



**UNIVERSIDADE FEDERAL DE PERNAMBUCO**  
**CENTRO DE INFORMÁTICA**  
**CURSO DE BACHARELADO EM ENGENHARIA DA COMPUTAÇÃO**

**Vinicius Thiago Leite dos Santos**

**Avaliação de cobertura de código de várias versões de Randoop**

**RECIFE**

**2021**

**UNIVERSIDADE FEDERAL DE PERNAMBUCO**

**CENTRO DE INFORMÁTICA**

**CURSO DE BACHARELADO EM ENGENHARIA DA COMPUTAÇÃO**

**Vinicius Thiago Leite dos Santos**

**Avaliação de cobertura de código de várias versões de Randoop**

Monografia apresentada ao Centro de Informática (CIN) da Universidade Federal de Pernambuco (UFPE), como requisito parcial para conclusão do Curso de Engenharia da Computação, orientada pelo professor Paulo Borba.

**RECIFE**

**2021**

**UNIVERSIDADE FEDERAL DE PERNAMBUCO**

**CENTRO DE INFORMÁTICA**

**CURSO DE BACHARELADO EM ENGENHARIA DA COMPUTAÇÃO**

**Vinicius Thiago Leite dos Santos**

**Avaliação de cobertura de código de várias versões de Randoop**

Monografia submetida ao corpo docente da Universidade Federal de Pernambuco, defendida e aprovada em 2 de Dezembro de 2021.

Banca Examinadora:

Paulo Borba

Orientador

XXXXXX

Doutor

Breno Miranda

Examinador

XXXXXXXX

Doutor



Dedico este trabalho à minha mãe, por nunca ter  
medido esforços para me proporcionar um ensino de  
qualidade .

## **AGRADECIMENTOS**

Gostaria de agradecer ao meu coorientador Léuson da Silva, que me ajudou em todas as etapas dessa pesquisa, sempre dedicando um pouco do seu tempo para tirar minhas dúvidas e me dar feedbacks construtivos.

“A visão é o mais avançado dos nossos sentidos,  
de forma que não é de surpreender que as  
imagens exerçam o papel mais importante na  
percepção humana.”  
Rafael C. Gonzalez

## RESUMO

Durante o desenvolvimento de software, a atividade de testes é fundamental para verificar e validar o software em desenvolvimento. Porém, trata-se de uma atividade que demanda um alto custo de tempo e esforço para criar e manter os testes. Considerando a importância dos testes, ferramentas de geração automática de testes como Randoop surgem como uma solução que reduz o custo e o esforço associados a esta etapa. Entretanto, em alguns casos, essas ferramentas de geração de testes apresentam deficiências gerando testes de baixa qualidade e cobertura. Assim, modificações nessas ferramentas podem ser implementadas para mitigar os efeitos de suas deficiências. Então, neste trabalho, nosso objetivo é avaliar os testes gerados pela ferramenta Randoop e por Randoop CLEAN, uma modificação de Randoop proposta em outro trabalho, quanto a cobertura de código, número e qualidade dos testes gerados.

**Palavras-chave:** Software, Testes, Randoop, Geração.



## ABSTRACT

During software development, the testing activity is essential to verify and validate the software under development. However, it is a activity that requires a high cost of time and effort to create and maintain the tests. Considering the importance of tests, automatic test generation tools like Randoop emerge as a solution that reduces the cost and effort associated with this step. However, in some cases, these test generation tools have shortcomings generating tests of low quality and coverage. Thus, modifications to these tools can be implemented to mitigate the effects of their deficiencies. So, in this work, our objective is to evaluate the tests generated by the Randoop tool and its proposed modifications in another work, regarding the code coverage, number and quality of the generated tests.

**Keywords:** Software, Tests, Randoop, Generation.

## Sumário

1.	Introdução	22
1.1.	Motivação	23
1.2.	Objetivos	28
2.	Conceitos Básicos	29
2.1	Cobertura de código	29
2.2	Conflitos semânticos	29
2.3	Ferramentas	29
3.	Trabalhos Relacionados (Estado da Arte)	31
3.1.	Ferramentas de Geração de teste	31
3.2.	Geração de testes	32
4	Algoritmo em Estudo	33
4.1	Visão Geral	33
4.2	Maximiza chamadas ao método alvo	33
4.3	Melhora criação de objetos utilizados no método alvo	34
5	Experimentos e Análise	35
5.1	Seleção dos cenários	35
5.2	Geração de testes	35
5.3	Instrumentação dos Jars	36
5.4	Execução da suíte de testes utilizando jars instrumentados	37
5.5	Obtenção das métricas de cobertura de código	37
5.6	Análise manual sobre a quantidade e qualidade dos testes gerados	43
6	Resultados	44
6.1	Resultado da execução de 30 segundos	44
6.2	Resultado da execução de 5 minutos	52
7	Ameaças a validade	55
8	Trabalhos Futuros	56
9	Bibliografia	57



## Lista de Figuras

Figura 1 -	Parte do método alvo dos testes	25
Figura 2 -	Exemplo de teste gerado por Randoop	26
Figura 3 -	Classe HikariConfig	27
Figura 4 -	Testes gerados por uma versão modificada de Randoop	28
Figura 5 -	Design do experimento	37
Figura 6 -	Exemplo de método instrumentado	39
Figura 7 -	Exemplo de arquivo usado para colher métrica SUA	41
Figura 8 -	Passo a passo para obter a cobertura	42
Figura 9 -	Exemplo de arquivo usado para análise de classe	43
Figura 10 -	Exemplo de arquivo usado para análise do método alvo	44
Figura 11 -	Método decode	48
Figura 12 -	Exemplo de teste gerado por Randoop	49

## **LISTA DE TABELAS**

Tabela 1 - Resultados obtidos para o projeto HikariCp	40
Tabela 2 - Descrição das Métricas	41
Tabela 3 - Cobertura para 30 segundos	46
Tabela 4 - Porcentagem para 30 segundos	49
Tabela 5 - Análise dos teste para 30 segundos	50
Tabela 6 - Cobertura para 5 minutos	51
Tabela 7 - Porcentagem para 5 minutos	53
Tabela 8 - Análise dos testes para 5 minutos	54

## **TABELA DE SIGLAS**

<b>Sigla</b>	<b>Significado</b>	<b>Página</b>
SUA	Sistema sob análise	41
CUA	Classe sob análise	43
MUA	Método sob análise	43



## 1. Introdução

Durante o desenvolvimento de software, teste é uma atividade essencial responsável por verificar e validar o software em construção. Seu objetivo é revelar falhas em um software para elas poderem ser corrigidas pela equipe de desenvolvimento antes que chegue ao usuário final. Diferentes tipos de testes podem ser usados para diferentes propósitos. Por exemplo, testes de unidade possuem o objetivo de avaliar as menores unidades de um software; como exemplo de unidade, temos uma classe Java onde *assertions* podem explorar o retorno de seus métodos. Testes de integração visam testar diferentes partes do sistema em conjunto de modo a buscar falhas entre as interfaces destes módulos do sistema. Testes de sistema testam o software completo simulando um usuário final; com isso, pode-se validar se o produto final satisfaz aos requisitos elicitados previamente. Por fim, testes de regressão é uma metodologia de teste que visa reduzir efeitos colaterais de novas versões do software; são testes usados para detectar mudanças de comportamento inesperadas entre duas versões de um programa. Da Silva et al. [1]. Para tanto, aplica-se um mesmo conjunto de testes em duas versões diferentes de um software buscando identificar estas mudanças de comportamento (regressões). Com isso evita-se que problemas resolvidos previamente retornem ao software, tornando a versão atual do software inválida. Porém, pesquisas mostram que o uso de testes de regressão é um processo caro que pode exigir mais de 33% dos recursos necessários para construção do software [2]. Logo, a criação e evolução de suítes de teste para este propósito torna-se uma atividade que demanda um alto custo de tempo e esforço. Para tornar este processo mais dinâmico e menos custoso, estes testes podem ser gerados automaticamente e integrados ao projeto.

Neste sentido, ferramentas de geração automática de testes em Java, como Randoop [3] e EvoSuite [4], podem ser usadas como uma alternativa para gerar testes de maneira automática. Essas ferramentas funcionam recebendo uma versão do sistema como entrada e retornam uma suíte de testes explorando a versão recebida. Com isso, mudanças de comportamento inesperadas poderiam ser detectadas rapidamente, reduzindo tempo e esforço do time de desenvolvimento. Como resultados, estudos prévios investigam a capacidade de detectar mudanças de comportamento em diferentes contextos. Silva et al. [3] investigam se



testes gerados por essas ferramentas podem ser usados para detectar mudanças de comportamento quando *refactorings* são realizados. Da Silva et al. [1] adota uma abordagem similar, onde utilizam testes gerados para detectar conflitos semânticos [5]. Entretanto, em ambos trabalhos, os autores reportam deficiências que impactam negativamente nas taxas de detecção de mudanças de comportamento pelos testes gerados.

