonte 15, o testador deve definir um caso de teste em que o tempo de execução seja inferior ao *timeout* original, porém, um valor próximo acima do *timeout* do mutante, ou seja, $1000 < tempodeexecucao \leq 10000$. Se o oráculo original e o mutante apresentam resultado positivo, ou seja, que o caso de teste passou, isso significa que o *timeout* original pode estar superdimensionado, ou ser desnecessário.

O testador deve tomar a decisão de quanto deve ser diminuído para que haja melhoria na execução do oráculo de teste. No entanto, decidir quanto é necessário diminuir deste valor de *timeout* não é uma decisão fácil, pois é necessário analisar quanto tempo leva o processamento do método que está sendo testado. Uma solução é utilizar alguns valores pré-definidos: $timeout - 10$, $timeout - 100$, $timeout - 1000$, $timeout/2$, $timeout/10$, etc.

4.3.8 ICtT (Incrementar constante no timeout)

ICtT é um operador de nível de anotação e classe de alteração. O operador ICtT incrementa um valor constante do *timeout* pré estabelecido, mata o mutante dependendo do valor incrementado e do valor do *timeout*. Se o oráculo depende do valor de *timeout* previamente estabelecido, aplicando este operador, o oráculo mutante passará a ter o resultado diferente do oráculo original.

O Código-fonte 16 realiza um teste de uma conexão no banco de dados, com o *timeout* de 1000 milissegundos. O operador ICtT incrementa o valor do *timeout*.

O oráculo mutante (linha 1) gerado pelo operador ICtT pode causar um resultado diferente do oráculo original, fazendo com que o valor de *timeout* seja suficiente, ou o oráculo mutante pode continuar vivo, dando o mesmo resultado do oráculo original, mostrando que o problema pode não estar no oráculo de teste, e sim no programa que realiza a conexão do banco de dados. Neste caso, cabe ao testador verificar o programa e identificar o erro.

O oráculo mutante é mais tolerante do que o oráculo original. Então, todo caso de teste que passar com o oráculo original deve também passar com o mutante. O mutante torna-se útil se o caso de teste falhar com o oráculo original mas passar com o mutante. Nesse caso, o testador deve verificar a fundamentação que levou a definição do *timeout* original para que o oráculo possa ser corrigido.

Código-fonte 16 – Exemplo operador ICfT.

```
1: \\ -> @Test(timeout=10000)
2: @Test(timeout=1000)
3: public void testGetConexao() {
4:        Connection con = Conexao.getConexao();
5:        assertNull("Não foi possível conectar no banco de dados", con);
6: }
```

No exemplo apresentado no Código-fonte 16, o testador deve definir um caso de teste cujo tempo de execução seja superior ao valor original de *timeout*, porém inferior ao valor mutado, ou seja $1000 < tempodeexecucao \leq 10000$.

As mesmas observações feitas para o operador DCfT em relação ao decremento do *timeout* valem para o incremento.

4.3.9 RTA (Remover constante timeout)

RTA é um operador de nível de anotação e classe de alteração. O operador RTA remove o *timeout* do oráculo e mata o mutante dependendo do valor do *timeout* e do tempo que o teste demora para ser realizado. Se o oráculo depende do valor de *timeout* previamente estabelecido, aplicando este operador, o oráculo mutante passará a ter o resultado diferente do oráculo original.

No JUnit é possível que um teste tenha um tempo máximo para ser executado. Por exemplo, caso o testador queira que o teste não demore mais que 500 milissegundos pode-se realizar a seguinte operação (Código-fonte 17, linha 2), no entanto, algumas operações podem demorar mais que o tempo estipulado no parâmetro *timeout*, e para isso, o operador RTA, remove esse parâmetro, fazendo com que a execução do teste utilize o tempo padrão de execução do JUnit.

Código-fonte 17 – Exemplo operador RTA.

```
1: \\ -> @Test
2: @Test(timeout = 500)
3: public void fastTestCase() {
4:     Assert.assertEquals(1500, calendar.getSize());
5: }
```

Esse operador de mutação faz com que o oráculo mutante não dependa do tempo de execução do caso de teste e, em teoria, poderia executar por um tempo infinito. No caso de o mutante ser morto, ou seja, indicar que o teste passou, enquanto o oráculo original indica que ele falhou, tem-se uma indicação de que, o caso de teste na realidade não depende do tempo de execução e que a cláusula *timeout* foi indevidamente utilizada.

4.4 Estrutura da “MuJava 4 JUnit”

Os operadores de mutação para oráculos foram incluídos na ferramenta MuJava. Assim, seguem a mesma estrutura de código já existente na ferramenta. Cada operador possui uma classe para gerar as alterações sintáticas nos códigos originais e uma classe para a geração dos arquivos dos programas mutantes.

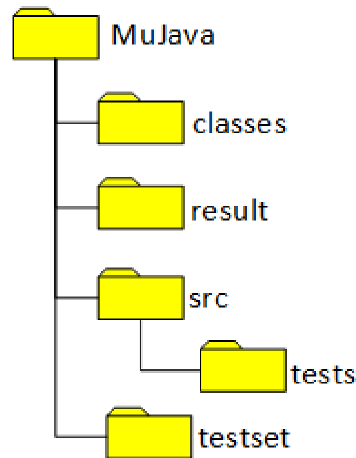
A “MuJava 4 JUnit” possui uma estrutura de diretórios, apresentada na Figura 8, similar à utilizada pela MuJava. Os arquivos binários da aplicação a ser testada devem ser armazenados na pasta *classes*, o código fonte deve estar na pasta *src*, o código fonte e os arquivos binários dos arquivos de teste devem ser copiados para a pasta *testset*. Ao aplicar os operadores de mutação às aplicações desejadas, os resultados são gravados na pasta *src*. Para a adaptação da MuJava ao contexto deste projeto, as classes JUnit são armazenadas numa subpasta da pasta *src*, chamada *tests*.

A interface gráfica da MuJava foi adaptada para que sejam utilizados os novos operadores de mutação para oráculos. Originalmente, a chamada da interface dá-se pelo comando:

```
> java mujava.gui.GenMutantsMain
```

Para que seja realizada a chamada da ferramenta “MuJava 4 JUnit”, será necessário

Figura 8 – Estrutura de diretórios da “MuJava 4 JUnit”.



Fonte: Elaborada pelo autor.

passar o parâmetro `-oracle`, executando a interface que é mostrada na Figura 9, por meio do comando:

```
> java mujava.gui.GenMutantsMain -oracle
```

4.5 Implementação

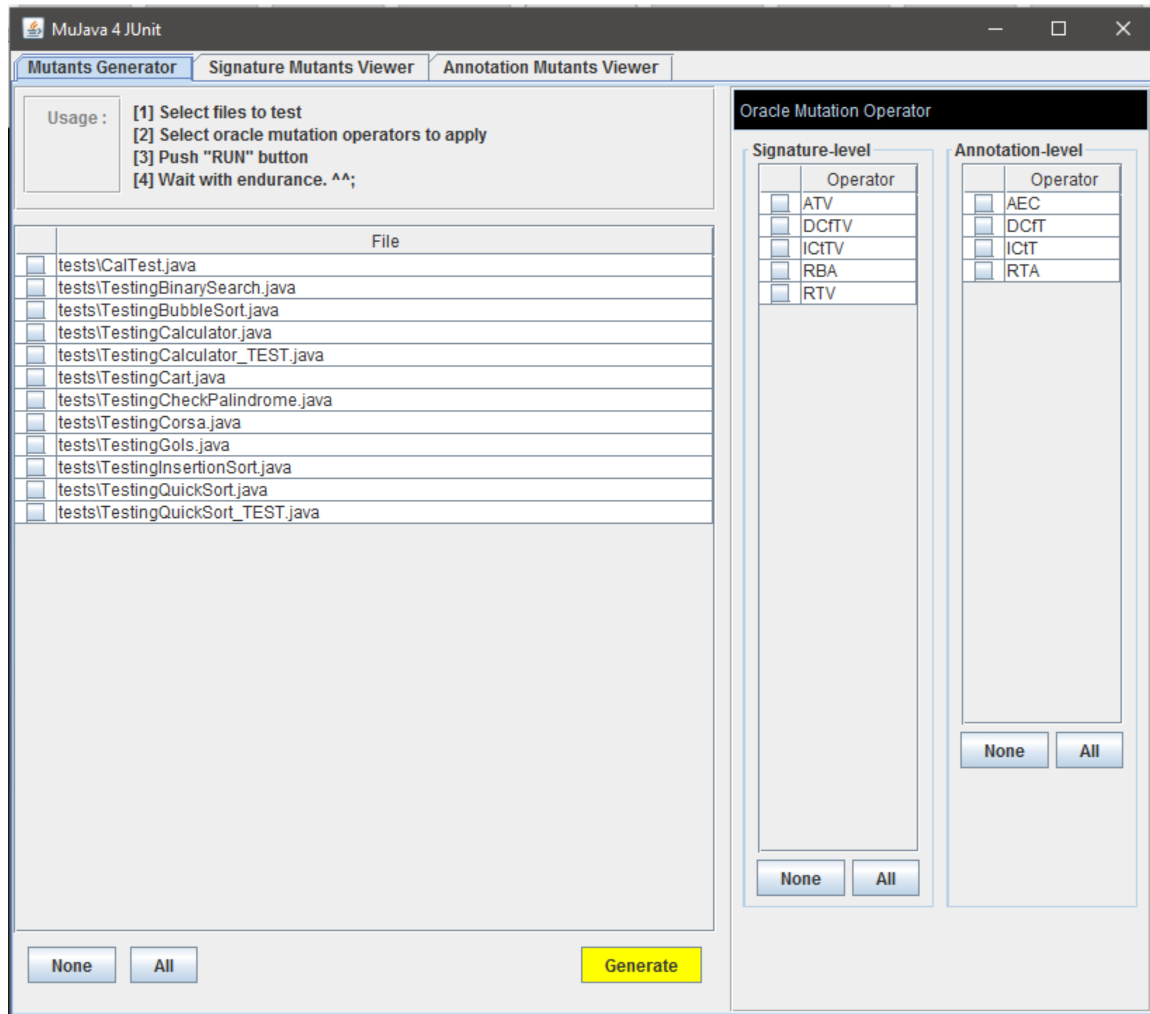
Antes de iniciar a implementação da ferramenta “MuJava 4 JUnit” foi necessário reconhecer toda a estrutura da ferramenta MuJava, que por ser uma ferramenta já consolidada e com diversas funções implementadas, possui uma grande quantidade de pacotes de classes, as quais precisaram ser analisadas para que fosse iniciada a implementação dos operadores de mutação para oráculos de teste baseados em assertivas propostos durante a realização do mestrado. A ferramenta “MuJava 4 JUnit” está disponível em https://goo.gl/ZGXqI5.

