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

CÁSSIO EDUARDO GABRIEL CORDEIRO

ANALISANDO SUÍTES DE TESTE MANUAIS E AUTOMÁTICAS PARA IDENTIFICAR FALTAS DE REFATORAMENTO

CAMPINA GRANDE - PB

2021

CÁSSIO EDUARDO GABRIEL CORDEIRO

ANALISANDO SUÍTES DE TESTE MANUAIS E AUTOMÁTICAS PARA IDENTIFICAR FALTAS DE REFATORAMENTO

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.

Orientador: Professor Dr. Everton L. G. Alves.

CAMPINA GRANDE - PB

2021

C794a Cordeiro, Cássio Eduardo Gabriel.
Analisando suítes de teste manuais e automáticas para identificar faltas de refatoramento. / Cássio Eduardo Gabriel Cordeiro. – 2021.

10 f.

Orientador: Prof. Dr. Everton Leandro Galdino Alves.
Trabalho de Conclusão de Curso - Artigo (Curso de Bacharelado em Ciência da Computação) - Universidade Federal de Campina Grande; Centro de Engenharia Elétrica e Informática.

1. Suítes de teste - análise. 2. Faltas de refatoramento. 3. Manutenibilidade de software. 4. Edições de refatoramento. 5. Faltas tipo Extract Method. 6. Randoop. 7. Evosuite. 8. Suite manual. 9. Suite automática. 10. Refatoramento. 11. Qualidade de software. 12. Engenharia de software. I. Alves, Everton Leandro Galdino. II. Título.

CDU:004.052.4(045)

Elaboração da Ficha Catalográfica:

Johnny Rodrigues Barbosa
Bibliotecário-Documentalista
CRB-15/626

CÁSSIO EDUARDO GABRIEL CORDEIRO

ANALISANDO SUÍTES DE TESTE MANUAIS E AUTOMÁTICAS PARA IDENTIFICAR FALTAS DE REFATORAMENTO

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:

Professor Dr. Everton L. G. Alves

Orientador – UASC/CEEI/UFCG

Professor Dr. Rohit Gheyi

Examinador – UASC/CEEI/UFCG

Professor Tiago Lima Massoni

Professor da Disciplina TCC – UASC/CEEI/UFCG

Trabalho aprovado em: 25 de MAIO de 2021.

CAMPINA GRANDE - PB

ABSTRACT

Developing systems with high quality involves activities that allow easy maintenance and provide confidence about the code produced. Software tests are related to reliability, already refactoring, with maintainability. By definition, refactoring editions aim to improve the code's structure, while preserving its behavior. However, bad refactoring can change the behavior of the system, they are called refactoring faults. Such faults may not be detected by unreliable test suites. An alternative for the systematic creation of test suites is the use of automatic generation tools. This work aims to evaluate the effectiveness of test suites generated manually and automatically to detect refactoring faults of the Extract Method type. For this, projects written in Java were selected, with test suites generated manually, new test suites were created automatically with the tools Randoop and EvoSuite, a set of faults were injected into the systems. Manual suites detected 61.9% of injected faults, while the Randoop suite detected only 46.7% and EvoSuite 55.8%. Randoop obtained a low detection rate, EvoSuite, however, obtained a result significantly comparable to that of manual suites.

Analisando Suítes de Teste Manuais e Automáticas para Identificar Faltas de Refatoramento

Cássio Eduardo Gabriel Cordeiro
cassio.cordeiro@ccc.ufcg.edu.br
Universidade Federal de Campina Grande
Campina Grande, Paraíba, Brasil

Everton L. G. Alves
everton@computacao.ufcg.edu.br
Universidade Federal de Campina Grande
Campina Grande, Paraíba, Brasil