Com base nestas deficiências, mudanças poderiam ser implementadas em Randoop com o intuito de melhorar o processo de geração de testes, e posteriormente, a cobertura de código alcançada por testes gerados automaticamente. Assim, neste trabalho, nós usamos a ferramenta Randoop CLEAN que consegue gerar mais testes com diversidade de entradas. Por exemplo, nos testes criados, a ferramenta Randoop CLEAN força a criação diversa de objetos das classes envolvidas e a chamada com mais frequências de um conjunto de métodos informados previamente, ao invés de manter a aleatoriedade de escolhas para criação de objetos e chamadas a métodos que Randoop faz. Em seguida, nós avaliamos as vantagens e desvantagens observadas após estas mudanças. Para tanto, nós realizamos um estudo empírico utilizando uma amostra de dados composta por 46 cenários de merge distribuídos em 31 projetos *open-source* desenvolvidos em Java. Para avaliar as mudanças realizadas em Randoop, nós calculamos as métricas de cobertura de código, o número e a qualidade dos testes gerados por Randoop para uma dada versão de um projeto. Dentre os resultados obtidos, observamos um aumento na cobertura de código onde dentre 159 casos no total a versão modificada de Randoop obteve uma cobertura melhor em 86 casos considerando a análise de linhas do sistema como um todo. Na criação e qualidade de teste criados o Randoop CLEAN obteve um melhor desempenho em 22 dos 36 projetos. Estes resultados indicam que a alteração realizada em Randoop trouxe melhoras significativas para a ferramenta.

### **1.1. Motivação**

Como discutido previamente, Randoop possui deficiências durante o processo de geração de testes. Estas deficiências ocorrem porque Randoop apresenta dificuldade para gerar testes que explorem todos os diferentes estados do código em análise. Por exemplo, nos casos em que o código em análise apresenta dependências a objetos mais complexos, com muitas dependências externas (APIs, banco de dados etc), a ferramenta não cria objetos bons,

com seus atributos inicializados com valores não default, e diversos. Em alguns casos, Randoop não consegue sequer criar os objetos das classes que estão sendo testadas ou usa valores *null* nos parâmetros. Nos casos em que a ferramenta consegue criar objetos, os testes gerados usam estes objetos em chamadas de métodos, mas sempre com os mesmos parâmetros. Com isso, a suíte gerada apresenta testes semelhantes revelando uma baixa qualidade, visto que estes testes exploram sempre as mesmas partes do código.

Como exemplo de testes de baixa qualidade gerados por Randoop, considere o teste gerado para o projeto HikariCP. Abaixo, na Figura 1, é apresentado o método alvo dos testes *validate()* pertencente à classe *HikariConfig* na versão do commit (9eb405e5b81082044daca981b2655396c4e9a6eb).

```

321 public void validate() {
322     Logger logger = LoggerFactory.getLogger(getClass());
323     if (this.acquireRetryDelay < 0L) {
324         logger.error("acquireRetryDelay cannot be negative.");
325         throw new IllegalStateException("acquireRetryDelay cannot be negative.");
326     }
327     if (this.acquireRetryDelay < 100L) {
328         logger.warn("acquireRetryDelay is less than 100ms, did you specify the wrong time unit? Using default instead.");
329         this.acquireRetryDelay = ACQUIRE_RETRY_DELAY;
330     }
331     if (this.connectionCustomizerClassName != null && this.connectionCustomizer == null)
332     {
333         try {
334             Class<?> customizerClass = getClass().getClassLoader().loadClass(this.connectionCustomizerClassName);
335             this.connectionCustomizer = (IConnectionCustomizer)customizerClass.newInstance();
336         } catch (Exception e) {
337             logger.warn("connectionCustomizationClass specified class '" + this.connectionCustomizerClassName + "' could not be loaded", e);
338             this.connectionCustomizerClassName = null;
339         }
340     }
341     if (this.connectionTimeout == 2147483647L) {
342         logger.warn("No connection wait timeout is set, this might cause an infinite wait.");
343     } else if (this.connectionTimeout < 100L) {
344         logger.warn("connectionTimeout is less than 100ms, did you specify the wrong time unit? Using default instead.");
345         this.connectionTimeout = CONNECTION_TIMEOUT;
346     }
347     if (this.dataSource == null && this.dataSourceClassName == null) {
348         logger.error("one of either dataSource or dataSourceClassName must be specified");
349         throw new IllegalStateException("one of either dataSource or dataSourceClassName must be specified");
350     }
351     if (this.dataSource != null && this.dataSourceClassName != null)
352     {
353         logger.warn("both dataSource and dataSourceClassName are specified, ignoring dataSourceClassName");
354     }
355     if (this.idleTimeout < 0L) {
356         logger.error("idleTimeout cannot be negative.");
357         throw new IllegalStateException("idleTimeout cannot be negative.");
358     }
359     if (this.idleTimeout < 30000L && this.idleTimeout != 0L) {
360         logger.warn("idleTimeout is less than 30000ms, did you specify the wrong time unit? Using default instead.");
361         this.idleTimeout = IDLE_TIMEOUT;
362     }
363     if (!this.isJdbc4ConnectionTest && this.connectionTestQuery == null) {
364         logger.error("Either jdbc4ConnectionTest must be enabled or a connectionTestQuery must be specified.");
365         throw new IllegalStateException("Either jdbc4ConnectionTest must be enabled or a connectionTestQuery must be specified.");
366     }
367     if (this.leakDetectionThreshold != 0L && this.leakDetectionThreshold < 10000L) {
368         logger.warn("leakDetectionThreshold is less than 10000ms, did you specify the wrong time unit? Disabling leak detection.");
369         this.leakDetectionThreshold = 0L;
370     }
371     if (this.maxPoolSize < this.minPoolSize) {
372         logger.warn("maxPoolSize is less than minPoolSize, forcing them equal.");
373         this.maxPoolSize = this.minPoolSize;
374     }
375     if (this.maxLifetime < 0L) {
376         logger.error("maxLifetime cannot be negative.");
377         throw new IllegalStateException("maxLifetime cannot be negative.");
378     }
379     if (this.maxLifetime < 120000L && this.maxLifetime != 0L) {
380         logger.warn("maxLifetime is less than 120000ms, did you specify the wrong time unit? Using default instead.");
381         this.maxLifetime = MAX_LIFETIME;
382     }
383 }

```

Figura 1 - Parte do método alvo dos testes

```

7731 @Test
7732 public void test0492() throws Throwable {
7733     if (debug)
7734         System.out.format("%n%s%n", "RegressionTest0.test0492");
7735     com.zaxxer.hikari.HikariConfig hikariConfig0 = new com.zaxxer.hikari.HikariConfig();
7736     hikariConfig0.addDataSourceProperty("", (java.lang.Object) 1L);
7737     com.zaxxer.hikari.HikariConfig hikariConfig5 = new com.zaxxer.hikari.HikariConfig();
7738     hikariConfig0.addDataSourceProperty("hi!", (java.lang.Object) hikariConfig5);
7739     java.util.Properties properties7 = hikariConfig0.dataSourceProperties;
7740     boolean boolean8 = hikariConfig0.isInitializationFailFast();
7741     // The following exception was thrown during execution in test generation
7742     try {
7743         hikariConfig0.validate();
7744         org.junit.Assert.fail("Expected exception of type java.lang.IllegalStateException; message");
7745     } catch (java.lang.IllegalStateException e) {
7746         // Expected exception.
7747     }
7748     org.junit.Assert.assertNotNull(properties7);
7749     org.junit.Assert.assertTrue("'" + boolean8 + "' != '" + false + "'", boolean8 == false);
7750 }

```

Figura 2 - Exemplo de teste gerado por Randoop

Na Figura 2 é apresentado um dos testes gerados por Randoop para esse projeto podemos observar na linha 7735, que o teste gerado por Randoop cria um objeto do tipo HikariConfig, mas pobre em atributos, alterando com valores diferentes dos de inicialização apenas o atributo *dataSourceProperties* do tipo *Properties* da classe dentre todos os possíveis atributos que podemos observar na Figura 3. Em seguida, na linha 7743, há uma chamada ao método *validate()*. Conforme a Figura 1, podemos ver que o método *validate()* é composto por declarações condicionais (*ifs*), onde são acessados atributos da classe. Porém, como apenas um atributo da classe é inicializado no teste (linha 7736) apenas o conteúdo do *if* da linha 347 é exercitado. Como resultado, o teste obterá uma baixa cobertura neste método e mudanças podem não ser detectadas.

```

public class HikariConfig implements HikariConfigMBean {
    public static long ACQUIRE_RETRY_DELAY = 750L;

    public static long CONNECTION_TIMEOUT = 5000L;

    public static long IDLE_TIMEOUT = TimeUnit.MINUTES.toMillis(10L);

    public static long MAX_LIFETIME = TimeUnit.MINUTES.toMillis(30L);

    static {
        JavassistProxyFactory.initialize();
    }

    public Properties dataSourceProperties = new Properties();

    public volatile int acquireIncrement = 1;

    public volatile int acquireRetries = 3;

    public volatile long acquireRetryDelay = ACQUIRE_RETRY_DELAY;

    public volatile long connectionTimeout = CONNECTION_TIMEOUT;

    public volatile long idleTimeout = IDLE_TIMEOUT;

    public boolean isAutoCommit = true;

    public boolean isJdbc4connectionTest = true;

    public volatile int minPoolSize = 10;

    public volatile int maxPoolSize = 60;

    public volatile long maxLifetime = MAX_LIFETIME;

    public String poolName = "HikariPool-" + poolNumber++;

    public int transactionIsolation = -1;

    public static int poolNumber;

    public volatile long leakDetectionThreshold;

    public String transactionIsolationName;

    public String connectionCustomizerClassName;

    public String connectionInitSql;

    public String connectionTestQuery;

    public String dataSourceClassName;

    public String catalog;

    public boolean isInitializationFailFast;

    public boolean isRegisterMbeans;

    public DataSource dataSource;

    public IConnectionCustomizer connectionCustomizer;
}