Como apresentado no Capítulo 2, Seção 2.6, a MuJava possui dois níveis de operadores: (i) operadores tradicionais; e (ii) operadores de classe. Os operadores de mutação para oráculos projetados e implementados nesse trabalho, possuem a estrutura semelhante aos operadores tradicionais implementados na MuJava.

A MuJava possui três pacotes principais:

- mujava: esse pacote inclui os geradores de mutantes (MutantsGenerator.java e suas subclasses) e os executores de teste (TestExecuter.java e suas subclasses). Como seus nomes implicam, eles funcionam como o núcleo da ferramenta. Para desenvolver os novos operadores, conhecer as classes desse pacote é essencial;

Figura 9 – Interface da ferramenta “MuJava 4 JUnit”.



Fonte: Elaborada pelo autor.

- mujava.op: esse pacote inclui a implementação de operadores de mutação. Normalmente, cada operador tem duas classes: uma classe para a geração dos mutantes e uma classe para gerar o arquivo com o código mutante. Por exemplo, para o operador ATV tem-se as classes: ATV.java e ATV_Writer.java; e

- mujava.gui: esse pacote inclui a interface gráfica da MuJava. Os operadores foram implementados e adicionados a uma interface gráfica adaptada da MuJava.

Cada classe geradora de mutante tem um conjunto de métodos `visit()` e `outputToFile()`, sendo esses, os principais métodos utilizados na implementação dos operadores de mutação. O método `visit()` faz a varredura no código fonte, altera o código de acordo com a implementação do operador e após realizar a alteração, passa para o método `outputToFile()` para a geração do mutante. O `outputToFile()` utiliza a classe de escrita do mutante (ATV_Writer.java).

Para modificar, por exemplo, a chamada do método `assertEquals(double, double, 0.0001)` para `assertEquals(double, double, 0.0001+constante)` é necessário implementar um método `visit(MethodCall)` para verificar cada chamada realizada para o método "assertEquals". Depois disso, utilizar métodos como `getArguments()` e `setArguments()` para mutar os argumentos da chamada do método. E finalmente, ter uma função correspondente do método `outputToFile()` para chamar a classe de escrita do mutante.

O Código-fonte 18 apresenta a classe de implementação do operador **ATV**, em que o parâmetro delta é adicionado ao método `assertEquals`. O método `visit(MethodCall)` realiza a mutação no oráculo por meio do parâmetro `MethodCall` que é a chamada do método em questão. Neste caso, o método a ser mutado é o `assertEquals`, portanto na Linha 4 do Código-fonte 18 existe um *if* que verifica se o método é o `assertEquals`. No caso da condição do *if* ser verdadeira, o próximo passo é verificar o tipo dos parâmetros utilizados no `assertEquals`.

As linhas 8 e 9 do Código-fonte 18 verificam se o tipo do parâmetro é *double* ou *float*, sendo algum desses dois tipos, é preciso verificar se o `assertEquals` tem 2 argumentos (Código-fonte 18, linha 12) ou 3 argumentos (Código-fonte 18, linhas 23, 24 e 25), mas no caso de 3 argumentos, o primeiro argumento precisa ser do tipo "String" para que o delta possa ser adicionado. Satisfazendo uma dessas condições, inicia-se o processo de adicionar o parâmetro delta.

Os parâmetros já definidos pelo oráculo original são mantidos, enquanto o parâmetro delta é adicionado por meio da classe `Literal` que é da biblioteca `OpenJava` (Seção 4.5.3), em que converte tipos de dados para "String". Depois de atribuídos todos os parâmetros do método `assertEquals` mutados, um novo objeto do tipo `MethodCall` é instanciado. Por fim, o método `outputToFile` é chamado para que seja gerado o arquivo do oráculo mutante.

Código-fonte 18 – Implementação do operador ATV.

```
 1: public class ATV{
 2:     public void visit(MethodCall p) throws ParseTreeException{
 3:       ExpressionList arguments = p.getArguments();
 4:       if ((p.getName().equals("assertEquals")){
 5:           try {
 6:                 OJClass varType = arguments.get(1)
 7:                     .getType(getEnvironment());
 8:                 if((varType.getName().contains("double"))||
 9:                   (varType.getName().contains("float"))){
10:                     ExpressionList mutantArgs = new
11:                       ExpressionList();
12:                     if(arguments.size()==2){
13:                         mutantArgs.add(arguments.get(0));
14:                         mutantArgs.add(arguments.get(1));
15:                         mutantArgs.add(Literal.makeLiteral
```

```
16:                          (0.001));
17:                     MethodCall mutant = new MethodCall(p.
18:                       getReferenceExpr(), p.getName(),
19:                       mutantArgs);
20:                     outputToFile(p, mutant);
21:                 }
22:             }
23:             if(arguments.size()==3 && (arguments.get(0).
24:               getType(getEnvironment()).getName().
25:               contains("String"))){
26:                 mutantArgs.add(arguments.get(0));
27:                 mutantArgs.add(arguments.get(1));
28:                 mutantArgs.add(arguments.get(2));
29:                 mutantArgs.add(Literal.makeLiteral(0.001));
30:                 MethodCall mutant = new MethodCall(
31:                   p.getReferenceExpr(), p.getName(),
32:                   mutantArgs);
33:                 outputToFile(p, mutant);
34:             }
35:         }
36:     } catch (Exception e) {
37:     e.printStackTrace();
38:     }
39: }
```

O método `outputToFile()` (Código-fonte 19) inicia o processo de escrita do oráculo mutante e recebe por parâmetro o método original e o método que foi mutado no método `visit()`. Primeiramente, a pasta que os mutantes devem ser armazenados é informada nas linhas 3 e 4 do Código-fonte 19. O mutante é passado por parâmetro para o construtor da classe `ATV_Writer`, na qual é escrito o mutante e adicionadas as informações sobre a geração do mutante em um arquivo de *log*. Por fim, as anotações são escritas no oráculo mutante.

Código-fonte 19 – Método `outputToFile()`.

```
 1: public void outputToFile(MethodCall original_field, MethodCall mutant
      ){
 2:     String f_name;
 3:     f_name = getSourceNameOracle("ATV");
 4:     String mutant_dir = getMutantID("ATV");
 5:     try{
 6:         PrintWriter out = getPrintWriter(f_name);
 7:         ATV_Writer writer = new ATV_Writer(mutant_dir, out);
 8:         writer.setMutant(original_field, mutant);
 9:         writer.setMethodSignature(currentMethodSignature);
10:         out.flush();
11:         out.close();
```

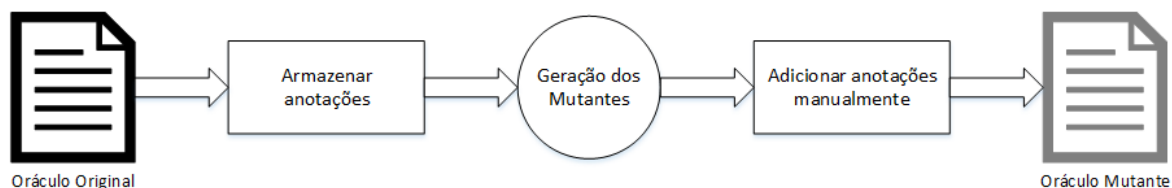
```
12:         OracleMutantCodeWriter.writeAnnotations(f_name);
13:     }catch (IOException e) {
14:         System.err.println("fails to create " + f_name );
15:     }catch (ParseTreeException e) {
16:         System.err.println("errors during printing "
17:           + f_name );
18:         e.printStackTrace();
19:     }
20: }
```

4.5.1 Adicionando anotações JUnit

As anotações tiveram que ser adicionadas ao final do processo de geração de mutantes porque a versão original da MuJava retira todas as anotações e comentários do programa em teste. Portanto, ao notar esse problema, foi necessário criar um processo complementar à geração dos mutantes para que as anotações fossem adicionadas ao final da geração dos mutantes.

Para os oráculos de teste escritos na forma de classe JUnit, as anotações são essenciais. Diante desse problema a solução implementada foi: (i) armazenar as anotações no início do processo de geração de mutantes; (ii) gerar os mutantes com as anotações excluídas do arquivo; e (iii) ao obter o arquivo do mutante, adicionar novamente as anotações. O processo pode ser visualizado na Figura 10.

Figura 10 – Processo para adicionar as anotações nos arquivos dos oráculos mutantes.



Fonte: Elaborada pelo autor.

Para armazenar as anotações antes do início do processo de geração de mutantes, foi criada uma classe para gerenciar as anotações, a `AnnotationManager`. Esta classe tem como atributos o nome da anotação, a linha que ela se encontra no arquivo original e a sequência em que ela ocorre.

As anotações são adicionadas logo após a geração dos mutantes. O método de escrita das anotações é chamado dentro do método `outputToFile` e pode ser visualizado na Linha 12 do Código-Fonte 19. O método `writeAnnotations()` recebe por parâmetro o caminho do arquivo do oráculo mutante e escreve as anotações neste mutante utilizando as informações que estão gravadas em um objeto `AnnotationManager`.

