



UNIVERSIDADE FEDERAL DE SANTA CATARINA – UFSC
CENTRO TECNOLÓGICO – CTC
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA - INE
CURSO DE SISTEMAS DE INFORMAÇÃO

Gilmar Douglas Baray de Souza

SUnit: Uma ferramenta para redução do tempo de execução de testes
automatizados com JUnit 5

Florianópolis

2025

Gilmar Douglas Baray de Souza

SUnit: Uma ferramenta para redução do tempo de execução de testes
automatizados com JUnit 5

Trabalho de Conclusão de Curso apresentado ao
Departamento de Informática e Estatística da
Universidade Federal de Santa Catarina como
requisito parcial para obtenção do grau de
bacharel em Sistemas de Informação.

Orientadora: Prof.^a Dr.^a Patrícia Vilain

Coorientador: Me. Lucas Pereira Da Silva

Florianópolis

2025

Souza, Gilmar Douglas Baray de
SUnit: Uma ferramenta para redução do tempo de execução
de testes automatizados com JUnit 5 / Gilmar Douglas Baray
de Souza ; orientadora, Patrícia Vilain, coorientador,
Lucas Pereira da Silva, 2025.
93 p.

Trabalho de Conclusão de Curso (graduação) -
Universidade Federal de Santa Catarina, Centro
Tecnológico, Graduação em Sistemas de Informação,
Florianópolis, 2025.

Inclui referências.

1. Sistemas de Informação. 2. Testes de software. 3.
Testes automatizados. 4. Testes de unidade. 5. Reuso de
execução. I. Vilain, Patrícia. II. Silva, Lucas Pereira
da. III. Universidade Federal de Santa Catarina. Graduação
em Sistemas de Informação. IV. Título.

Gilmar Douglas Baray de Souza

SUnit: Uma ferramenta para redução do tempo de execução de testes
automatizados com JUnit 5

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do título
de bacharel em Sistemas de Informação e aprovado em sua forma final pelo curso
de Sistemas de Informação.

Florianópolis, 10 de Julho de 2025.

Coordenação do Curso

Banca examinadora

Profa. Patrícia Vilain, Dra.

Orientadora

Lucas Pereira da Silva, Me.

Coorientador

Profa. Fabiane Barreto Vavassori Benitti, Dra.

Prof. Raul Sidnei Wazlawick, Dr.

Florianópolis, 2025

RESUMO

Atualmente, testes automatizados se fazem de grande importância na linha de produção de software de qualidade, proporcionando uma redução nos custos de manutenção e aumentando a capacidade de entrega de uma equipe ágil. Este trabalho visa implementar uma extensão do framework JUnit 5, chamado SUnit, com o objetivo de reduzir o tempo gasto durante a execução de testes automatizados, através da abordagem de reutilização da configuração implícita de cada teste, sempre que possível, sem ferir o princípio de independência dos testes. A reutilização da configuração implícita de cada teste é feita através da identificação dos testes que não alteram o estado inicial dessa configuração, chamados de testes seguros. A redução no tempo gasto durante a execução dos testes é possível porque a configuração implícita incluída em uma classe de teste é executada uma única vez antes da execução de todos os testes seguros. Foi realizado um experimento para verificar a eficácia da ferramenta em um projeto real, com uma suíte de teste apresentando 391 métodos de teste; e o resultado mostrou uma redução de aproximadamente 46% no tempo de execução dos testes.

Palavras-chave: Testes de software. Testes automatizados. Testes de unidade. Reuso de execução. JUnit.

ABSTRACT

Currently, automated tests are of great importance in the software quality production pipeline, providing a reduction in maintenance costs and increasing the delivery capability of an agile team. This work aims to implement an extension of the JUnit 5 framework, called SUnit, with the objective of reducing the time spent during the execution of automated tests, through the approach of reusing the implicit setup of each test, whenever possible, without violating the principle of test independence. The reuse of each test's implicit setup is achieved by identifying tests that do not alter the initial state of this setup, called safe tests. The reduction in time spent during test execution is possible because the implicit setup included in a test class is executed only once before the execution of all safe tests. An experiment was conducted to verify the tool's effectiveness in a real project, with a test suite presenting 391 test methods; and the result showed a reduction of approximately 46% in test execution time.

