UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

LEVI RIOS GOMES

AVALIANDO A QUALIDADE DE SUÍTES DE TESTE GERADAS AUTOMATICAMENTE EM DETECTAR FALTAS INTRODUZIDAS POR REFATORAMENTOS.

CAMPINA GRANDE - PB

2023

LEVI RIOS GOMES

AVALIANDO A QUALIDADE DE SUÍTES DE TESTE GERADAS AUTOMATICAMENTE EM DETECTAR FALTAS INTRODUZIDAS POR REFATORAMENTOS.

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

Orientadora: Professora Dra. Melina Mongiovi Brito Lira.

CAMPINA GRANDE - PB

2023

LEVI RIOS GOMES

AVALIANDO A QUALIDADE DE SUÍTES DE TESTE GERADAS AUTOMATICAMENTE EM DETECTAR FALTAS INTRODUZIDAS POR REFATORAMENTOS.

Trabalho de Conclusão Curso apresentado ao Curso Bacharelado em Ciência da Computação do Centro de Engenharia Elétrica e Informática da Universidade Federal de Campina Grande, como requisito parcial para obtenção do título de Bacharel em Ciência da Computação.

BANCA EXAMINADORA:

Professora Dr.(a.) Melina Mongiovi Brito Lira
Orientador – UASC/CEEI/UFCG

Professora Dr.(a.) Patricia Duarte De Lima Machado
Examinador – UASC/CEEI/UFCG

Professor Tiago Lima Massoni
Professor da Disciplina TCC – UASC/CEEI/UFCG

Trabalho aprovado em: 14 de Fevereiro de 2023.

CAMPINA GRANDE - PB

ABSTRACT

Refactorings are the practice where developers alter their code in a way that does not alter the system's behavior. Such a practice is usually accompanied by the use of regression suites to detect unwanted changes in a system's behavior. However, such test suites may not guarantee fault detection, creating a false sense of security during refactorings, since the suite may not detect certain changes. In this work, we propose an approach that aims to evaluate such test suites in their capacity to detect refactoring faults, as well as compare these suites to automated test suites alternatives. For this, a study related to the refactoring faults of the Extract Method refactoring type was carried out, as well as the development of a tool that facilitates the evaluation of a test suite, through a Eclipse IDE plugin made using the Java programming language. From this, refactoring mutants(faults) were created and utilized in a quantitative study, in which we evaluate the regression suites of 3 different open source projects, as well as test suites generated by the generation tool EvoSuite. Our studies show that there is a possible relation between test coverage and refactoring mutant detection in a system, as well as negligence of less common cases during the development of these suites, since about 38.7% of refactoring faults inserted where not detected by the manual test suites, and 45.3% by the automated suites, showing that there is room for improvement in the test suites focused in this context.

Keywords: Refactoring, Test Suites, Mutation Tests.

Avaliando a qualidade de suítes de teste geradas automaticamente em detectar faltas introduzidas por refatoramentos.

Levi Rios Gomes
levi.gomes@ccc.ufcg.edu.br
Universidade Federal de Campina Grande
Campina Grande, Paraíba

Melina Mongiovi
melina@computacao.ufcg.edu.br
Universidade Federal de Campina Grande
Campina Grande, Paraíba

RESUMO

Refatoramentos são a prática em que desenvolvedores alteram seu código de forma que não altere o comportamento do sistema. Tal prática costuma vir acompanhada do uso de suítes de regressão para detectar mudanças de comportamento indesejadas em um sistema. Porém, tais suítes de teste podem não garantir a detecção de faltas, criando um falso senso de segurança durante refatoramentos, visto que a suíte pode não perceber certas alterações. Neste trabalho, propomos uma abordagem que tem por objetivo avaliar tais suítes de teste na sua capacidade de detectar faltas de refatoramento, assim como as comparar com alternativas de geração de suítes de teste automatizadas. Para isso, foi realizado um estudo relacionado às faltas do refatoramento Extract Method, e o desenvolvimento de uma ferramenta que facilita a avaliação de uma suíte de testes, por meio de um plugin da IDE Eclipse feito com a linguagem de programação Java. A partir disso, foram criados mutantes de refatoramento (faltas) e utilizados em um estudo quantitativo, no qual avaliamos as suítes de regressão de 3 diferentes projetos open source, assim como suítes de testes geradas pela ferramenta de geração EvoSuite. Nossos estudos mostraram que existe uma possível relação entre cobertura de testes e detecção de mutantes de refatoramento em um sistema, assim como uma negligência de casos menos comuns durante o desenvolvimento dessas suítes, visto que cerca de 38,7% das faltas de refatoramento injetadas não foram detectadas nas suítes manuais, e 45,3% nas suítes automatizadas, demonstrando que existe espaço para melhoria das suites de teste focadas neste contexto.