RESUMO

Desenvolver sistemas com alta qualidade envolve atividades que permitam fácil manutenção e forneçam confiança sobre o código produzido. Testes de software se relacionam com a confiabilidade, já refatoramentos, com manutenibilidade. Por definição, edições de refatoramento objetivam melhorar estrutura do código, mas preservando seu comportamento. Porém, refatoramentos mal feitos podem alterar o comportamento do sistema, são as chamadas faltas de refatoramento. Tais faltas, podem não ser detectadas por suítes de teste pouco confiáveis. Uma alternativa para criação sistemática de suítes de teste é a utilização de ferramentas de geração automática. Este trabalho tem como objetivo avaliar a efetividade de suítes de teste geradas manual e automaticamente para detectar faltas de refatoramento do tipo Extract Method. Para isso, foram selecionados projetos escritos em Java, com suítes de teste geradas manualmente, novas suítes de testes foram criadas automaticamente com as ferramentas Randoop e EvoSuite, um conjunto de faltas foram injetadas nos sistemas. As suítes manuais detectaram 61,9% das faltas injetadas, enquanto a suíte Randoop detectou apenas 46,7% e a EvoSuite 55,8%. A Randoop obteve uma taxa de detecção baixa, a EvoSuite, no entanto, obteve um resultado significantemente comparável ao de suítes manuais.

1 INTRODUÇÃO

A Engenharia de software aborda técnicas para o desenvolvimento de sistemas com alta qualidade. Duas características de qualidade são a Manutenibilidade e a Confiabilidade [14]. Os testes de software se relacionam com a Confiabilidade, já que fornecem à equipe de desenvolvimento maior confiança sobre o código produzido. Testes que falham podem apontar possíveis divergências entre o comportamento implementado e o esperado do sistema. Suítes de teste podem ser criadas de duas formas: manual ou automaticamente. Testes gerados manualmente, são aqueles escritos pelos próprios desenvolvedores. Já os automáticos, são gerados por ferramentas especializadas, como Randoop [4] e EvoSuite [2].

Durante o desenvolvimento e a manutenção do software, uma das atividades que ocorre com bastante frequência é o refatoramento. Refatoramentos são edições de código com o objetivo de aperfeiçoar sua estrutura interna, melhorando a legibilidade, coesão e organização [6], mas mantendo o comportamento original. Existem diversos tipos de refatoramento, como por exemplo o *Extract Method*, o *Rename Variable* e o *Move Method*.

Efeitos colaterais do refatoramento de código podem alterar indevidamente o comportamento externo de um sistema. São as chamadas *faltas de refatoramento*. Algumas delas são difíceis de detectar, já que podem não provocar erros de compilação ou falhas de sistema, podendo, inclusive, ocorrer apenas em situações específicas e/ou em funcionalidades menos utilizadas. Desta forma, suítes de teste de regressão são normalmente usadas para aumentar a confiabilidade que edições de refatoramento não introduziram novas faltas [5].

Suítes de teste com pouca qualidade podem oferecer o falso sentimento de que edições de refatoramento foram seguras [11, 13]. Desta forma, faz-se necessário avaliar a qualidade dessas suítes. Uma vez que as suítes geradas automaticamente estão cada vez mais ganhando espaço, também é necessário avaliar se os testes gerados são mais ou menos efetivos para a validação de edições de refatoramento, em comparação com testes manuais. É interessante saber também se a combinação de testes manuais e automáticos trazem ganhos para este contexto.

Este trabalho se propõe a realizar um estudo investigativo com o objetivo de avaliar a efetividade de suítes de teste geradas manual e automaticamente para detectar faltas de refatoramento. Para tal, selecionamos 7 projetos open-source. Os projetos possuem uma suíte manual, escrita pelos desenvolvedores. No contexto do nosso estudo, foram geradas suítes de teste usando as ferramentas Randoop e EvoSuite. Faltas de refatoramento relacionadas a edições do tipo *Extract Method* foram injetadas sistematicamente. Por fim, analisamos a capacidade de detecção de cada suíte.

