UNIVERSIDADE ESTADUAL DE CAMPINAS

FACULDADE DE TECNOLOGIA

ISABEL CRISTINA SARTI SPROGIS

UM ESTUDO DA ANÁLISE DE MUTANTES E BENEFÍCIOS PARA O TESTE UNITÁRIO

LIMEIRA - SP

2020

ISABEL CRISTINA SARTI SPROGIS

UM ESTUDO DA ANÁLISE DE MUTANTES E BENEFÍCIOS PARA O TESTE UNITÁRIO

Monografia apresentada à Faculdade de Tecnologia da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Bacharel em Sistemas de Informação.

Orientador: Prof. Dr. PLÍNIO ROBERTO SOUZA VILELA

Este exemplar corresponde à versão final da Monografia defendida pela aluna Isabel Cristina Sarti Sprogis e orientada pelo Prof. Dr. Plínio Roberto Souza Vilela

LIMEIRA - SP

2020

A ship in harbor is safe,

but that is not what ships are built for.

(John A. Shedd)

AGRADECIMENTOS

Agradeço primeiramente a minha família, nesses tempos tão difíceis de pandemia, por me proporcionar encorajamento, força e companhia além de me ouvir e propor idéias e até me auxiliar a desenvolver melhor minhas idéias. Também pelos bons momentos que tivemos esse ano, pelas brincadeiras e bastante comilança e drinks para desestressar nos fins de semana só nós quatro. Eu amo vocês.

Quero agradecer aos meus amigos que sempre estiveram comigo me apoiando, mesmo que virtualmente neste ano. É um grande prazer ver todos nós crescendo como profissionais e adultos, nos ajudando uns aos outros também. Quero poder abraçar cada um de vocês de novo assim que der.

Agradeço também ao Professor Plínio não só pela orientação neste trabalho, mas também desde suas aulas de engenharia de software que solidificaram minha vontade de seguir carreira na área de testes de software e qualidade, e também na iniciação científica que se tornou a base deste trabalho. Sem sua sugestão de tema, que foi extremamente inusitada, nossas reuniões constantes e discussões dos experimentos este trabalho nunca teria sido possível.

Por fim agradeço a você leitor, que dedicou tempo para ler este trabalho, e espero que eu possa passar um pouco mais do conhecimento que obtive ao focar nesse tema.

RESUMO

Entre as diversas técnicas de testes de software podemos destacar as técnicas Funcionais, Estruturais e Baseadas em Erros. O Teste de Mutação é uma técnica Baseada em Erros com grande crescimento como objeto de estudos teóricos, muitas vezes para afirmar a qualidade do conjunto de testes. A Análise de Mutantes aplica um conjunto de regras de mutação ao programa em teste, essas regras promovem a geração de programas modificados chamados mutantes. Cada mutante possui apenas uma modificação sintática em relação ao programa original. O princípio de uso desta técnica como um critério de teste é avaliar a qualidade de um conjunto de teste com base na quantidade de mutantes que a execução do conjunto de teste consegue diferenciar do original. Para tanto executa-se o programa original com cada um dos dados de entrada presentes no conjunto de casos de teste e compara-se a saída obtida com a apresentada pela execução de cada mutante. Quando essa saída é diferente o mutante é considerado morto. Quanto mais mutantes mortos melhor o conjunto de teste. Este estudo tem por objetivo explorar as Técnicas de Mutação e Análise de Mutantes para deliberar a hipótese de que a Análise de Mutantes contribui com a elaboração e otimização dos casos de testes de um conjunto, pois imagina-se que os mutantes sobreviventes são capazes de identificar oportunidades de melhoria através da utilização de dados de entrada não triviais ou de novos casos capazes de matá-los. Para validar essa hipótese foi desenvolvida uma metodologia de pesquisa que consiste em dois experimentos: o primeiro para analisar os mutantes sobreviventes de um programa e buscar oportunidades de inserir novos casos de testes com entradas que os matem. E o segundo inserindo dados de entradas aleatórios em um conjunto de testes, que não produzia mutantes sobreviventes previamente na técnica de mutação com as entradas originais, e verificando se mutantes sobreviventes irrompem com os dados aleatórios. O operador de mutação *Changed conditional boundary* se sobressaiu

ao outros por ter mais mutações sobreviventes nos casos onde não é usado como entrada de dados os valores necessários para desvios de caminho. Ao final dos experimentos as entradas necessárias para se matar os mutantes obtidos foram melhor clarificadas, possibilitando novos casos de testes ou maior entendimento das entradas necessárias não triviais requeridas pelos programas submetidos sob a análise de mutação.

Palavras-Chave: Casos de Teste. Cobertura de Código. Engenharia de Software. Teste de Mutação. Teste de Software.

ABSTRACT

Among the various software testing techniques we can highlight Functional, Structural and Fault Based types. Mutation Testing is a Fault Based technique with increased growth as an object of academic studies, often used to assert the quality of a test suite. The Mutation Analysis technique implements a set of mutation rules to the program in test, those rules generate a version of the program which differs from the original by a single syntactic modification applied by the rule applied and is referred as a mutant. The main use of this technique as a test criteria is to evaluate the quality of a test suite based on the number of mutants that the execution of the test suite can distinguish from the original program. To achieve that both the original program is executed with the data present in the test suite and compared to the output obtained by the execution of each mutant. Whenever this output differs from the mutant it is considered dead. The more mutants are classified as dead the better is the test suite. This study aims to explore mutation technique and analysis to confirm the hypothesis in which mutation analysis contributes to the elaboration and optimization of test suites, since it's believed surviving mutants are capable of identifying opportunities of improvement through non trivial entry data usage or new cases capable of killing them. To validate this hypothesis, a research methodology was designed consisting of two experiments: the first to analyse the surviving mutants of a program and search for opportunities to insert new test cases with entries which kill the surviving mutants. And the second by inserting random data entries in a test suite, which didn't have any surviving mutants before with the original data, and verifying if surviving mutants erupt. The mutation operator *Changed conditional boundary* has stood out by having more surviving mutations in test cases which didn't had the necessary data entry to exercise deviations in the program path. At the end of the experiments the necessary data entry to kill the surviving mutants were better clarified, enabling new test cases or greater

understanding of necessary non trivial entries required by the programs submitted to mutation analysis.

Keywords: Test Cases. Code Coverage. Software Engineering. Mutation Testing. Software Testing.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1.INTRODUÇÃO

1.1 Contexto

A Engenharia de Software é uma disciplina que evoluiu significativamente nos últimos anos, procurando garantir a produção de software de qualidade que atenda às necessidades dos clientes e usuários, além de garantir um processo de desenvolvimento eficiente e eficaz. A atividade de testes tem grande importância em todas as partes do ciclo de desenvolvimento de software, tendo por objetivo identificar a ocorrência de defeitos, minimizar os riscos associados e assegurar a qualidade do software (Pressman, 2006).

Entre as técnicas utilizadas para testes de software temos as técnicas Funcionais, que estabelecem os requisitos de teste a partir exclusivamente da especificação, Estruturais, que estabelecem os critérios do teste baseado na estrutura interna (ou seja, a partir de informações da implementação do software), e Baseadas em Erros em que os requisitos de teste provêm do conhecimento de principais erros no desenvolvimento.

O Teste pode ser dividido em vários níveis: unitário, integração, validação e sistema. Neste trabalho será considerado somente o Teste Unitário. O Teste Unitário, também chamado de teste de componentes, é o teste das unidades mínimas do software de forma independente. Na orientação a objetos tais unidades individuais são métodos e objetos das classes. O teste unitário exercita tais unidades através de chamadas para os métodos passando diferentes dados de entrada e comparando o resultado obtido com o que é esperado segundo a especificação do software, validando assim o comportamento dos componentes do software (Firesmith, 1993).

Uma questão relevante nos estudos envolvendo teste de software é a de verificar quando testou-se suficientemente um software, ou, de forma análoga como definir a qualidade de um conjunto de testes que já não é mais capaz de identificar a presença de defeitos em um programa. É um paradoxo pois se o programa for bom o suficiente o teste não vai mais ser capaz de demonstrar que ele possui defeitos. Assim, se um

conjunto de testes não revela a presença de defeitos, como saber se esse conjunto de testes é ruim ou se o software é que já está bom o suficiente?

Para tanto, os critérios de teste podem ser usados como uma maneira de avaliar a qualidade do conjunto de teste, através do conceito de cobertura, e assim auxiliar a avaliar a qualidade do software de maneira indireta. É possível avaliar quanto do software aquele conjunto de testes foi capaz de exercitar, segundo algum critério escolhido. Caso esse valor seja alto o suficiente, pode-se ter uma garantia de que o conjunto de testes é bom e como ele não identifica mais defeitos, supõe-se que o software também esteja bom o suficiente (Pressman, 2006).

O Teste de Mutação (ou Análise de Mutantes) é uma técnica de teste Baseada em Erros, que necessita de um conjunto de testes unitários, a fim de verificar a qualidade do próprio conjunto de casos de teste (Soares, 2000).

A Análise de Mutantes proposta por DeMillo; Buddy; Lipton; Sayward (1978) funciona aplicando-se operadores de mutação (regras que aplicam uma determinada variação sintática) em um programa. Tais operações inserem uma modificação sintática no programa, criando para cada modificação uma nova versão do programa original chamada mutante. Cada mutante do programa deve conter apenas uma única[1] modificação sintática em relação ao programa original.

Ao executar novamente os casos de teste no programa com a alteração (dito mutante) duas situações são esperadas:

1. A saída da execução foi diferente da saída do programa original em pelo menos um dos casos de testes do conjunto, pois a modificação sintática presente no mutante deve produzir uma alteração semântica que modifique um operador para que ele tenha outro sentido e portanto introduza uma falha, que exercitada pelo caso de teste produz um resultado diferente do programa original, e o mutante é dito morto.

[1]Este é o caso da mutação de primeira ordem, variações da técnica que colocam mais de uma modificação sintática em cada mutante (segunda ordem, terceira ordem, etc) mas esses não serão abordados neste trabalho.

2. As saídas obtidas com a execução de todos os casos de testes é igual ao programa original, por isso o mutante é dito vivo.

Neste último caso, se nenhum caso de teste consegue mostrar essa diferença para um mutante ele é considerado sobrevivente e temos dois possíveis resultados: ou o mutante introduzido é equivalente semanticamente mesmo com a variação sintática ou os casos de testes não estão adequados. Há grande esforço gasto para determinar se mutantes são equivalentes, pois eles mantêm a semântica do programa inalterada atuando como falsos positivos e portanto, não conseguem ser detectados por nenhum conjunto de testes, não importa o quão bom seja (Jia; Harman, 2011).

1.2 Motivação

Existe uma conjectura de que a maior parte dos mutantes gerados pela aplicação da Análise de Mutantes seja relativamente fácil de matar. Ou seja, mesmo um conjunto de casos de teste relativamente fraco mataria a maior parte dos mutantes. Desta conjectura deriva-se o interesse em estudar aqueles mutantes que promovem realmente um incremento na qualidade do conjunto de casos de teste, ou seja, eles não são mortos pelos casos de teste mais simples, exigindo que se crie casos de teste que exercitem situações de execução mais específicas no programa ou entradas de dados específicas para eles. Esses casos de teste têm alta probabilidade de revelar a presença de defeitos ainda não descobertos no programa. Daí a importância da análise desse tipo de mutante e sua habilidade em agregar novos casos ou dados ao conjunto final de testes.

1.3 Objetivo

O objetivo desta pesquisa é utilizar a Análise de Mutantes como técnica para melhoria do conjunto de testes tanto na possibilidade de adicionar novos casos quanto corrigir casos de testes já existentes ou encontrar melhores entradas de dados que matem mutantes sobreviventes após a execução da Técnica de Mutação.

Para isto, este trabalho é separado em dois experimentos, sendo o primeiro deles com o objetivo em explorar e aprender a Análise de Mutantes com uma biblioteca open source e seu o próprio conjunto de testes, analisar quais operadores de mutação geram mais mutantes sobreviventes a fim de poder trazer resultados interessantes à técnica caso algum operador se sobressaia aos outros, e empenhar-se em utilizar os mutantes sobreviventes para identificar novos casos possíveis de serem adicionados ao conjunto de testes ou melhorá-los para aumentar sua qualidade.

O segundo experimento tem por objetivo utilizar a Análise de Mutantes como fator de identificação de melhores entradas de dados em um conjunto de testes, amparado pelo primeiro experimento no qual foi verificado que diversos mutantes sobrevivem pelo fato dos dados de entradas de alguns testes não terem sido bem escolhidos. Dessa forma será utilizado também um programa open source com seu conjunto de casos de testes e criada uma classe que gere conjuntos de entradas aleatórias para esses testes, e verificar se usando entradas aleatórias mutantes sobreviventes emergem.

A hipótese deste trabalho é que a Análise de Mutantes é uma boa técnica para atestar a qualidade do conjunto de testes, seja pela inclusão de novos casos, refatoração de casos existentes ou utilização de melhores entradas de dados proporcionados pelos mutantes sobreviventes após a análise individual de cada sobrevivente, principalmente os não equivalentes.

1.3 Organização do trabalho

As demais seções deste trabalho estão organizadas da seguinte forma:

No Capítulo 2 temos a Revisão Bibliográfica na qual apresenta-se o resultado do estudo do tema e diversos autores a fim de formar a base teórica necessária para desenvolver esta pesquisa. Inicia-se com o conceito de Engenharia e Testes de Software e Cobertura de testes. É apresentado o Critério baseado em Erros do qual a técnica de teste de mutantes faz parte antes de apresentar o que é a Análise de Mutantes e o problema dos Mutantes Equivalentes no contexto desta técnica.