Keywords: Software test. Automated test. Unit test. Execution reuse. JUnit.

LISTA DE FIGURAS

Figura 1 - Exemplo de uma classe de teste em linguagem de programação Java...	12
Figura 2 - Relação entre os níveis, tipos e técnicas de teste.....	16
Figura 3 - Técnicas de teste: caixa preta e caixa branca.....	17
Figura 4 - Gastos com desenvolvimento de código de software através do tempo...	18
Figura 5 - Pirâmide de automação de testes.....	20
Figura 6 - Exemplo de configuração local.....	21
Figura 7 - Exemplo de configuração implícita.....	22
Figura 8 - Exemplo de configuração delegada.....	22
Figura 9 - Categorias de ferramentas de automação de testes.....	23
Figura 10 - Configuração implícita de teste.....	32
Figura 11 - Fluxograma de reutilização da configuração implícita.....	33
Figura 12 - Diagrama de atividades da proposta.....	34
Figura 13 - Exemplo da classe Funcionario.....	35
Figura 14 - Exemplo da classe FuncionarioTest.....	37
Figura 15 - Diagrama de classes do SUnit.....	40
Figura 16 - Arquivo de configuração “sunit.properties”.....	41
Figura 17 - Árvore de sintaxe abstrata da classe Funcionario.....	42
Figura 18 - Classificação dos métodos de classe.....	44
Figura 19 - Árvore de sintaxe abstrata da classe FuncionarioTest.....	46
Figura 20 - Classificação dos métodos de teste.....	47
Figura 21 - Ordenação dos métodos de teste.....	49
Figura 22 - Trecho de código que define a execução da configuração implícita.....	50
Figura 23 - Definindo o diretório resource de teste.....	52
Figura 24 - Inserindo o SUnit no projeto.....	53
Figura 25 - Utilização do SUnit.....	53
Figura 26 - Diagrama de classe do projeto usado para validar o SUnit.....	55
Figura 27 - Resultados obtidos pelo SUnit no projeto criado pelo autor.....	56

SUMÁRIO

1 INTRODUÇÃO.....	10
1.1 Problema.....	11
1.2 Objetivo Geral.....	13
1.3 Objetivos Específicos.....	13
1.4 Organização deste trabalho.....	13
2 FUNDAMENTAÇÃO TEÓRICA.....	15
2.1 Testes de software.....	15
2.2 Testes automatizados.....	17
2.3 Teste de unidade.....	19
2.4 Configurações de testes.....	20
2.5 Ferramentas de automação de testes.....	23
2.6 JUnit 5.....	24
3 TRABALHOS RELACIONADOS.....	25
3.1 Estoria.....	26
3.2 SolUnit.....	27
3.3 GridUnit.....	28
3.4 Unit-Test Framework for Database.....	29
3.5 E2E test execution optimization for web application based on state reuse....	30
4 PROPOSTA.....	32
4.1 - Visão geral da proposta.....	32
4.2 - Etapas da Proposta.....	34
4.2.1 Análise do código fonte.....	34
4.2.2 Identificação dos métodos seguros.....	35
4.2.3 Identificação dos testes seguros.....	36
4.2.4 Ordenação e execução dos métodos de testes.....	36
5 IMPLEMENTAÇÃO.....	38
5.1 Diagrama de Classes.....	39
5.2 - Análise do código fonte.....	39

5.3 - Identificação dos métodos das classes seguros e não seguros.....	41
5.4 - Identificação dos testes seguros e não seguros.....	45
5.5 - Ordenação e execução dos métodos de teste.....	48
5.6 - Utilização do SUnit.....	51
6 EXPERIMENTOS.....	54
6.1 - Validação da classificação dos métodos de classe e testes.....	54
6.2 - Validação do SUnit em um projeto real.....	56
7 CONCLUSÃO.....	59
REFERÊNCIAS.....	60
APÊNDICE A - RESULTADO DA BUSCA DE TRABALHOS RELACIONADOS....	65
APÊNDICE B - MUDANÇA DO TEMA DESTE TRABALHO.....	68
APÊNDICE C - ARTIGO.....	71

1 INTRODUÇÃO

É comum que grandes empresas adotem um processo de qualidade rigoroso na etapa final de desenvolvimento de *software* e esse controle de qualidade pode ser realizado por meio de testes manuais, onde a execução desses testes precisa ser rápida e efetiva. Contudo, com a manutenção e evolução contínua do código, a execução de um vasto conjunto de testes manuais torna-se progressivamente cansativa e dispendiosa (BERNARDO; KON, 2008).