```

Figura 3 - Classe HikariConfig

A seguir, na Figura 4, podemos observar um exemplo de um teste gerado com a versão Modificada de Randoop para o mesmo método previamente apresentado na Figura 1. No intervalo das linhas 1800 e 1805, dentro do método de teste *test0090()*, pois o arquivo possui diversos outros testes, podemos observar melhorias como a iniciação de alguns atributos na geração dos objetos usados como parâmetros na chamada ao método *validate()*, linha 1808. Com essa melhoria na criação dos atributos os *ifs* encontrados nas linhas 326, 330, 347 e 358 do método *validate()* (figura 1) são exercitados.

```

1796  @Test
1797  public void test0090() throws Throwable {
1798      if (debug)
1799          System.out.format("%n%s%n", "RegressionTest0.test0090");
1800      com.zaxxer.hikari.HikariConfig hikariConfig0 = new com.zaxxer.hikari.HikariConfig();
1801      hikariConfig0.idleTimeout = (byte) 10;
1802      int int3 = hikariConfig0.minPoolSize;
1803      javax.sql.DataSource dataSource4 = null;
1804      hikariConfig0.dataSource = dataSource4;
1805      hikariConfig0.setAcquireRetryDelay(10L);
1806      // The following exception was thrown during execution in test generation
1807      try {
1808          hikariConfig0.validate();
1809          org.junit.Assert.fail("Expected exception of type java.lang.IllegalStateException; message: one of
1810      } catch (java.lang.IllegalStateException e) {
1811          // Expected exception.
1812      }
1813      org.junit.Assert.assertTrue("'" + int3 + "' != '" + 10 + "'", int3 == 10);
1814  }

```

Figura 4 - Testes gerados por uma versão modificada de Randoop

Como podemos observar, ao contrário da versão original de Randoop, a versão modificada cria o objeto com mais atributos que podem ser explorados melhorando a qualidade da suíte de testes. Com isso temos uma maior probabilidade de aumento/melhoria na cobertura de código.

## **1.2. Objetivos**

Esse trabalho busca avaliar as vantagens e desvantagens observadas na geração de testes pela ferramenta Randoop e sua versão modificada. Para tanto, nós avaliamos, para cada versão, a cobertura de código dos testes que elas geram, número e qualidade dos testes gerados.

## 2. Conceitos Básicos

Nesta seção, são apresentados os conceitos, terminologia e ferramentas adotadas na realização do trabalho.

### 2.1. Cobertura de código

O principal objetivo da cobertura de código é encontrar trechos dentro do software que não foram testados. A cobertura também ajuda na avaliação da suíte de testes do sistema. Então, um sistema com alta cobertura de código, possui uma menor chances de apresentar falhas, pois significa que foi testado mais exaustivamente.

### 2.2. Conflitos semânticos

São conflitos que não são detectados através do *merge* realizado, pois o código consegue ser mesclado, porém mudam o comportamento do software que passa a obter um comportamento não esperado. Podem ser detectados através de testes de regressão que visam manter o comportamento do sistema.

### 2.3 Cenário de Merge

*Merge* é a mesclagem da alteração de código realizada por 2 desenvolvedores. Cenários de *merge* são os *commits* relacionados a mesclagem do código realizada. Dentre esse cenário, existem 4 *commits* envolvidos, o *commit* base que contém a versão original do código, os *commits left* e *right*, que são as alterações realizadas pelos desenvolvedores e por último o *commit merge*, que é responsável pelo código que foi mesclado pelas alterações dos desenvolvedores.

### 2.3. Ferramentas

Abaixo estão as ferramentas de desenvolvimento que são utilizadas no estudo.

#### i. Jacoco

Criado pela equipe de desenvolvimento do EclEmma, *Jacoco* é uma biblioteca gratuita de cobertura de código para a linguagem Java. A biblioteca fornece a tecnologia padrão para análise de cobertura de código em ambientes baseados em Java VM (*virtual machine*),



conhecida por sua flexibilidade e não exigir muitos recursos para sua operação. Além disso, é uma biblioteca com versões estáveis e boa documentação associada, usada por projetos populares e relevantes como *Ant Tasks*, *Maven plug-in* e *EclEmma Eclipse plug-in* [7]. Neste trabalho, nós usamos *Jacoco.cli*, que é uma interface de linha de comando, usado para realizar operações básicas da ferramenta Jacoco [8]. Dentre as operações disponíveis para uso, nós usamos as operações para instrumentação de *jars* e geração de reporte da análise de cobertura.

## **ii. Randoop**

Randoop é uma ferramenta *open source* para geração automática de testes para projetos Java. Randoop gera testes de unidade usando geração de teste aleatório direcionado por feedback. Essa técnica, de forma inteligente, gera sequências de invocações de método / construtor para as classes em teste, com isso Randoop cria testes a partir das sequências de código e asserções. Randoop é usado principalmente para dois tipos de teste: testes de unidade, que servem para revelar erros em código, e teste de regressão, para detecção de mudanças de comportamento [9].

## **iii. SMAT**

Projeto desenvolvido em Python para a detecção de conflitos semânticos, por meio da geração automática de testes, usando ferramentas como Randoop, executa suítes de teste em cenários de *merge*. SMAT disponibiliza a infraestrutura necessária para a geração e execução de testes a partir de cenários de *merge*, disponibilizados através de um arquivo CSV de entrada, e reporte de conflitos semânticos encontrados, arquivo CSV de saída [10].

### 3. Trabalhos Relacionados (Estado da Arte)

Nesta seção, nós apresentamos trabalhos relacionados ao nosso estudo. Inicialmente, nós discutimos trabalhos que realizam uma comparação entre diferentes ferramentas de geração automática de testes. Em seguida, nós discutimos trabalhos que realizam transformações de código com o sentido de auxiliar as ferramentas na geração dos testes.

#### 3.1. Ferramentas de Geração de Testes

Estudos prévios comparam diferentes ferramentas de geração de testes baseados em diferentes métricas e motivações. Sina et al. [12] comparam as ferramentas Randoop, EvoSuite e AgitarOne quanto a cobertura de código e quantidade de *bugs* reportados pelos testes de cada ferramenta. Para realizar essa comparação, os autores realizam um estudo com base em um *dataset* composto 357 casos formados por dois commits (*bug*, *fix*) detectados em 5 projetos *open-source*. Para cada *bug*, os autores tinham acesso à versão responsável pela introdução do *bug*, bem como a sua versão com o reparo aplicado. Inicialmente, as ferramentas foram usadas para a geração de testes a partir das versões sem *bugs*. Posteriormente, os testes gerados foram executados nas versões falhas de cada sistema com o intuito de detectar o *bug* ali presente. Assim, os autores avaliaram o desempenho de cada ferramenta. Com base nos resultados, uma lista de melhorias que poderiam ser adotadas pelas ferramentas de geração de testes é reportada.

Apesar do estudo focar na comparação das ferramentas em relação à detecção de falhas, os autores também avaliam a cobertura de código das suítes de testes geradas, objetivo deste trabalho. Entretanto, neste trabalho, nós também avaliamos outras métricas relacionadas às suítes de testes, como a quantidade e qualidade dos testes gerados. Em relação à ferramenta usada para avaliar a métrica de cobertura, baseado em estudos prévios [14], nós adotamos a ferramenta Jacoco, enquanto que Sina et al. [12] adotaram duas ferramentas; Cobertura [14] para obter a cobertura de código dos testes gerados por Randoop e EvoSuite, enquanto que a ferramenta AgitarOne possui uma ferramenta própria para avaliar a cobertura de código.

### 3.2. Geração de testes

Transformações de testabilidade podem ser aplicadas para aumentar as chances, que um código tem de ser diretamente executado por casos de testes. Arcuri e Galeotti. [13] aplicam transformações para auxiliar a ferramenta EvoMaster na geração de testes, ferramenta baseada no método de geração de testes *search-based software testing* (SBST). Os autores comentam que EvoMaster apresentava dificuldades na geração de testes para APIs e REST *web-services*. Assim, os autores buscam avaliar se as transformações realizadas auxiliam EvoMaster na geração de testes automatizados em específicos cenários. Neste estudo, transformações de testabilidade também foram aplicadas nos commits, usados para gerar as suítes de testes (ver Seção 5.1). Estas transformações foram usadas para aumentar as chances de executar diretamente métodos de uma classe, que originalmente não tinham visibilidade pública, dentre outros aspectos[16]. Entretanto, o foco do nosso estudo é avaliar a cobertura alcançada pelos testes gerados.

## 4 Algoritmo em Estudo

Nesta seção, nós apresentamos as modificações realizadas em Randoop, derivando assim a versão de Randoop CLEAN.

### 4.1 Visão Geral

Grande parte do comportamento original de Randoop foi preservado. Então, abaixo explicaremos as duas principais modificações implementadas em Randoop CLEAN.

### 4.2 Maximiza chamadas ao método alvo

Antes de apresentar como o Randoop CLEAN funciona, é importante entender o funcionamento original de Randoop. Assim, seguindo o comportamento original de Randoop, antes da ferramenta começar a gerar testes, ela seleciona todos os métodos e construtores públicos de uma determinada lista de classes fornecida como entrada. Esta lista de métodos é salva internamente em uma lista, chamada conceitualmente por *pool*. Em seguida, Randoop usa essa lista para selecionar aleatoriamente um elemento, e então, gerar sequências que serão usadas para criar casos de teste. Estas sequências representam entradas para os testes, como por exemplo, instruções de criações de objetos e chamadas de método. Adicionalmente à lista de classes informadas como entrada para Randoop, é possível informar um conjunto de métodos como entrada, que devem ser cobertos pelos testes gerados pela ferramenta. Entretanto, Randoop não oferece garantias de que este método será exercitado pelos testes gerados, considerando que as sequências criadas envolvendo estes métodos podem ser avaliadas como inválidas, e posteriormente, descartadas.