Palavras-chave

Refatoramento, Suíte de teste, Testes de Mutação.

1. INTRODUÇÃO

Refatoramentos são a prática de modificar um sistema de software sem que seja alterado seu comportamento, tendo como objetivo melhorar a estrutura interna do código [3]. De acordo com Soares [1] , refatoramentos são edições de código bastante frequentes durante o desenvolvimento de um software, ocorrendo em cerca de 30% de todas as alterações que acontecem durante o seu desenvolvimento.

Considerando fatores como a complexidade da preservação semântica, assim como refatormentos serem uma prática comumente manual, tentativas de aplicação de refatoramentos podem levar à introdução de faltas em um software previamente estável. Além disso, ferramentas que possibilitam o refatoramento de forma automática, como por exemplo Eclipse e NetBeans, podem conter bugs [2]. Estudos que analisam o histórico de projetos de software encontraram uma relação forte entre a aplicação e localização de refatoramentos com o aumento de bugs reportados [4, 5]. Um estudo em específico demonstra que mais de 80% das mudanças em APIs que levam a erros nas aplicações são refatoramentos [6].

Visando ganhar confiança que suas edições recentes não introduziram faltas, desenvolvedores tendem a combinar a aplicação de refatoramentos com a utilização de Conjuntos de Casos de Teste de Regressão, ou Suítes de Regressão [7]. Estas suítes de regressão, ou suítes de testes, são compostas por casos de teste que refletem o comportamento de uma dada versão estável do sistema, e que devem ser utilizados para garantir a permanência do comportamento do sistema após edições.

Porém, ao utilizar uma suíte de testes não eficiente, o desenvolvedor pode ter uma falsa sensação de segurança que seus refatoramentos não inseriram faltas, quando na verdade mudanças de comportamento podem ter sido inseridas e não terem sido detectadas. Tais faltas, quando não detectadas e/ou corrigidas, podem aumentar drasticamente os custos de desenvolvimento de um projeto, assim como afetar negativamente a qualidade do código do sistema.

Tais suítes de regressão também podem ser geradas automaticamente por diversas ferramentas, como o Randoop e o EvoSuite [9, 10]. Essas suítes podem carregar os mesmos problemas das suítes geradas manualmente, porém trazem consigo seus próprios problemas, por não sempre se adaptarem às particularidades de um projeto que um desenvolvedor conseguiria perceber.

Para avaliar este problema, elaboramos uma abordagem que avalie a efetividade de suítes de testes para detectar faltas de refatoramento, para isso, baseamos na metodologia de testes de mutação [8]. Essa metodologia se baseia em testar um sistema inserindo mutantes, entidades que não deveriam existir no sistema, e avaliando a capacidade da suíte de testes em detectar essas entidades. Em nosso caso, vamos inserir refatoramentos com faltas em sistemas selecionados, e vamos avaliar a efetividade da sua suíte de testes em detectar essas faltas.

Elaboramos um catálogo de faltas relacionadas aos possíveis tipos de faltas do *Extract Method*, bem como uma ferramenta que aplica a abordagem proposta para projetos Java.

Podemos ver na Figura 1 um exemplo de falta de refatoramento do tipo *Extract Method*, onde um método novo, printDetails é criado com as funcionalidades das 2 últimas linhas do método original, printClient, porém ele não é invocado no método printClient após a extração, alterado o comportamento do código.

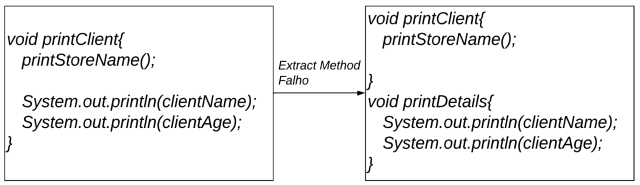

Figura 1: Exemplo de um Extract Method Falho

Fonte: Autoria Própria, 2022

Um estudo de avaliação foi desenvolvido usando três projetos open source. De um total de 75 faltas de refatoramento injetadas nesses sistemas (mutantes de refatoramento), 38,7% permaneceram não detectadas em suítes manuais, e 45,3% em suítes automatizadas, indicando possíveis fraquezas das suítes criadas pelos desenvolvedores dos projetos na detecção de certas mudanças no sistema.