2 EXEMPLO MOTIVACIONAL

Para exemplificar um Extract Method realizado manualmente de forma incorreta, considere a figura 1. Para realizar o refatoramento, foi criado o novo método, o código das linhas 28 a 33 foi substituído por uma chamada do novo método e, por fim, ele foi implementado, resultando no código da figura 2. Porém, durante este processo, o *statement* responsável pela ordenação da coleção (linhas 28 e 29) foi esquecido, causando uma mudança de comportamento no sistema, e consequentemente, uma falta de refatoramento. Este é uma falta comum de ocorrer [8] e difícil de identificar, uma vez que não gera erro de compilação.

Caso as suítes não incluam testes em que as disciplinas são inseridas não ordenadas, possuam somente testes com apenas uma disciplina ou todas as disciplinas tenham a mesma carga horária,

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

elas podem não detectar a falta, deixando o desenvolvedor que realizou o refatoramento com uma falsa sensação de segurança.

```java
class Disciplina {
    private String nome;
    private int cargaHoraria;

    // ...

    @Override
    public String toString() {
        return this.nome + " - " + this.cargaHoraria + " horas";
    }
}

class Aluno {
    private String nome;
    private String matricula;
    private List<Disciplina> disciplinas;

    // ...

    public String getAlunoDetalhes() {
        StringBuilder detalhes = new StringBuilder();
        detalhes.append("Nome: " + this.nome + "\n")
                .append("Matricula: " + this.matricula + "\n")
                .append("Disciplinas: \n");

        Collections.sort(this.disciplinas,
                (d1, d2) -> d1.getCargaHoraria() - d2.getCargaHoraria());

        for (Disciplina disciplina : this.disciplinas) {
            detalhes.append("\t" + disciplina.toString() + "\n");
        }

        return detalhes.toString();
    }
}
```

Figura 1: Exemplo de um Extract Method (código original).

```java
class Disciplina {
    private String nome;
    private int cargaHoraria;

    // ...

    @Override
    public String toString() {
        return this.nome + " - " + this.cargaHoraria + " horas";
    }
}

class Aluno {
    private String nome;
    private String matricula;
    private List<Disciplina> disciplinas;

    // ...

    public String getAlunoDetalhes() {
        StringBuilder detalhes = new StringBuilder();
        detalhes.append("Nome: " + this.nome + "\n")
                .append("Matricula: " + this.matricula + "\n")
                .append("Disciplinas: \n")
                .append(getDisciplinasDetalhes());

        return detalhes.toString();
    }

    private String getDisciplinasDetalhes() {
        StringBuilder detalhesDisciplinas = new StringBuilder();

        for (Disciplina disciplina: this.disciplinas) {
            detalhesDisciplinas.append("\t" + disciplina.toString() + "\n");
        }

        return detalhesDisciplinas.toString();
    }
}
```

Figura 2: Exemplo de um Extract Method que causou uma falta (código refatorado).

3 FUNDAMENTAÇÃO TEÓRICA

3.1 Extract Method

No seu livro, Fowler define Extract Method como o processo de transformar um trecho de código em um método, cujo nome explica o seu propósito [6]. Refatoramentos do tipo Extract Method são dos mais comuns feitos por desenvolvedores [10]. Na ferramenta JDeodorant [3], ele aparece como o mais aplicado [3].

Fowler define a seguinte mecânica de passos para aplicar um Extract Method de forma segura [6]:

- Criar um novo método e nomeá-lo de acordo com a sua função;
- Copiar o código extraído do método de origem para o novo método;
- Verificar escopos de variáveis, instanciando ou recebendo como parâmetro, caso necessário;
- Substituir o código extraído no método de origem por uma chamada do novo método;
- Compilar;
- Testar.

Faltas de refatoramento do tipo Extract Method podem ocorrer quando algum dos passos básicos é negligenciado (*missing edit*) ou quando elementos extra de código são adicionados *(extra edit*) [13].

Apesar de existir ferramentas que auxiliam na aplicação de edições Extract Method, na prática, grande parte ocorre de forma manual [7], aumentando a possibilidade de introduzir faltas e provocar mudanças de comportamento no sistema.