A primeira diferença implementada em Randoop CLEAN é referente a tentativa de maximizar o número de chamadas para um método alvo. Para tanto, a ferramenta tem acesso internamente a uma lista de sequências válidas, geradas durante o processo de criação dos testes. Randoop CLEAN usa a lista de sequências válidas para verificar cada vez que um determinado número de sequências; este número é definido com base no número de classes fornecidas como entrada para a ferramenta. Quando este número é atingido, Randoop CLEAN

adicionar uma nova chamada ao método alvo, também previamente fornecido como entrada. A ferramenta adota um intervalo de tempo entre as chamadas forçadas, pois outras chamadas de métodos, que podem modificar os objetos usados pelo método alvo, podem ser executadas. Assim, a ferramenta busca não apenas aumentar o número de chamadas a um método alvo, como também usando objetos diversos nestas chamadas. Como exemplo da maximização da chamada ao método alvo, manualmente nós analisamos os testes gerados para o cenário descrito na Seção (1.1) e nós obtivemos o seguinte resultado. Randoop gerou 2215 métodos de teste e dentre esses testes o método alvo foi chamado 76 vezes. A ferramenta Randoop CLEAN gerou um total de 1687 métodos de testes resultando numa diminuição de 23.83% em relação a ferramenta Randoop, dentre eles o método alvo foi chamado 117 vezes o que resultou num incremento de 35.04% de chamadas ao método alvo, mesmo gerando menos métodos de teste.

### **4.3 Melhora criação de objetos utilizados no método alvo**

Sobre a criação de objetos, Randoop adota um comportamento semelhante à escolha dos métodos para gerar os testes, como explicado previamente. Isso ocorre porque a criação de objetos ocorre por meio de chamadas aos construtores de objetos, e portanto, fazem parte da lista de métodos que a ferramenta seleciona de maneira aleatória. Esta aleatoriedade pode representar uma limitação, quando um método é testado usando sempre os mesmos objetos como parâmetro, por exemplo. Logo, seria necessário a criação de diferentes objetos de um dado tipo, permitindo assim a avaliação do comportamento de um método baseado em diferentes entradas.

Assim, a segunda diferença implementada em Randoop CLEAN tem o intuito de maximizar o número de chamadas para a criação dos objetos usados por um determinado método. Para tanto, a ferramenta usa duas variáveis internamente: o número de etapas executadas e a lista de classes alvo dadas como entrada. Essas etapas representam tentativas de gerar sequências, que podem, portanto, falhar e serem descartadas. Quando Randoop CLEAN executa um determinado número de etapas, a ferramenta adiciona chamadas para a criação de todos os objetos relacionados com o método alvo. Por exemplo, objetos da classe onde o método alvo encontra-se declarado, bem como seus parâmetros, quando aplicável. Assim, Randoop CLEAN identifica o tipo de objeto requerido, e em seguida, escolhe

aleatoriamente um método ou construtor, da lista de métodos disponíveis (*pool*), e assim o objeto do tipo específico é retornado.

## 5 Experimentos e Análise

Para avaliar os benefícios observados pelas alterações implementadas em Randoop, nós realizamos um estudo empírico composto de seis etapas, a Figura 5 ilustra essas etapas utilizando como exemplo o *commit* base, de um cenário de *merge*. Na primeira etapa, nós utilizamos a mesma amostra usada no trabalho realizado de Da Silva et al. [1]. Adicionalmente, nós selecionamos mais 3 novos projetos. Na segunda etapa, para cada cenário de *merge* em nossa amostra, nós usamos as ferramentas Randoop e Randoop CLEAN para a criação de suítes de testes. Na terceira etapa, nós instrumentamos os *Jars* gerados na primeira etapa. Na quarta etapa, os testes, que foram gerados pelas ferramentas na segunda etapa, foram executados em cada um dos *Jars* do cenário de *merge* em análise para obtenção da análise de cobertura. Na quinta etapa, nós obtemos os resultados da cobertura de código para cada cenário de *merge* avaliado. Em seguida, nós agrupamos o resultado obtido por cada ferramenta para cada versão de *merge* em um único arquivo CSV. Por fim, os testes gerados pelas ferramentas foram analisados manualmente para obtenção das métricas de quantidade e qualidade dos testes gerados. Detalhes adicionais sobre as etapas descritas serão apresentadas posteriormente ao longo da seção.

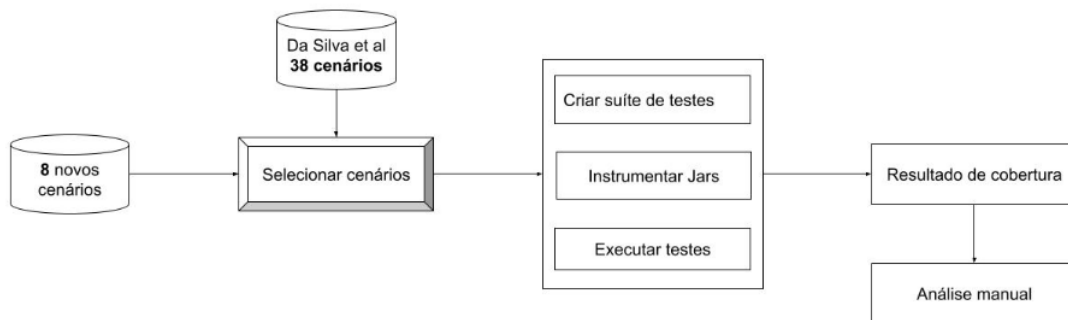


Figura 5 - Design do experimento

### 5.1 Seleção dos cenários

No trabalho realizado por Da Silva et al. [1], projetos foram selecionados contendo cenários de *merge* que apresentavam mudanças feitas por left ou right em um mesmo método ou inicialização de atributos. Neste trabalho, nós reutilizamos os 38 cenários de *merge* analisados pelo trabalho prévio [1], que pertencem a 28 diferentes projetos. Além dos cenários de *merge*, nós também usamos os arquivos *jars*, que foram transformados e gerados através

do processo de *build* para cada versão de *merge*. Adicionalmente, neste trabalho, nós selecionamos 8 novos cenários de 3 novos projetos gerando uma amostra composta de 46 cenários de *merge* de 31 projetos. Estes novos cenários foram selecionados seguindo os mesmos critérios utilizados no trabalho de Da Silva et al. [1]; por exemplo, são projetos *open-source*, hospedados no GitHub, escritos na linguagem Java e com mudanças de dois desenvolvedores em um mesmo método ou atributo.

## 5.2 Geração de testes

A versão do sistema correspondendo a cada commit do cenário de *merge* selecionado na etapa anterior, foi compilado e transformado em um arquivo *Jar*. Randoop e Randoop CLEAN usam esse arquivo *Jar* para criar suítes de teste. Alguns testes obtiveram falhas ao serem criados, devido a falta de memória em alguns casos, e esses cenários foram descartados. Nesta etapa, os testes foram gerados para cada um dos cenários de *merge* (*base*, *left*, *right* e *merge*) adotando dois intervalos de tempo: 30 segundos e 5 minutos para cada ferramenta envolvida, cada ferramenta foi usada apenas uma vez para gerar testes para cada *commit* do cenário de *merge*. Esses intervalos de tempo foram escolhidos para analisarmos se o tempo adotado para a criação dos testes poderia interferir em melhores resultados de cobertura.

## 5.3 Instrumentação dos Jars

Instrumentação é uma técnica que insere “sondas” dentro do código-fonte. Essas sondas são ativadas em tempo de execução para coletar as informações necessárias sobre a cobertura de código. Em Java existem duas abordagens principais para essa técnica de instrumentação: *source code level* e *bytecode level* [11]. Neste trabalho, nós usamos a ferramenta Jacoco [7] para realizar essa etapa de instrumentação. Essa ferramenta usa a abordagem *source code level*, onde modifica o código-fonte do programa adicionando as sondas para coletar as informações de cobertura.



```

321 public void validate() {
321     boolean[] arrayOfBoolean = $jacocoInit();
321     Logger logger = LoggerFactory.getLogger(getClass());
322     if (this.acquireRetryDelay < 0L) {
322         arrayOfBoolean[77] = true;
323         logger.error("acquireRetryDelay cannot be negative.");
323         arrayOfBoolean[78] = true;
324         arrayOfBoolean[79] = true;
324         throw new IllegalStateException("acquireRetryDelay cannot be negative.");
324     }
326     if (this.acquireRetryDelay >= 100L) {
326         arrayOfBoolean[80] = true;
326     } else {
326         arrayOfBoolean[81] = true;
327         logger.warn("acquireRetryDelay is less than 100ms, did you specify the wrong time unit? Using default instead.");
328         this.acquireRetryDelay = ACQUIRE_RETRY_DELAY;
328         arrayOfBoolean[82] = true;
328     }
330     if (this.connectionCustomizerClassName == null) {
330         arrayOfBoolean[83] = true;
330     } else if (this.connectionCustomizer != null) {
330         arrayOfBoolean[84] = true;
330     } else {
330         try {
330             arrayOfBoolean[85] = true;
332             Class<?> clazz = getClass().getClassLoader().loadClass(this.connectionCustomizerClassName);
332             arrayOfBoolean[86] = true;
333             this.connectionCustomizer = (IConnectionCustomizer)clazz.newInstance();
333             arrayOfBoolean[87] = true;
338         } catch (Exception customizerClass) {
338             arrayOfBoolean[88] = true;
338             logger.warn("connectionCustomizationClass specified class '' + this.connectionCustomizerClassName + '' could not be loaded", (Throwable)customizerClass);
338             this.connectionCustomizerClassName = null;
338             arrayOfBoolean[89] = true;
338         }
340     }
340     if (this.connectionTimeout == 2147483647L) {
340         arrayOfBoolean[90] = true;
341         logger.warn("No connection wait timeout is set, this might cause an infinite wait.");
341         arrayOfBoolean[91] = true;
343     } else if (this.connectionTimeout >= 100L) {
343         arrayOfBoolean[92] = true;
343     } else {
343         arrayOfBoolean[93] = true;
344         logger.warn("connectionTimeout is less than 100ms, did you specify the wrong time unit? Using default instead.");
345         this.connectionTimeout = CONNECTION_TIMEOUT;
345         arrayOfBoolean[94] = true;
345     }
347     if (this.dataSource != null) {
347         arrayOfBoolean[95] = true;
347     } else if (this.dataSourceClassName != null) {
347         arrayOfBoolean[96] = true;
347     } else {
347         arrayOfBoolean[97] = true;
348         logger.error("one of either dataSource or dataSourceClassName must be specified");
348         arrayOfBoolean[98] = true;
349         arrayOfBoolean[99] = true;
349         throw new IllegalStateException("one of either dataSource or dataSourceClassName must be specified");
349     }