4.5.2 Geração de relatório sobre os oráculos mutantes

A geração dos relatórios em relação aos mutantes vivos e mortos foi toda implementada, pois o conceito de mutantes vivos e mortos é diferente do utilizado na ferramenta MuJava. Um mutante morto, no contexto deste trabalho, é quando o caso de teste original "passa" pelo oráculo original e "falha" com o oráculo mutante, ou vice-versa. Um mutante vivo é quando o oráculo original e o mutante apresentam o mesmo comportamento.

A implementação do relatório utilizou a classe `JUnitCore`, que suporta a execução de testes escritos no JUnit 4, JUnit 3.8.x e misturas entre essas versões. `JUnitCore` usa reflexão para encontrar um `Runner` apropriado para as classes de teste aprovadas, ou seja, procura uma anotação `@RunWith` na classe de teste. Se nenhum outro `Runner` for encontrado, o `Runner` padrão (`BlockJUnit4ClassRunner`) é usado. O `Runner` é instanciado e a classe de teste passada para o `Runner`.

Foi implementado um `Runner` próprio do qual se pudesse extrair as informações necessárias para avaliar mutantes vivos ou mortos. A classe `MyRunner` foi implementada sobrescrevendo o método `run()`, pois para a geração do relatório foi necessária a implementação de uma classe chamada `JUnitExecutionListener` herdando de `RunListener`. A classe `RunListener` é utilizada para responder aos eventos durante uma execução de teste, portanto ela é estendida e os métodos necessários são implementados.

O Código-fonte 20 mostra a implementação da classe `JUnitExecutionListener` que configura os métodos de acordo com as necessidades para a geração do relatório sobre os oráculos mutantes vivos e mortos. Pode-se obter informações como número de métodos executados no arquivo JUnit, o método que iniciou a execução, o método que finalizou a execução, os métodos que falharam e os métodos que estão sendo ignorados.

Código-fonte 20 – Implementação da classe `JUnitExecutionListener`.

```
1: public class JUnitExecutionListener extends RunListener {
2:   public void testRunStarted(Description description)
3:     throws Exception {
4:      RunnerOracleMain.methods.clear();
5:      System.out.println("Number of tests to execute: "
6:      + description.testCount());
7:      TestClass.arquivo+= "Number of tests to execute: "
8:      + description.testCount()+"\n";
9:   }
10:
11:   public void testRunFinished(Result result)
12:     throws Exception {
13:     System.out.println("Number of tests executed: "
14:       + result.getRunCount());
15:   }
```

```
16:
17:   public void testStarted(Description description)
18:     throws Exception {
19:     System.out.print("MethodName: " + description.
20:       getMethodName()+" ");
21:   }
22:
23:   public void testFinished(Description description)
24:     throws Exception {
25:     System.out.println("Finished: " + description.
26:       getMethodName());
27:     System.out.println();
28:   }
29:
30:   public void testFailure(Failure failure)
31:     throws Exception {
32:     System.out.print(" Failed ");
33:     RunnerOracleMain.methods.add(failure.
34:       getDescription().getMethodName());
35:   }
36:
37:   public void testAssumptionFailure(Failure failure) {
38:     System.out.print(" Failed ");
39:     RunnerOracleMain.methods.add(failure.
40:       getDescription().getMethodName());
41:   }
42:
43:   public void testIgnored(Description description)
44:     throws Exception {
45:     System.out.print("Ignored ");
46:   }
47: }
```

O oráculo original é executado, utilizando a `MyRunner` que é a classe responsável por chamar a `JUnitExecutionListener` e grava o nome dos métodos que falharam em um arquivo `original.txt`. Posteriormente, todos os oráculos mutantes são executados, armazenando os métodos que falharam e comparando com o oráculo original por meio do arquivo `original.txt`. Além do número de mutantes vivos e mortos, o número de mutantes gerados por operador são apresentados no relatório.

O modelo do relatório apresentado após a execução dos oráculos pode ser visualizado na Figura 11, em que para cada operador é possível visualizar "número de mutantes vivos/número de mutantes mortos", por exemplo, no operador ATV do relatório apresentado na Figura 11 têm-se 2 mutantes vivos e 1 mutante morto.

Figura 11 – Exemplo de exibição do relatório sobre mutantes vivos e mortos.

```
**********SIGNATURE**********
ATV: 2/1 -- RTV: 0/0
DCfTV: 0/0 -- ICtTV: 0/0
RBA: 1/1
**********ANNOTATION**********
AEC: 5/2 -- DCfT: 0/0
ICtT: 0/0 -- RTA: 0

*****************************
Mutantes vivos: 8
Mutantes mortos: 4
Total mutantes: 12
```

Fonte: Elaborada pelo autor.

4.5.3 Biblioteca `OpenJava`

`OpenJava`[1] é um sistema de macro com uma estrutura de dados que representa uma estrutura lógica de um programa Orientado a Objetos (TATSUBORI *et al.*, 2000).

Na implementação dos operadores de mutação, a classe `OJClass`, da `OpenJava`, foi bastante utilizada para obter a superclasse de determinada classe, retornando um meta objeto de classe. Como resultado, o meta programa pode usar o meta objeto da classe retornada para obter informações diretamente em relação à superclasse.

A `OpenJava` gera automaticamente os meta objetos da classe, mesmo para classes declaradas em outro arquivo de origem ou para classes disponíveis somente na forma de *bytecode*, ou seja, classes cujo código fonte não está disponível.

A classe `OJMethod`, que é o tipo de retorno de `getDeclaredMethods()` em `OJClass`, representa a estrutura lógica de um método. Assim, de forma semelhante à classe `OJClass`, essa classe tem métodos para examinar ou modificar os atributos do método.

A MuJava utiliza a `OpenJava` para reconhecer componentes sintáticos como expressões binárias, expressões unárias, estrutura de repetição (for, while, do...while, etc.), estruturas de seleção, etc. A "MuJava 4 JUnit" também utilizou a biblioteca *OpenJava* para reconhecer os métodos `assert`, as anotações JUnit e seus respectivos parâmetros.

4.6 Considerações finais

Este capítulo apresentou inicialmente o conjunto de operadores de mutação desenvolvidos para oráculos JUnit. Esse operadores limitam-se a atuar em conceitos específicos do JUnit relacionados à implementação dos oráculos dos casos de teste. Procurou-se mostrar a

[1] http://openjava.sourceforge.net/

fundamentação para a implementação de cada um dos operadores, bem como realizar uma avaliação analítica do seu comportamento, indicando situações em que os mutantes podem ser úteis. Embora a implementação atual da ferramenta “MuJava 4 JUnit” não contemple todas as estratégias, foram discutidas possíveis alternativas para a implementação dos operadores de mutação projetados.

Em seguida, discutiu-se a implementação da ferramenta de suporte à validação dos oráculos JUnit utilizando mutação. Para que a ferramenta fosse implementada, diversos aspectos tiveram que ser levados em consideração. O primeiro foi a implementação dos operadores de mutação, que se baseia na versão original da ferramenta, a MuJava. Cada um dos operadores projetados foi convertido em classes Java que cuidam de modificar o programa original e gerar a sua versão modificada. Várias adaptações foram feitas nesse sentido, principalmente considerando que a versão original da ferramenta remove anotações do programa e estes são essenciais na estratégia aqui apresentada. Finalmente, foi necessária a criação de mecanismos específicos para a execução dos oráculos originais e dos mutantes, de modo que se pudesse colher o resultado da corporação entre eles.

Assim, grande parte da contribuição dessa dissertação está descrita neste capítulo, com o projeto e análise dos operadores e a criação de uma ferramenta que contempla as necessidades da estratégia de validação proposta para oráculos JUnit. No próximo capítulo, um estudo empírico realizado para avaliar os resultados dessa contribuição é detalhado.

CAPÍTULO

5

ESTUDO EMPÍRICO

5.1 Considerações iniciais

Esta seção apresenta um estudo empírico envolvendo a mutação dos oráculos de teste. O objetivo do estudo é aplicar os operadores específicos para oráculos de teste baseados em assertivas (escritos na forma de classe JUnit) e gerar os mutantes. As mudanças sintáticas produzidas pelos oráculos de teste mutantes reproduzem defeitos nas assinaturas ou anotações do métodos `assert`.

Além disso, os detalhes do estudo empírico realizado com a "MuJava 4 JUnit" são apresentados, bem como são apresentados os passos realizados durante o estudo, questões de pesquisa, programas utilizados no experimento, execução detalhada do experimento realizado com cinco programas simples e seus respectivos oráculos de teste, resultados por meio das respostas das questões de pesquisa e vantagens e desvantagens da técnica desenvolvida.

5.2 Um estudo empírico com a "MuJava 4 JUnit"

Este estudo foi conduzido seguindo conceitos e recomendações do Wohlin *et al.* (2012). Depois de projetar os operadores específicos para oráculos de teste na forma de assertivas e implementá-los na ferramenta "MuJava 4 JUnit" (Seção 4.5), um experimento foi desenvolvido com o objetivo de avaliar os operadores. Para tal objetivo, foram definidas questões de pesquisa, selecionados programas para a execução do experimento e realizada uma coleta de dados em relação aos oráculos mutantes.

5.2.1 Design do experimento

O experimento foi conduzido com o objetivo de verificar se os oráculos são capazes de identificar falhas que não foram encontradas anteriormente pelos oráculos originais e analisar

os oráculos mutantes para avaliar se podem contribuir com a qualidade dos oráculos originais, identificando erros de implementação.

Adicionalmente à aplicação da mutação nos oráculos de teste e a análise dos resultados neste contexto, os programas utilizados no experimento foram mutados utilizando a MuJava com o objetivo de verificar se os oráculos de teste originais e mutantes teriam comportamento diferente nos programas do experimento mutados pela MuJava.