A Figura 3 apresenta a aplicação correta do refatoramento do código da Figura 1. Foi criado um novo método, getDisciplinasDetalhes, as linhas 28 a 33 foram extraídas para ele, foram instanciadas as variáveis necessárias e, por fim, o código extraído foi substituído por uma chamada do método criado.

3.2 Randoop

A ferramenta Randoop[4] gera automaticamente suítes de testes unitários para projetos Java. Ela utiliza uma abordagem dirigida por *feedback*, que gera sequências de invocações de métodos e a partir da execução delas, usa os resultados para criar as verificações que capturam o comportamento do sistema. Os testes gerados focam em trechos que possam causar violações básicas de contrato. Uma das finalidades da Randoop é criar suítes de teste de regressão, que passaram a ser utilizadas na validação de refatoramentos. A Figura 4 apresenta um teste gerado pela Randoop para o código da Figura 1.

3.3 Evosuite

EvoSuite[2] é outra ferramenta de geração automática de testes para Java. Ela utiliza uma abordagem híbrida, que gera e otimiza suítes de teste inteiras para satisfazer um critério de cobertura. A EvoSuite adiciona conjuntos pequenos de verificações, que capturam o comportamento atual do sistema, fazendo com que seja possível detectar desvios de comportamento. A Figura 5 apresenta um teste gerado pela EvoSuite para o código da Figura 1;

```java
class Disciplina {
    private String nome;
    private int cargaHoraria;

    // ...

    @Override
    public String toString() {
        return this.nome + " - " + this.cargaHoraria + " horas";
    }
}

class Aluno {
    private String nome;
    private String matricula;
    private List<Disciplina> disciplinas;

    // ...

    public String getAlunoDetalhes() {
        StringBuilder detalhes = new StringBuilder();
        detalhes.append("Nome: " + this.nome + "\n")
                .append("Matricula: " + this.matricula + "\n")
                .append("Disciplinas: \n")
                .append(getDisciplinasDetalhes());

        return detalhes.toString();
    }

    private String getDisciplinasDetalhes() {
        StringBuilder detalhesDisciplinas = new StringBuilder();

        Collections.sort(this.disciplinas,
                (d1, d2) -> d1.getCargaHoraria() - d2.getCargaHoraria());

        for (Disciplina disciplina: this.disciplinas) {
            detalhesDisciplinas.append("\t" + disciplina.toString() + "\n");
        }

        return detalhesDisciplinas.toString();
    }
}
```

Figura 3: Exemplo de um Extract Method (código refatorado).

```java
@Test
    public void test2() throws Throwable {
        if (debug)
            System.out.format("\n%s%n", "RegressionTest0.test2");
        java.util.List list2 = null;
        Aluno aluno3 = new Aluno("", "hi!", list2);
        // The following exception was thrown during execution in
  test generation
        try {
            java.lang.String str4 = aluno3.getAlunoDetalhes();
            org.junit.Assert.fail("Expected exception of type
  java.lang.NullPointerException; message: null");
        } catch (java.lang.NullPointerException e) {
            // Expected exception.
        }
    }
```

Figura 4: Exemplo de teste criado pela Randoop para o código da figura 1.

4 ESTUDO INVESTIGATIVO

4.1 Objetivos e Questões de pesquisa

Este estudo visa analisar a eficácia de suítes de teste manuais e automáticas na detecção de faltas de refatoramento, bem como identificar se existe um tipo de suíte de teste, ou conjunto de suítes, mais eficaz em detectar tais problemas e os aspectos que impactaram na taxa de detecção das suítes criadas pelas ferramentas de geração automática.

```java
@Test(timeout= 4000)
    public void test2() throws Throwable {
        LinkedList<Disciplina> linkedList0 = new LinkedList<>();
        Disciplina disciplina0 = new Disciplina("", 0);
        linkedList0.add(disciplina0);
        Aluno aluno0 = new Aluno("\t", "", linkedList0);
        String string0 = aluno0.getAlunoDetalhes();
        assertEquals("Nome: \t\nMatr\u00EDcula: \nDisciplinas: \n\t
  - 0 horas\n", string0);
    }
```

Figura 5: Exemplo de teste criado pela EvoSuite para o código da figura 1.