```

Figura 6 - Exemplo de método instrumentado

A Figura 6 exemplifica as alterações que foram realizadas nos *Jars* nesta etapa. Onde nós podemos encontrar o mesmo método *validate()* com sua versão instrumentada, método previamente apresentado na Figura 1. Nós podemos observar nas linhas 321, 323 e 324, por exemplo, que as sondas inseridas pela ferramenta Jacoco buscam informar se estas respectivas linhas foram executadas ou não durante a execução de uma suíte de testes, e, portanto, informando a cobertura de testes associada a cada suíte.

## 5.4 Execução de Suítes de Testes utilizando os Jars Instrumentados

Após a etapa de instrumentação dos jars, nós executamos as suítes de testes geradas previamente para os commits do cenário de *merge*, os testes gerados para cada *commit* do

cenário de *merge* foi executado apenas uma vez. Esta execução é feita a partir dos *Jars* instrumentados, resultando em um arquivo chamado *jacoco.exec*, único para cada execução. Este arquivo representa a análise de cobertura dos testes em um formato codificado, que é usado pela ferramenta do Jacoco para gerar o reporte final de cobertura.

## 5.5 Obtenção das Métricas de Cobertura de Código

Após a geração de cada arquivo *jacoco.exec*, para cada execução das suítes de testes em cada commit de um cenário de merge, nós utilizamos a ferramenta Jacoco para gerar o reporte final com as informações de cobertura. Este processo é feito usando o comando *report*, responsável por decodificar o arquivo *jacoco.exec* e gerando um arquivo CSV com as informações de cobertura.

A seguir, nós apresentamos as métricas que serão utilizadas no estudo. Para cada métrica adotada, nós apresentaremos sua definição bem como seu cálculo através de um exemplo obtido do projeto HikariCP, ver Seção (1.1). Os resultados obtidos para cada métrica podem ser encontrados na Tabela 1.

Ferramenta	MÉTRICA					
	SUA			CUA		MUA
	MC	CC	LC	MC	LC	LC
Randooop	21.7%	28.5%	19.0%	96.2%	70.8%	35.8%
Randooop CLEAN	21.7%	28.5%	19.0%	96.2%	79.1%	62.2%

Tabela 1 - Resultados obtidos para o projeto HikariCp *commit*  
(9eb405e5b81082044daca981b2655396c4e9a6eb)

SUA → Sistema sob análise
CUA → Classe sob análise
MUA → Método sob análise
MC → Cobertura de Método
CC → Cobertura de Classe
LC → Cobertura de Linha

Tabela 2 - Descrição das Métricas

## I. SUA

SUA (*System Under Analysis*) é uma métrica usada para medir a cobertura de código considerando todo o sistema em análise. Essa métrica é obtida através da leitura de um dos arquivos gerados pelo Jacoco ao final de cada *report* da ferramenta. Como exemplo de arquivo gerado pela execução de 30 segundos usando a ferramenta Randoop, nós temos a Figura 7.

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
javassist.bytecode		26%		18%	1,749	2,218	4,130	5,490	828	1,155	67	115
javassist.compiler		27%		19%	1,462	1,736	3,156	4,301	263	411	13	26
javassist		16%		13%	1,174	1,364	2,852	3,458	579	722	38	60
javassist.bytecode.analysis		0%		0%	689	689	1,601	1,601	200	200	21	21
javassist.util.proxy		0%		0%	286	286	858	858	131	131	20	20
javassist.bytecode.stackmap		44%		42%	394	632	752	1,436	99	213	7	21
javassist.expr		0%		0%	260	260	749	749	131	131	18	18
javassist.bytecode.annotation		0%		0%	279	279	617	617	187	187	19	19
javassist.tools.reflect		0%		0%	143	143	382	382	68	68	10	10
javassist.convert		0%		0%	167	167	386	386	50	50	10	10
org.slf4j.helpers		0%		0%	224	224	433	433	136	136	12	12
com.zaxxer.hikari		23%		16%	114	176	296	425	42	93	5	6
javassist.tools.rmi		0%		0%	76	76	291	291	42	42	8	8
javassist.compiler.ast		25%		8%	189	252	309	429	134	194	11	24
javassist.tools.web		0%		0%	83	83	242	242	34	34	4	4
com.zaxxer.hikari.proxy		30%		37%	82	102	262	335	56	62	6	7
javassist.scopedpool		0%		0%	80	80	187	187	47	47	5	5
com.zaxxer.hikari.util		15%		17%	58	66	133	155	25	29	4	6
org.slf4j		0%		0%	61	61	163	163	27	27	3	3
org.slf4j.impl		49%		43%	70	102	115	235	38	64	2	6
javassist.util		0%		0%	28	28	94	94	18	18	3	3
javassist.runtime		0%		0%	39	39	71	71	20	20	5	5
com.zaxxer.hikari.hibernate		0%		0%	19	19	39	39	12	12	2	2
javassist.tools		0%		0%	4	4	20	20	2	2	2	2
Total	76,077 of 94,836	19%	8,070 of 9,534	15%	7,730	9,086	18,138	22,397	3,169	4,048	295	413

Figura 7 - Exemplo de arquivo usado para colher métrica SUA

Nesta figura, a coluna *Element* representa os pacotes do projeto, enquanto que as colunas *Classes*, *Methods* e *Lines* representam a quantidade de classes, métodos e linhas por pacotes, respectivamente. Ao lado de cada uma dessas colunas, há uma coluna *Missed*, que representa a quantidade de classes, métodos e linhas que não foram cobertos pelos testes, respectivamente. Ao final do arquivo, há uma linha *Total*, que representa a soma dos valores de cada coluna, exceto para coluna *Cov*, que representa a porcentagem de *Instructions* e *Branches* que foram cobertas. A Figura 8 mostra como funciona o passo a passo para obtenção da cobertura. Então, nós subtraímos o valor obtido na linha *Total* para cada uma das colunas *Classes*, *Methods* e *Lines*, pelo valor obtido na coluna *Missed*, respectivo a cada coluna. Assim, nós obtemos a quantidade de *Classes*, *Methods* e *Lines* que foram cobertos por uma dada suíte de testes (linhas 3, 5 e 7). Por fim, o resultado é dividido pelo total de *Classes*, *Methods* e *Lines* e multiplicado por 100 para obtenção da porcentagem (linhas 10, 12 e 14). Com isso, nós podemos obter a análise de classes, métodos e linhas para o sistema como um todo (linha 16).

```
1 analiseCobertura(totalClasses, totalClassesNaoCobertas, totalMetodos, totalMetodosNaoCobertos, totalLinhas,
2 totalLinhasNaoCobertas)
3 totalClassesCobertas <- totalClasses - totalClassesNaoCobertas
4
5 totalMetodosCobertos <- totalMetodos - totalMetodosNaoCobertos
6
7 totalLinhasCobertas <- totalLinhas - totalLinhasNaoCobertas
8
9
10 coberturaClasses <- (totalClassesCobertas / totalClasses) * 100
11
12 coberturaMetodos <- (totalMetodoCobertos / totalMetodos) * 100
13
14 coberturaLinhas <- (totalLinhasCobertas / totalLinhas) * 100
15
16 return {coberturaClasses, coberturaMetodo, coberturaLinhas}
```

Figura 8 - Passo a passo para obter a cobertura

## II. CUA

CUA (*Class under analysis*) é uma métrica utilizada para medir a cobertura de código considerando apenas uma classe alvo dentre todas as classes do sistema. Observando novamente a Figura 7, na coluna *Element* existem pacotes. Estes pacotes podem ser compostos de outros pacotes, onde as classes de um sistema são agrupadas. Por exemplo, explorando a hierarquia do pacote *com.zaxxer.hikari*, nós podemos ter acesso às informações de cobertura das classes presentes no pacote *Element*, como representado na Figura 9.