No Capítulo 3 detalha-se a metodologia utilizada para o desenvolvimento da pesquisa, com a separação entre os dois experimentos entre suas ferramentas e escolha de programa, Extração de Mutantes e sua análise separada em cada experimento.

No Capítulo 4 temos os resultados dos experimentos detalhados nos mutantes sobreviventes obtidos, o porquê são considerados sobreviventes, e como possivelmente matá-los com novos casos de testes ou com entradas específicas.

O Capítulo 5 faz o fechamento do trabalho com as Conclusões discutindo os achados e se o objetivo proposto foi alcançado, as limitações e dificuldades encontradas e sugestões para trabalhos futuros.

2.REVISÃO BIBLIOGRÁFICA

Este capítulo apresenta as ideias, técnicas e discussões de autores a respeito do teste de software que fundamentam esta pesquisa. Começando com tipos e critérios de testes, conceito de cobertura de software e fluxo de controle, continuando com a técnica de testes de mutante proposto por DeMillo; Buddy; Lipton; Sayward (1978) e por fim a existência de mutantes equivalentes e seu impacto na Análise de Mutantes.

2.1 Engenharia e testes de software

Na disciplina de Engenharia de Software, qualidade de software é um conceito que pode ser definido como "*uma gestão de qualidade efetiva aplicada de modo a criar um produto útil que forneça valor mensurável para aqueles que o produzem e para aqueles que o utilizam*" (Pressman, 2006, p.360). Com o intuito de assegurar a qualidade do software, é necessário estabelecer critérios e se utilizar de técnicas conhecidas para garantir tal propriedade.

O teste de Software é uma atividade imprescindível em qualquer tipo de ciclo de desenvolvimento de software, e seu objetivo primário é garantir que não haja erros que serão entregues ao cliente junto do produto. Segundo (Pressman, 2006, p.428) "*Para encontrar o maior número possível de erros, devem ser executados testes sistematicamente, e os casos de teste devem ser projetados usando técnicas disciplinadas*". Dessa forma os testes expõem falhas de funcionamento do software que são registradas, analisadas para saber sua origem, a correção de sua causa é realizada e em seguida é necessário testar-se novamente para verificar que a falha não mais ocorre.

As técnicas usadas para testes podem ser separadas em três tipos: Funcionais, Estruturais e Baseadas em Erros. As técnicas Funcionais, também conhecidas como caixa-preta, estabelecem os requisitos dos testes a partir da especificação do produto, sem conhecer seu funcionamento interno do código, para determinar se o programa satisfaz seus requisitos funcionais e não funcionais. As técnicas Estruturais, também chamadas de caixa-branca, necessitam de conhecimento da estrutura lógica interna do

código para que possa-se exercitar os caminhos independentes, decisões de verdadeiros e falsos, ciclos e estruturas de dados. As técnicas Baseadas em Erros criam seus testes através do conhecimento dos principais erros no desenvolvimento do software (Soares, 2000).

Dentre essas três técnicas o teste de software também é dividido, segundo Firesmith (1993) entre teste unitário, integração, validação e de sistema. O teste unitário (ou teste de componentes) é o teste de unidades individuais, e que na orientação a objetos significa os métodos e objetos das classes. Nos testes unitários são passados valores de entrada para os métodos segundo os requisitos do software e validado o comportamento com base nesses dados de entradas. O teste unitário pode ser tanto de caixa-preta validando apenas entradas e saídas esperadas quanto mensagens de resposta, quanto de caixa-branca nos quais validam-se interações entre atributos e operações, caminhos lógicos como condicionais ou ciclos de repetições.

O teste de integração é o teste da junção das unidades como um conjunto. Na orientação a objetos, isso significa testar diversos objetos e classes funcionando em conjunto, principalmente regras de herança e polimorfismo. O teste de validação é para validar se o software atende aos requisitos funcionais e não funcionais de sua especificação que foram levantados na fase de análise de requisitos. O teste de sistema significa testar o sistema completo, principalmente envolvendo integração entre software e hardware (Firesmith 1993).

2.2 Cobertura de software e fluxo de controle

A pergunta principal no trabalho de testes, no entanto, é: como podemos saber que já testamos o suficiente nosso software? Para se ter a deliberação de que o conjunto de testes é adequado e suficiente, existem métricas para cada uma das técnicas de testes.

No caso de testes estruturais, os quais são baseados na estrutura interna do programa, o objetivo é exercitar todos os possíveis caminhos da estrutura desse programa (ou outros elementos estruturais dependendo do critério). Para facilitar o

mapeamento de tais caminhos utiliza-se normalmente um grafo de fluxo de controle. De acordo com Vilela (1998), o grafo de fluxo de controle é um grafo dirigido que contém um único nó de entrada e um único nó de saída, cada nó representa uma sequência de comandos a serem executados em conjunto como um bloco (sem desvios de fluxo de controle), e cada arco é um desvio lógico do programa, ou seja a transferência de controle entre esses blocos. Podemos dizer também que um caminho é uma sequência começando no nó de entrada através de uma sequência de arcos até o nó de saída (Vilela, 1998)

Três critérios existem a partir do grafo de fluxo de controle do programa: Todos os Nós, Todos os Arcos e Todos os Caminhos. O Critério Todos os Nós requer que todos os Nós sejam exercitados ao menos uma vez. O Critério Todos os Arcos requer que todos os arcos - transferências de controle entre nós - sejam exercitados ao menos uma vez. E o Critério Todos os Caminhos requer que todos os caminhos lógicos possíveis do programa sejam exercitados.

A métrica de Complexidade Ciclomática foi elucidada por Thomas McCabe (1976) para mensurar a complexidade de um programa através do grafo de fluxo de controle. A Complexidade Ciclomática indica a complexidade e se relaciona com o número de caminhos independentes de um programa.

McCabe (1976) também apresenta uma metodologia para criar testes por meio da visualização trazida pelo grafo de fluxo de controle, os caminhos e nós identificados se tornam os requisitos de teste: assume-se que dado um programa P, que tenha sua complexidade ciclomática V calculada através da utilização do grafo de fluxo de controle igual ao número de caminhos independentes do programa, e com um número AC de caminhos testados pelo conjunto de testes. Se o número de caminhos AC testados pelo conjunto de testes é menor que a complexidade ciclomática V umas das seguintes definições é verdadeira:

- É necessário adicionar mais casos de testes até que todos os caminhos sejam cobertos, e AC seja igual a V.

- O grafo de fluxo de controle pode ser simplificado em sua complexidade levando em conta os caminhos testados AC.
- Partes do programa P podem ser simplificadas (a complexidade pode ter aumentado para economizar espaço).

Através das métricas de Complexidade Ciclomática, do grafo de fluxo de controle, e dos critérios Todos os Nós, Todos os Arcos e Todos os Caminhos se torna mais fácil elaborar conjuntos de casos de testes de forma tal que todas as instruções sejam executadas pelo menos uma vez durante o teste unitário.

2.3 Critérios baseados em erros

A técnica de testes de software baseada em erros, como mencionado na seção anterior, enfatiza erros que podem ser cometidos pelo desenvolvedor durante o desenvolvimento do software. Nesta técnica existem dois critérios que podem ser adotados para buscar tais erros, são elas a Análise de Mutantes e Semeadura de Erros (Soares, 2000).

A Análise de Mutantes, mais detalhada na próxima seção, é um critério no qual o programa passa por mudanças sintáticas através de operadores de mutação, obtendo um conjunto de novos programas modificados chamados mutantes. Cada mutante contém apenas uma modificação sintática comparado ao original. Depois de se testar com um conjunto de testes nos mutantes é possível encontrar neste conjunto casos de testes que possam evidenciar as diferenças entre o mutante e o programa original.

A Semeadura de Erros é o critério no qual, ao se inserir erros no programa, faz-se uma análise durante os testes de quais erros obtidos eram os inseridos e quais eram naturais ao programa. Dessa forma têm-se a métrica de número de erros semeados com número de erros naturais revelados e uma indicação de quantos erros naturais restantes existem.

2.4 Análise de mutantes.

A Análise de Mutantes proposta por DeMillo; Buddy; Lipton; Sayward (1978) baseia-se em aplicar variações sintáticas de operadores em um programa para que erros sejam introduzidos através dos mutantes criados por essas variações. Deve-se aplicar o conjunto de casos de testes para verificar essa diferença criada e se necessário criar novos casos para evidenciar as diferenças entre os mutantes e o programa original.

Mais detalhadamente, segundo DeMillo; Buddy; Lipton; Sayward (1978), o processo de análise de mutação é descrito como: dado um programa P e um conjunto de testes T considerado adequado (já foi aplicado o conjunto T ao programa P que parece estar correto), é então submetido o programa P a diversos tipos de operadores de mutação para gerar programas diferentes de P em apenas uma mudança sintática denominados mutantes. O conjunto de casos de teste T é então aplicado para cada mutante novamente e analisado o resultado. Caso todos os mutantes morram, infere-se que o conjunto de testes T é um bom conjunto de testes e o programa P tem qualidade. No entanto se alguns mutantes não forem mortos quando passarem novamente pelo casos de testes temos duas possíveis condições: (1) o mutante é equivalente ao programa P original mesmo após a mudança pelo operador de mutação; ou (2) o conjunto de casos de teste T está inadequado e não contém um caso de teste que evidencie a diferença entre o mutante e o programa original P.

No fim, o conjunto de testes T só é considerado adequado quando todos os mutantes vivos considerados não equivalentes são mortos através da adição de novos casos em T que possam diferenciar o mutante de P.

Outra hipótese fundamental da técnica de Análise de Mutantes é o Efeito de Acoplamento, que afirma que os dados usados nos testes nos quais mutantes simples são mortos são tão sensíveis que é implícito que esses dados também são capazes de matar mutantes mais complexos, pois estes em sua maioria, são compostos de mutantes mais simples (Jia; Harman, 2011).

2.5 Mutantes equivalentes

A parte de análise dos mutantes vivos é o passo com maior esforço necessário pois é preciso analisar individualmente cada mutante com relação ao programa original a fim de separar os mutantes equivalentes dos não equivalentes ao programa original. Mutantes equivalentes são mutantes que, apesar da mudança sintática com o operador de mutação, mantém a semântica do programa original.

Grun; Schuler; Zeller (2009) definem o problema dos mutantes equivalentes na técnica de mutação de software como falsos positivos, pois apesar de parecerem com um mutante sobrevivente, não existe caso de teste que seja possível de matá-los. O maior problema é que os mutantes equivalentes acabam por popular o conjunto de resultados, e o trabalho de localizá-los e retirá-los do conjunto é um esforço completamente manual para definir se a alteração sintática pelo operador de mutação gerou uma diferença semântica ou se no fim o mutante é semelhante ao programa original.

Soares (2000) completa afirmando que não existe uma solução algorítmica completa para o problema de equivalência de mutação. Isto é, determinar a equivalência entre dois programas ou dois mutantes é um problema indecidível. E o esforço necessário baseia-se no conhecimento do programa, da linguagem e das técnicas de testes.

Tais mutantes equivalentes devem ser retirados do conjunto de mutantes finais na Análise de Mutação pois, como falsos positivos, eles não contribuem com a melhoria do conjunto de testes ou do programa.

2.6 Considerações finais

Neste capítulo foram apresentados os conceitos base deste trabalho amparados em autores e suas publicações prévias no assunto. O processo de teste de software é uma ciência guiada por diversas técnicas e critérios. Cada uma dessas técnicas e critérios nos apresentam visões diferentes do software e possibilitam encontrar

diferentes falhas ou pontos problemáticos, buscando sempre melhorar a qualidade do que já existia.

A Análise de Mutantes já assume que exista um programa e seu conjunto de testes, logo seu foco é melhorar o conjunto de testes tanto quanto o código do programa. Por isso traz uma visão diferenciada na gestão de qualidade do processo de criação de software.

Contudo, a técnica tem sua própria dificuldade: os mutantes equivalentes. Diferenciar o mutante gerado pela aplicação do operador de mutação do programa original no sentido semântico não é uma tarefa atualmente automatizável, sendo necessário esforço humano para fazer a diferenciação e classificação.

3.METODOLOGIA

Este capítulo descreve as atividades, ferramentas e experimentos realizados segundo a proposta do trabalho, amparado pelas discussões e definições dos autores descritas na revisão bibliográfica. As tarefas de escolher programas base, ferramentas para realizar a análise de mutante, obter relatórios e tabelas e a própria análise são aqui pormenorizadas.

3.1 Organização das atividades

O foco da pesquisa é experimental, buscando se utilizar da técnica de Análise de Mutantes para trazer maior qualidade a conjuntos de testes através da busca de novos casos a serem adicionados ou melhores entradas de dados proporcionados pela análise dos mutantes sobreviventes.

Tendo este objetivo, foram realizados dois experimentos para achar conjuntos de mutantes sobreviventes e analisá-los. O primeiro experimento descrito na seção 3.2, com foco de encontrar os mutantes sobreviventes de um programa, e caso não sejam equivalentes, encontrar um meio de matá-los através de um novo caso ou melhor entrada de dados. E o segundo descrito na seção 3.3, usando dados de entradas aleatórios em um conjunto de testes, conjunto no qual existiam mutantes sobreviventes previamente na técnica de mutação com suas entradas originais, e verificando se mutantes sobreviventes irrompem com os dados aleatórios.

Entre as atividades comuns podemos destacar (1) Pesquisa Bibliográfica para embasamento teórico em testes unitários de aplicações orientadas à objetos, nas técnicas de mutação e Análise de Mutantes (2) Escolha de Ferramentas de mutação, programa em Java que atenda aos objetivos do experimento e um conjunto de testes unitários para tal programa.