2. METODOLOGIA

Esta seção descreve a metodologia adotada para a construção da abordagem que avalia a efetividade das suítes de testes em detectar faltas de refatoramento, foram realizadas as seguintes atividades:

2.1: Criação do catálogo de faltas: Descrevemos como foi elaborado o catálogo de faltas[1], que descreve os possíveis tipos de falta de um *Extract Method*.

2.2: Plug-ins para refatoramento e inserção de faltas: Descrevemos como foi realizado o desenvolvimento dos plug-ins que estão disponíveis para uso em um repositório do GitHub[2], assim como seu funcionamento.

2.3: Estudo empírico quantitativo: Descrevemos como foi realizado a análise das suítes de teste avaliadas.

2.1 Criação do catálogo de faltas

Primeiramente, realizamos a criação de um catálogo de faltas, detalhando os diferentes tipos de possíveis faltas em um refatoramento do tipo *Extract Method*. Esse tipo de refatoramento inclui a extração de um trecho de um método específico para um novo método externo, que será chamado pelo método original.

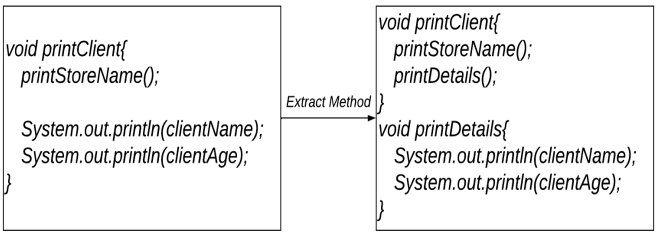

Figura 2: Exemplo de um Extract Method realizado corretamente

Fonte: Autoria Própria, 2022

Os passos para aplicar um *Extract Method* de forma segura são ilustrados por Fowler da seguinte maneira [3]:

1 - Criação de um novo método e nomeado de acordo com a sua função.

2 - Copiar o código extraído do método original para o novo método.

3 - Verificar os escopos de variáveis, as instanciando ou recebendo como parâmetro, caso seja necessário.

4 - Remover o código extraído no método de origem e o substituir pela chamada do método novo.

5 - Finalmente, compilar e testar o sistema após o refatoramento.

O catálogo de faltas, que se encontra disponível para análise, exemplifica alguns possíveis tipos de faltas que podem acontecer durante o *Extract Method*, assim como meios de reproduzi-las, para propósitos de teste. Destes tipos, produzimos 3 tipos de simulações de falta que seriam implementadas posteriormente por meio de um plug-in do Eclipse:

1. Last Statement Removal(LSR): Simula a remoção indevida de um statement durante o refatoramento, em específico o único possivelmente removível.

2. Return Replacement(RR): Simula a alteração indevida de um return statement, trocando o valor que deveria ser retornado pelo método alterado por um valor do mesmo tipo diferente do valor original do método.

3. Null Assignment(NA): Simula a alteração indevida de um *statement* de atribuição, alterando a atribuição para um valor nulo.

2.2 Plug-ins para refatoramento e inserção de faltas

Após a criação do catálogo de faltas, foi necessário a criação de uma ferramenta que pudesse ser utilizada para facilitar a inserção de faltas em um sistema, para que a suíte de regressão relacionada a ela fosse avaliada. A partir disso, foi decidido o desenvolvimento da ferramenta por meio de dois plugins para a Eclipse IDE, um para o refatoramento e outro para a inserção da falta no código após o refatoramento.