### com.zaxxer.hikari

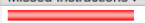
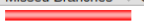










Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
HikariPool		0%		0%	55	55	161	161	19	19	1	1
HikariConfig		62%		42%	23	85	53	182	2	53	0	1
HikariDataSource		0%		0%	22	22	40	40	15	15	1	1
HikariMBeanElf		0%		0%	5	5	23	23	3	3	1	1
HikariPool.HouseKeeper		0%		0%	8	8	18	18	2	2	1	1
HikariPool.AddConnectionStrategy		0%		0%	n/a	1	1	1	1	1	1	1
Total	1,279 of 1,677	23%	138 of 165	16%	114	176	296	425	42	93	5	6

Figura 9 - Exemplo de arquivo usado para análise de classe

A Figura 9 representa o exemplo de um arquivo usado para obtenção dos dados relativos à análise de cobertura de uma classe alvo dos testes; por exemplo, a classe *HikariConfig*. Seguindo a mesma sequência de passos descrita na análise de cobertura SUA, podemos obter a análise de cobertura de métodos e linhas, que considera apenas a classe em análise.

## III. MUA

MUA (*Method Under Analysis*) é uma métrica usada para medir a cobertura de código considerando apenas um método dentre todos os métodos declarados em uma classe. Esta é a métrica mais importante para o propósito do estudo, pois as alterações implementadas em Randoop CLEAN, são focadas em melhorar a cobertura do método alvo dos testes, por isso é nela que nós podemos observar a maior diferença de cobertura. Observando a Figura 9, na coluna *Element*, nós temos as classes presentes neste pacote. Para cada classe deste pacote, as informações de cobertura podem ser obtidas buscando o arquivo relativo a classe. A Figura 10 apresenta as informações de cobertura da classe *HikariConfig*.

## HikariConfig

Source file "com/zaxxer/hikari/HikariConfig.java" was not found during generation of report.

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
• validate()	<div><div></div></div>	35%	<div><div></div></div>	29%	17	23	34	53	0	1
• copyState(HikariConfig)	<div><div></div></div>	0%	<div><div></div></div>	0%	3	3	8	8	1	1
• HikariConfig(String)	<div><div></div></div>	48%	<div><div></div></div>	50%	1	2	9	13	0	1
• setAcquireRetryDelay(long)	<div><div></div></div>	61%	<div><div></div></div>	50%	1	2	1	4	0	1
• getConnectionCustomizer()	<div><div></div></div>	0%	<div><div></div></div>	n/a	1	1	1	1	1	1
• HikariConfig()	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	15	0	1
• setConnectionTimeout(long)	<div><div></div></div>	100%	<div><div></div></div>	100%	0	3	0	6	0	1
• static {...}	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	6	0	1
• setAcquireIncrement(int)	<div><div></div></div>	100%	<div><div></div></div>	100%	0	2	0	4	0	1
• setAcquireRetries(int)	<div><div></div></div>	100%	<div><div></div></div>	100%	0	2	0	4	0	1
• setMinimumPoolSize(int)	<div><div></div></div>	100%	<div><div></div></div>	100%	0	2	0	4	0	1
• setMaximumPoolSize(int)	<div><div></div></div>	100%	<div><div></div></div>	100%	0	2	0	4	0	1
• addDataSourceProperty(String, Object)	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	2	0	1
• HikariConfig(Properties)	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	3	0	1
• setDataSourceProperties(Properties)	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	2	0	1
• setCatalog(String)	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	2	0	1
• setConnectionCustomizerClassName(String)	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	2	0	1
• setConnectionTestQuery(String)	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	2	0	1
• setConnectionInitSql(String)	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	2	0	1
• setDataSource(DataSource)	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	2	0	1
• setDataSourceClassName(String)	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	2	0	1
• setIdleTimeout(long)	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	2	0	1
• setAutoCommit(boolean)	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	2	0	1
• setInitializationFailFast(boolean)	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	2	0	1
• setJdbc4ConnectionTest(boolean)	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	2	0	1
• setRegisterMbeans(boolean)	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	2	0	1
• setLeakDetectionThreshold(long)	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	2	0	1
• setMaxLifetime(long)	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	2	0	1
• setPoolName(String)	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	2	0	1
• setTransactionIsolation(String)	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	2	0	1
• getAcquireIncrement()	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	1	0	1
• getAcquireRetries()	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	1	0	1
• getAcquireRetryDelay()	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	1	0	1
• getCatalog()	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	1	0	1
• getConnectionCustomizerClassName()	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	1	0	1
• getConnectionTestQuery()	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	1	0	1
• getConnectionInitSql()	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	1	0	1
• getConnectionTimeout()	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	1	0	1
• getDataSource()	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	1	0	1
• getDataSourceClassName()	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	1	0	1
• getDataSourceProperties()	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	1	0	1
• getIdleTimeout()	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	1	0	1
• isAutoCommit()	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	1	0	1
• isInitializationFailFast()	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	1	0	1
• isJdbc4ConnectionTest()	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	1	0	1
• isRegisterMbeans()	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	1	0	1
• getLeakDetectionThreshold()	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	1	0	1
• getMaxLifetime()	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	1	0	1
• getMinimumPoolSize()	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	1	0	1
• getMaximumPoolSize()	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	1	0	1
• getPoolName()	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	1	0	1
• getTransactionIsolation()	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	1	0	1
• setUseInstrumentation(boolean)	<div><div></div></div>	100%	<div><div></div></div>	n/a	0	1	0	1	0	1
Total	234 of 632	62%	37 of 64	42%	23	85	53	182	2	53

Figura 10 - Exemplo de arquivo usado para análise do método alvo

Nesta figura, nós podemos encontrar na primeira linha o método alvo dos testes, o método *validate()*, ver Seção (1.1). Seguindo a mesma análise descrita na métrica de SUA, nós podemos obter a análise de cobertura de linhas para o método alvo.

## 5.6 Análise Manual sobre a Quantidade e Qualidade dos Testes Gerados

Após a obtenção da cobertura de código para cada suíte de testes para um dado commit, nós avaliamos todos os testes gerados manualmente explorando dois objetivos principais. Inicialmente, nós identificamos o número de testes gerados em uma dada suíte. Para tanto, nós verificamos a assinatura dos casos de teste. Isso porque os nomes dos testes gerados pelas ferramentas são formados por uma *String* (*test*) e um valor inteiro, em ordem crescente, como exemplo temos na Figura 4 o método de teste *test0090()*. Assim, para identificar o número de testes gerados, nós extraímos o valor inteiro do último caso de teste gerado. Sobre a qualidade dos testes gerados, nós verificamos em cada arquivo de teste gerado, a quantidade de vezes que o método alvo foi chamado naquele conjunto de testes. Por fim, nós somamos o resultado de chamados observado em cada arquivo de teste, obtendo o número de chamadas realizadas em uma única suíte.

## 6 Resultados

Nesta seção, nós apresentaremos os resultados de nosso estudo empírico, previamente motivado e apresentado na seção anterior. Inicialmente, nós apresentamos os valores alcançados pela cobertura de código, bem como a quantidade e qualidade de testes gerados para os 46 cenários de *merge* de 31 projetos, ver Seção (5.1). Para tanto, nós agrupamos nossos resultados baseados nos diferentes intervalos de tempo usados para a criação das suítes de testes, 30 segundos e 5 minutos, respectivamente, para cada ferramenta. Em seguida, nós discutimos nossos resultados, suas contribuições, ameaças à validade e trabalhos futuros.

### 6.1 Resultado da execução de 30 segundos

A seguir, nós apresentaremos o resultado dos testes gerados por 30 segundos. Inicialmente apresentaremos os resultados da análise de cobertura dos cenários e logo a seguir, o resultado da análise sobre a quantidade e qualidade dos testes gerados.

Tempo		MÉTRICA					
		SUA			CUA		MUA
		MC	CC	LC	MC	LC	LC
30s	Randoop	13	7	16	10	13	2
	Randoop CLEAN	50	40	86	16	25	5
	Empates	96	112	57	133	121	54
	%	31.44%	25.15%	54.08%	10.06%	15.72%	3.14%

Tabela 3 - Cobertura para 30 segundos



Na Tabela 3, nós apresentamos a comparação da cobertura de código entre cada ferramenta adotando o intervalo de tempo de execução de 30 segundos. Para tanto, 159 *commits* (46 cenários de merge X 4 *commits* por cenário - 25 *commits* descartados) foram analisados por cada ferramenta. A comparação é feita de acordo com a cobertura reportada por cada ferramenta para cada *commit* analisado. O empate acontece quando as ferramentas apresentam a mesma cobertura reportada naquele caso. A linha de porcentagem representa o percentual de casos onde Randoop CLEAN foi superior considerando o total de casos analisados. Cada valor encontrado na linha de Randoop e Randoop CLEAN representa a quantidade de casos onde cada ferramenta foi superior, levando em consideração as métricas apresentadas na Seção (2.2), então a soma total de cada coluna é 159, pois representa a quantidade total de *commits* analisados. A quantidade apresentada em empates na coluna MUA (linha 7, coluna 8), está desconsiderando os casos onde as ferramentas obtiveram cobertura em 0% e os casos onde o método *target* não foi encontrado no *Jar* usado como base para geração da suíte de testes. Nós também desconsideramos os casos onde houveram problemas na checagem da cobertura do método *target*. Por isso, nossa comparação referente à métrica de cobertura de linha do método alvo (MUA) envolveu apenas 61 casos.