No entanto as atividades de experimentação e análise foram particulares de cada experimento, e serão melhor descritas nas seções 3.2 e 3.3 respectivamente. Porém, seguiram uma base que pode ser descrita em (3) Pesquisa Experimental aplicando a

ferramenta de mutação e coletando todos mutantes sobreviventes e (4) Análise Experimental dos resultados com base no objetivo de cada experimento.

3.2 Experimento 1: Calendário Persa

O primeiro experimento planejado tem o objetivo de usar a Técnica de Mutação para conseguir obter possíveis mutantes sobreviventes de um programa escolhido que seja complexo, com pelo menos mais de 3 classes envolvidas e com diversas funções matemáticas, tipos de retornos e caminhos lógicos de forma que tenhamos uma maior quantidade de operadores de mutação aplicados. Também precisamos de um conjunto de testes para essa aplicação para poder usar a Técnica de Mutação, e seria um diferencial se o conjunto já existir e tiver uma boa cobertura, apesar de não precisar ser completa nesse experimento, pois queremos achar espaço para melhoria através da Análise de Mutantes.

O foco do experimento é poder identificar oportunidades de acrescentar ao conjunto de testes existentes melhores casos ou melhores entradas não pensados pelo programador.

3.2.1 Escolha do programa e das ferramentas

Nesta seção trata do programa escolhido e das ferramentas utilizadas para se preparar o experimento, tanto na garantia de cobertura pelo conjunto, análise do programa original e o mais importante qual a ferramenta de mutação escolhida.

Para o primeiro experimento foi necessário escolher uma aplicação Java Open Source robusta como estudo de caso, para que os mais diversos operadores de mutação pudessem ser exercitados. Vários programas em Java foram pesquisados pela aluna através do site Github, entre eles havia a Torre de Hanói, um Cronômetro, a Cifra de César e o Calendário Persa. O Calendário Persa se sobressaiu aos demais por aparentar ser complexo computacionalmente porém de fácil entendimento ao ler e estar bem documentado, além de conter um extenso conjunto de testes escritos pelo próprio programador. Por isso escolheu-se utilizar a biblioteca open source de funções em Java

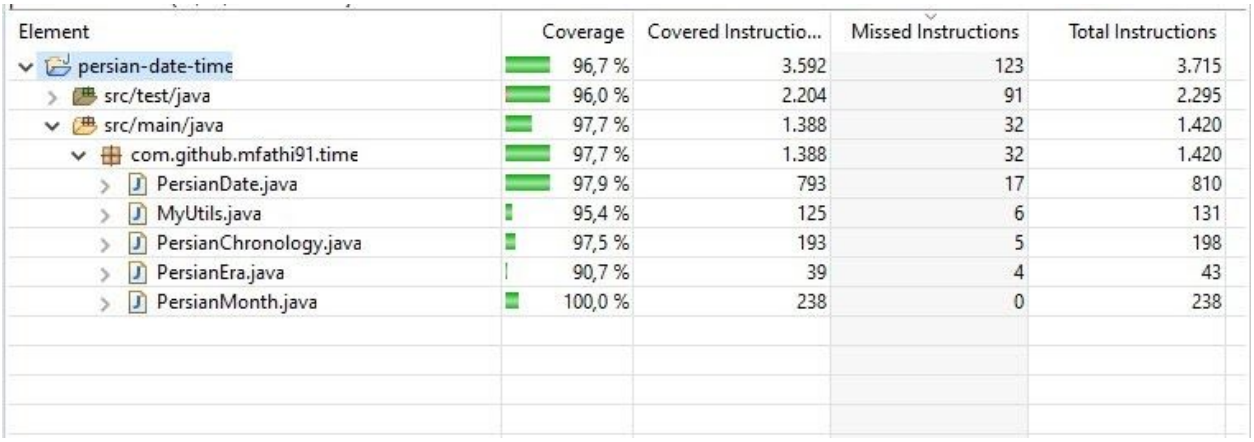
para implementar o Calendário Persa: https://github.com/mfathi91/persian-date-time (Fathi, 2019).

Para entender como funciona a conversão do Calendário Persa e para nosso calendário Gregoriano foi usado o site de conversão de calendários Fourmilab (FOURMILAB, 2020). Este site além de ter um conversor online também explica as diferenças e como funciona o cálculo das datas, ciclos e épocas.

O diferencial dessa biblioteca é que, além de ser grande, contendo 1.231 linhas de código divididas entre 5 classes e que portanto tem capacidade de exercitar mais operadores de mutações diferentes, ela contém um conjunto de 132 testes unitários escritos em JUnit que abrangem todos os métodos de cada classe e por vezes testam mais de uma vez cada método, o qual tornou-se o conjunto base deste primeiro experimento. O mais importante de ter tal conjunto de testes é que primeiramente ele foi escrito pelo programador que já conhece o programa, de forma que temos então um conjunto original para poder testar sua qualidade com a Técnica de Mutação e tentar achar novas oportunidades para melhorá-lo além do que o programador conseguiu. Segundo, isso auxilia muito no tempo e dificuldade do trabalho ao invés de ter que escrever testes unitários para um programa ao qual não escrevemos.

A Figura 1 apresenta a cobertura de código dos casos de testes pela ferramenta JaCoCo (https://www.eclemma.org/jacoco/) (JaCoCo, 2020) apresentando 97% do código sendo coberto pelo conjunto de testes que ele já tinha. Todas as classes e métodos importantes foram abrangidos tanto em atributos quanto operações para o teste de Caixa Branca, além de mensagens e exceções para testes de Caixa Preta (Firesmith, 1993).

A cobertura de testes é verificada pela ferramenta JaCoCo, um plugin instalado na IDE Eclipse, que realiza o teste de cobertura e sinaliza áreas cobertas (em verde) ou que não foram cobertas pelos casos (em vermelho), além da percentagem total de todos os códigos, e que exporta todos os dados no formato HTML para documentação.



Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
persian-date-time	96,7 %	3.592	123	3.715
src/test/java	96,0 %	2.204	91	2.295
src/main/java	97,7 %	1.388	32	1.420
com.github.mfathi91.time	97,7 %	1.388	32	1.420
PersianDate.java	97,9 %	793	17	810
MyUtils.java	95,4 %	125	6	131
PersianChronology.java	97,5 %	193	5	198
PersianEra.java	90,7 %	39	4	43
PersianMonth.java	100,0 %	238	0	238

Figura 1: Resultado da cobertura em percentual total do código da biblioteca persa pelo conjunto de casos de testes originais através da ferramenta JaCoCo.

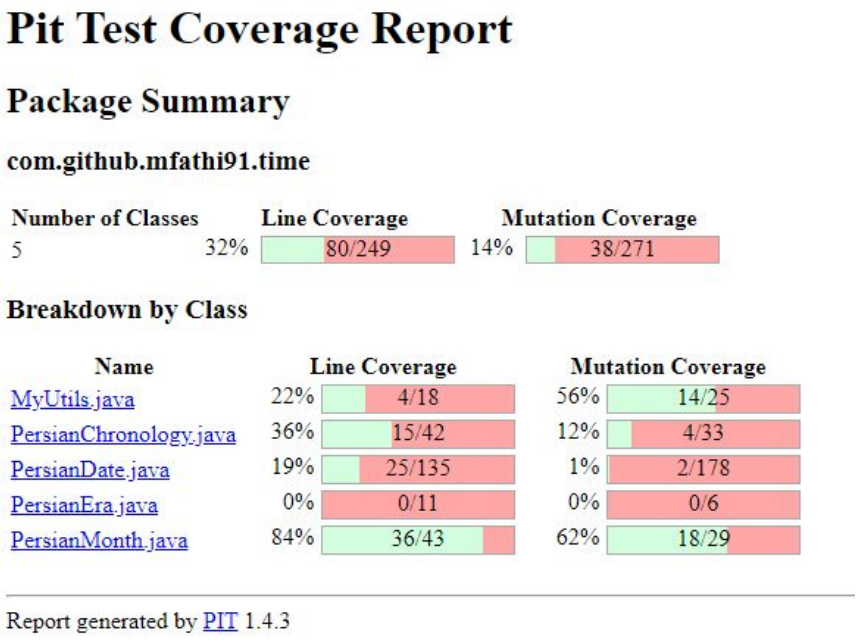
Para estudar a técnica de testes de mutação foi necessária a escolha de uma ferramenta que pudesse automaticamente propagar as mutações baseadas em conjuntos de operadores diferentes no código de um programa escolhido. Após pesquisar diversas ferramentas foi escolhido o PITEST (*https://pitest.org/*) (PITEST, 2019) como ferramenta de mutação por sua facilidade de apresentar as mutações aplicadas no código e o resultado de sua sobrevivência quando os testes são executados novamente através de relatórios HTML e XML gerados. A mutação vive (*survived* e marcado em rosa) se os casos de testes não conseguirem mostrar a diferença do programa original, ou morre (*killed* e marcado em verde) quando os testes falham ao passar pela mutação. Pode-se ver tal notação na Figura 2 e Figura 3.

Imediatamente após a finalização da escolha dos objetos de estudos e suas ferramentas foi necessário desenvolver a pesquisa de forma que obtivéssemos um conjunto de mutações aplicadas pelo PITEST no código, e que tais mutações pudessem ser classificadas entre elas de acordo com sua sobrevivência segundo a técnica de Análise de Mutantes, para assim poder discorrer das características dos mutantes sobreviventes encontrados nos resultados.

3.2.2 Técnica de mutação e extração de mutantes

Esta seção explica o processo de aplicação da técnica de mutação no programa através da ferramenta PITEST, e como foi feita a agregação de dados em uma tabela.

A Pesquisa Experimental começa por aplicar a ferramenta de mutação na biblioteca e submeter, nas diversas mutações realizadas, apenas um único dos casos de testes unitários do conjunto. O PITEST automaticamente faz a aplicação dos operadores de mutação e submete ao conjunto de testes, retornando o relatório dos resultados da Técnica de Mutação. Dessa forma, podemos analisar por todo o conjunto de mutações sofridas quais delas sobrevivem e para quais casos de testes. Cada relatório HTML e XML gerado foi salvo manualmente em pastas separadas, com o nome do teste unitário que submeteu-se ao programa após mutações.



Pit Test Coverage Report

Package Summary

com.github.mfathi91.time

Number of Classes	Line Coverage		Mutation Coverage	
5	32%	80/249	14%	38/271

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
MyUtils.java	22%	4/18	56%	14/25
PersianChronology.java	36%	15/42	12%	4/33
PersianDate.java	19%	25/135	1%	2/178
PersianEra.java	0%	0/11	0%	0/6
PersianMonth.java	84%	36/43	62%	18/29

Report generated by PIT 1.4.3

Figura 2: Relatório automático HTML do Persian Calendar gerado pelo PITEST no caso de teste testOnIsEqual. Visão da cobertura de linhas e mutações de todas as classes.

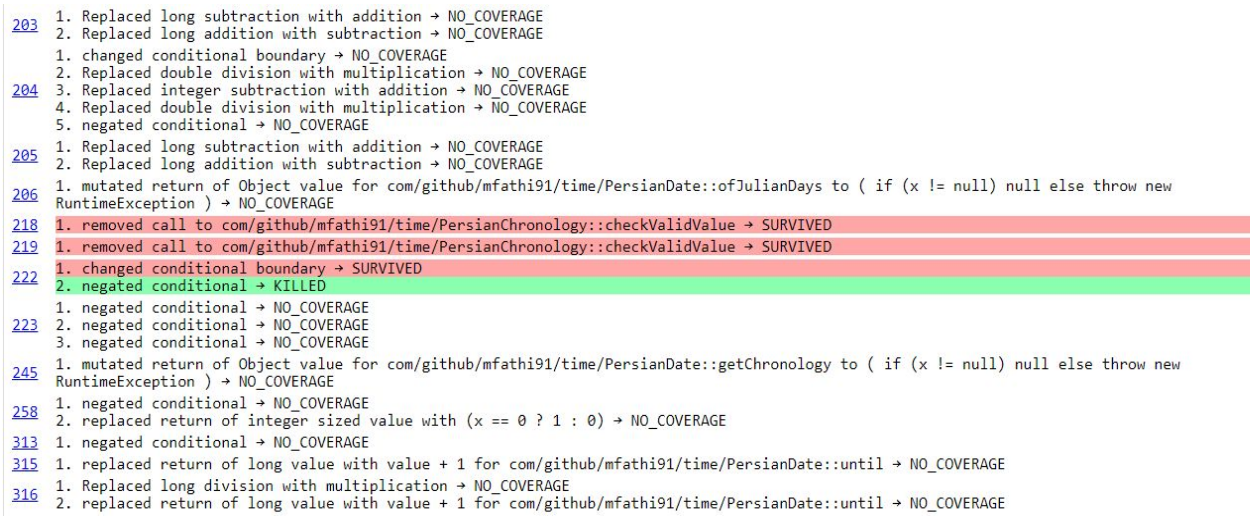


Figura 3: Relatório automático HTML do Persian Calendar gerado pelo PITEST no caso de testes testOnIsEqual. Visão do resultado das mutações nas linhas da classe Persian Date.

A ferramenta PITEST foi parametrizada para cada um dos 132 casos de testes aplicados individualmente para cada mutante gerado. Ao final do processo foi criado pela ferramenta um arquivo XML com as informações mostradas a seguir:

1. detected: indica se o mutante foi detectado (Verdadeiro = detectado / Falso = Não detectado)
2. status: indica o status de sobrevivência do mutante (SURVIVED=vivo; KILLED=morto; NO_COVERAGE= não coberto pelo caso de teste).
3. numberOfTestsRun: número de testes executados contra o mutante.
4. sourceFile: nome do arquivo JAVA que sofreu a mutação.
5. mutatedClass: nome da classe que sofreu a mutação.
6. mutatedMethod: nome do método que sofreu a mutação.
7. methodDescription: descrição do método.
8. lineNumber: número da linha que sofreu a mutação.
9. Mutator: nome da classe do operador de mutação aplicado.
10. Index: índice de controle (não utilizado).
11. Block: bloco de controle (não utilizado).
12. killingTests: relação dos casos de teste que mataram o mutante.