[1]https://docs.google.com/document/d/1FpcHinJhwUYzGZOK8Of6WlGGEvmdfrFlabGRWuixmOU/edit?usp=sharing
[2]https://github.com/LeviG99/TCC-Plugins

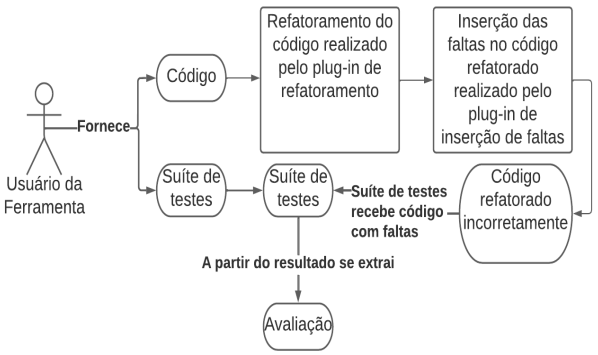

Figura 3: Diagrama da abordagem de avaliação

Fonte: Autoria Própria, 2022

O plug-in de refatoramento recebe o código fornecido pelo usuário, e, após selecionados o método e o intervalo que será realizado o refatoramento, é realizado o *Extract Method* no método selecionado, com o método auxiliar sendo criado logo após o original. Esse plug-in se utiliza das próprias capacidades de refatoramento do Eclipse, realizando o refatoramento programaticamente em código Java.

O plug-in de inserção de falta é executado logo após o fim da execução do refatoramento, realizando um dos 3 tipos de simulação de faltas: LSR, RR e NA. Após requisitar os atributos necessários para a inserção destas faltas, é realizada a inserção de uma dessas faltas no código, podendo agora ele ser inserido no sistema original para avaliação do sistema.

Nossa abordagem para a avaliação da suíte de teste, tendo como base a utilização do plug-in, pode ser vista na figura 3, que demonstra como é o funcionamento da abordagem como um todo.

2.3 Estudo Empírico Quantitativo

Para o nosso estudo, selecionamos 3 projetos *open source*, selecionados pela sua diversidade de código assim como sua compatibilidade com a abordagem:

- Tracking Things: Sistema de cadastro de itens e gerenciamento de empréstimos a usuários cadastrados. Selecionado por ser um projeto de menor escopo.
- La4j: Biblioteca Java que fornece primitivas (matrizes e vetores) e algoritmos de Álgebra Linear. Selecionado por ser um projeto que trabalha com uma variedade de operações matemáticas e pelo seu uso de variáveis primitivas.
- Graphhopper/core: É um mecanismo de roteamento Java que promete agir de forma rápida e com uso eficiente de memória. Selecionado por seu escopo maior, com vários casos de utilização de diferentes tipos de objetos.

Na tabela 1, podemos ver as métricas das suítes de testes destes projetos, obtidas por meio da avaliação de cobertura provida pelo Eclipse IDE.

Projeto	Tamanho (LOC)	Quantidade de Testes	Cobertura dos testes (linhas)	Cobertura dos testes (branches)
TrackingThings	1760	169	85.1%	71.5%
La4j	7846	835	83.2%	69.2%
Graphhopper-core	3541	2016	91.3%	76.0%

Tabela 1: Informações sobre os projetos estudados e suas suítes de teste

Fonte: Autoria Própria, 2022

Para a realização do estudo empírico desses projetos, foram inserido de forma isolada 75 faltas de de refatoramento, sendo 30 LSR e 30 RR, por serem tipos de faltas que podem passar mais facilmente despercebidas, e 15 faltas do tipo NA, essas sendo faltas que, pelo fato de terem uma chance maior de causarem erros de compilação, são mais facilmente percebidas, porém podem causar muitos problemas quando não detectadas.

Após isso, analisamos a efetividade das suítes de testes manuais existentes no projeto em detectar as faltas inseridas, e foram geradas suítes de testes utilizando o EvoSuite, e estas também foram avaliadas.

3. RESULTADOS

3.1 Suítes de teste manuais

Após a inserção das 75 faltas, 38.7% de todas as faltas inseridas não foram detectadas, como pode ser percebido no gráfico 1. Dessas, 43.3% das faltas do tipo LSR não foram detectadas, 43.3% das faltas do tipo RR não foram detectadas, e 20% das faltas do tipo NA não foram detectadas, como pode ser visto nos gráficos 2, 3 e 4 respectivamente.

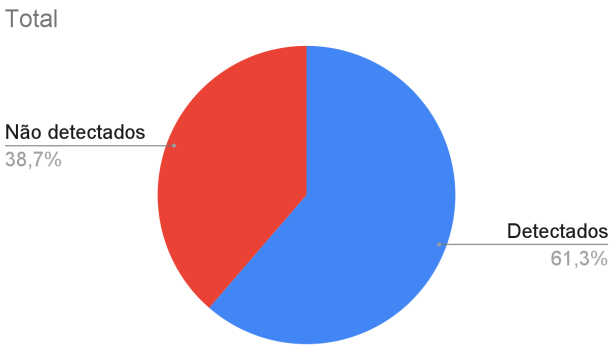

Gráfico 1: Detecção de faltas em todos os 3 projetos de todos os tipos

Fonte: Autoria Própria, 2022

Essa porcentagem demonstra que mesmo essas suítes de teste sendo criadas de forma manual, uma parte significativa das falhas não foi detectada.