Para guiar o estudo, foram estabelecidas as seguintes questões de pesquisa:

- **RQ1**: Testes automaticamente gerados são mais eficientes que os testes manuais para detecção de faltas de refatoramento?
- **RQ2**: Qual ferramenta gera suítes mais eficientes para detectar faltas de refatoramento?
- **RQ3**: A combinação de diferentes suítes geradas melhora a detecção de faltas de refatoramento?

4.2 Metodologia

Para o nosso estudo, selecionamos sete projetos *open-source*. Os projetos são todos escritos em Java, hospedados no GitHub e possuem suítes de teste manuais escritas pelos desenvolvedores. A seguir, apresentamos uma breve descrição de cada projeto.

- Tracking Things[1]: Sistema de cadastro de itens e gerenciamento de empréstimos a usuários cadastrados;
- Santuário XML Security Java[2]: O projeto visa fornecer a implementação dos padrões de segurança primários para XML: Sintaxe e processamento de assinatura XML e Sintaxe e processamento de criptografia XML;
- TCC manager[3]: Sistema de gerenciamento de TCC 's por professores;
- La4j[4]: Biblioteca Java que fornece primitivas (matrizes e vetores) e algoritmos de Álgebra Linear;
- Graphhopper/core[5]: É um mecanismo de roteamento Java rápido e com uso eficiente de memória;
- Dubbo-remoting-api[6]: É uma estrutura RPC de código aberto baseada em Java de alto desempenho;
- SAD-UFCG[7]: É um sistema que tem o objetivo de obter o feedback das matrículas em relação à qualidade da docência com o intuito de melhorar a experiência de aula dos matrículas;

Para cada projeto, foram geradas automaticamente duas suítes de teste, uma com a ferramenta Randoop e outra com a EvoSuite.

[1]https://github.com/marcelovitorino/TrackingThings
[2]https://github.com/apache/santuario-xml-security-java
[3]https://github.com/brunojdo/tcc-manager
[4]https://github.com/vkostyukov/la4j
[5]https://github.com/graphhopper/graphhopper/tree/master/core
[6]https://github.com/apache/dubbo/tree/master/dubbo-remoting/dubbo-remoting-api
[7]https://github.com/sad-ufcg/back-end

Tabela 1: Projetos (LOC - Linhas de código, TR - Testes Randoop, TE - Testes EvoSuite, TM - Testes manuais, CC - Cobertura de código das suítes manuais).

Projeto	LDC	TR	TE	TM	CC
Tracking Things	1760	4754	564	169	86,9%
Santuário XML Security Java	20454	3091	4970	1100	69%
Tcc manager	4028	3630	407	195	92,3%
La4j	7846	1738	1857	835	88,9%
Graphhopper/core	3541	1271	3017	2016	94,7%
Dubbo-remoting-api	3028	450	867	357	50%
SAD-UFCG	1363	346	303	33	20%

A Tabela 1 apresenta o número de linhas de código e tamanho das suítes para os projetos usados no estudo.

Foram definidos sete tipos de faltas de refatoramento a serem injetadas. Todas podem ser causadas no contexto de um refatoramento de um Extract Method incorreto. As faltas foram selecionadas a partir de trabalhos que identificaram as principais edições e faltas que ocorrem junto com esse tipo de refatoramento [8, 13]. São elas:

- **Falta 1**: O retorno do Extract Method não é atribuído à variável no método que o chama;
- **Falta 2**: O método não chama o método criado pelo Extract Method;
- **Falta 3**: O comportamento do código original é implementado de forma errada no método criado pelo Extract Method;
- **Falta 4**: O comportamento dos métodos estão corretos, mas são passados argumentos errados para o método criado;
- **Falta 5**: O método é criado, mas a implementação não é extraída para o método;
- **Falta 6**: O código é extraído da classe pai para a classe filha, mas o comportamento é implementado de forma errada;
- **Falta 7**: O código é extraído da classe filha para a classe pai, mas o comportamento é implementado de forma errada.