13. succedingTests: outros casos de testes executados e que não mataram o mutante.
14. Description: descrição do operador de mutante aplicado.

Em seguida, o arquivo XML gerado foi transformado em uma planilha Excel. Ao final do processo, pode-se trabalhar com várias perspectivas dos mutantes obtidos, como as apresentadas a seguir:

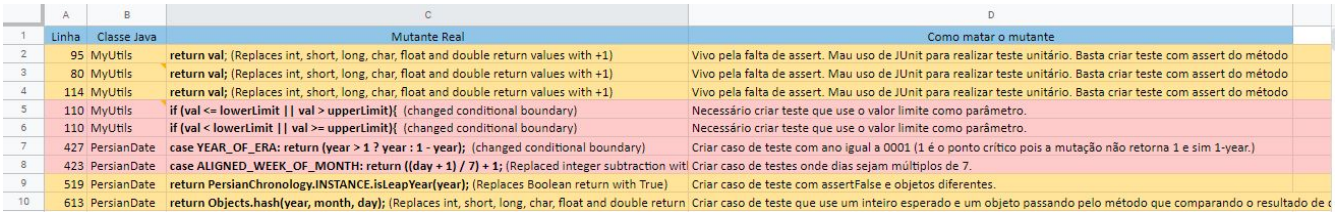
- Quais mutantes sobreviveram ao conjunto de testes?
- Existe uma classe de operadores de mutação que tem mais mutantes sobreviventes não equivalentes?
- Existe uma classe de operadores de mutação que tem mais mutantes mortos, ou seja vários casos de teste são capazes de matá-los?

3.2.3 Compilação da tabela de mutantes

O relatório dinâmico descrito acima foi apresentado em uma tabela final com duas abas, a qual está disponibilizada no link respectivo na seção 3.4. A primeira aba contém apenas os mutantes que sobreviveram dividida em 7 colunas: (1) Quantas vezes aquela mutação sobreviveu, (2) a classe Java que pertence a mutação, (3) a linha de código na classe respectiva, (3) uma cópia linha de código para referência, (4) a mutação sofrida, (5) o operador de mutação aplicado, (6) Comentários da análise feita e (7) Classificação final após a análise manual feita pela autora como M - Mutante ou E - Equivalente.

É necessário ressaltar que as colunas 1 a 6 foram resultados da Ferramenta PITEST, porém a coluna 7, que é a Classificação, foi um processo de análise realizada pela autora para classificar em Mutante (M) ou Equivalente (E). O critério de análise feita foi baseado na diferença semântica entre o código original e o mutado. Caso fosse percebido que o mutante tivesse a mesma semântica que a linha de código original, ou que era impossível criar algum teste que conseguisse diferenciar o programa original do mutante, a classificação é Equivalente.

A segunda aba, Figura 4, tem o descritivo dos mutantes sobreviventes não equivalentes e a forma encontrada de como matar tal mutante, baseada na análise mencionada.



	A	B	C	D
1	Linha	Classe Java	Mutante Real	Como matar o mutante
2	95	MyUtils	**return val**; (Replaces int, short, long, char, float and double return values with +1)	Vivo pela falta de assert. Mau uso de JUnit para realizar teste unitário. Basta criar teste com assert do método
3	80	MyUtils	**return val;** (Replaces int, short, long, char, float and double return values with +1)	Vivo pela falta de assert. Mau uso de JUnit para realizar teste unitário. Basta criar teste com assert do método
4	114	MyUtils	**return val;** (Replaces int, short, long, char, float and double return values with +1)	Vivo pela falta de assert. Mau uso de JUnit para realizar teste unitário. Basta criar teste com assert do método
5	110	MyUtils	**if (val <= lowerLimit || val > upperLimit){** (changed conditional boundary)	Necessário criar teste que use o valor limite como parâmetro.
6	110	MyUtils	**if (val < lowerLimit || val >= upperLimit){** (changed conditional boundary)	Necessário criar teste que use o valor limite como parâmetro.
7	427	PersianDate	**case YEAR_OF_ERA: return (year > 1 ? year : 1 - year);** (changed conditional boundary)	Criar caso de teste com ano igual a 0001 (1 é o ponto crítico pois a mutação não retorna 1 e sim 1-year.)
8	423	PersianDate	**case ALIGNED_WEEK_OF_MONTH: return ((day + 1) / 7) + 1;** (Replaced integer subtraction with	Criar caso de testes onde dias sejam múltiplos de 7.
9	519	PersianDate	**return PersianChronology.INSTANCE.isLeapYear(year);** (Replaces Boolean return with True)	Criar caso de teste com assertFalse e objetos diferentes.
10	613	PersianDate	**return Objects.hash(year, month, day);** (Replaces int, short, long, char, float and double return	Criar caso de teste que use um inteiro esperado e um objeto passando pelo método que comparando o resultado de

Figura 4: Tabela de Mutantes sobreviventes do experimento Calendário Persa na segunda aba, apenas os mutantes sobreviventes não equivalentes detalhados, e descrição de como escrever um teste unitário para matá-lo.

3.2.4 Análise dos mutantes

Esta seção detalha o processo de Análise de Mutantes realizado manualmente, buscando separar os mutantes sobreviventes em Equivalentes ou Sobreviventes Não Equivalentes, além de começar o processo de criar novos casos ou utilizar novas entradas.

A Análise Experimental dos dados a partir da tabela é empírica. É necessário conhecer a linguagem Java para entender o que é esperado na linha de código do programa original sem mutação, a lógica do programa - em nosso estudo pesquisar e entender como funciona o calendário persa (FOURMILAB, 2020), o que se altera quando se aplica o operador de mutação e quais dados de entrada ou saída podem ser afetados com essa alteração.

Com base nessas análises individuais para cada mutante, foram feitas tentativas de criar testes unitários que exercitem a diferença do programa original com o mutante para demonstrar que tal mutante só pode ser morto apenas com entradas de dados não triviais. Mutações sobreviventes pela nossa hipótese possibilitam a criação de tais casos exercitando características singulares que anteriormente não foram refletidas no conjunto de testes originais. Se um novo caso de testes pudesse ser criado com tais dados era então utilizada novamente a ferramenta PITEST para verificar o resultado

deste novo caso de teste, a morte deste mutante com esse novo caso, e classificar a mutação como sobrevivente e não equivalente.

Posto isto, para cada mutante que sobreviveu foi utilizada a classificação de “E - Equivalente” quando não é possível mostrar a diferença entre o programa original e a mutação ou quando isso foi impossível pela limitação da entrada de dados, e mantendo-se a estrutura original do programa, não consigo criar um caso de testes no qual eu exercite essa diferença com uma entrada válida. “M - Mutantes” são aqueles que através da análise pude criar ou alterar um ou mais casos, no qual a diferença semântica criada pela mutação é evidenciada, e portanto o mutante é morto.

3.3 Experimento 2: Calcular Aprovação

O segundo experimento foi planejado posteriormente ao do Calendário Persa, com a intenção de corroborar a hipótese de usar a Análise de Mutantes como fator para encontrar bons casos de testes e melhores entradas que elevem a qualidade do conjunto de testes. Para que essa hipótese possa ser comprovada, a linha metodológica do segundo experimento pretende colocar em foco as entradas de dados dos testes e como elas determinam a sobrevivência de um mutante ou a adequação do conjunto de testes.

Para isso é necessário um programa mais simples para melhor visualizar seu grafo de fluxo de controle, calcular sua complexidade ciclomática e ter entendimento de seus Arcos e Caminhos. Também é necessário seu conjunto de casos de testes que atenda as métricas de Cobertura de Caminhos, seja um conjunto mínimo para atingir essa cobertura (caso retirado qualquer caso de testes do conjunto sua cobertura total deixa de ser 100%), e que não contenha nenhum mutante sobrevivente ao rodar a ferramenta de mutação PITEST no conjunto e programa originais. Para finalizar os materiais necessários é importante gerar conjuntos de entradas aleatórias, pertencentes à faixa de aceitação de valores das variáveis de entrada dos testes, para que a integridade do conjunto continue, e não se altere a validade dos testes unitários.

Com estes materiais em mãos e a técnica de mutação três perguntas precisam ser respondidas por este experimento:

1. Se eu gerar conjuntos de entradas aleatórias é possível que mutantes sobreviventes apareçam?
2. Caso apareçam, tais mutantes sobreviventes reproduzidos pelas diferentes entradas aleatórias só existem pois tais entradas são triviais demais para serem as melhores entradas possíveis?
3. E será que esse mesmo conjunto com diversas entradas aleatórias geram conjuntos de mutantes sobreviventes diferentes?

3.3.1 Escolha do programa e das ferramentas

Para ser capaz de realizar tal experimento, como mencionado anteriormente, foi necessária a busca por um programa em Java mais simples com um conjunto de casos de testes que passasse por três critérios da análise de sua qualidade: (1) atender integralmente ao critério de Todos os Caminhos tendo cobertura completa do código; (2) ser um conjunto mínimo de forma que, retirando-se qualquer caso de teste do conjunto, sua cobertura do código não seja mais completa; (3) o Conjunto original com suas entradas originais não gere nenhum mutante sobrevivente ao se utilizar a ferramenta de mutação PITEST.

O programa base deste experimento foi escolhido pela plataforma de aprendizado para programadores Devmedia em um artigo instrutivo (*https://www.devmedia.com.br/teste-unitario-com-junit-e-complexgraph/31382*) (TESTE, 2020) sobre como escrever testes unitários com JUnit e utilizar ferramentas de cobertura, além de explicar os conceitos de testes de caixa-branca, complexidade ciclomática e grafo de fluxo de controle. O programa Calcular Aprovação contém uma única classe, com 24 linhas e um conjunto de testes com 5 casos de testes. Embora pareça demasiadamente simples, sua complexidade ciclomática é de ordem 5, com 4 regiões e 4 comandos de desvios para alcançar as diferentes regiões, 14 arestas e 11

vértices, como pode ser visto segundo a Figura 5 apresentada no artigo. O grafo, também retirado do artigo, pode ser visto na Figura 6.

O conjunto de casos de testes original do programa foi, segundo o artigo, elaborado utilizando a teoria de complexidade ciclomática, cobertura de Todos os Caminhos e dados de teste. De forma que dado a complexidade ciclomática 5, o algoritmo possui 5 caminhos de execução diferente, então para ter cobertura completa são necessários no mínimo 5 casos de testes que percorrem cada um desses caminhos exercitando os comandos de desvios, e tais desvios exercitados pelos dados de entrada como vistos na Tabela 1.

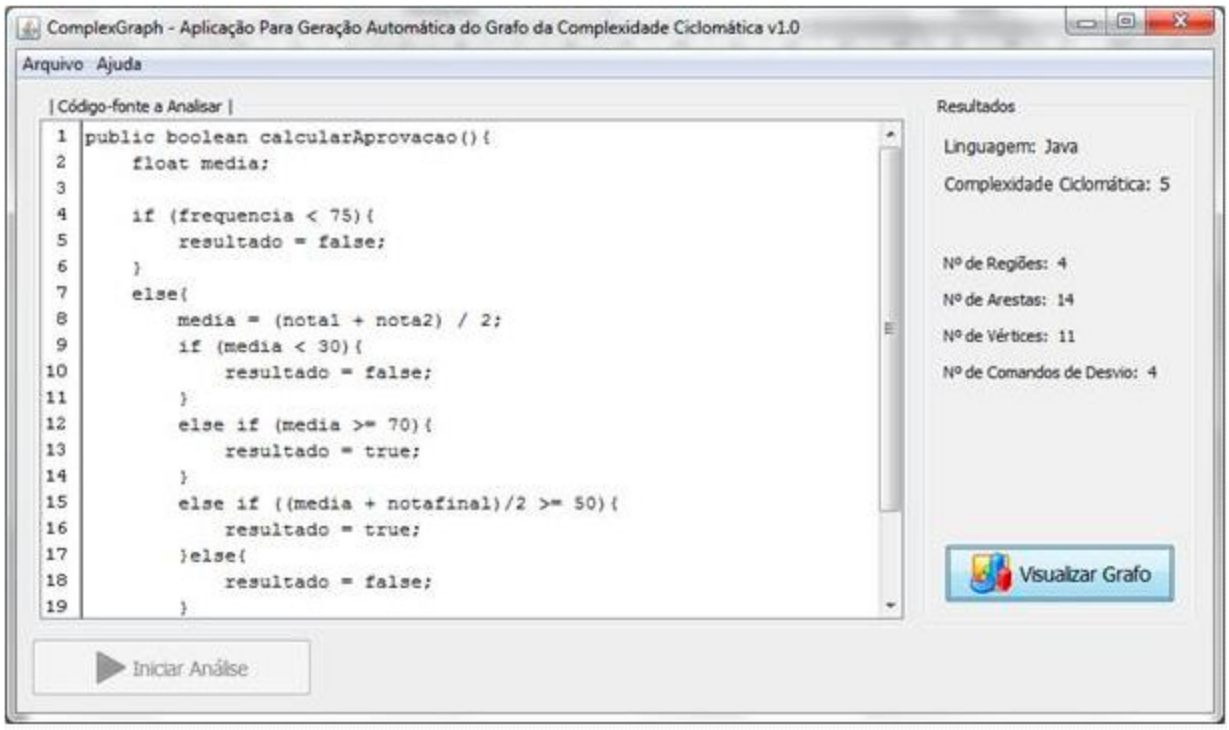


Figura 5: Resultados das métricas de Todos os Caminhos da ferramenta ComplexGraph aplicado ao programa Calcular Aprovação, adaptado do artigo “Teste unitário com JUnit e ComplexGraph” no site Devmedia.