Bernardo e Kon (2008) destacam ainda o efeito “bola de neve” que pode ocorrer quando módulos do sistema deixam de funcionar após a correção de um erro ou evolução do sistema, ocasionando erros de regressão. Isto acontece devido à não execução de todo o conjunto de testes manuais por causa do grande esforço necessário para sua execução a cada alteração de código.

Para enfrentar esse desafio, garantindo a efetividade e reduzindo o tempo gasto na etapa final do desenvolvimento de *software* são introduzidos os testes automatizados, que podem diminuir em até 80% do tempo gasto durante a execução dos testes. (FEWSTER; GRAHAM, 1999).

Meszaros (2007) ainda aponta que a busca por *software* de qualidade aliado a entregas rápidas e contínuas vem crescendo muito e, com isso, os testes automatizados já são partes fundamentais do ciclo de desenvolvimento, pois garantem que as funcionalidades do sistema se comportem conforme o esperado, reduzindo riscos e custos com falhas em produção.

Contudo, à medida que o sistema cresce, e com isso a quantidade de testes automatizados desenvolvidos escala consideravelmente, o tempo de execução dos mesmos tende a aumentar significativamente, impactando diretamente na eficiência das equipes de desenvolvimento. Emery (2009) ainda destaca que um dos principais motivos para que equipes de desenvolvimento deixem de lado a inserção de testes automatizados está no fato de que testes tendem a se tornar frágeis e de alto custo. Tsai et al. (2003) relata que atividades de testes automatizados dentro do desenvolvimento de sistemas podem consumir em torno de 50% de todo o esforço de desenvolvimento e manutenção do código do *software*. Entretanto, Meszaros (2007) aponta que o tempo gasto com o desenvolvimento de testes automatizados em *softwares* deve ser de fato recompensado no esforço reduzido que

posteriormente deverá ser despendido nas etapas de manutenção e de garantia de qualidade do *software*.

Dessa forma, torna-se cada vez mais necessário buscar estratégias que possam otimizar o tempo de execução de testes automatizados e ainda manter a confiabilidade do processo de validação do *software*. Com isso em mente, este trabalho apresenta uma proposta de diminuição do tempo despendido durante a execução de testes automatizados.

1.1 Problema

Um dos principais obstáculos atuais das equipes de desenvolvimento é a execução excessivamente longa de testes automatizados, impactando diretamente na eficácia dos ciclos de integração contínua. Nesses ambientes, onde ocorre a busca por entrega contínua e rápida de valor, o tempo do *feedback* é fundamental. Segundo Humble e Farley (2010), um dos princípios fundamentais da entrega contínua é obter *feedback* rápido a partir da execução de testes automatizados, permitindo que falhas sejam detectadas e corrigidas rapidamente. Quanto mais demorada a execução dos testes automatizados, maior será o tempo de resposta pelo *feedback* dos testes, o que pode comprometer a agilidade de tais equipes.