Considerando os valores reportados pelas métricas na Tabela 3, nós podemos observar que, para quase todos os critérios envolvidos, a ferramenta Randoop CLEAN obteve um ganho em mais de 100%, em cenários onde conseguiu obter uma maior cobertura nos testes gerados, em comparação com Randoop. É possível observar também, que apesar da porcentagem superior de casos, onde Randoop CLEAN obteve uma maior cobertura, em média, 70% dos casos resultaram em empate na cobertura obtida por ambas as ferramentas. Desta forma, podemos concluir que as ferramentas ainda possuem muitas semelhanças apesar das modificações realizadas.

Para exemplificar um caso, onde as ferramentas obtiveram um empate na cobertura de código, nós analisamos manualmente o caso do projeto *netty* na versão do *commit* 193acdb36cd3da9bfc62dd69c4208dff3f0a2b1b com o método *target decode()*, que nós podemos encontrar na Figura 11.

Na análise da cobertura desse projeto, ambas as ferramentas apresentaram uma cobertura de 4% na métrica de análise da cobertura do método alvo (MUA). Ao analisar manualmente os testes gerados para esse *commit*, nós observamos que o método *target* é invocado 355 vezes dentre todos os 10362 testes criados pela ferramenta Randoop, e invocado

331 vezes dentre todos os 11236 testes criados pela ferramenta Randoop CLEAN, considerando o tempo para criação dos testes de 30 segundos.

```
312 @Override
313 protected Object decode(
314     ChannelHandlerContext ctx, Channel channel, ChannelBuffer buffer) throws Exception {
315
316     if (discardingTooLongFrame) {
317         long bytesToDiscard = this.bytesToDiscard;
318         int localBytesToDiscard = (int) Math.min(bytesToDiscard, buffer.readableBytes());
319         buffer.skipBytes(localBytesToDiscard);
320         bytesToDiscard -= localBytesToDiscard;
321         this.bytesToDiscard = bytesToDiscard;
322         failIfNecessary(ctx, false);
323         return null;
324     }
325
326     if (buffer.readableBytes() < lengthFieldEndOffset) {
327         return null;
328     }
329
330     int actualLengthFieldOffset = buffer.readerIndex() + lengthFieldOffset;
331     long frameLength;
332     switch (lengthFieldLength) {
333     case 1:
334         frameLength = buffer.getUnsignedByte(actualLengthFieldOffset);
335         break;
336     case 2:
337         frameLength = buffer.getUnsignedShort(actualLengthFieldOffset);
338         break;
339     case 3:
340         frameLength = buffer.getUnsignedMedium(actualLengthFieldOffset);
341         break;
342     case 4:
343         frameLength = buffer.getUnsignedInt(actualLengthFieldOffset);
```

Figura 11 - Método *decode()*

Para explicar a cobertura de 4%, primeiramente, nós observamos um exemplo de teste criado pelas ferramentas, onde podemos observar na Figura 12, que os objetos criados e passados como parâmetro para o método *decode()* (linhas 440, 441, 442) são nulos. Na linha 318 do método *target*, apresentado na Figura 11, uma propriedade de um dos objetos passado como parâmetro é testada, porém, como o objeto em questão está nulo, uma exceção do tipo *NullPointerException* é lançada encerrando a execução do teste na terceira linha do método *target*.

```

435  @test
436  public void test0027() throws Throwable {
437      if (debug)
438          System.out.format("%n%s%n", "RegressionTest0.test0027");
439      org.jboss.netty.handler.codec.frame.LengthFieldBasedFrameDecoder lengthFieldBasedFrameDecoder0 = new org.jboss.netty.handler.
440      org.jboss.netty.channel.ChannelHandlerContext channelHandlerContext1 = null;
441      org.jboss.netty.channel.Channel channel2 = null;
442      org.jboss.netty.buffer.ChannelBuffer channelBuffer3 = null;
443      // The following exception was thrown during execution in test generation
444      try {
445          java.lang.Object obj4 = lengthFieldBasedFrameDecoder0.decode(channelHandlerContext1, channel2, channelBuffer3);
446          org.junit.Assert.fail("Expected exception of type java.lang.NullPointerException; message: null");
447      } catch (java.lang.NullPointerException e) {
448          // Expected exception.
449      }
450  }

```

Figura 12 - Exemplo de teste gerado por Randoop.

Nós podemos explicar o comportamento de Randoop ao gerar testes com objetos nulos. Existe uma limitação na geração de objetos em Randoop, quando os objetos a serem gerados para os métodos são interfaces, Randoop não consegue identificar quais são as classes que implementam a interface e com isso não consegue gerar objeto com atributos, criando assim objetos nulos.

### 6.1.1 Ganho aumenta com o refinamento da métrica

A seguir, nós apresentamos o resultado do percentual médio de ganho de Randoop CLEAN bem com o desvio padrão dos ganhos.

Tempo		MÉTRICAS					
		SUA			CUA		MUA
		MC	CC	LC	MC	LC	LC
30s	Percentual Médio de Ganho	0.62%	1.75%	0.46%	9.99%	7.80%	11.92%
	Desvio Padrão	0.98	2.79	0.83	7.94	7.95	8.93

Tabela 4 - Porcentagem para 30 segundos

Na Tabela 4, nós apresentamos a porcentagem média do ganho e o desvio padrão referente aos casos onde o Randoop CLEAN obteve uma maior cobertura. Para o cálculo do

Percentual Médio de Ganho, nós calculamos o percentual de ganho de cobertura por *commit*, dos casos onde Randoop CLEAN obteve maior percentual de cobertura de código e após isso nós calculamos a média. Observando a Tabela 4, nós podemos notar que apesar do baixo ganho na métrica SUA (análise de cobertura referente a todo o sistema), a ferramenta apresentou uma melhora significativa de porcentagem de cobertura com o refinamento da métrica. Por exemplo, a métrica MUA (análise de cobertura de método) apresenta um média de quase 12% de cobertura de linha do método alvo dos testes. Então, para o tempo de 30 segundos, que pode ser considerado um tempo baixo, levando em consideração o tamanho dos sistemas que nós analisamos, Randoop CLEAN possui vantagem na cobertura, porém sem considerar um número grande de *commits* e uma intensidade significativa, tendo vista que a média ficou em torno de 12%.

### 6.1.2 Melhora na quantidade e qualidade dos testes

A seguir, nós apresentamos a comparação da quantidade e qualidade dos testes gerados pelas ferramentas em análise.

<b>Tempo</b>		Quantidade de testes gerados	Qualidade dos testes gerados
30s	Randoop	13	2
	Randoop CLEAN	22	22
	Empates	1	12
	Total	36	36

Tabela 5 - Análise dos testes para 30 segundos

Na Tabela 5, nós apresentamos a quantidade e qualidade dos testes gerados por cada ferramenta (Seção 5.6), onde o método alvo é diretamente chamado pelos testes. Cada valor

representa a quantidade de casos onde cada ferramenta foi superior considerando a métrica envolvida. Para nós obtermos o valor total, a amostra foi dividida por método *target*, onde 36 (46 métodos targets - 10 descartados pois as ferramentas não conseguiram gerar testes) representam a quantidade total de métodos *target* que os testes conseguiram abordar. A amostra foi dividida dessa forma para nós obtermos a métrica de qualidade dos testes. Na linha 2, nós temos a comparação da quantidade de testes gerados por cada ferramenta, onde nós podemos notar que a ferramenta Randoop CLEAN gerou mais testes em mais de 61% dos casos totais. Posteriormente, nós podemos observar também a baixa quantidade de empates, apenas 1 (linha 2, coluna 5). Na linha 3, nós temos a comparação da qualidade dos testes gerados, onde nós avaliamos o número de vezes que o método alvo é usado na suíte de testes gerada por cada ferramenta. Observando os valores obtidos, podemos notar que Randoop CLEAN obtém um ganho de mais de 1000%, porém ainda existe uma grande quantidade de empates que representa em média 30% dos casos totais. Com isso, podemos concluir que a modificação realizada em Randoop, possui impacto na qualidade dos testes gerados e isso resulta na melhoria da cobertura de código apresentada nos resultados anteriores. Por outro lado, a quantidade de empates indica que melhorias ainda podem ser feitas.

## 6.2 Resultado da execução de 5 minutos

A seguir, nós apresentaremos o resultado dos testes gerados por 5 minutos. Inicialmente apresentaremos os resultados da análise de cobertura dos cenários e logo a seguir, o resultado da análise sobre a quantidade e qualidade dos testes gerados.

Tempo		MÉTRICAS					
		SUA			CUA		MUA
		MC	CC	LC	MC	LC	LC
5min	Randoop	3	2	3	2	2	0
	Randoop CLEAN	36	36	38	11	15	2
	Empates	110	110	108	136	132	54
	%	24.16%	24.16%	25.50%	7.38%	10.06%	1.34%

Tabela 6 - Cobertura para 5 min

Na Tabela 6, nós apresentamos a comparação de cobertura de código entre cada ferramenta adotando o intervalo de tempo de execução de 5 minutos. Para tanto, foram analisados 149 *commits* por cada ferramenta. Nossa comparação foi feita seguindo os mesmos critérios informados na apresentação dos resultados da Tabela 3. Neste sentido, também foram desconsiderados os casos onde as ferramentas obtiveram cobertura de 0%, ou quando os métodos alvo não foram encontrados nos arquivos *Jar*, usados como base para gerar a suíte de teste, bem como os casos onde houveram problemas na checagem da cobertura do método *target*. Por isso, nossa comparação referente à métrica de cobertura de linha do método alvo (MUA) envolveu apenas 56 casos. O número total de casos apresentado na coluna relativa à análise de classe (CC) não soma 149; o motivo disso é que existe um caso onde a cobertura de ambas as ferramentas foi de 0%, e ele foi desconsiderado dos resultados. Assim, nossa comparação referente a esta métrica envolveu 148 *commits*.