A Figura 12 ilustra os passos realizados no experimento, são eles:

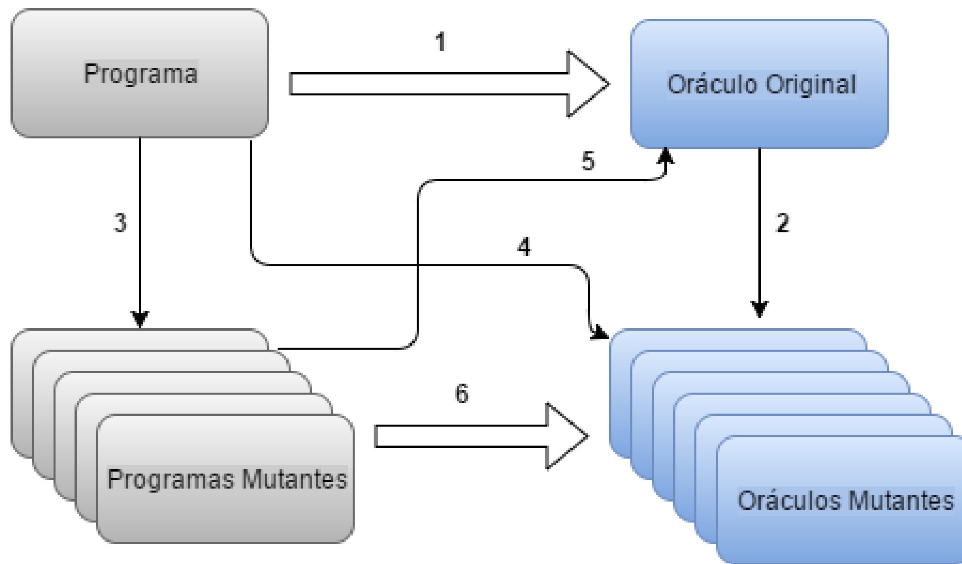


Figura 12 – Passos do experimento.

1. Executar o oráculo original contra os programas utilizados no experimento: cada programa utilizado no experimento possui um oráculo de teste, os oráculos originais. Neste passo do experimento, os oráculos originais são executados e o resultado da execução é armazenado para que seja utilizado posteriormente na comparação com o resultado da execução dos oráculos mutantes;

2. Aplicar os operadores de mutação da "MuJava 4 JUnit" em todos os oráculos de teste: neste passo do experimento, todos os operadores de mutação para oráculos de teste baseados em assertivas que foram implementados na ferramenta "MuJava 4 JUnit" são aplicados nos oráculos de teste originais, gerando os oráculos mutantes;

3. Aplicar todos os operadores de mutação tradicionais da MuJava nos programas utilizados no experimento: mutantes dos programas originais são gerados com o objetivo de inserir defeitos nos programas originais e verifica-se se isso causa alteração no resultado dos oráculos;

4. Executar todos os oráculos mutantes nos programas utilizados no experimento: os oráculos mutantes, gerados no Passo 2, são executados contra os programas utilizados no

experimento. Os resultados das execuções dos oráculos mutantes são comparados com o resultado da execução do oráculo original (Passo 1), gerando resultados sobre quais mutantes estão vivos ou mortos;

5. Executar todos os programas mutantes gerados pela MuJava contra o oráculo original: os oráculos originais são executados contra os programas mutantes gerados no Passo 3 e os resultados dessa execução são armazenados; e

6. Executar todos os programas mutantes, gerados pela MuJava, contra todos os oráculos mutantes, gerados pela "MuJava 4 JUnit": neste passo final, os resultados da execução dos oráculos mutantes com os programas mutantes são comparados com os resultados obtidos no Passo 5 e armazenadas informações sobre mutantes vivos e mortos dessa execução.

5.2.2 Questões de pesquisa

As seguintes questões de pesquisa foram definidas:

QP1: Os oráculos mutantes são capazes de melhorar a qualidade do oráculo original?

QP2: A eficiência dos operadores muda dependendo do programa em teste?

Visando responder essas questões de pesquisa, os operadores de mutação foram aplicados em oráculos de teste na forma de assertivas de 5 programas, os quais geram oráculos mutantes em que supostamente melhoram a qualidade do oráculo original. Estatísticas em relação aos oráculos mutantes vivos e mortos também foram coletadas.

Os operadores tradicionais da MuJava também foram aplicados nos programas utilizados no experimento, com o objetivo de verificar se a mutação diretamente nos programas é capaz de mudar o resultado dos oráculos.

Para identificar o comportamento dos operadores e a capacidade de melhorar a qualidade dos oráculos originais, apenas as estatísticas de mutantes vivos e mortos não são suficientes. Portanto, uma análise manual dos oráculos mutantes foi realizada para verificar se os oráculos mutantes foram eficientes em melhorar a qualidade dos oráculos, identificando erros ou gerando novos casos de teste.

A geração de códigos mutantes de oráculos de teste sugere reparos nos testes unitários definidos previamente. Além disso, novos casos de teste podem ser encontrados para melhorar a qualidade do conjunto original de casos de teste.

Um operador é considerado eficiente, no contexto desse experimento, de acordo com a capacidade de melhorar a qualidade dos oráculos de teste, ou seja, sugerir novos casos de teste ou identificar erros nos programas que estão sendo executados pelos oráculos de teste.

5.2.3 Seleção de programas

Foram selecionados 5 programas de complexidades ciclomáticas diferentes, variando de 1 a 6, para verificar a efetividade dos oráculos mutantes em revelar defeitos nos oráculos originais. Os programas utilizados no experimento são apresentados na Tabela 7. Cada programa possui uma classe de testes com oráculo escrito na forma de classe JUnit. As informações sobre os oráculos estão apresentadas na Tabela 8.

Tabela 7 – Aplicações utilizadas no experimento.

Programa	#Complexidade Ciclomática	#Linhas de código
Calculator	1	19
CheckPalindrome	3	16
BinarySearch	4	31
BubbleSort	4	66
ShoppingCart	6	117

Tabela 8 – Casos de teste utilizados no experimento.

Programa	#Complexidade Ciclomática	#Linhas de Código	#Casos de Teste	
			Passou	Falhou
TestingCalculator	1	58	3	7
TestingCheckPalindrome	1	61	8	2
TestingBinarySearch	1	114	8	2
TestingBubbleSort	1	156	7	3
TestingShoppingCart	1	212	2	13

Os oráculos descritos na Tabela 8 utilizam os métodos `assertEquals`, `assertTrue` e `assertFalse`. Os operadores geram os mutantes de acordo com os métodos `Assert` utilizados e quais parâmetros estão sendo utilizados. A anotação `@Test` é utilizada em todos os oráculos, podendo estar com os parâmetros *expected* que especifica uma exceção esperada na execução do oráculo, ou com o parâmetro *timeout* que define o tempo máximo de execução do oráculo em milissegundos.

5.2.4 Execução do experimento

O experimento foi executado em duas partes, dividido em 6 passos (Figura 12). Na primeira parte do experimento, cinco programas foram selecionados. Cada programa tem uma classe de teste correspondente com alguns oráculos baseados em assertivas escritos no JUnit. Então, os mutantes específicos para oráculos de teste foram aplicados por meio da ferramenta "MuJava 4 JUnit", e informações sobre os mutantes vivos e mortos foram coletadas.

Um oráculo mutante vivo é quando o oráculo original tem resultado *verdadeiro*, ou seja, passou, e o oráculo mutante também tem resultado *verdadeiro*, ou quando o oráculo original tem

resultado *falso*, ou seja, falhou, e o oráculo mutante também tem resultado *falso*. Um oráculo mutante morto é quando o oráculo original tem resultado *verdadeiro* e o oráculo mutante tem resultado *falso* e vice-versa.

Os programas utilizados no experimento não possuem defeitos, portanto se o oráculo apresenta falha, pode-se afirmar que o oráculo que apresenta defeito e não o programa. Porém, ao inserir defeitos no programa, não é possível realizar a mesma afirmação e é necessário analisar o oráculo para que seja verificado onde o defeito está.

Na primeira parte do experimento, os programas não contém defeitos. Por outro lado, na segunda parte do experimento, defeitos são inseridos por meio da aplicação do teste de mutação, aplicando todos os operadores tradicionais da MuJava nos 5 programas utilizados no experimento. Adicionalmente, os programas mutantes gerados foram executados contra oráculos mutantes, gerados pela ferramenta "MuJava 4 JUnit", e os oráculos originais (Tabela 8).

5.3 Resultados

No total, a "MuJava 4 JUnit" implementa 9 operadores de mutação para oráculos de teste, dos quais 5 são do nível de assinatura e 4 são do nível de anotação. Nesta seção, é apresentada uma análise detalhada dos efeitos de utilizar versões ligeiramente modificadas de oráculos de teste para melhorar a qualidade das classes de teste. Foi realizada uma análise do comportamento dos operadores e verificado se os mutantes gerados foram vivos ou mortos.

Os resultados sobre os mutantes vivos e mortos foram checados e a eficiência dos oráculos mutantes em revelar defeitos nos oráculos originais foi analisada e são apresentados de forma resumida na Tabela 9, em que estão separados por programa. A coluna "Original" apresenta os números referentes aos oráculos mutantes vivos (V) e mortos (M) utilizando o programa original, sem mutação. A coluna "MuJava" apresenta os números relacionados aos oráculos mutantes aplicados aos programas mutantes do experimento.

5.4 Respostas das questões de pesquisa

QP1: Os oráculos mutantes são capazes de melhorar a qualidade do oráculo original?

Alguns operadores geram mais mutantes que outros. A geração de mutantes depende da assertiva utilizada, dos parâmetros definidos nas assertivas e qual anotação está sendo executada. Neste experimento, foram coletados dados dos mutantes gerados por cada operador implementado na "MuJava 4 Junit", número de mutantes vivos e número de mutantes mortos (Tabela 9).