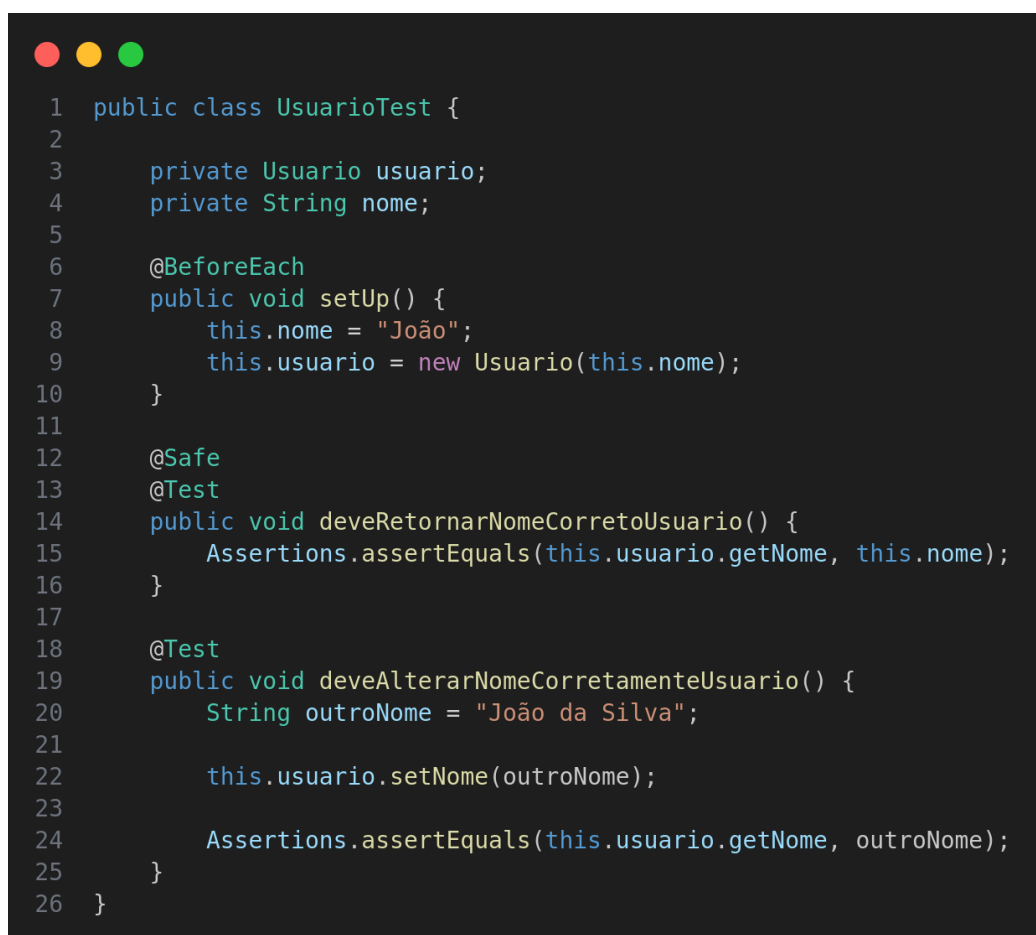
Outro fator importante é que testes automatizados muito demorados podem não incentivar desenvolvedores a executá-los com maior frequência, levando à negligência e ao acúmulo de dívidas técnicas, perdendo seu valor como mecanismo de prevenção de falhas e gerando aumento de gastos com manutenção (MESZAROS, 2007).

Além disso, processos de testes automatizados ineficientes podem impactar negativamente no tempo total de entrega de um projeto, o que aumenta o *lead time* do projeto e reduz a capacidade das equipes ágeis em responder rapidamente a mudanças nos requisitos de negócio (FORSGREN; HUMBLE; KIM, 2018). Com isso, otimizar o tempo de execução de testes automatizados permite otimizar processos de desenvolvimento de *software*, garantindo economia e aumentando a capacidade de entrega e evolução mais rápida de códigos de *software* de uma organização.

Nesse contexto, Silva e Vilain (2017) apresentaram uma proposta para reduzir o tempo de execução de testes automatizados através da reutilização das

configurações iniciais implícitas¹ de cada teste. Os autores partem do princípio que nem todos os métodos de teste fazem alteração das configurações implícitas de teste, e nesse caso as configurações podem ser reutilizadas entre esses métodos de teste. Contudo, na proposta de Silva e Vilain (2017) a classificação dos métodos de teste que reutilizarão as configurações implícitas de teste é feita de forma manual, sendo o desenvolvedor o responsável por marcar os métodos de testes que irão reutilizar as configurações implícitas. Na Figura 1 é possível visualizar um exemplo onde a configuração implícita de teste, representada pela notação *@BeforeEach*, deve ser executada antes de cada método de teste, representado pela notação *@Test*. Na Figura 1 também é possível visualizar, na linha 12, a notação *@Safe* que indica que o método de teste não altera a configuração implícita.