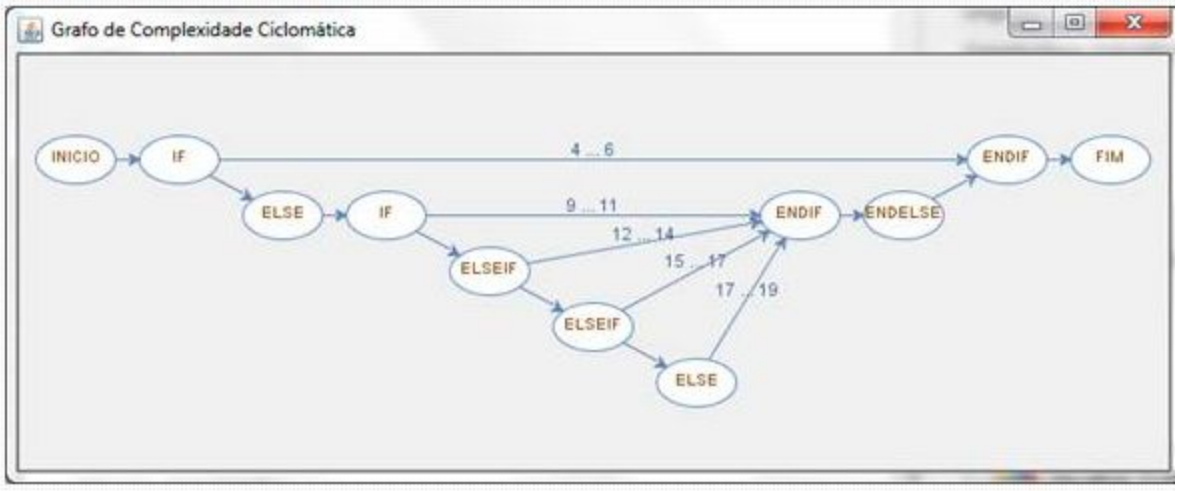


Figura 6: Grafo de complexidade ciclomática do programa Calcular Aprovação, gerado pela ferramenta ComplexGraph, adaptado do artigo “Teste unitário com JUnit e ComplexGraph” no site Devmedia.

Casos de Teste	Entradas	Saída
Caso de Teste 1	Frequência = 74 (F < 75)	Reprovado
Caso de Teste 2	Frequência = 75, Nota1 = 29 e Nota2 = 30 (F ≥ 75, N1 < 30, N2 ≥30)	Reprovado
Caso de Teste 3	Frequência = 75, Nota1 = 70 e Nota2 = 70 (F ≥ 75, N1 ≥ 70, N2 ≥ 70)	Aprovado
Caso de Teste 4	Frequência = 75, Nota1 = 30, Nota2 = 30 e Nota Final = 70 (F ≥ 75, N1 ≥ 30, N2 ≥ 30, NF ≥ 70)	Aprovado
Caso de Teste 5	Frequência = 75, Nota1 = 30, Nota2 = 30 e Nota Final = 70 (F ≥ 75, N1 ≥ 70, N2 ≥ 70, NF < 70)	Reprovado

Tabela 1: Tabela de entrada de dados para o conjunto de 5 casos de teste que percorre Todos os Caminhos do código.

Tanto a ferramenta de Análise de Cobertura JaCoCo quanto a ferramenta de mutação PITEST são as mesmas utilizadas no experimento passado.

3.3.2 Programa de entradas randômicas - RandomInputs

De forma a responder as perguntas feitas ao final da seção 3.3.1, um programa foi escrito para substituir as entradas do conjunto de testes original com entradas aleatórias no formato texto (para melhor visualização dos dados gerados) mas que ainda exercite todos os caminhos do código e não afete o resultado dos testes pelo JUnit. Para isso, foi necessário se basear nas regras definidas no conjunto de casos de teste originais.

Uma fórmula matemática, transformada em um método Java escrito pela autora, guia os limites entre os quais o valor randômico pode pertencer:

```
public static int generateRandomNumberInRange(int min,
int max) {
     return (int) ((Math.random()*((max-min)+1))+min);
}
```

Tal método significa que o número randômico gerado está entre o valor máximo que ele pode assumir e o mínimo, com a adição do +1 para que o valor máximo seja incluso na possibilidade. Então para cada dado de entrada, levando em conta os limites de entradas de dados válidos vistos da Tabela 1, podemos gerar os números aleatórios que vão estar adequados ao teste como visto na Figura 7.

```
66          freqAbaixo75 = generateRandomNumberInRange(74,0);
67          writeInFile(freqAbaixo75);
68
69          freq75Mais = generateRandomNumberInRange(100,75);
70          writeInFile(freq75Mais);
71
72          nota1Baixa = generateRandomNumberInRange(50,30);
73          writeInFile(nota1Baixa);
74
75          nota1BaixaMenor30 = generateRandomNumberInRange(29,0);
76          writeInFile(nota1BaixaMenor30);
77
78          nota1Alta = generateRandomNumberInRange(100,70);
79          writeInFile(nota1Alta);
80
81          nota2Baixa = generateRandomNumberInRange(50,30);
82          writeInFile(nota2Baixa);
83
84          nota2BaixaMenor30 = generateRandomNumberInRange(29,0);
85          writeInFile(nota2BaixaMenor30);
86
87          nota2Alta = generateRandomNumberInRange(100,70);
88          writeInFile(nota2Alta);
89
90          notaFinalBaixa = generateRandomNumberInRange(69,0);
91          writeInFile(notaFinalBaixa);
92
93          notaFinalAlta = generateRandomNumberInRange(100,70);
94          writeInFile(notaFinalAlta);
```

Figura 7: Programa RandomInputs e as entradas aleatórias a serem escritas num arquivo texto para usar no conjunto de casos de teste de Calcular Aprovação.

E para poder utilizar os dados aleatórios no arquivo texto gerado, o conjunto de casos foi alterado para ler a linha correspondente ao valor do dado necessário segundo a Tabela 1. A diferença entre o caso original e o alterado pode ser visto nas diferenças entre as Figuras 8 e 9.

```
38      @Test
39      public void testMediaMenor30() {
40          // Media < 30
41          int frequencia = 75;
42          int nota1 = 29;
43          int nota2 = 30;
44          int notafinal = 0;
45          EstudoCaso1 instance = new EstudoCaso1();
46          boolean expResult = false;
47          boolean result = instance.calcularAprovacao(nota1, nota2, notafinal, frequencia);
48          assertEquals(expResult, result);
49      }
```

Figura 8: Caso de teste original que exercita o caminho de Média menor que 30 resultando em reprovação.

```
50  @Test
51  public void testMediaMenor30() {
52      // Media < 30
53      int frequencia = Integer.parseInt(lines.get(1)); //frequencia maior igual 75
54      int nota1 = Integer.parseInt(lines.get(3)); //nota1baixa menor q 30
55      int nota2 = Integer.parseInt(lines.get(6)); //nota2baixa menor q 30
56      int notafinal = Integer.parseInt(lines.get(8));
57      EstudoCaso1 instance = new EstudoCaso1();
58      boolean expResult = false;
59      boolean result = instance.calcularAprovacao(nota1, nota2, notafinal, frequencia);
60      assertEquals(expResult, result);
61  }
```

Figura 9: Caso de teste que exercita o caminho Média menor que 30 alterado, lendo do arquivo texto de entrada dos respectivos dados aleatórios gerados para garantir sua integridade.

Para cada conjunto de casos de teste com entradas aleatórias gerado foi validado através do JUnit que o conjunto continua com cobertura 100% do programa e que nenhum caso de teste falha com a nova entrada qualquer seja ela. Os resultados dessa verificação geraram relatórios em HTML disponibilizados no link do github da seção 3.4.

3.3.3 Extração de mutantes

O próximo passo do experimento é aplicar a ferramenta de mutação ao programa - com o conjunto de casos de testes alterados para receber as entradas aleatórias - para cada conjunto de entradas geradas. Dessa forma, conseguiremos extrair os mutantes sobreviventes para cada conjunto de entradas aleatórias.

Desta vez, todo o conjunto de testes é rodado ao invés de um único caso pela ferramenta, pois queremos analisar se em um conjunto que tem cobertura completa; é mínimo, de forma que retirando-se qualquer caso o conjunto deixe de ter a cobertura completa do programa; e não continha nenhum mutante previamente. Ter entradas entradas de diversos dados triviais - porém válidos - faz com que mutantes sobreviventes apareçam.

Foram gerados 12 conjuntos de dados de entrada aleatórios gerados pelo programa RandomInputs submetidos ao PITEST, automaticamente a ferramenta realizou a mutação do programa e submeteu ao conjunto de testes para aplicar a técnica e gerar os relatórios de resultados. Foram coletados os respectivos relatórios HTML e XML semelhantemente ao Experimento 1 para criar-se a planilha. Poderiam ser gerados infinitos conjuntos e inicialmente eram 10, porém foram adicionados mais dois para tentar obter alguma diferença maior que os 10 iniciais, porém percebeu-se que não houve grandes diferenças então não foram acrescentados mais pois pareceu redundante.

Como o programa Calcular Aprovação é consideravelmente menor que a biblioteca Calendário Persa, a planilha foi criada manualmente com duas abas. A primeira aba, que contém as descrições dos mutantes sobreviventes apenas, foi dividida nas seguintes colunas: (1) o número da linha de código da mutação que sobreviveu, (2) uma cópia linha de código para referência, (3) a mutação sofrida, (4) descrição do operador de mutação aplicado e (5) quais conjuntos apresentaram mutação viva. A primeira aba da planilha pode ser vista na Figura 10.

A segunda aba, da planilha Figura 11, são as entradas aleatórias geradas para cada uma das variáveis, de forma a verificar que tais dados de entrada ainda fazem parte de domínio de entradas válidas segundo a Tabela 3 vista anteriormente.

	A	B	C	D	E
1	lineNumber	Código	Mutante	description	Conjunto de Casos
2	11	if (frequencia < 75)	< para <=	changed conditional boundary	C1, C2, C3, C4, C5, C6, C7, C8, C9, C10, C11, C12
3	16	if (media < 30)	< para <=	changed conditional boundary	C1, C2, C3, C4, C5, C6, C7, C8, C9, C10, C11, C12
4	19	else if (media >= 70)	>= para >	changed conditional boundary	C1, C2, C3, C4, C5, C6, C7, C8, C9, C10, C11, C12
5	22	else if ((media + notafinal)/2 >= 50)	>= para >	changed conditional boundary	C1, C2, C3, C4, C5, C6, C7, C8, C9, C10, C11, C12

Figura 10: Primeira Aba da Tabela de Mutantes Sobreviventes do Experimento 2 Calcular Aprovação contendo os mutantes sobreviventes por linha, o código, a mutação sofrida, o operador de mutação e quais conjuntos de entradas aleatórias sobreviveram.

	A	B	C	D	E	F	G	H	I	J	K
1	ID	requencia Abaixo 75	encia Maior Igual 75	Nota 1 entre 30 e 50	Nota 1 Menor que 30	Nota 1 Alta	Nota 2 entre 30 e 50	Nota 2 Menor que 30	Nota 2 Alta	lota Final Baixa	Nota Final alta
2	C1	13	93	45	15	71	42	19	84	12	89
3	C2	1	89	47	2	89	36	2	89	30	88
4	C3	47	93	39	11	73	46	19	78	48	97
5	C4	37	97	33	2	86	32	1	73	48	91
6	C5	64	81	41	26	83	37	14	97	49	74
7	C6	1	76	41	24	72	44	16	77	48	94
8	C7	48	87	49	17	85	45	15	96	5	98
9	C8	33	81	45	26	73	32	13	99	24	80
10	C9	26	92	47	23	71	32	4	77	39	74
11	C10	42	92	36	27	91	33	14	75	12	87
12	C11	58	92	35	20	82	45	5	87	26	73
13	C12	57	90	34	21	71	40	22	80	34	98

Figura 11: Segunda Aba da Tabela de Mutantes Sobreviventes do Experimento 2 Calcular Aprovação contendo os conjuntos de entradas aleatórias pormenorizados por variável.

3.3.4 Análise dos dados

Para cada mutante sobrevivente foi investigado o porquê as entrada de dados para cada conjunto randômico C1 a C12 resultou na sobrevivência do mutante através do exercício mental de seguir os caminhos descritos no grafo com os valores de entrada. Tais resultados são posteriormente apresentados no Capítulo 4 e especificamente na seção 4.1.

3.4 Disponibilização do projeto e dados

Todos os códigos do Experimento 1 Calendário Persa, assim como relatórios HTML gerados e os novos casos de testes adicionados ao conjunto com o comentário `//meu próprio caso de teste` poderão ser consultados através do seguinte link:

https://github.com/IsabelSprogis/persian-date-time-master

Todos os códigos do Experimento 2 Calcular Aprovação, assim como relatórios HTML gerados poderão ser consultados através do seguinte link:

https://github.com/IsabelSprogis/test-master

O programa escrito pela aluna que gera um arquivo de texto contendo as entradas aleatórias de dados - ainda que dentro do limite de aceitação de dados de forma que todos os casos continuem íntegros ao passar pelo JUnit - pode ser consultado através do link:

https://github.com/IsabelSprogis/RandomInputs

A planilha de resultados de Mutantes Sobreviventes do Experimento 1 Calendário Persa pode ser acessada através do link:

http://bit.ly/3bJJLn8

A planilha de resultados de Mutantes Sobreviventes do Experimento 2 Calcular Aprovação pode ser acessada através do link:

http://bit.ly/3oXaPmY

3.5 Considerações Finais

Neste capítulo foram apresentados os dois experimentos realizados, suas ferramentas, metodologia para cada um, e o objetivo de cada experimento.