Na Tabela 6, nós podemos observar que apesar da adoção de um tempo maior disponibilizado para as ferramentas gerarem testes, ocorreu uma diminuição de 10 casos analisados, devido à ocorrência de erros na geração dos testes, ver Seção (5.3). É possível observar também, que apesar de Randoop CLEAN continuar apresentando um ganho maior, houve um aumento no número de casos que as ferramentas apresentaram empate (mesmo percentual de cobertura), exceto para a cobertura de classe (CC, linha 4, coluna 5). Por exemplo, na Tabela 3, para a cobertura de método (MC, linha 6, coluna 3) a quantidade de empates é 96, porém considerando a mesma métrica na Tabela 6, nós podemos observar o valor de 110 empates, um aumento de 14.58%. Observando a métrica de cobertura de linha (LC, linha 6, coluna 5) existe um aumento de 89.47% no número de empates. Comparando as demais métricas, podemos concluir que a mesma observação é válida. Desta forma, podemos concluir que a partir de um certo tempo disponibilizado para geração dos testes, as ferramentas se tornam equivalentes e, portanto, geram suítes de testes que apresentam a mesma cobertura de código.

### 6.2.1 Melhora na média de ganho

Tempo		MÉTRICAS					
		SUA			CUA		MUA
		MC	CC	LC	MC	LC	LC
5min	Percentual Médio de Ganho	0.76%	1.79%	0.64%	9.50%	9.19%	3.67%
	Desvio Padrão	1.03	2.86	0.94	5.45	6.76	0.14

Tabela 7 - Porcentagem para 5 minutos

Semelhante à Tabela 4, a Tabela 7 apresenta a porcentagem média de ganho dos casos onde Randoop CLEAN obteve uma cobertura superior, porém considerando o tempo de 5 minutos para geração da suíte de testes. Nesta tabela, nós podemos notar uma pequena melhora no ganho da cobertura do sistema (SUA); especificamente, nas métricas relativas à cobertura de linha (MC, linha 4, coluna 3) passando de 0.62% na Tabela 4, para 0.76%, e na cobertura de classes (CC, linha 4, coluna 4) passando de 1.75% na Tabela 4, para 1.79%. Porém, nós podemos notar uma baixa na média da cobertura de método (MUA, linha 4, coluna 8), onde a porcentagem média passou de 11.92%, na Tabela 4, para 3.67%. Esse resultado de baixo ganho, nos indica mais uma vez que com maior tempo disponível para criação de testes, as ferramentas tendem a se tornarem equivalentes.

### 6.2.2 Melhora na qualidade dos testes gerados

Tempo		Quantidade de testes gerados	Qualidade dos testes gerados
5min	Randoop	10	6
	Randoop CLEAN	24	18
	Empates	1	11
	Total	35	35

Tabela 8 - Análise dos testes para 5 minutos

A Tabela 8 é semelhante à Tabela 5, alterando o tempo de geração dos testes para 5 minutos. A quantidade total de casos é menor, pois para um método alvo<sup>1</sup>, ambas as ferramentas apresentaram erros e não conseguiram gerar testes. Na Tabela 8, nós podemos observar uma perda na qualidade dos testes gerados por Randoop CLEAN (linha 3, coluna 4), passando de 22 observado, na Tabela 5, para 18, e consequentemente uma melhora de Randoop (linha 3, coluna 3), passando de 2, observado na Tabela 5, para 6. Sobre a quantidade de testes, nós podemos observar uma melhora em Randoop CLEAN (linha 2, coluna 4) passando de 22 para 24, enquanto para Randoop (linha 2, coluna 3), nós podemos observar uma redução, passando de 13 para 10. Com esses resultados, nós podemos concluir que com maior tempo para gerar os testes, Randoop CLEAN conseguiu melhorar seu desempenho na geração de testes, porém houve uma perda na qualidade dos testes.



## 7 Ameaças a validade

Nosso trabalho apresenta algumas ameaças à validade discutidas a seguir. Inicialmente, como explicado durante a Seção de Metodologia (Seção 5.5), nós utilizamos a ferramenta Jacoco para obter a cobertura do código testado. Com isso, nossos resultados são diretamente baseados na cobertura reportada por esta ferramenta, e, portanto, eles podem apresentar inconsistências devido à forma que a cobertura é avaliada. Neste sentido, nós avaliamos manualmente um cenário para garantir que a cobertura reportada pela ferramenta para o projeto netty (ver Seção 6.1), de fato representa a cobertura da suíte de teste. Adicionalmente, a ferramenta Jacoco foi adotada baseada em estudos prévios que também a adotaram para tal função.

Outra ameaça é referente aos resultados da quantidade e qualidade de testes gerados pelas ferramentas. O processo foi feito checando o número de testes gerados e contando as chamadas feitas ao método alvo dos testes. Esta análise foi realizada de maneira manual por apenas um pesquisador, e com isso, erros podem ter ocorrido.

Apesar de nós avaliarmos *commits* de cenários de *merge* neste trabalho, tais cenários foram identificados a partir de integrações de código por meio do comando *git merge*. Assim, integrações de código realizadas por meio de opções avançadas, como *rebase*, *squash*, *cherry-pick* e *stash-apply*, não são avaliadas neste trabalho.

A amostra deste estudo foi obtida a partir de projetos *open-source* em linguagem Java; logo, nossos resultados podem não ser generalizados para projetos em outra linguagem de programação ou privados.

Outra ameaça é referente à quantidade de projetos, cenários e *commits* que foram poucos. E por último, nós não executamos várias vezes os testes gerados pelas ferramentas para eliminar a aleatoriedade e testes *flakyness*.

## 8 Trabalhos Futuros

Ao longo deste trabalho, observações surgiram e acreditamos que elas podem ser investigadas de maneira mais detalhada. A seguir, nós listamos algumas dessas situações que podem ser objetos de futuros trabalhos:

- Realizar o mesmo trabalho, estendendo a infraestrutura utilizada para coletar automaticamente os resultados relacionados a quantidade e qualidade dos testes gerados;
- Na análise da cobertura de linha do método *target* (MUA), mais de 50% dos casos não foram analisados devidos a deficiências de nossos *scripts* (ver Seção 6.1). Assim, como trabalho futuro, seria importante ajustar a infraestrutura para que mais casos possam ser avaliados usando os valores desta métrica;
- Com base nos resultados obtidos, seria importante analisar os casos onde ambas as ferramentas apresentaram empate ou não houve cobertura. Logo, é necessário entender as causas para este comportamento, e posteriormente, Randoop gerando uma nova versão melhorada.

## 9 Bibliografia

- [1] DA SILVA, Leuson; BORBA, Paulo; MAHMOOD, Wardah; BERGER, Thorsten; MOISAKIS, João. **Detecting Semantic Conflicts via Automated Behavior Change Detection**, 2020
- [2] P.K. Chittimalli , M.J. Harrold , **Recomputing coverage information to assist regression testing**, 2009
- [3] PACHECO, Carlos; LAHIRI, Shuvendu; ERNST, Michel; BALL Thomas. **Feedback-directed Random Test Generation**, 2007
- [4] SILVA, Indy; ALVES, Everton; ANDRADE, Wilkerson. **Analyzing Automatic Test Generation Tools for Refactoring Validation**, 2017
- [5] FRASER, G; ARCURI, A. Evosuite: **Automatic Test Suite Generation for Object-Oriented Software**, 2011
- [6] SARMA, Anite; HOEK, André. Palantír: **Early Detection of Development Conflicts Arising from Parallel Code Changes**, 2012
- [7] Mountainminds GmbH & Co. KG and Contributors. Jacoco - Mission, 2009. Disponível em <<https://www.jacoco.org/jacoco/trunk/doc/mission.html>>. Acesso em 10/09/2021
- [8] Mountainminds GmbH & Co. KG and Contributors. Command Line Interface, 2009. Disponível em <<https://www.jacoco.org/jacoco/trunk/doc/cli.html>>. Acesso em 10/09/2021
- [9] Randoop. Randoop Manual, 2020. Disponível em <<https://randoop.github.io/randoop/manual/index.html#Introduction>>. Acesso em 10/09/2021

[10] DA SILVA, Leuson. SMAT, 2019. Disponível em

<<https://github.com/leusonmario/SMAT/blob/master/MANIFEST.in>>. Acesso em 10/09/2021

[11] TENGERI, Dávid; FERENC, Horváth; ÁRPÁD, Beszédes; TAMÁS, Gergely; TIBOR, Gyimóthy.

**Negative Effects of Bytecode Instrumentation on Java Source Code Coverage, 2016**

[12] SHAMSHIRI, Sina; JUST, René; ROJAS, José; FRASE, Gordon; MCMINN, Phil; ARCURI, Andrea.

**Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study Of Effectiveness and Challenges, 2015**

[13] ARCURI, Andrea; GALEOTTI, Juan.

**Enhancing Search-based Testing with Testability Transformations for Existing APIs, 2021**

[14] Cobertura. Cobertura, 2021. Disponível em <<http://cobertura.github.io/cobertura/>>. Acesso em 27/10/2021

[15] SILVA, Indy; ALVES, Everton; MACHADO, Patrícia.

**Can automated test case generation cope with extract method validation? 2018**

[16] MOISAKIS, João; BORBA, Paulo; DA SILVA, Leuson

**Usando Transformações de Código para Melhorar Detecção de Conflitos de Teste através de Ferramentas de Geração de Testes de Unidade, 2021**