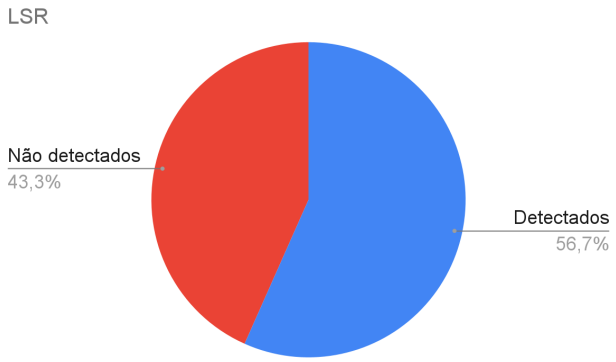

Gráfico 2: Detecção de faltas em todos os 3 projetos do tipo LSR

Fonte: Autoria Própria, 2022

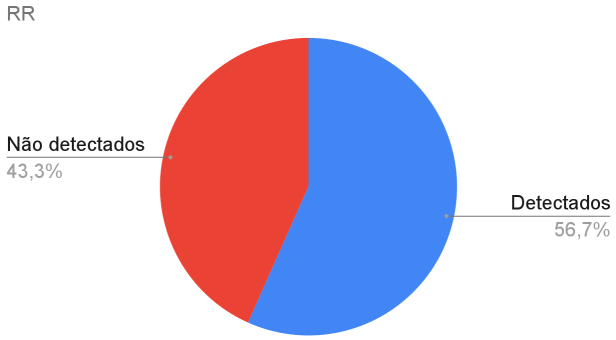

Gráfico 3: Detecção de faltas em todos os 3 projetos do tipo RR

Fonte: Autoria Própria, 2022

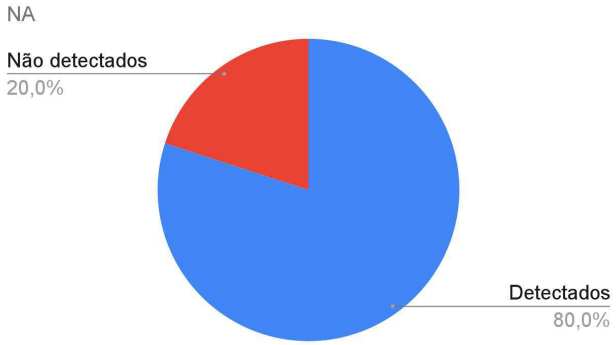

Gráfico 4: Detecção de faltas em todos os 3 projetos do tipo NA

Fonte: Autoria Própria, 2022

Podemos perceber que enquanto a porcentagem de detecção de faltas do tipo LSR e RR se mostram próximas ao do total de falhas, o percentual de falhas NA não detectadas é significativamente menor. Isso se dá pelo fato de que a atribuição de valores para um valor nulo, de forma proposital, possui uma chance maior de gerar erros de compilação do que os outros dois tipos de falta. Essa falta foi desenvolvida com o propósito de causar erros de compilação e ser mais facilmente detectada, porém, caso não seja, ela é mais grave do que a não detecção de outros tipos de falta.

Os projetos, individualmente, apresentaram particularidades em sua capacidade de detecção. Enquanto TrackingThings apresenta uma menor detecção de mutantes do tipo RR, detectando somente 20% das falhas desse tipo, o Graphhopper-core apresenta uma capacidade maior de detectar esses tipos de falha, porém detecta somente 40% das faltas do tipo LSR. TrackingThings e Graphhopper-core ambos detectam a maioria dos mutantes do tipo NA, detectando 60% e 80%, respectivamente. O número de faltas não detectadas nos projetos TrackingThings, la4j e Graphhopper-core são ilustrados nos gráficos 5, 6 e 7, respectivamente.