A porcentagem de mutantes vivos e mortos por cada operador da ferramenta "MuJava 4

Tabela 9 – Resumo número de mutantes vivos e mutantes mortos coletados no experimento.

	Calculator				CheckPalindrome				BinarySearch				BubbleSort				ShoppingCart				TOTAL		
	Original		MuJava		Original		MuJava		Original		MuJava		Original		MuJava		Original		MuJava				
	V	M	V	M	V	M	V	M	V	M	V	M	V	M	V	M	V	M	V	M	V	M	TODOS
ATV	2	2	144	117	0	0	0	0	0	0	0	0	0	0	0	0	34	0	6902	0	7082	119	7201
DCfTV	3	2	216	117	0	0	0	0	4	1	160	25	5	0	240	45	4	1	240	30	872	221	1093
ICfTV	3	1	216	79	0	0	0	0	4	1	160	25	5	0	240	45	4	2	240	100	872	253	1125
RBA	0	0	0	0	7	7	392	261	7	7	138	89	0	0	0	0	14	14	2842	2667	3400	3045	6445
RTV	3	2	216	115	0	0	0	0	0	0	0	0	0	0	0	0	2	1	140	30	361	148	509
AEC	60	18	4260	1212	60	30	3360	1488	60	12	8880	2850	60	18	14518	7815	90	12	18270	2187	49618	15642	65260
DCfT	3	0	50	0	0	0	0	0	0	0	0	0	0	0	0	0	4	1	250	42	307	43	350
ICfT	3	1	25	5	0	0	0	0	0	0	0	0	0	0	0	0	4	2	250	65	282	73	355
RTA	3	0	12	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	10	0	29	0	29

JUnit", considerando apenas a primeira parte do experimento, ou seja, apenas os programas sem defeitos estão sumarizados na Tabela 10. Pode-se observar que alguns operadores matam mais mutantes que outros. Também observa-se que os operadores da ferramenta "MuJava 4 JUnit" trabalharam bem na geração dos oráculos mutantes.

Como discutido no Capítulo 4, os oráculos mutantes mais interessantes são aqueles que permanecem dando resultado igual aos oráculos originais, pois eles podem sugerir novos casos de teste, indicar fraquezas no caso de teste, ou então, identificar erros no programa que está sendo testado. O Código-fonte 21 apresenta um caso de teste do programa *ShoppingCart*, utilizado no experimento, em que um produto é adicionado ao carrinho e o oráculo verifica se o preço final da compra será 4,5. Em sistemas financeiros, é essencial que a precisão das casas decimais seja exata, portanto adicionar o valor de delta auxilia o testador a ajustar o valor correto. O operador ATV adiciona o valor de delta de 10^{-3} (Código-fonte 22). O sistema de real utiliza centavos (10^{-2}) nos cálculos. No entanto, duas casas decimais podem não resolver todos os cálculos possíveis e pode ser necessário realizar um arredondamento das casas decimais, portanto o delta 10^{-3} é útil para os casos em que apenas duas casas decimais não resolvem.

Código-fonte 21 – Exemplo do operador ATV no experimento (original).

```
1: @Test
2: public void testAddOneProduct(){
3:     cart.addProduct(blackPen, 2);
4:     assertEquals(4.50, cart.getTotalPrice());
5: }
```

Código-fonte 22 – Exemplo do operador ATV no experimento (mutante).

```
1: @Test
2: public void testAddOneProduct(){
3:     cart.addProduct(blackPen, 2);
4:     assertEquals(4.50, cart.getTotalPrice(), 0.001);
5: }
```

Pode-se observar na Tabela 10 que os mutantes gerados possuem maior taxa de mutantes vivos, comparados aos mutantes mortos. Assim sendo, a resposta para a questão de pesquisa QP1 diante do contexto deste experimento, é sim. Porém, futuramente devem ser realizadas análises minuciosas dos oráculos mutantes vivos para que esse resultado seja consolidado.

QP2: A eficiência dos operadores muda dependendo do programa em teste?

Como discutido na Seção 5.3, cada operador gera mutantes de acordo com o método `Assert` e seus respectivos parâmetros, ou pelas anotações utilizadas. Portanto, ao realizar o experimento, chegou-se a conclusão de que o tipo de dados que o programa está usando, é o que vai determinar qual operador é mais eficiente naquela situação.

Tabela 10 – Mutantes vivos e mortos por operador da “MuJava 4 JUnit”.

	Vivos(%)	**Mortos(%)**
ATV	94,74	5,26
DCfTV	80,00	20,00
ICtTV	80,00	20,00
RBA	50,00	50,00
RTV	62,50	37,50
AEC	78,57	21,43
DCfT	87,50	12,50
ICtT	70,00	30,00
RTA	100,00	0,0

No contexto do experimento apresentado, o programa *CheckPalindrome*, por exemplo, trabalha com valores booleanos, portanto o operador que usa esses valores são mais eficientes, nesse caso será o operador RBA. O programa *Calculator*, trabalha com valores inteiros e *double*, fazendo com que os operadores ATV, DCfTV, ICtTV e RTV sejam mais eficientes. Os programas *BinarySearch*, *BubbleSort* e *ShoppingCart*, realizam operações com valores booleanos, inteiros e *double*, utilizando portanto todos os operadores desses gêneros. A Tabela 11 evidencia a quantidade de mutantes gerados por cada operador em cada programa utilizado no experimento.

Tabela 11 – Quantidade de mutantes gerados em cada programa separados por operador.

	Calculator	**CheckPalindrome**	**BinarySearch**	**BubbleSort**	**ShoppingCart**
ATV	4	0	0	0	34
DCfTV	5	0	5	5	5
ICtTV	4	0	5	5	6
RBA	0	14	14	0	28
RTV	5	0	0	0	3
AEC	78	90	72	78	102
DCfT	3	0	0	0	5
ICtT	4	0	0	0	6
RTA	3	0	0	0	4

5.5 Vantagens e desvantagens

Este estudo apresenta uma proposta sobre operadores de mutação para oráculos de teste baseados em assertivas, iniciando uma forma sistemática de avaliação da qualidade de oráculos. Por outro lado, os operadores foram projetados apenas para oráculos escritos em formato de JUnit, por meio da sua estrutura assertiva.

Os operadores tiveram um bom desempenho e foram observados seus comportamentos em diferentes situações. O principal objetivo desse estudo foi analisar o comportamento dos operadores propostos para oráculos de teste escritos na forma de assertivas, coletar dados sobre os mutantes vivos e mutantes mortos e analisá-los.

Os operadores ATV, DCfTV, ICfTV e RTV são úteis quando algum mutante fica morto, pois indica que o valor de precisão está fazendo a diferença no resultado do oráculo. Na prática, ao obter um mutante nessa condição, o testador deve atentar para o fato de que o caso de teste não esteja preciso quanto aos dados de teste e talvez alterar o seu oráculo para que o valor de precisão não precise interferir para alterar o valor final da execução do oráculo.

O operador AEC gera mutantes que podem ser mortos ou vivos. Na prática, ao obter um mutante nessa condição, o testador deve atentar para o fato de que o caso de teste necessita da exceção adicionada pelo operador e é interessante que ele projete o oráculo levando em consideração todas as situações que possam ocorrer para que as exceções sejam necessárias.

Os operadores DCfT, ICfT e RTA geram mutantes que podem ser mortos ou vivos. Eles são úteis quando algum mutante fica morto, pois mostra que o *timeout* deve ser reconsiderado pelo testador ao projetar o oráculo de teste.

5.6 Limitações e ameaças à validade

Não existe na literatura uma taxonomia que considere defeitos ou falhas em oráculos de teste. À vista disso, os operadores foram baseados nos operadores de mutação de outras linguagens e domínios e isso pode ter sido notado na realização do estudo empírico e não refletido as necessidades específicas dos oráculos baseados em assertivas utilizados no estudo.

A quantidade de registros de programas que pudessem ser utilizados como objeto de estudo neste trabalho é pequena em relação a programas convencionais. Além disso, não existem *logs* de erros de implementação de oráculos, ou seja, não são conhecidas implementações incorretas dos oráculos. Portanto, selecionar os objetos de experimentação que possuíssem defeitos e validassem o presente trabalho de maneira satisfatória foi uma das limitações para a realização deste experimento.

As ameaças à validade deste estudo, podem ser analisadas por meio de quatro perspectivas diferentes:

Validade interna: O estudo é projetado com um escopo limitado – oráculos de teste baseados em assertivas. O experimento foi concebido para responder às nossas questões de pesquisa. Os resultados são consistentes para as suas respostas, levando a uma validade interna alta e aceitável.

Validade externa: O estudo avalia a efetividade dos operadores de mutação para oráculos de teste baseados em assertivas em cinco programas pequenos. No entanto, o experimento realizado não fornece resultados para supor que o comportamento da nossa técnica será o mesmo em sistemas reais no cenário industrial. É necessário mais trabalho neste contexto.

Validade construtiva: O conceito de mutação é útil em vários contextos, tornando a nossa validade construtiva alta. Assim, o tamanho e complexidade, das aplicações escolhidas são

adequados para mostrar a eficácia dos operadores de mutação para oráculos de teste baseado em assertivas.

Validade de conclusão: A metodologia utilizada foi apresentada em detalhes e nós estamos fornecendo o código da ferramenta ("MuJava 4 JUnit") que foi desenvolvida pela autora. Neste contexto, os resultados estão associados com a ferramenta, e portanto, alega-se que o estudo possui alta validade de conclusão.

5.7 Considerações finais

Este capítulo apresentou os resultados obtidos durante a realização do mestrado, apresentando o comportamento dos operadores específicos para oráculos de teste baseados em assertivas e a efetividade dos mesmos em melhorar os oráculos originais e sugerir novos casos de teste.