O primeiro experimento utilizou o programa open source do Calendário Persa aplicado à ferramenta automática de mutação PITEST. A aplicação da ferramenta foi realizada para cada um dos 132 testes unitários que o programa já continha, e os resultados foram exportados automaticamente pela ferramenta em relatórios HTML e XML.

Tais relatórios foram adicionados em uma planilha e a análise dos mutantes foi realizada manualmente pela autora. A análise buscou diferenciar o programa mutado do original através da criação de novo caso de teste ou nova entrada de dados. Tais análises foram adicionadas na segunda aba da planilha, e os casos novos ou novos dados que foram acrescentados ao conjunto original foram marcados com o comentário `//meu próprio caso de teste.`

O segundo experimento utilizou o programa de Calcular Média com seu respectivo conjunto de testes, que era perfeito segundo a Técnica de Mutação aplicada inicialmente, visto que o conjunto de testes estava adequado e não produzia nenhum mutante sobrevivente ao se aplicar a ferramenta PITEST. Podemos atribuir a alta qualidade do conjunto deste programa devido às métricas de complexidade ciclomática e escolha dos dados para criar os casos de testes, segundo o artigo do qual ele se originou.

O experimento alterou as entradas de dados do conjunto de teste para entradas aleatórias, podendo chamá-las de triviais, que ainda respeitam a estrutura do conjunto de testes. Foram 12 conjuntos de entradas aleatórias, aplicadas novamente a ferramenta de mutação automática PITEST.

Foi verificado que, mutantes sobreviventes que anteriormente não existiam no conjunto com as entradas originais emergiram, e tais casos foram adicionados a planilha para análise do motivo da introdução dos mutantes sobreviventes com as entradas aleatórias.

4. RESULTADOS

Este capítulo contém os resultados de ambos os experimentos realizados separados por experimentos.

4.1 Resultado do Experimento 1: Calendário Persa

Levando em conta o processo descrito na metodologia do Experimento 1, das 30 mutações sobreviventes temos:

- 21 mutações foram consideradas equivalentes logicamente ao programa original. Ou seja, 70% do conjunto de sobreviventes é equivalente ao programa original ou não é possível evidenciar essa diferença.
- 9 mutações foram consideradas mutantes sobreviventes não equivalentes. Apenas 30% dos mutantes não foram mortos por nenhum caso de testes do conjunto original, mas morrem ao conceber novos casos com valores específicos ou asserções melhores.

Entre estes últimos, é possível separá-los em dois grupos: os que sobrevivem por conta de casos de testes de baixa qualidade e que exercitam pouco ou nada os cenários da unidade pelas entradas; e os que necessitam de uma entrada específica, comumente valores ligados à lógica da linha em questão, como valores limites em um escopo e pontos críticos como valores classificatórios para fazer parte de um conjunto.

Todos os testes criados que matam mutantes previamente sobreviventes poderiam substituir ou mesmo complementar os que já existiam no conjunto e antes não matavam tais mutações. Testes unitários com JUnit permitem diversas asserções por caso de teste unitário a fim de abranger quantos tipos de entradas e saídas desejar, obviamente lembrando de tentar se exercitar diferentes cenários de entrada e saída mais do que qualquer número.

Um dos primeiros casos a se notar que existem mutantes que necessitam ser testados com valores limites para matar tais mutações foi o caso da linha 110 na Figura 12 da classe Java MyUtils. Tal linha faz parte do método longRequireRange, que recebe um valor, um limite mínimo, um limite máximo e uma string caso haja exceção. A linha

110 é o que define a lógica do método comparando se o valor é menor que o limite mínimo ou maior que o limite máximo, e caso seja retorna uma exceção. As mutações sobreviventes que essa linha sofre são do tipo *changed conditional boundary,* transformando a comparação de menor ou maior respectivamente que os limites em menores ou maiores iguais. Um perfeito exemplo para aplicar a técnica da Análise de Valores Limites - criando casos de teste com entradas que coincidem com ambos os lados de cada fronteira, incluindo o valor limite da própria fronteira visto na Figura 13. O resultado de tal aplicação de técnica foi a morte desses mutantes quando se utiliza o próprio valor limite na Figura 14.

```
 99⊕     * Checks whether a long is in a range or not. If {@code val} is less than…
109⊖    static long longRequireRange(long val, long lowerLimit, long upperLimit, String valName) {
110         if (val < lowerLimit || val > upperLimit){
111                 throw new IllegalArgumentException(valName + " " + val +
112                         " is out of valid range [" + lowerLimit + ", " + upperLimit + "]");
113             }
114         return val;
115     }
116 }
```

Figura 12: Imagem do método original LongRequireRange da classe Java MyUtils que verifica se um dado valor está dentro do limite passado por parâmetro.

```
64⊖     @Test
65      public void testOnLongRequireRange() {
66          MyUtils.longRequireRange(Long.MAX_VALUE - 1,
67                  Long.MAX_VALUE - 2, Long.MAX_VALUE, "exception");
68          MyUtils.longRequireRange(Long.MIN_VALUE +1,
69                  Long.MIN_VALUE, Long.MIN_VALUE + 2, "exception");
70      }
71
72
73⊖     @Test
74      public void testOnLongRequireRange2() {         //Meu proprio teste
75          MyUtils.longRequireRange(Long.MAX_VALUE,
76                  Long.MAX_VALUE, Long.MAX_VALUE, "exception");
77          MyUtils.longRequireRange(Long.MIN_VALUE,
78                  Long.MIN_VALUE, Long.MIN_VALUE, "exception");
79      }
```

Figura 13: Diferença entre o caso de teste original e o caso de testes utilizando como entradas apenas valores iguais como MAX_VALUE do programa ou MIN_VALUE, exercitando o método LongRequireRange.

Figura 14: Resultado da morte dos mutantes do tipo changed conditional boundary da linha 110 quando aplicada a técnica de análise dos valores limites criando o caso de testes testOnLongRequireRange2.

Outro mutante não equivalente também do operador *changed conditional boundary* é o da linha 427, parte do método get Long da classe Java Persian Date. Tal método retorna o valor do campo de uma data especificada pela entrada, no caso da linha 427 ele verifica em que ano da era esta como visto na Figura 15, a mutação no entanto se torna case YEAR_OF_ERA: return (year > 1 ? year : 1 - year). A análise feita para a mutação dessa linha é de que deixou-se de verificar na lógica quando o ano é 1, e tal caso de testes usando ano igual a 0001 não fazia parte do conjunto original. Bastava incluir um caso de testes contendo o como entrada o ano 0001 para matar a mutação, assim como visto na Figura 16.

```
414     public long getLong(TemporalField field) {
415         if (field instanceof ChronoField) {
416             switch ((ChronoField) field) {
417                 case DAY_OF_WEEK: return getDayOfWeek().getValue();
418                 case ALIGNED_DAY_OF_WEEK_IN_MONTH: return ((day - 1) % 7) + 1;
419                 case ALIGNED_DAY_OF_WEEK_IN_YEAR: return ((getDayOfYear() - 1) % 7) + 1;
420                 case DAY_OF_MONTH: return this.day;
421                 case DAY_OF_YEAR: return this.getDayOfYear();
422                 case EPOCH_DAY: return this.toEpochDay();
423                 case ALIGNED_WEEK_OF_MONTH: return ((day - 1) / 7) + 1;
424                 case ALIGNED_WEEK_OF_YEAR: return ((getDayOfYear() - 1) / 7) + 1;
425                 case MONTH_OF_YEAR: return month;
426                 case PROLEPTIC_MONTH: return (year * 12L + month - 1);
427                 case YEAR_OF_ERA: return (year >= 1 ? year : 1 - year);
428                 case YEAR: return year;
429                 case ERA: return (year >= 1 ? 1 : 0);
430             }
431         }
432         throw new UnsupportedTemporalTypeException("Unsupported field: " + field);
433     }
```

Figura 15: Imagem do método getLong da classe PersianDate que retorna um campo de uma data especificado pela entrada.

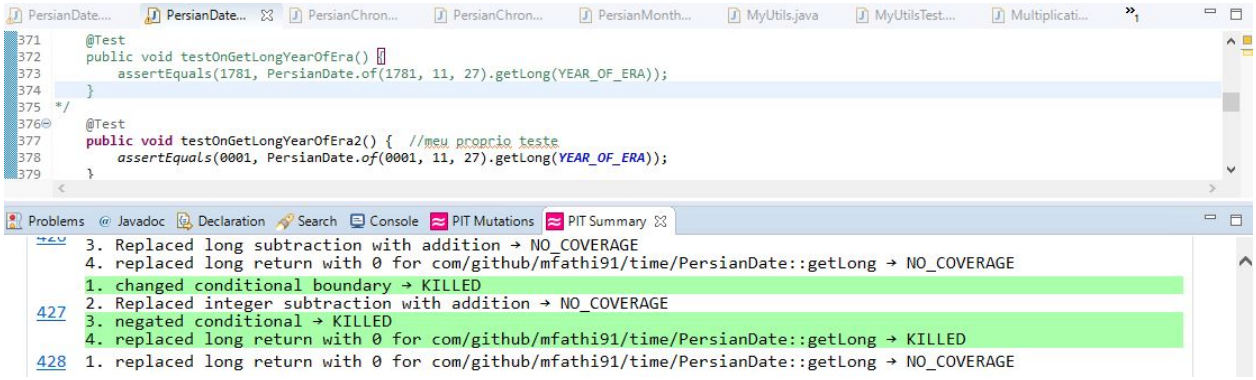


Figura 16: Diferença entre o caso de teste original e o novo caso que utiliza ano igual a 0001. Também relatório HTML gerado com a morte da mutação changed conditional boundary da linha 427 do método getLong da classe PersianDate.

O quarto e último mutante não equivalente encontrado ocorre no mesmo método getLong na linha 423 que pode ser visto também na Figura 15. O operador de mutação aplicado é o *Replaced integer subtraction with addition* de forma que o mutante é return (day + 1)/7 +1; Analisando a linha original temos que seu propósito é retornar, de 1 a 4, qual semana do mês a data se encontra (pois um mês é formado sempre de 4 semanas). Novamente foi constatado que no conjunto de casos de testes iniciais não

houve nenhum caso de teste que utilizasse entradas com múltiplos de 7, o que pela lógica é a data limite entre uma semana e outra. Apenas criando um caso que levasse em conta datas múltiplas de sete nesse cenário já foi possível matar tal mutação como é possível ver na Figura 17.

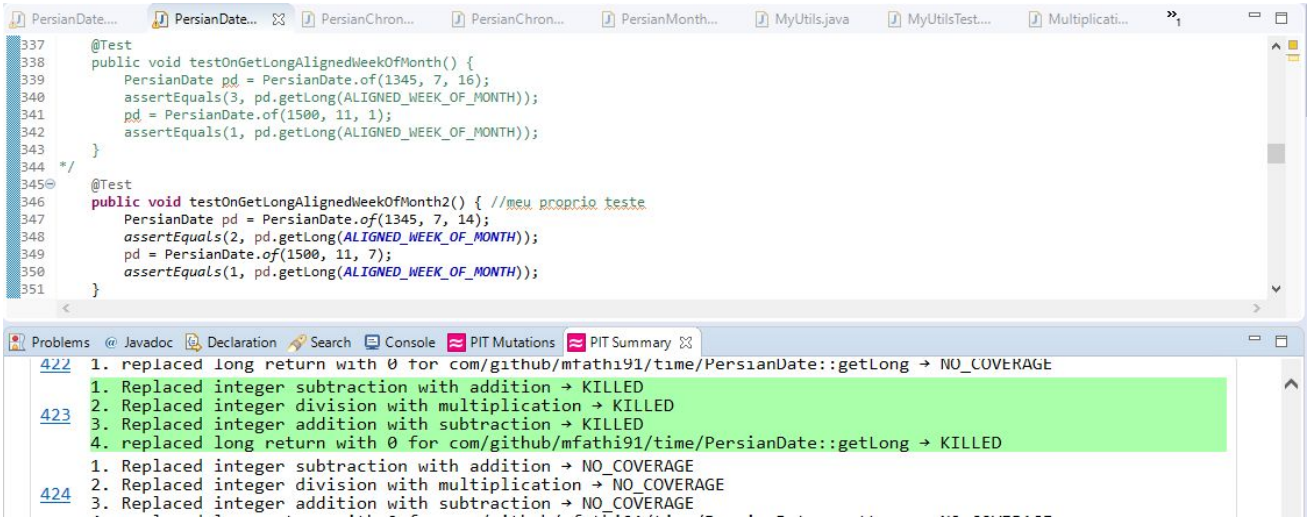


Figura 17: Diferença entre o caso de teste original e o novo caso que utiliza ano igual datas com dias múltiplos de 7, relatório HTML gerado com a morte da mutação replaced integer subtraction with addition da linha 423 do método getLong da classe PersianDate.