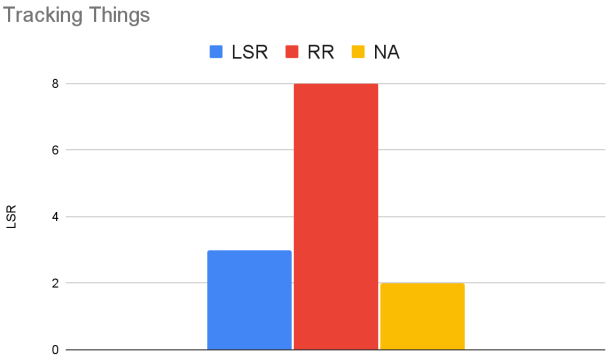

Gráfico 5: Não Detecção de faltas no projeto Tracking Things

Fonte: Autoria Própria, 2022

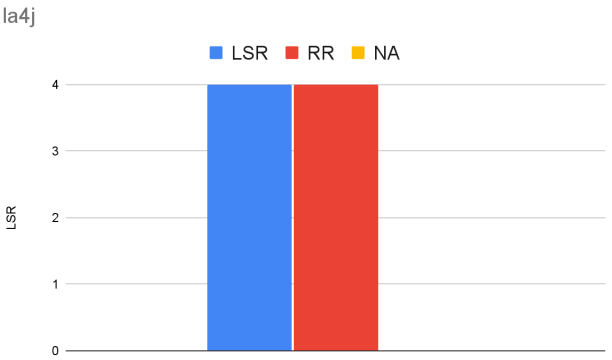

Gráfico 6: Não Detecção de faltas no projeto la4j

Fonte: Autoria Própria, 2022

Podemos perceber no gráfico 6 que todas as 5 faltas NA inseridas no projeto foram detectadas. Isso se dá ao fato de que este projeto trabalha de forma principal com variáveis primitivas, e essas quando atribuídas para um valor nulo, sempre causam erro de compilação.

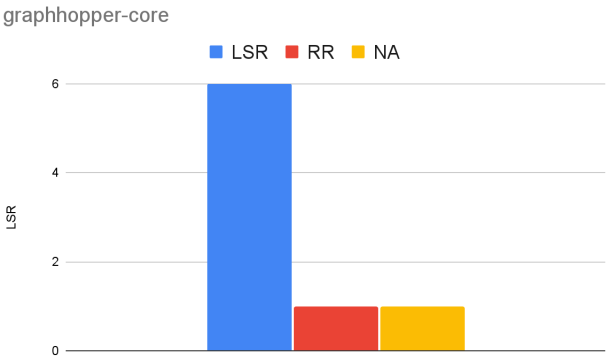

Gráfico 7: Não Detecção de faltas em todos no projeto graphhopper-core

Fonte: Autoria Própria, 2022

Durante a avaliação, algumas falhas foram inseridas com focos diferentes:

- 8 faltas do tipo LSR foram inseridas especificamente em trechos de código relacionados a exceções.
- 9 faltas do tipo RR foram inseridas especificamente em trechos de código relacionados a códigos condicionais.

Das faltas inseridas em contextos de exceção, 100% foram detectadas no projeto TrackingThings, e nenhuma dessas faltas foram detectadas nos outros dois projetos. Isso pode indicar que os dois projetos de maior porte não tendem a avaliar exceções durante a criação de seus testes.

Das faltas inseridas em contextos condicionais, 6 das faltas inseridas não foram detectadas, por mais que possivelmente pudessem alterar o comportamento do código, isso pode indicar que essas suítes de teste não tendem a avaliar situações não comuns no código.

Vale ressaltar que, durante a realização dos testes, foi percebido que nos casos de atribuições em conjunto com operações, como "+=", demonstravam mais vulnerabilidade do que as atribuições sem operações.

3.2 Suítes de teste automatizadas

Após analisados os resultados das suítes de teste manuais, voltamos nossa atenção para as suítes geradas pelo EvoSuite.

EvoSuite [10] é uma ferramenta de geração automática de testes para Java. Ela se baseia em uma abordagem híbrida, que gera e otimiza suítes de teste inteiras para satisfazer um critério de cobertura. O EvoSuite adiciona conjuntos pequenos de verificações, que capturam o comportamento atual do sistema, fazendo com que seja possível detectar desvios de comportamento.

Para o contexto do nosso projeto, foram geradas suítes de teste para cada um dos 3 projetos, e avaliamos a capacidade dessas suítes em detectar as faltas que inserimos.