Como citado anteriormente, não existe na literatura uma taxonomia que considere defeitos ou falhas em oráculos de teste. Este fato limitou a validação empírica do trabalho, pois são encontrados poucos programas disponíveis para realizar os experimentos com oráculos de teste, tendo em vista que essa área começou a atrair pesquisadores recentemente. Diante desses fatos, foram realizadas apenas análises numéricas em relação aos mutantes gerados pelos operadores projetados e implementados para oráculos de teste no formato de assertivas. Diante desses fatos, foram realizadas apenas análises numéricas para oráculos de teste no formato de assertivas, em relação ao mutantes gerados pelos operadores projetados e implementados.

O próximo capítulo apresenta as principais conclusões tiradas por meio da realização deste trabalho. Além disso são apresentadas algumas limitações e ameaças à validade do estudo e, por fim, alguns trabalhos futuros a serem realizados.

CAPÍTULO

6

CONCLUSÕES

Para detectar falhas, o comportamento de execução do programa deve ser verificado em relação ao comportamento esperado. Esta verificação é normalmente realizada usando um oráculo de teste que verifica as mudanças de estado e a saída do programa em teste enquanto ele é executado. A precisão de um oráculo refere-se à tolerância do oráculo para falhas, e é normalmente o que determina seu custo (VOAS; MILLER, 1992).

Não foram encontradas na literatura formas sistemáticas de avaliar a qualidade ou a precisão de um oráculo de teste automatizado. Assim, é possível que em alguns casos os resultados da execução de um conjunto de testes apresentem resultados indesejados, não por problemas de dados de teste ou programa de teste, mas por erros na implementação do oráculo. Este estudo projeta operadores de mutação desenvolvidos para testar oráculos escritos no formato de assertivas.

6.1 Contribuições

Por meio deste trabalho de mestrado, operadores específicos para oráculos de teste escritos na forma de assertivas foram implementados e incluídos na ferramenta MuJava, gerando uma nova ferramenta, a "MuJava 4 JUnit". O experimento realizado neste estudo destaca o comportamento dos operadores quando aplicados a programas simples e de diferentes complexidades ciclomáticas. Os dados foram coletados de mutantes vivos e mortos, assim como o comportamento dos operadores propostos e implementados durante a realização deste trabalho.

Este trabalho contribui com uma das questões importantes do teste de software que é verificar a qualidade dos oráculos de teste. Durante a realização do trabalho vários obstáculos foram encontrados, pois a ideia de projetar operadores de mutação para oráculos é inédita e foi baseada principalmente em operadores desenvolvidos para outras linguagens e no conhecimento das assertivas e do *framework* JUnit.

Conclui-se que o uso de mutação colabora na melhoria da qualidade de oráculos de teste. O trabalho também contribui apresentando uma forma sistemática de avaliação da qualidade dos oráculos, que ainda não foi encontrada na literatura. Os operadores cumprem seu papel e pode-se observar o comportamento dos operadores em situações diversas em que a qualidade dos oráculos podem ser melhoradas quando esses operadores são utilizados.

6.2 Discussões

Os operadores da ferramenta "MuJava 4 JUnit" podem ser generalizados para outras APIs de teste unitários, oráculos diferentes dos escritos no formato de classe JUnit, mas que utilizam assertivas. O presente trabalho representa uma amortização de custo no desenvolvimento de novos operadores, pois o esforço é grande para implementar novos operadores de modo efetivo, dentro do contexto de outras ferramentas.

Operadores de mutação para oráculos de teste não têm uma alta taxa de geração de mutantes mortos, no entanto, podem revelar fraquezas no casos de teste, melhorando o oráculo original ou gerando novos casos de teste. Portanto, como foi observado na análise individual de cada operador de mutação, existem casos em que os mutantes vivos também devem ser examinados pois podem revelar fraquezas nos oráculos ou casos de teste originais.

Observou-se também que alguns operadores são mais efetivos que outros, alguns geram mais mutantes mortos que outros e dependendo do tipo de programa em teste, alguns operadores são mais recomendados que outros. Assim sendo, este trabalho também serve para guiar outros pesquisadores que possam querer projetar oráculos mais efetivos, podendo implementar operadores específicos de acordo com suas necessidades.

É importante salientar que os trabalhos encontrados na literatura que realizaram avaliação da qualidade de oráculos de teste não apresentaram nenhuma forma sistematizada, e cada avaliação foi feita dentro de um contexto específico. Já este trabalho, apresenta uma forma sistematizada de avaliar os oráculos de teste escritos na forma de assertivas por meio da utilização dos novos operadores de mutação específicos para oráculos de teste implementados na ferramenta "MuJava 4 JUnit".

6.3 Trabalhos futuros

A definição de operadores de mutação específicos para oráculos de teste escritos na forma de assertivas definiu o início de uma forma sistemática de avaliar oráculos de teste. No entanto, esta dissertação apresentou um estudo empírico utilizando cinco programas simples para verificar o comportamento dos operadores implementados na "MuJava 4 JUnit", e diante disso, como trabalhos futuros, umas das atividades é a realização de experimentos que utilizem

programas reais, projetos da indústria, que tenham defeitos conhecidos, em busca de afirmar os resultados obtidos durante a realização do presente trabalho.

Os operadores foram desenvolvidos com base em funções específicas do JUnit relacionadas à implementação dos oráculos dos casos de teste. A implementação da ferramenta "MuJava 4 JUnit" pode ser complementada com novas estratégias para os operadores já existentes. Atualmente, os valores de incremento do parâmetro delta, decremento do parâmetro delta e *timeout* devem ser definidos, obrigatoriamente, pelo testador. Todavia, estratégias podem ser definidas e complementar a implementação dos operadores já existentes auxiliando o testador em definir esses valores.

A mutação dos oráculos de teste produz alterações nas assertivas em que são geradas assertivas semelhantes a qualquer outro teste unitário. Os operadores implementados na "MuJava 4 JUnit" foram desenvolvidos adicionando ou removendo parâmetros dos métodos `asserts`, sendo que alterações em outras partes do caso de teste não são realizadas. Portanto, novos operadores podem ser desenvolvidos complementarmente, realizando alterações nas outras partes dos casos de teste, pois geralmente os desenvolvedores preferem executar novos casos de teste do que resolver todas as falhas que podem ocorrer durante a execução do teste (DANIEL *et al.*, 2009).

Um operador de mutação é um conjunto de instruções para gerar mutantes de um tipo particular (ANDREWS; BRIAND; LABICHE, 2005). Por exemplo, quando um operador de mutação de substituição da ferramenta "MuJava 4 JUnit" é utilizado, um novo oráculo de teste com outra assertiva é projetado. Na maioria das vezes, os operadores de mutação produzem mutantes que demonstram a necessidade de mais casos de teste (FRANKL; WEISS; HU, 1997).

Em programas "comuns", ao aplicar a mutação uma cópia modificada do código de implementação é criada contendo uma ou mais alterações com base em erros comuns do programador e/ou alterações que podem ter um impacto importante na forma como os componentes no sistema interagem (DEMILLO, 1980). Nos oráculos de teste os mutantes podem ser gerados analisando a forma de projeção do oráculo, assinaturas de métodos e parâmetros utilizados. Não existe forma definida de como realizar a mutação em oráculos de teste. Portanto, tendo como base o trabalho realizado nesta dissertação, pode-se gerar operadores de mutação para outros contextos além do JUnit.

A análise de mutação é útil não apenas para avaliar a eficácia da detecção de falhas de conjuntos de testes, mas também para avaliar a aplicação de novas abordagens de teste para domínios específicos. Como tal, a aplicação de novas abordagens em programas mutantes representa uma forma de avaliar os benefícios e desvantagens das técnicas (OLIVEIRA *et al.*, 2015). Assim sendo, a ideia de mutação pode ser estendida para outros tipos de oráculos e não somente naqueles baseados em assertivas.

A Seção 2.3.1 apresenta diversos tipos diferentes de taxonomias de oráculos que podem,

futuramente, utilizar a ideia de mutação para avaliar a qualidade dos oráculos de teste. Diferentes autores (BARESI; YOUNG, 2001; BEIZER, 1990; HARMAN *et al.*, 2013; STAATS; WHALEN; HEIMDAHL, 2011; MCDONALD; STROOPER, 1998; SHRESTHA; RUTHERFORD, 2011) definiram e classificaram taxonomias de oráculos de teste. Devido à diversidade de domínios de software, atualmente não existe taxonomia de teste de oráculo padronizada e as classificações existentes podem depender da fonte de informação, da saída do SUT ou do método de automatização considerado pelo oráculo. Portanto, estudos devem ser conduzidos para utilização da mutação para avaliação destes oráculos.

REFERÊNCIAS

AMMANN, P.; OFFUTT, J. **Introduction to Software Testing**. 1ª. ed. Cambridge, UK: Cambridge University Press, 2008. ISBN 0521880386, 9780521880381. Citado 3 vezes nas páginas 23 e 39.

ANDREWS, J. H.; BRIAND, L. C.; LABICHE, Y. Is mutation an appropriate tool for testing experiments? In: **Proceedings of the 27th International Conference on Software Engineering**. New York, NY, EUA: ACM, 2005. (ICSE '05), p. 402–411. ISBN 1-58113-963-2. Citado na página 91.

BARBOSA, E. F. **Uma Contribuição para a Determinação de um Conjunto Essencial de Operadores de Mutação no Teste de Programas C**. Dissertação (Mestrado) — ICMC-USP, São Carlos, SP, Novembro 1998. Citado na página 40.

BARBOSA, E. F.; MALDONADO, J. C.; VINCENZI, A. M. R.; DELAMARO, M. E.; SOUZA, S.; JINO, M. Introdução ao teste de software. **Minicurso apresentado no XIV Simpósio Brasileiro de Engenharia de Software**, 2000. Citado 2 vezes nas páginas 30 e 32.

BARESI, L.; YOUNG, M. Test oracles. **Technical Report CISTR-01**, University of Oregon, Dept. of Computer and Information Science, Eugene, Oregon, EUA, v. 2, p. 9, 2001. Citado 12 vezes nas páginas 24, 33, 35, 37, 38 e 92.