Exemplos dos tipos de falta podem ser encontrados em nosso site [1].

Em cada projeto, foram injetadas até cinco faltas de cada tipo. Apenas uma falta foi injetada por vez e as suítes executadas. Vale destacar que a escolha dos métodos a serem refatorados e as faltas injetadas foi feito aleatoriamente. Uma falta foi considerada detectada se ao menos um caso de teste da suíte apresentou falha ou erro.

Em alguns projetos, não foi possível injetar cinco diferentes faltas de um mesmo tipo. Isso ocorreu por características do projeto, principalmente, poucos casos de herança, o que dificultou a implantação de todas as repetições dos tipos 6 e 7. A Tabela 2 apresenta a quantidade de faltas injetadas de cada tipo por projeto.

Tabela 2: Faltas injetadas de cada tipo por projeto.

Projeto	Tipo de falta						
	1	2	3	4	5	6	7
Tracking Things	5	5	5	5	5	5	5
Santuário XML Security Java	5	5	5	5	5	3	2
Tcc manager	5	5	5	3	3	0	0
La4j	5	5	5	5	5	3	3
Graphhopper/core	5	5	5	5	5	1	2
Dubbo-remoting-api	5	5	5	5	5	0	2
SAD-UFCG	5	5	5	5	5	0	0

Figura 6: Taxa de detecção das suítes geradas pela Randoop.

Figura 7: Taxa de detecção das suítes geradas pela EvoSuite.

5 RESULTADOS E DISCUSSÕES

5.1 RQ1: Testes automaticamente gerados são mais eficientes que os testes manuais para detecção de faltas de refatoramento?

Das 197 faltas injetadas, as suítes manuais detectaram 61,9%, as da Randoop 46,7% e da EvoSuite 55,8%. A fim de verificar se existem diferenças estatisticamente significativas entre as taxas de detecção, realizamos um teste de proporção [9]. A hipótese nula deste teste indica que as suítes possuem taxas de detecção equivalentes. Com 95% de confiança, o teste rejeitou a hipótese nula, indicando que existe diferença significativa entre as taxas de detecção (p-valor = 0,009345).

Da mesma forma, realizamos um teste de proporção par-a-par para os diferentes suítes (Randoop-manual, Evosuite-manual, Randoop-Evosuite). A Tabela 3 sumariza os resultados desta análise. Os testes mostraram que há diferença significativa entre as taxas de detecção

Figura 8: Taxa de detecção das suítes manuais.

Tabela 3: Teste de proporção par-a-par.

Hipótese Nula	p-valor
Randoop = manual	0,0016
EvoSuite = manual	0,13
Randoop = EvoSuite	0,043

Tabela 4: Detecção por tipo de falta.

Falta	Randoop	EvoSuite	Manual
Falta 1	28,6%	34,3%	62,9%
Falta 2	37,1%	51,4%	51,4%
Falta 3	57,1%	57,1%	62,9%
Falta 4	54,5%	48,5%	63,6%
Falta 5	54,5%	72,7%	45,5%
Falta 6	50%	58,3%	100%
Falta 7	50%	92,9%	85,7%

da Randoop e manual e entre as taxas da Randoop e EvoSuite. Já a EvoSuite, obteve uma taxa de detecção sem diferença significativa em relação à taxa manual. Porém, em números absolutos, as suítes manuais detectaram mais faltas do que a EvoSuite, 122 e 110, respectivamente. Desta forma, podemos ordenar as suites com relação a taxas de detecção de faltas da seguinte forma: manual = Evosuite > Randoop.

Desta forma, podemos responder RQ1 indicando que, dependendo da ferramenta usada, suítes geradas podem ser tão eficientes quanto às manuais.