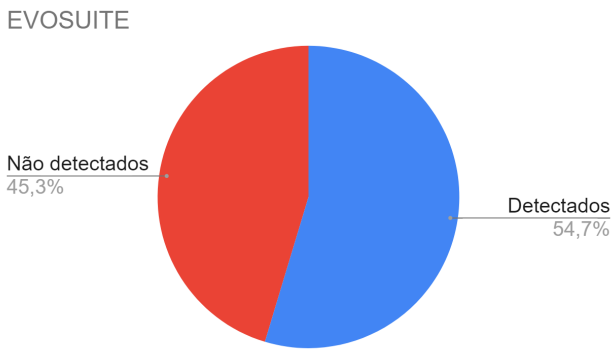

Gráfico 8: Detecção de faltas em todos os 3 projetos nas suíte de testes geradas pelo EvoSuite

Fonte: Autoria Própria, 2022

Como podemos ver no gráfico 8, as suítes de teste automatizadas detectaram 54,7% das faltas inseridas nos projetos, demonstrando uma efetividade menor do que as suítes de teste manuais.

Tipo de Falta	Suítes de teste manuais	Suítes de teste automatizadas
LSR	56.7%	40.0%
RR	56.7%	56.7%
NA	80.0%	80.0%
Total	61.3%	54.7%

Tabela 2: Informações sobre a detecção de faltas nas suítes manuais e automatizadas

Fonte: Autoria Própria, 2022

As suítes de teste geradas automaticamente demonstraram certas particularidades, como:

- Uma baixa detecção de faltas inseridas em contextos de exceção, tendo uma taxa de detecção ainda menor do que as suítes manuais.
- Uma maior habilidade de detecção de faltas do tipo RR
- Uma menor habilidade na detecção de faltas do tipo LSR

Além dessas particularidades, as características de detecção de falta das suítes geradas automaticamente se assemelhavam às suítes manuais, porém com resultados menos eficazes, causados principalmente pela dificuldade de detecção de faltas em contextos de exceção.

Vale também ressaltar que as essas suítes detectaram certas faltas de refatoramento não detectadas pelas suítes manuais, demonstrando uma detecção de 77,3% das faltas quando ambas são utilizadas em conjunto, demonstrando uma percepção do comportamento do código em certas situações que não são percebidas pelas suítes manuais.

4. AMEAÇAS A VALIDADE

Os resultados deste estudo não são generalizáveis, se aplicando somente aos projetos avaliados e as faltas inseridas. Mas acreditamos que os projetos selecionados e faltas inseridas sejam representativos de um contexto geral.

O processo de injeção de faltas por meio da ferramenta desenvolvida poderia ser considerado uma possível ameaça. Porém, as faltas foram inseridas de forma aleatória, e de forma padronizada por meio do comportamento da ferramenta, o que garante um padrão na inserção das faltas do sistema e uma confiança na detecção e não detecção das faltas no sistema.

Por mais que existam outras possíveis métricas de avaliação de efetividade das suítes de regressão, optamos por avaliar sua taxa de detecção de faltas, por ser uma métrica utilizada comumente em trabalhos similares, e por ser mais representativo da funcionalidade da ferramenta criada.

Diferentes ferramentas de geração de suítes de teste poderiam ter sido utilizadas para o estudo, como o Randoop. Porém, selecionamos o Evo Suite por ser a ferramenta mais famosa, e de utilização mais acessível.

5. TRABALHOS RELACIONADOS

Alves et al. [12] analisaram a eficácia de suítes manuais na validação de refatoramentos. Eles apresentaram que as suítes de teste feitas manualmente podem não ser suficientes para validação de código. Logo, nesse estudo utilizamos também as suítes de teste automatizadas por meio do Evo Suite, e utilizamos da ferramenta de inserção de faltas para melhor validar a eficácia das suítes.

Silva et al. [11] avaliou a eficácia de suítes automatizadas em detectar faltas de refatoramento. Nos aprofundamos essa avaliação por meio da criação de uma ferramenta visando padronizar e facilitar a inserção de faltas para avaliação de um sistema, e as aplicando tanto a suítes manuais como automatizadas.

6. CONCLUSÃO

Nesta pesquisa, foi elaborada abordagem que utiliza-se de plugins da Eclipse IDE que facilitam a avaliação de qualidade de suítes de teste, tendo como foco o tipo de refatoramento *Extract Method*. Para isso, desenvolvemos uma ferramenta que aplica automaticamente uma simulação de refatoramentos feitos de maneira incorreta, tendo como tipos de faltas LSR, RR e NA, estas faltas simulam faltas comuns relacionadas ao *Extract Method*.