BARR, E.; HARMAN, M.; MCMINN, P.; SHAHBAZ, M.; YOO, S. The oracle problem in software testing: A survey. **IEEE Transactions on Software Engineering**, v. 41, n. 5, p. 507–525, 2014. Citado na página 40.

BEIZER, B. **Software Testing Techniques**. 2ª. ed. New York, NY, EUA: Van Nostrand Reinhold Co., 1990. 550 p. Citado 8 vezes nas páginas 25, 35 e 92.

BERTOLINO, A. Software Testing Research and Practice. In: **Proceedings of the Abstract State Machines 10th International Conference on Advances in Theory and Practice**. Berlin, Heidelberg: Springer-Verlag, 2003. (ASM'03), p. 1–21. Citado na página 29.

BUDD, T. A.; DEMILLO, R. A.; LIPTON, R. J.; SAYWARD, F. G. Theoretical and Empirical Studies on Using Program Mutation to Test the Functional Correctness of Programs. In: **Proceedings of the 7th Symposium on Principles of Programming Languages**. New York, NY, EUA: ACM, 1980. p. 220–233. Citado na página 34.

CHEON, Y. Abstraction in Assertion-Based Test Oracles. In: **Proceedings of the 7th International Conference on Quality Software, 2007**. Portland, Oregon, EUA: IEEE, 2007. p. 410–414. Citado 5 vezes nas páginas 48, 50 e 51.

CHEON, Y.; LEAVENS, G. T. A Simple and Practical Approach to Unit Testing: The JML and JUnit Way. Springer-Verlag, London, UK, p. 231–255, 2002. Citado na página 42.

CHOI, B.; DEMILLO, R.; KRAUSER, E.; MARTIN, R.; MATHUR, A.; OFFUTT, A.; PAN, H.; SPAFFORD, E. The Mothra tool set. In: **Proceedings of the 22nd Annual Hawaii International Conference on System Sciences**. West Lafayette, Indiana, EUA: IEEE, 1989. v. 2, p. 275–284 vol.2. Citado 5 vezes nas páginas 41 e 57.

COPELAND, L. **A practitioner's guide to software test design**. 1a. ed. Norwood, EUA: Artech House, 2004. Citado na página 31.

COPPIT, D.; HADDOX-SCHATZ, J. On the Use of Specification-Based Assertions as Test Oracles. In: **Proceedings of the 29th Annual IEEE/NASA Software Engineering Workshop**. Washington, DC, EUA: IEEE Computer Society, 2005. p. 305–314. Citado 5 vezes nas páginas 48 e 51.

DANIEL, B.; JAGANNATH, V.; DIG, D.; MARINOV, D. ReAssert: Suggesting Repairs for Broken Unit Tests. In: **2009 IEEE/ACM International Conference on Automated Software Engineering**. Washington, DC, USA: IEEE Computer Society, 2009. p. 433–444. Citado na página 91.

DELAMARO, M.; MALDONADO, J.; JINO, M. **Introdução ao teste de software**. 1a. ed. São Paulo, SP: Elsevier, 2007. 394 p. Citado 10 vezes nas páginas 23, 24, 25, 30, 31, 38, 39 e 42.

DELAMARO, M. E.; MALDONADO, J. C.; JINO, M.; CHAIM, M. L. Proteum: Uma Ferramenta de Teste Baseada na Análise de Mutantes. In: **Software Tools Session – VII Brazilian Symposium on Software Engineering**. Rio de Janeiro - RJ: SBES, 1993. p. 31–33. Citado 5 vezes nas páginas 41 e 57.

DELAMARO, M. E.; MALDONADO, J. C.; VINCENZI, A. M. R. Proteum/IM: Uma ferramenta de apoio ao teste de integração. In: **Software Tools Session – XI Brasilian Symposium on Software Engineering**. Fortaleza - CE - Brazil: SBES, 1997. Citado na página 41.

DELAMARO, M. E.; NUNES, F. d. L. d. S.; OLIVEIRA, R. A. P. de. Using concepts of content-based image retrieval to implement graphical testing oracles. **Software Testing, Verification and Reliability**, v. 23, n. 3, p. 171–198, 2013. Citado na página 34.

DELAMARO, M. E.; OFFUTT, J.; AMMANN, P. Designing Deletion Mutation Operators. In: IEEE. **Proceedings of the 7th International Conference on Software Testing, Verification and Validation**. Washington, DC, EUA, 2014. p. 11–20. Citado 5 vezes nas páginas 41 e 57.

DEMILLI, R.; OFFUTT, A. Constraint-based automatic test data generation. **IEEE Transactions on Software Engineering**, IEEE Press, Piscataway, NJ, EUA, v. 17, n. 9, p. 900–910, Setembro 1991. Citado na página 42.

DEMILLO, R. **Mutation Analysis as a Tool for Software Quality Assurance**. Georgia, EUA: School of Information and Computer Science, Georgia Institute of Technology, 1980. (GIT-ICS). Citado 3 vezes nas páginas 38 e 91.

DEMILLO, R. A.; LIPTON, R. J.; SAYWARD, F. G. Hints on test data selection: Help for the practicing programmer. **IEEE Computer Society Press**, IEEE Computer Society Press, Los Alamitos, CA, EUA, v. 11, n. 4, p. 34–41, 1978. Citado na página 38.

DURELLI, V. H. S.; ARAUJO, R. F.; SILVA, M. A. G.; OLIVEIRA, R. A. P. d.; MALDONADO, J. C.; DELAMARO, M. E. A scoping study on the 25 years of research into software testing in Brazil and an outlook on the future of the area. **Journal of Systems and Software**, Elsevier, v. 86, n. 4, p. 934–950, 2013. Citado na página 34.

DUSTIN, E.; GARRETT, T.; GAUF, B. **Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality**. 1ª. ed. Boston, EUA: Addison-Wesley Professional, 2009. 368 p. Citado na página 24.

FABBRI, S. C. P. F.; MALDONADO, J. C.; MASIERO, P. C.; DELAMARO, M. E.; WONG, E. W. Mutation Analysis Applied to Validate Specifications based on Petri Nets. In: **Proceedings of the 8th International Conference on Formal Description Techniques**. London, UK: Chapman & Hall, Ltd., 1995. p. 329–337. Citado na página 41.

FABBRI, S. C. P. F.; MALDONADO, J. C.; MASIERO, P. C.; DELAMARO, M. E. Proteum/FSM: A Tool to Suport Finite State Machine Validation. In: **Proceedings of the International Conference of the Chilean Computer Society**. Talca - Chile: IEEE Computer Society, 1999. Citado na página 41.

FABBRI, S. C. P. F.; MALDONADO, J. C.; SUGETA, T.; MASIERO, P. C. Mutation testing applied to validate specifications based on statecharts. In: IEEE. **Proceedings of the 10th International Symposium on Software Reliability Engineering**. Washington, DC, EUA: IEEE Computer Society, 1999. p. 210–219. Citado na página 41.

FANTINATO, M.; CUNHA, A. C. R. da; DIAS, S. V.; CARDOSO, S. A. M.; CUNHA, C. A. Q. AutoTest–Um framework reutilizável para a automação de teste funcional de software. **CPqD Tecnologia**, v. 1, n. 1, p. 119–131, 2005. Citado na página 33.

FRANKL, P. G.; WEISS, S. N.; HU, C. All-uses vs mutation testing: An experimental comparison of effectiveness. **Journal of Systems and Software**, Elsevier Science Inc., New York, NY, EUA, v. 38, n. 3, p. 235–253, set. 1997. Citado na página 91.

GONÇALVES, V. M.; NUNES, F. L.; DELAMARO, M. E.; OLIVEIRA, R. A. Avaliação de funções de similaridade em um framework de teste para programas com saídas gráficas. **XXXVII Conferência Latino Americana de Informática**, Quito, EQU, 2011. Citado na página 25.

HADDOX, J.; KAPFHAMMER, G. An approach for understanding and testing third party software components. In: IEEE. **Proceedings of the Reliability and Maintainability Symposium**. Seattle, Washington, EUA, 2002. p. 293–299. Citado na página 51.

HARMAN, M.; MCMINN, P.; SHAHBAZ, M.; YOO, S. **A Comprehensive Survey of Trends in Oracles for Software Testing**. Sheffield, South Yorkshire, England, 2013. Citado 11 vezes nas páginas 24, 25, 35, 37 e 92.

HOFFMAN, D. Using oracles in test automation. In: **Nineteenth Annual Pacific Northwest Software Quality Conference**. Portland, Oregon, EUA: Citeseer, 2001. p. 90–117. Citado na página 33.

HUNT, A.; THOMAS, D. **Pragmatic unit testing in Java with JUnit**. Raleigh, NC: Pragmatic Bookshelf, 2015. 163 p. Citado na página 42.

IEEE. IEEE Standard Glossary of Software Engineering Terminology. **IEEE Std 610.12-1990**, p. 1–84, Dec 1990. Citado na página 30.

IVORY, M. Y.; HEARST, M. A. The state of the art in automating usability evaluation of user interfaces. **ACM Computing Surveys**, ACM, New York, NY, EUA, v. 33, n. 4, p. 470–516, 2001. Citado na página 33.

JIA, Y.; HARMAN, M. An analysis and survey of the development of mutation testing. **IEEE Transactions on Software Engineering**, IEEE Press, Piscataway, NJ, EUA, v. 37, n. 5, p. 649–678, 2011. Citado na página 40.

KANER, C.; BACH, J.; PETTICHORD, B. **Lessons learned in software testing**. 1ª. ed. New York, NY, EUA: John Wiley & Sons, 2008. 360 p. Citado na página 32.