É notável que os mutantes que sobreviveram devido a baixa qualidade dos testes que exercitam suas unidades são em maior quantidade dos operadores de *change return type,* dos 9 mutantes não equivalentes 5 deles se enquadram nessa categoria, aproximadamente 56% dos sobreviventes não equivalentes. Um erro muito comum encontrado no conjunto de testes original provenientes da biblioteca é criar testes com falta de asserção sobre uma entrada e um resultado esperado, evidenciado principalmente no mutante da linha 80 de MyUtils como demonstrado na Figura 18 e Figura 19.

```
76    static int intRequirePositive(int val, String valName) {
77        if (val <= 0) {
78            throw new IllegalArgumentException(valName + " is not positive: " + val);
79        }
80        return val;
81    }
```

Figura 18: A linha 80 sofre a mutação do tipo replaced int return with 0, que faz com que o valor retornado seja 0. O mutante só sobrevive pois não há asserção do resultado no caso de teste original.

```
48    @Test
49    public void testOnIntRequirePositive() {
50        MyUtils.intRequirePositive(1, "one");
51    }
52
53    @Test
54    public void testOnIntRequirePositive2() { //meu proprio teste
55        int expected = 1;
56        assertEquals(MyUtils.intRequirePositive(1, "one"), expected);
57    }
```

Figura 19: Diferença entre o caso de teste usando apenas a chamada do método intRequirePositive de um caso de testes que faz a asserção do valor esperado de saída com entrada do método intRequirePositive.

Algumas das mutações sobreviventes equivalentes notáveis foram as que substituíram algum operador matemático em linhas de códigos, como a linha 579 do método toJulianDay da classe PersianDate visto na Figura 20, que continham fórmulas matemáticas onde são usado os dados 2820 - que é a quantidade de anos em um grande ciclo no calendário persa, assim como 1029983 que é 2137 anos normais de 365 dias e 683 anos bissextos de 366 dias dentro desse ciclo.

Tais mutações dessas linhas, como mostradas na Figura 21, só conseguiam ser mortas por casos de testes nos quais os tipos de entradas e de variáveis eram alteradas de inteiro para double e que o ano fosse especificamente *ano= 0,00273790926646362124423412813 61.* Tal número foi retirado da conta

*1029983*ano/2820=1 então ano=2820/1029983 logo*

ano=0,00273790926646362124423412813 61.

Porém, uma das premissas do Teste de Mutação é não alterar o programa original em nenhuma forma. Desse modo o tipo de entrada necessária para evidenciar a diferença dessa mutação para o programa original não é válida, tampouco é possível que um ano não seja do domínio de inteiros, e portanto tais casos foram considerados como equivalentes.

```
576    static long toJulianDay(int year, int month, int dayOfMonth) {
577        int epbase = year - 474;
578        int epyear = 474 + (epbase % 2820);
579        return dayOfMonth + PersianMonth.of(month).daysToFirstOfMonth() +
580                (epyear * 682 - 110) / 2816 +
581                (epyear - 1) * 365 +
582                (epbase / 2820 * 1029983) +
583                (1948320 - 1);
584    }
```

Figura 20: Método toJulianDay que sofre mutação Replaced integer multiplication with division na linha 582 de forma que o mutante final é (epbase / 2820 / 1029983).

```
52    @Test
53    public void testMatarMutiplicacaoParaDivisao() {
54        //se 1029983x/2820 = 1, x= 2820/1029983, logo x = 0,0027379092664636212442341281361
55        //simpleToJulianDays(1) é 1948346
56        double ano = 0.0027379092664636212442341281361;
57        double expected = 1774858.15880981;
58        double delta = 2.0000000;
59        assertEquals(MultiplicationToDivision.simpleToJulianDays(ano), expected, delta);
70        }
71
```

Problems | Javadoc | Declaration | Search | Console | PIT Mutations | PIT Summary

MultiplicationToDivision.java

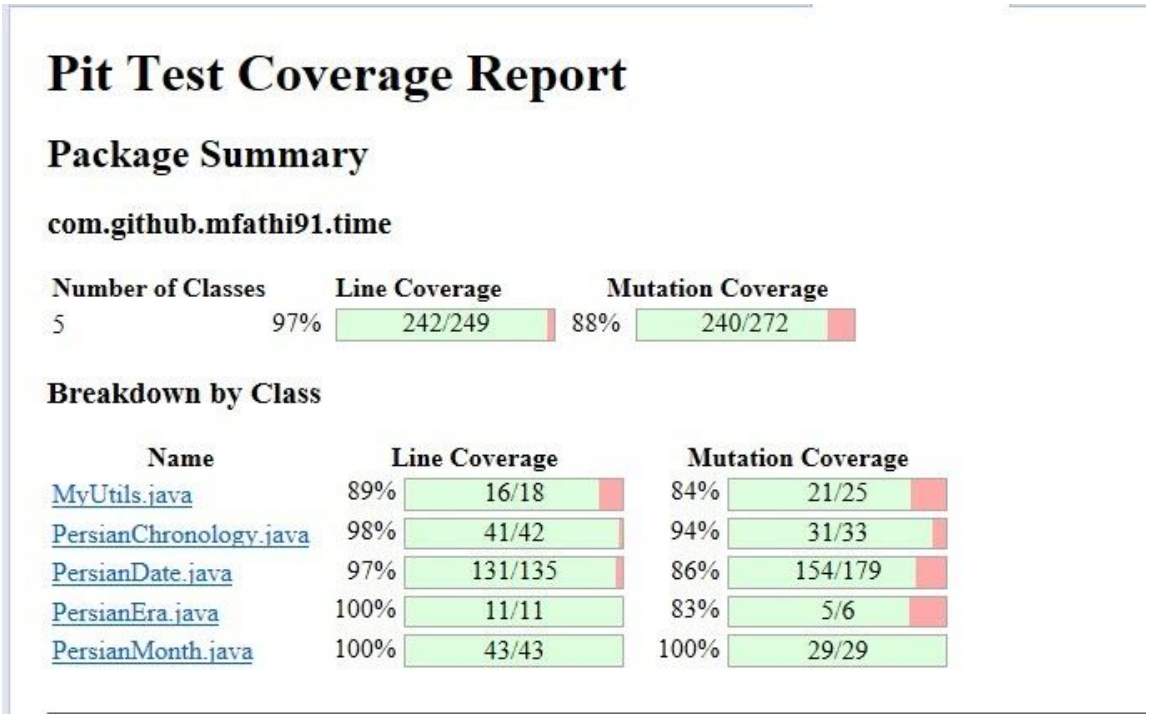
```
1   package MultiToDiv;
2
3   public class MultiplicationToDivision {
4
5           static long simpleToJulianDays(double ano){
6   1               double anobase = ano - 474;
7   2               double anoAtual = 474 + (anobase % 2820);
8   8               return (long) (12 + 15 +(anoAtual * 682 - 110) / 2816 +
9   2                                           (anoAtual - 1) * 365 +
10  2                                           (anobase/ 2820 * 1029983) +
11                                              (1948320 - 1));
```

Figura 21: Teste criado a partir da cópia do método toJulianDay em outra classe criada para testes, onde se altera as variáveis para double e o return para long, entrando com o ano = 0,0027379092664636212442341281361. A cor verde no PITEST demonstra que a mutação multiplicação para divisão na linha (epbase/ 2820 * 1029983) foi morta.

Por fim, neste experimento devemos ressaltar que, apesar de obtermos novos casos ou ter casos alterados com novas entradas de dados específicos para matar os mutantes, a cobertura total do código não foi alterada. Isso ocorre, pois mesmo que um caso de teste não seja bom o suficiente segundo a técnica de mutação, ele ainda pode cobrir um certo caminho ou arco. Um bom exemplo disso foram os testes do MyUtils nos quais não eram utilizadas asserções dos métodos e apenas chamadas para eles, considerado até um erro do ponto de vista de testes unitários, porém que no fim ainda é visto como coberto pelas ferramentas de cobertura.

A Análise de mutantes consegue ir além do critério de Todos os Arcos por selecionar dados específicos que, apesar de não aumentar a cobertura para todos os arcos, consegue matar os mutantes que sobreviveram ao conjunto de testes inicial. O número de mutantes mortos por número de mutantes totais de cada classe e do programa total traz um bom parâmetro da melhoria incluída por experimento no conjunto de testes como pode ser visto entre a Figura 22 e a Figura 23.

Note que todas as mutações do programa na classe MyUtils agora são mortas após os resultados obtidos através da Análise de Mutantes, e que tivemos um aumento de 2% de eliminação de mutantes na classe PersianDate. A biblioteca teve um aumento de 3% na cobertura total de suas mutações.



Pit Test Coverage Report

Package Summary

com.github.mfathi91.time

Number of Classes	Line Coverage		Mutation Coverage	
5	97%	242/249	88%	240/272

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
MyUtils.java	89%	16/18	84%	21/25
PersianChronology.java	98%	41/42	94%	31/33
PersianDate.java	97%	131/135	86%	154/179
PersianEra.java	100%	11/11	83%	5/6
PersianMonth.java	100%	43/43	100%	29/29

Figura 22: Relatório de cobertura de linhas e mutantes pela ferramenta PITEST no conjunto de testes original.

Pit Test Coverage Report

Package Summary

com.github.mfathi91.time

Number of Classes	Line Coverage		Mutation Coverage	
5	97%	242/249	91%	247/272

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
MyUtils.java	89%	16/18	100%	25/25
PersianChronology.java	98%	41/42	94%	31/33
PersianDate.java	97%	131/135	88%	157/179
PersianEra.java	100%	11/11	83%	5/6
PersianMonth.java	100%	43/43	100%	29/29

Figura 23: Relatório de cobertura de linhas e mutantes pela ferramenta PITEST no conjunto de testes com novos casos, casos refatorados e novas entradas após a análise de mutantes.

4.2 Resultado do Experimento 2: Calcular Aprovação

Seguindo a metodologia do Experimento 2, obtivemos:

- 4 mutações sobreviventes diferentes, todas do mesmo tipo de operador de mutação - *changed conditional boundary*.
- Nenhuma dessas mutações é equivalente logicamente, comprovado pelo fato delas não existirem no conjunto de testes originais sem as entradas alteradas para entradas aleatórias.
- Essas quatro mutações se repetiram para todos os conjuntos de entradas de dados aleatórias C1 a C12 criadas.

É necessário esclarecer que ocorreram algumas poucas repetições dos dados entre os conjuntos gerados randomicamente, mas isso não afeta os resultados negativamente pois queremos observar o comportamento que qualquer dado seja gerado, e nenhum deles foi alterado após escritos no arquivo texto.

O exercício de determinar porque essas entradas não matam o mutante tal qual os dados de entrada dos testes originais pode ser feito pela análise de cada mutante sobrevivente, que não coincidentemente são os pontos de desvios de caminhos do programa.

Todas as entradas geradas pelo programa satisfazem os requisitos para percorrer todos os caminhos e são válidas como discutido previamente e vistos nos relatórios do JaCoCo. Porém, ao se utilizar a ferramenta de mutação PITEST, as mutações obtidas de changed conditional boundary não podem ser diferenciadas do programa original através dessas entradas, por isso a sobrevivência dos mutantes.

O primeiro mutante não equivalente encontrado está no primeiro desvio de caminho, no primeiro IF da linha 11 do programa tal qual a Figura 24. Neste IF é validado se a frequência é menor que 75, e caso seja, o resultado dado será reprovação. Pela teoria de caminhos do software, esse IF pode ser visto também como caso a frequência seja maior ou igual a 75 ele continua para o outro caminho do programa. O PITEST, por aplicar o operador *changed conditional boundary,* transforma a comparação da entrada do primeiro IF de menor para menor igual.

Desta forma, mesmo que as entradas válidas para os casos que tomariam o segundo caminho no qual a frequência deveria ser acima ou igual a 75, como nenhum dado de entrada aleatório foi exatamente 75 a mutação sobrevive. O único dado de entrada que garante a diferenciação do mutante do programa original é utilizar a frequência igual a 75, como foi utilizado no conjunto de testes original em pelo menos um teste. De forma análoga, isto ocorre para os outros desvios de caminhos do programa.

```
1  package test;
2
3
4  public class EstudoCaso1
5  {
6
7      public boolean calcularAprovacao(float nota1, float nota2, float notafinal, int frequencia){
8          float media;
9          boolean resultado;
10
11         if (frequencia < 75){
12             resultado = false;
13         }
14         else{
15             media = (nota1 + nota2) / 2;
16             if (media < 30){
17                 resultado = false;
18             }
19             else if (media >= 70){
20                 resultado = true;
21             }
22             else if ((media + notafinal)/2 >= 50){
23                 resultado = true;
24             }else{
25                 resultado = false;
26             }
27         }
28         return resultado;
29     }
30 }
31
```

Figura 24: Programa Calcular aprovação original. As linhas nas quais os mutantes sobrevivem no Experimento 2 são as 11, 16, 19 e 22 após passarem pela ferramenta PITEST.

A próxima mutação sobrevivente é o IF da linha 16, que originalmente verifica se a média é menor que 30 e se for o resultado é reprovação e o outro caminho é alcançado caso a média seja pelo menos maior ou igual a 30. A mutação pelo operador *changed conditional boundary* transforma o IF de média menor que 30 para menor ou

igual a 30. Os únicos dados de entrada que garantem que a mutação possa ser evidenciada seriam se ambas as nota 1 como a nota 2 dessem uma média exatamente igual a 30. No conjunto original de testes isso foi atingido colocando ambas as notas como 30 em pelo menos um teste.

Na linha 19, no IF que valida se a média é maior ou igual a 70 temos o caso da mutação sobrevivente que transforma, através do mesmo operador de mutação, para validar se a média é apenas maior que 70. Para esta mutação ser morta é necessário que as notas 1 e 2 dessem a média exata como 70, o que nos testes originais foi obtido com ambas as notas iguais a 70.

Por fim, a última mutação sobrevivente é da linha 22, no IF que a soma da média com a nota final se ela é maior ou igual a 50, e que transforma tal condição para validar apenas se ela é maior que 50 pelo operador de mutação. Então, apenas em casos onde a soma da média mais a nota final for exatamente igual a 50 é que tal mutação pode ser morta.