Quando analisamos os resultados por tipo de falta, as suítes EvoSuite apresentaram resultados melhores do que as suítes manuais em dois tipos, 5 e 7, já as suítes Randoop não superaram os resultados das manuais em nenhum tipo, mas superou a taxa de detecção da EvoSuite em faltas do tipo 4.

5.2 RQ2: Qual ferramenta gera suítes mais eficientes para detectar faltas de refatoramento?

Com base nos resultados encontrados e testes estatísticos, podemos verificar que as suítes da EvoSuite tiveram um resultado melhor que as Randoop. O algoritmo de geração de testes da EvoSuite busca cobrir a maior quantidade de código possível, essa característica pode ter facilitado a detecção, porém possui dificuldades de criar objetos com atributos que não são de tipos primitivos. Já a Randoop, cria testes limitados a métodos públicos, uma restrição que afeta sua capacidade de detecção, as asserções criadas por ela são, na maioria das vezes, verificações de que o objeto é ou não null, é ou não vazio ou de lançamento de exceções, a dificuldade de criar objetos com atributos complexos também está presente nessa ferramenta. Como métodos privados e a composição de objetos são partes básicas da programação orientada a objetos, apresentar falhas na geração de testes para esses aspectos impacta negativamente na qualidade das suítes geradas.

A EvoSuite gera asserções melhores do que a Randoop, elas fazem chamadas a métodos públicos e criam asserções para eles, característica que aproxima os testes gerados por ela a testes manuais criados por desenvolvedores. As asserções criadas pela Randoop são fracas em detectar retornos de métodos não primitivos que tiveram seus atributos modificados por uma falta.

5.3 RQ3: A combinação de diferentes suítes geradas melhora a detecção de faltas de refatoramento?

Figura 9: Taxa de detecção das suítes geradas pela Randoop e EvoSuite.

Quando combinadas, as duas ferramentas obtiveram uma taxa de detecção de 69% (136 faltas). Combinadas, as ferramentas apresentaram um resultado melhor do que os resultados obtidos por cada ferramenta separadamente e também melhor do que as suítes manuais. Das 197 faltas injetadas, 25,4% (50) foram detectadas pela combinação das ferramentas e não foram detectadas pelas suítes manuais. Já as suítes manuais, detectaram 18,3% (36) das faltas que as ferramentas de geração automática não conseguiram. Assim, utilizar uma abordagem com a combinação das ferramentas parece

ser um caminho promissor. Já se combinarmos as três abordagens, temos uma taxa de detecção de 87,3% (172), o que representa a melhor abordagem.

6 AMEAÇAS À VALIDADE

Os resultados deste estudo não são generalizáveis além do contexto dos projetos utilizados e faltas injetadas. Porém, acreditamos que os projetos (sistemas *open-source* de diferentes contextos) e faltas (baseadas em trabalhos da área) escolhidas são representativas.

O processo de injeção de faltas também poderia ser contestado. Porém, foram realizadas escolhas randômica, quando viável, de quais métodos e faltas aplicar. Além disso, todas as faltas injetadas foram detectadas por ao menos um caso de teste, o que garante que de fato alteraram o comportamento dos sistemas.

Outras métricas poderiam ter sido usadas para avaliar a efetividade das suítes de teste. Porém, optamos por usar a taxa de detecção de faltas, por ser uma métrica bastante usada em trabalhos similares (e.g., taxa de mutantes mortos).

Outras ferramentas de geração de testes poderiam ser usadas no estudo. Porém, Randoop e Evosuite são as ferramentas mais famosas neste contexto. Além disso, nossa inspeção manual não identificou que as ferramentas geraram falsos positivos.

7 TRABALHOS RELACIONADOS

Alves et al. [5] analisaram a eficácia de suítes manuais na validação de refatoramentos. Eles identificaram que suítes manuais podem não ser suficientes para detectar faltas. Deste modo, neste estudo, ampliamos os tipos de suítes, incluindo testes automaticamente gerados pelas ferramentas Randoop e EvoSuite.