Nosso estudo quantitativo apresenta que as suítes de teste manuais avaliadas possuíam a capacidade de detectar a maioria das faltas, mas ainda cerca de 38.7% dos mutantes de refatoramentos permaneceram não detectados, o que indica vulnerabilidade dessas suítes em certas circunstâncias.

Observamos ainda que certos trechos de código, em específico aqueles que não são garantidos de ocorrerem durante a execução do programa e aqueles com atribuições em conjunto com operações, podem não estar recebendo a devida atenção durante a criação de suítes.

No contexto de suítes de teste geradas automaticamente, estas se demonstraram menos eficazes em detectar faltas do que as realizadas manualmente, detectando somente 54,7% das faltas inseridas, sendo especialmente vulneráveis a faltas inseridas nos contextos de exceções.

Observamos também que as suítes geradas pelo EvoSuite podem detectar mudanças de comportamento que não são detectadas pelas suítes manuais, demonstrando que o uso de suítes de teste manuais e automatizadas em conjunto diminui a chance de faltas serem inseridas no sistema, podendo detectar até 77,3% das faltas inseridas.

Acreditamos que estes resultados podem ajudar os testadores a facilitar seu processo de avaliar suas suítes, perceber áreas vulneráveis a faltas de refatoramento em seus sistemas, assim como guiar esforços de teste.

A partir disso, um possível trabalho futuro com a avaliação de suítes de testes geradas automaticamente seria vantajoso para comparar a efetividade dessas suítes quando comparadas com as criadas de forma manual.

7. REFERÊNCIAS

[1] SOARES, G. et al. Analyzing refactorings on software repositories. In: IEEE. Software Engineering (SBES), 2011 25th Brazilian Symposium on. [S.l.], 2011. p. 164–173

[2] MONGIOVI, M.; GHEYI, Rohit ; SOARES, Gustavo ; RIBEIRO, MARCIO ; BORBA, Paulo ; TEIXEIRA, Leopoldo. Detecting Overly Strong Preconditions in Refactoring Engines. IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, v. 44, 2018.

[3] FOWLER, Martin et al. Refactoring: Improving the Design of Existing Code. 2. ed. [S. l.: s. n.], 2018.

[4] KIM, M.; CAI, D.; KIM, S. An empirical investigation into the role of api-level refactorings during software evolution. In: ACM. Proceedings of the 33rd International Conference on Software Engineering. [S.l.], 2011. p. 151–160.

[5] WEISSGERBER, P.; DIEHL, S. Are refactorings less error-prone than other changes? In: ACM. Proceedings of the 2006 international workshop on Mining software repositories. [S.l.], 2006. p. 112–118

[6] DIG, D.; JOHNSON, R. How do apis evolve? a story of refactoring. Journal of software maintenance and evolution: Research and Practice, Wiley Online Library, v. 18, n. 2, p. 83–107, 2006.

[7] MOSER, R. et al. A case study on the impact of refactoring on quality and productivity in an agile team. Balancing Agility and Formalism in Software Engineering, Springer, p. 252–266, 2008.

[8] BROWN, D. B. . Mutation Testing : Algorithms and Applications. Estados Unidos: University of Wisconsin-Madison, 2020.

[9] Carlos Pacheco, Michael D. Ernst. Randoop: Feedback-directed random testing for java. In Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion, OOPSLA '07, New York, NY, USA, 2007. ACM.

[10] Gordon Fraser and Andrea Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE '11, New York, NY, USA, 2011. ACM

[11] SILVA, Indy et al. Avaliando Suites Automaticamente Geradas para Validação de Refatoramentos. Orientadores: Everton Leandro Galdino Alves, Patrícia Duarte de Lima Machado. 2018. Dissertação (Mestrado em Ciência da computação.) - Universidade Federal de Campina Grande, [S. l.], 2018.

[12] Everton L.G. Alves, Tiago Massoni, and Patrícia Duarte de Lima Machado. 2017. Test coverage of impacted code elements for detecting refactoring faults: An exploratory study. Journal of Systems and Software 123 (2017), 223–238. https: //doi.org/10.1016/j.jss.2016.02.001