Utilizando C1 para elucidar esse exercício, temos 10 tipos de entradas aleatórias geradas em C1:

1. A Frequência baixa é 13.
2. A Frequência alta é 93.
3. A Nota 1 é 45.
4. A Nota 1 baixa é 15.
5. A Nota 1 alta é 71.
6. A Nota 2 é 42.
7. A Nota 2 baixa é 19.
8. A Nota 2 alta é 84.
9. A Nota Final baixa é 12.
10. A Nota Final alta é 89.

A primeira mutação da linha 11, como mencionado anteriormente, modifica a verificação se a frequência é menor que 75 para menor ou igual. Isso significa que o mutante não pode ser diferenciado do programa original no fato de a frequência alta ser

93, tal dado de entrada não pode diferenciar a mudança de condicional de troca de caminhos ao contrário do número 75.

Na mutação da linha 16, que altera a validação da média menor que 30 para menor ou igual a 30, os dados de entradas de C1 relacionados são as notas baixas, no caso foram 15 e 19. Porém, como elas são aleatórias dificilmente a média de ambas daria exatamente 30, que é o dado necessário para diferenciar e portanto matar esse mutante. Similarmente isso acontece para a próxima mutação, da linha 19, a qual verificava se a média era maior ou igual a 70 e se torna apenas maior que 70,e as notas altas de C1 foram 71 e 84.

Por fim, a última mutação é ainda mais dependente de todos os dados de entradas pois é a mudança da conferência de se a média entre a média e o nota final é maior ou igual a 50 para somente maior que 50. Dessa forma, as notas da média formadas pelos dados Nota 1 e Nota 2, que foram 45 e 42 respectivamente, além da Nota Final que pode ser tanto baixa ou alta para seguir os últimos dois caminhos, de reprovação ou aprovação respectivamente, deveriam ter dado 50 ao ter feito a média final (para poder matar tal mutante), o que é ainda mais improvável com 3 entradas aleatórias.

Isso não quer dizer que as entradas estão erradas, pois elas são aceitas pelo JUnit e pela ferramenta de cobertura igual ao conjunto de testes original como pode-se ver na Figura 25. Porém isso, significa que, apesar de aceitáveis como valores de entradas, elas não são as entradas ótimas evidenciado pelo fato de terem mutantes sobreviventes vistos na Figura 26 e no relatório associado no github. Tais entradas de dados não utilizam os valores das condicionais que exercitam os desvios dos caminhos como dados de entradas.

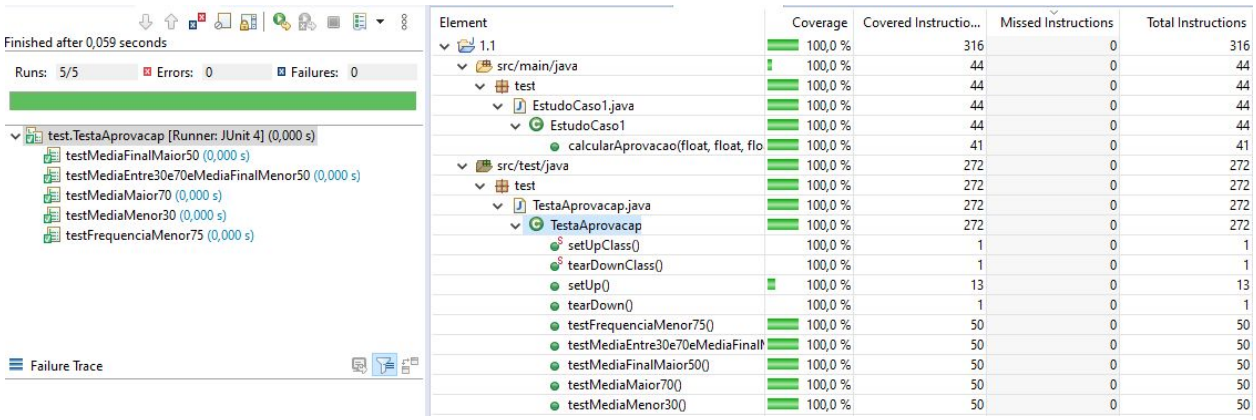


Figura 25: Resultados do conjunto de testes unitários C1 ao passar pelo JUnit e JaCoCo.

```
11 2        if (frequencia < 75){
12              resultado = false;
13          }
14          else{
15 2            media = (nota1 + nota2) / 2;
16 2            if (media < 30){
17                  resultado = false;
18              }
19 2            else if (media >= 70){
20                  resultado = true;
21              }
22 4            else if ((media + notafinal)/2 >= 50){
23                  resultado = true;
24              }else{
25                  resultado = false;
26              }
27          }
28 2        return resultado;
29      }
30  }
31
```

Mutations

```
11  1. changed conditional boundary → SURVIVED
    2. negated conditional → KILLED
15  1. Replaced float addition with subtraction → KILLED
    2. Replaced float division with multiplication → KILLED
16  1. changed conditional boundary → SURVIVED
    2. negated conditional → KILLED
19  1. changed conditional boundary → SURVIVED
    2. negated conditional → KILLED
    1. changed conditional boundary → SURVIVED
22  2. Replaced float addition with subtraction → KILLED
    3. Replaced float division with multiplication → KILLED
    4. negated conditional → KILLED
28  1. replaced boolean return with false for test/EstudoCaso1::calcularAprovacao → KILLED
    2. replaced boolean return with true for test/EstudoCaso1::calcularAprovacao → KILLED
```

Figura 26: Resultados do conjunto de testes unitários C1 ao passar pelo PITEST com as quatro mutações de *changed conditional boundary* nos quatro ifs de desvios de caminhos.

4.3 Considerações Finais

Neste capítulo foram apresentados os resultados de cada experimento, separadamente segundo seus objetivos, e conclusões começam a ser formadas pelos resultados apresentados. A proposta de cada experimento influenciou os resultados observados, posto que cada experimento tinha um foco diferente: Encontrar melhores dados e casos de testes como no Calendário Persa, ou verificar que entradas e testes triviais não são adequados ao conjunto de testes, e devem ser estudadas antes de se escrever os testes.

No experimento do Calendário Persa, os maiores resultados foram a obtenção de 30 mutantes sobreviventes, dos quais foram retirados 21 equivalentes, sobrando então 9 mutantes sobreviventes não equivalentes. Tais mutantes foram importantes por propiciar a oportunidade de criar-se 6 novos casos de testes e 3 correções de casos existentes, pela má utilização do JUnit faltando a asserção do método. O conjunto de testes no final foi melhorado utilizando-se a Análise de Mutantes.

No experimento do Calcular Média, todos os pontos de desvio de caminho do código, quando aplicados aos operadores de mutação, geraram mutantes sobreviventes não equivalentes ao introduzir entradas de dados aleatórias. Isso comprova o objetivo do experimento de verificar que entradas triviais influenciam o resultado da qualidade do conjunto de testes segundo a Análise de Mutantes, apesar de as entradas serem válidas segundo JUnit, nem sempre elas são ótimas.

Em ambos os experimentos, podemos destacar que o operador com maior número de mutantes sobreviventes não equivalentes foi o *Changed conditional boundary.* Novamente corroborando o fato de, em casos de desvios de caminhos ou comparações lógicas, as melhores entradas são as associadas aos valores críticos ou de limite.

5. CONCLUSÕES

Uma das maiores dificuldades encontradas foi não existir atualmente ferramentas de mutação que separe o conjunto de casos de testes e rode cada caso separadamente nas mutações. Foi necessário rodar individualmente cada caso comentando todos os outros manualmente durante o Experimento 1: Calendário Persa, o que no final foi um processo trabalhoso para conseguir todos os relatórios de cada caso de testes únicos aplicados ao programa utilizando o PITEST. Montar a tabela com os resultados dos relatórios HTML e XML exportados do Calendário Persa foi uma tarefa complexa também pelo número de casos de testes e classes envolvidos neste experimento, que poderia ter sido reduzida caso a ferramenta já exportasse em formato .csv por exemplo para melhor análises.

A técnica de Análise de Mutantes sofre muito com a necessidade de se avaliar a equivalência semântica de um mutante. É um trabalho árduo e manual, pois é necessário analisar caso por caso mesmo que o operador de mutação seja igual para duas linhas diferentes. Mas, apesar de todas as dificuldades encontradas, o estudo empírico sobre técnica de Testes de Mutação mostrou-se muito útil não só para analisar a qualidade do código e do conjunto de testes pela métrica de mutantes sobreviventes obtidos, como também pela proposta de testar e avaliar a qualidade do conjunto de testes.

A grande adição trazida pela análise dos mutantes encontrados no primeiro experimento foi poder criar novos casos de testes com mais cenários de dados. Tais dados não triviais são completamente dependentes da semântica em questão e vem da análise de requisitos e entendimento da linguagem de programação. Dessa forma, é muito valioso poder acrescentar ao conjunto casos que matem os mutantes sobreviventes.

E da mesma forma, o segundo experimento trouxe grande acréscimo ao estudo por demonstrar que apesar de uma entrada ser válida em um teste unitário nem sempre ela é a melhor das entradas que poderia ser utilizada, e que a análise de mutação nos

auxilia a encontrar tais entradas não triviais para que o conjunto de testes não só seja aceitável, mas ótimo segundo o critério da técnica de Análise de Mutantes.

A partir disso, uma lição que podemos tirar é a boa prática de construir casos de testes que pensem e exercitem os valores limites da lógica do programa se utilizando de métricas de testes como Todos os Nós, Todos os Arcos e Todos os Caminhos. Se utilizar de ferramentas que mapeiam o Grafo de Fluxo de Controle e também montar uma tabela de entradas e saídas esperadas auxiliam extremamente no processo e na qualidade da escrita dos testes unitários.

Em ambos os experimentos é ressaltado que as mutações mais notáveis são criadas pelo operador *Changed conditional boundary*, as quais chamam atenção as entradas de dados dos testes unitários. Tais entradas são não triviais para matar esses mutantes, os quais chamamos de resistentes, pois elas devem exercitar especificamente ou um valor “crítico” da lógica do programa ou um desvio de caminho.

A execução dos experimentos e seus resultados asseguraram o cumprimento dos objetivos deste trabalho de poder utilizar a Análise de Mutantes como técnica de melhoria de conjuntos de casos de testes ou de examinar sua qualidade através dos mutantes sobreviventes e as informações que eles nos retornam sobre possíveis defeitos no código. Apesar de ser uma técnica muito ligada ao programa em foco e portanto muito manual, o valor que ela traz pode ser extremamente útil para assegurar a qualidade do conjunto de testes. Futuramente, é possível imaginar alguma ferramenta de automação com auxílio de Inteligência Artificial que faça sugestões enquanto se elabora um conjunto de testes unitários baseados na Análise de Mutantes.

Também estudos e experimentos podem ser desenvolvidos para se obter outros tipos de mutações possíveis e destacar além das mutações de *changed conditional boundary* outros possíveis operadores que mais produzem mutantes sobreviventes, e por fim utilizá-los como heurística na criação e análise de conjunto de casos de testes unitários.

6.BIBLIOGRAFIA

DEMILLO, R.; BUDDY, T.; LIPTON, R.; SAYWARD, F. The Design Of A Prototype Mutation System For Program Testing. Georgia Institute of Technology, Atlanta, Georgia, 1978. DOI: 10.1109/AFIPS.1978.195.

FIRESMITH, DONALD G. Testing object-oriented software. In Testing Object Oriented Languages and Systems (TOOLS), Março 1993.

FATHI, MAHMOUD. GitHub, 2018. Persian Date Time. Disponível em: <https://github.com/mfathi91/persian-date-time>. Acesso em: 6 de Agosto de 2019.

FOURMILAB Calendar Converter. Fourmilab.cn. Disponível em: <http://www.fourmilab.ch/documents/calendar/>. Acesso em: 19 de Junho de 2020.

GRUN, B.; SCHULER, D.; ZELLER, A. The Impact of Equivalent Mutants. In: IEEE International Conference on Software Testing Verification and Validation Workshops (ICSTW). Washington, DC, USA. IEEE Computer Society, 2009. DOI: 10.1109/ICSTW.2009.37.

JACOCO Ferramenta de Cobertura de Software. eclemma.org. Disponível em: <https://www.eclemma.org/jacoco/>. Acesso em: 16 de Outubro de 2020.

JIA, Y.; HARMAN, M. An Analysis and Survey of the Development of Mutation Testing. IEEE Transactions on Software Engineering, v.37 , n.5 , Oct. 2011. DOI: 10.1109/TSE.2010.62.

PITEST Mutation Testing Tool. Pitest.org. Disponível em: <https://pitest.org/>. Acesso em: 14 de Julho de 2019.

MCCABE, THOMAS J. A complexity measure. IEEE Transactions on software Engineering, n. 4, p. 308-320, 1976. DOI: 10.1109/TSE.1976.233837.

PRESSMAN, ROGER S. Engenharia de Software. – Sexta Edição. São Paulo McGraw-Hill, 2006.

SOARES, INALI WISNIEWSKI. Análise de Mutantes e Critérios Restritos no Contexto de Teste de Software: Resultados de uma Avaliação Empírica. 2000. 109f. Dissertação de Mestrado - Universidade Federal do Paraná, Curitiba, 2000.

TESTE unitário com JUnit e ComplexGraph. Devmedia.com.br. Disponível em: <https://www.devmedia.com.br/teste-unitario-com-junit-e-complexgraph/31382>. Acesso em: 14 de Outubro de 2020.

VILELA, PLÍNIO ROBERTO SOUZA. Critérios potenciais usos de integração: Definição e análise. Dissertação de Doutorado - Universidade Estadual de Campinas, Campinas, 1998.