KIM-PARK, D. S.; RIVA, C. de la; TUYA, J. An Automated Test Oracle for XML Processing Programs. In: **Proceedings of the First International Workshop on Software Test Output Validation**. New York, NY, EUA: ACM, 2010. p. 5–12. Citado 6 vezes nas páginas 48 e 52.

KOSCIANSKI, A.; SOARES, M. dos S. **Qualidade de software: aprenda as metodologias e técnicas mais modernas para o desenvolvimento de software**. São Paulo, SP, BRA: Novatec Editora, 2007. 395 p. Citado na página 32.

MA, Y.-S.; OFFUTT, J.; KWON, Y. R. MuJava: an automated class mutation system. **Software Testing, Verification and Reliability**, Wiley Online Library, v. 15, n. 2, p. 97–133, 2005. Citado 5 vezes nas páginas 26 e 57.

MA, Y.-S.; OFFUTT, J.; KWON, Y.-R. MuJava: A Mutation System for Java. In: **Proceedings of the 28th International Conference on Software Engineering**. New York, NY, USA: ACM, 2006. p. 827–830. ISBN 1-59593-375-1. Citado 3 vezes nas páginas 41 e 44.

MALDONADO, J. C. **Critérios potenciais usos: Uma contribuição ao teste estrutural de software**. Tese (Doutorado) — Universidade de Campinas, Campinas, SP, 1991. Citado na página 32.

MATHUR, A. P. **Foundations of Software Testing**. 1st. ed. London, UK, UK: Pearson Education, 2008. 258 p. Citado na página 32.

MCDONALD, J.; STROOPER, P. Translating Object-Z specifications to passive test oracles. In: **2nd International Conference on Formal Engineering Methods**. Brisbane, Australia: IEEE Computer Society, 1998. p. 165–174. Citado 7 vezes nas páginas 35 e 92.

MYERS, G.; SANDLER, C.; BADGETT, T.; THOMAS, T. **The Art of Software Testing**. Nova Jérsia, EUA: Wiley, 2004. 224 p. (Business Data Processing: A Wiley Series). Citado 4 vezes nas páginas 23, 30 e 31.

NARDI, P. A. **On test oracles for Simulink-like models**. Tese (Doutorado) — Universidade de São Paulo, São Carlos, SP, 2013. Citado na página 33.

NUNES, F. L.; DELAMARO, M. E.; OLIVEIRA, R. A. Oráculo gráfico como apoio na avaliação de sistemas de auxílio ao diagnóstico. In: **IX Workshop de Informática Médica**. Bento Gonçalves, RS, Brasil: [s.n.], 2009. Citado na página 34.

OFFUTT, A. J. Investigations of the software testing coupling effect. **ACM Transactions on Software Engineering and Methodology**, ACM, New York, NY, EUA, v. 1, n. 1, p. 5–20, 1992. Citado na página 40.

OLIVEIRA, R. A.; DELAMARO, M. E.; NUNES, F. L. Estrutura para utilização de recuperação de imagens baseada em conteúdo em oráculos de teste de software com saída gráfica. In: **Anais do IV Workshop de Visão Computacional**. Bauru, SP, BRA: SBC, 2008. v. 1, p. 205–210. Citado na página 25.

OLIVEIRA, R. A.; KANEWALA, U.; NARDI, P. A. Automated test oracles: State of the art, taxonomies, and trends. **Advances In Computers, Vol 95**, ELSEVIER ACADEMIC PRESS INC 525 B STREET, SUITE 1900, SAN DIEGO, CA 92101-4495 USA, v. 95, p. 113–199, 2014. Citado 8 vezes nas páginas 24, 35, 37, 47, 53 e 54.

OLIVEIRA, R. A.; OLIVEIRA, L. B. R.; CAFEO, B. B. P.; DURELLI, V. H. Automating test oracles for systems with complex outputs. In: **Proceedings of the 22nd International Conference on Computers in Education**. Nara, Japão: Asia-Pacific Society for Computers in Education, 2014. Citado na página 42.

OLIVEIRA, R. A. P.; ALÉGROTH, E.; GAO, Z.; MEMON, A. Definition and evaluation of mutation operators for GUI-level mutation analysis. In: **2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)**. Graz, Austria: IEEE, 2015. p. 1–10. Citado na página 91.

OLIVEIRA, R. A. P. d. **Apoio à automatização de oráculos de teste para programas com interfaces gráficas**. Dissertação (Mestrado) — Universidade de São Paulo, São Carlos, SP, 2010. Citado na página 33.

ORZESZYNA, W. **Solutions to the equivalent mutants problem: A systematic review and comparative experiment.** Dissertação (Mestrado) — School of Computing Blekinge Institute of Technology, Estocolmo, Sweden, 2011. Citado na página 40.

PEZZÈ, M.; ZHANG, C. Automated test oracles: A survey. **Advances in Computers**, Academic Press, v. 95, p. 1–48, 2014. Citado 2 vezes nas páginas 34 e 36.

PRESSMAN, R. S. **Software Engineering: A Practitioner's Approach**. 6ª. ed. New York, EUA: McGraw Hill Brasil, 2005. 880 p. Citado na página 29.

ROCHA, A. R. C. d.; MALDONADO, J. C.; WEBER, K. C. **Qualidade de software: teoria e prática**. 2ª. ed. São Paulo, SP: Prentice Hall, 2001. 303 p. Citado na página 23.

SCHULER, D.; ZELLER, A. Assessing Oracle Quality with Checked Coverage. In: **Proceedings of the 4th International Conference on Software Testing, Verification and Validation**. Washington, DC, EUA: IEEE Computer Society, 2011. p. 90–99. Citado 5 vezes nas páginas 48, 52 e 53.

SHAHAMIRI, S.; KADIR, W.; MOHD-HASHIM, S. A comparative study on automated software test oracle methods. In: **Proceedings of the 4th International Conference on Software Engineering Advances**. Porto, Portugal: IEEE, 2009. p. 140–145. Citado 3 vezes nas páginas 48 e 50.

SHAHAMIRI, S. R.; KADIR, W. M. N. W.; IBRAHIM, S.; HASHIM, S. Z. M. An automated framework for software test oracle. **Information and Software Technology**, Butterworth-Heinemann, Newton, MA, EUA, v. 53, n. 7, p. 774–788, 2011. Citado 4 vezes nas páginas 48 e 49.

SHIN, D.; JEE, E.; BAE, D.-H. Empirical Evaluation on FBD Model-Based Test Coverage Criteria Using Mutation Analysis. In: **Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems**. Berlin, Heidelberg: Springer-Verlag, 2012. (MODELS'12), p. 465–479. Citado na página 31.

SHRESTHA, K.; RUTHERFORD, M. An Empirical Evaluation of Assertions as Oracles. In: **Proceedings of the 4th International Conference on Software Testing, Verification and Validation**. Washington, DC, EUA: IEEE Computer Society, 2011. p. 110–119. Citado 15 vezes nas páginas 25, 36, 48, 49 e 92.

SMITH, B. H.; WILLIAMS, L. An empirical evaluation of the mujava mutation operators. In: **Testing: Academic and Industrial Conference Practice and Research Techniques**. Windsor, UK: IEEE Computer Society, 2007. p. 193–202. Citado na página 44.

SOUZA, S. D. R. S. D.; MALDONADO, J. C.; FABBRI, S. C. P. F.; SOUZA, W. L. D. Mutation Testing Applied to Estelle Specifications. **Software Quality Control**, Kluwer Academic Publishers, Hingham, MA, EUA, v. 8, n. 4, p. 285–301, December 1999. Citado na página 41.

SOUZA, S. R. S.; SOUZA, P. S. L.; BRITO, M. A. S.; SIMAO, A. S.; ZALUSKA, E. J. Empirical evaluation of a new composite approach to the coverage criteria and reachability testing of concurrent programs. **Software Testing, Verification and Reliability**, 2015. Citado na página 31.

STAATS, M.; WHALEN, M.; HEIMDAHL, M. Programs, tests, and oracles: the foundations of testing revisited. In: **Proceedings of the 33rd International Conference on Software Engineering**. New York, NY, EUA: ACM, 2011. p. 391–400. Citado 7 vezes nas páginas 38 e 92.

TAN, R. P.; EDWARDS, S. H. Experiences evaluating the effectiveness of JML-JUnit testing. **ACM SIGSOFT Software Engineering Notes**, ACM, New York, NY, EUA, v. 29, n. 5, p. 1–4, 2004. Citado 8 vezes nas páginas 48 e 49.

TATSUBORI, M.; CHIBA, S.; ITANO, K.; KILLIJIAN, M.-O. OpenJava: A Class-Based Macro System for Java. In: **Proceedings of the 1st OOPSLA Workshop on Reflection and Software Engineering: Reflection and Software Engineering,**. London, UK, UK: Springer-Verlag, 2000. p. 117–133. Citado na página 77.

TRAKHTENBROT, M. New Mutations for Evaluation of Specification and Implementation Levels of Adequacy in Testing of Statecharts Models. In: **Proceedings of the 3rd Workshop on Mutation Analysis**. Windsor, UK: IEEE, 2007. p. 151–160. Citado na página 41.

VOAS, J. M.; MILLER, K. W. Improving the software development process using testability research. In: **[1992] Proceedings Third International Symposium on Software Reliability Engineering**. Arlington, VA, USA: IEEE, 1992. p. 114–121. Citado na página 89.

WAZLAWICK, R. S. **Engenharia de software: conceitos e práticas**. 1a. ed. São Paulo, SP: Campus, 2013. 360 p. Citado na página 30.

WEYUKER, E. J. On testing non-testable programs. **The Computer Journal**, Brazil Computer Society, v. 25, n. 4, p. 465–470, 1982. Citado 3 vezes nas páginas 33 e 38.

WOHLIN, C.; RUNESON, P.; HST, M.; OHLSSON, M. C.; REGNELL, B. **Experimentation in Software Engineering.** 2a. ed. Norwell, MA, EUA: Springer, 2012. 236 p. Citado na página 79.