Rachatasumrit e Kim [11] identificaram que uma parte pequena dos métodos refatorados são testados por suítes de teste manuais e que desenvolvedores hesitam em realizar tarefas de refatoramento por causa da baixa cobertura de testes. Moreira et al. [8] mostraram que desenvolvedores frequentemente combinam Extract Method com edições extras no código.

Silva et al. [12] analisaram a eficácia de suítes de teste automaticamente geradas em detectar faltas de refatoramento. Nosso estudo analisa também a capacidade de detecção das faltas por suítes manuais e pela combinação de diferentes abordagens. Silva et al. [13] apontaram fatores que podem influenciar a performance das ferramentas Randoop e EvoSuite para gerar testes efetivos.

8 CONCLUSÕES

Neste trabalho, realizamos um estudo com o objetivo de analisar a eficácia de diferentes tipos de suítes na detecção de faltas de refatoramento. Para isso, selecionamos projetos escritos em Java, com suítes de teste manuais, criamos suítes de teste com as ferramentas Randoop e EvoSuite, injetamos faltas de refatoramento causados por Extract Method. Nossos resultados mostraram que a melhor abordagem é a utilização das ferramentas em conjunto com testes manuais. Isoladamente, a Randoop obteve uma taxa de detecção baixa, a EvoSuite, no entanto, obteve um resultado significantemente comparável ao de suítes manuais. Os resultados obtidos neste estudo ajudam equipes de desenvolvimento na escolha de abordagens para realizar testes nos sistemas, a entender as consequências de usar testes automaticamente gerados e o melhor cenário de utilizá-las.

Como trabalhos futuros, pretendemos ampliar nosso estudo incluindo mais suítes automáticas geradas, a fim de reduzir o impacto da aleatoriedade da geração de testes. Também pretendemos avaliar as suítes com outros tipos de faltas introduzidas relacionadas a diferentes tipos de refatoramento.

REFERÊNCIAS

[1] [n. d.]. Analisando suites de teste. https://sites.google.com/ccc.ufcg.edu.br/analisandosuitesdeteste/início

[2] [n. d.]. EvoSuite. https://www.evosuite.org/

[3] [n. d.]. JDeodorant. https://users.encs.concordia.ca/~nikolaos/stats.html

[4] [n. d.]. Randoop. https://www.randoop.com/

[5] 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

[6] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code (Object Technology Series)* (illustrated edition ed.). Addison-Wesley Longman, Amsterdam.

[7] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2012. A Field Study of Refactoring Challenges and Benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. Association for Computing Machinery, New York, NY, USA, Article 50, 11 pages. https://doi.org/10.1145/2393596.2393655

[8] Jaziel S Moreira, Everton LG Alves, and Wilkerson L Andrade. 2020. An exploratory study on extract method floss-refactoring. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*. 1532–1539.

[9] Pedro Alberto Moretin and Wilton de Oliveira Bussab. 2017.. *Estatística básica* / (9.ed. ed.). Saraiva„ São Paulo:.

[10] Emerson Murphy-Hill, Chris Parnin, and Andrew P Black. 2011. How we refactor, and how we know it. *IEEE Transactions on Software Engineering* 38, 1 (2011), 5–18.

[11] Napol Rachatasumrit and Miryung Kim. 2012. An empirical investigation into the impact of refactoring on regression testing. In *2012 28th ieee international conference on software maintenance (icsm)*. IEEE, 357–366.

[12] Indy P.S.C. Silva, Everton L.G. Alves, and Wilkerson L. Andrade. 2017. Analyzing Automatic Test Generation Tools for Refactoring Validation. In *2017 IEEE/ACM 12th International Workshop on Automation of Software Testing (AST)*. 38–44. https://doi.org/10.1109/AST.2017.9

[13] Indy PSC Silva, Everton LG Alves, and Patrícia DL Machado. 2018. Can automated test case generation cope with extract method validation?. In *Proceedings of the XXXII Brazilian Symposium on Software Engineering*. 152–161.

[14] I. Sommerville. 2011. *Engenharia de software.* PEARSON BRASIL.