Figura 1 - Exemplo de uma classe de teste em linguagem de programação Java



```
1 public class UsuarioTest {
2
3     private Usuario usuario;
4     private String nome;
5
6     @BeforeEach
7     public void setUp() {
8         this.nome = "João";
9         this.usuario = new Usuario(this.nome);
10    }
11
12    @Safe
13    @Test
14    public void deveRetornarNomeCorretoUsuario() {
15        Assertions.assertEquals(this.usuario.getNome, this.nome);
16    }
17
18    @Test
19    public void deveAlterarNomeCorretamenteUsuario() {
20        String outroNome = "João da Silva";
21
22        this.usuario.setNome(outroNome);
23
24        Assertions.assertEquals(this.usuario.getNome, outroNome);
25    }
26 }
```

Fonte: Elaborado pelo autor.

¹ Ver seção 2.4

Já Medeiros (2020) apresenta uma proposta similar com aplicação em *smart contract* em uma *blockchain* de forma automatizada, utilizando o mesmo princípio da proposta anterior onde as configurações implícitas de teste podem ser reaproveitadas caso os métodos de teste sejam considerados seguros. A proposta de Medeiros (2020) realiza a classificação dos métodos de teste de forma automatizada e não de maneira manual, como a proposta de Silva e Vilain (2017) . Contudo, deve-se ter em mente que o desenvolvimento realizado em um *smart contract* tem características diferentes de um projeto desenvolvido em linguagem Java.

1.2 Objetivo Geral

O objetivo geral deste trabalho é propor a redução do tempo de execução de testes automatizados utilizando o *framework* JUnit através da reutilização das configurações implícitas de cada classe de teste quando possível, ocorrendo de forma automatizada e sem acarretar na alteração do resultado esperado durante a execução de cada teste.

1.3 Objetivos Específicos

- Identificação dos testes automatizados que podem reutilizar as configurações implícitas da classe de teste;
- Projeto de uma ferramenta que reutiliza a configuração implícita de testes de unidade;
- Desenvolvimento da ferramenta proposta que estende JUnit 5.

1.4 Organização deste trabalho

O presente documento está estruturado da seguinte forma: o Capítulo 2 apresenta a fundamentação teórica pesquisada para elaboração deste trabalho; o Capítulo 3 apresenta o estado da arte onde são apresentadas ferramentas que tem por objetivo reduzir o tempo de execução e aumentar o reuso de código; o Capítulo

4 trata sobre a proposta deste trabalho; o Capítulo 5 detalha a implementação da ferramenta SUnit; o Capítulo 6 apresenta os resultados obtidos com a execução de experimentos com a ferramenta deste trabalho; e o Capítulo 7 apresenta as conclusões.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os conceitos básicos no qual a presente pesquisa se fundamentou, elencados na seguinte ordem: testes de *software*, testes automatizados, testes de unidade, configurações de testes, ferramentas de automação de teste, JUnit 5, SolUnit.

2.1 Testes de *software*

O teste de *software* tem por objetivo verificar se o produto que foi projetado está funcionando corretamente, revelando falhas que devem ser resolvidas pelo time de desenvolvimento antes de entrar em na fase de produção (DIAS NETO, 2007). As atividades de teste de *software* não devem se restringir apenas ao produto final, elas devem ser executadas desde a sua concepção para garantir que o software está sendo construído de acordo com o especificado (DELAMARO; JINO; MALDONADO, 2013).

Vasconcelos *et al.* (2006) ressalta que a fase de teste não garante que o *software* está livre de falhas, pois só é possível garantir com eles que o *software* sob teste contém erros. Quando os testes não encontram erros haverá sempre duas possibilidades: a qualidade do *software* é aceitável ou os testes foram inadequados. Dias Neto (2007) por sua vez, afirma que em virtude da sua natureza de revelar falhas em *softwares*, os testes de *software* são de natureza destrutiva e não construtiva, onde a confiança em um produto é sustentada pela revelação de suas falhas antes da entrega final. Além disso, testes de *software* não ajudam apenas na detecção de falhas antes da entrega do produto final, mas também contribui para a melhoria contínua da qualidade do código do *software* (SOMMERVILLE, 2011).

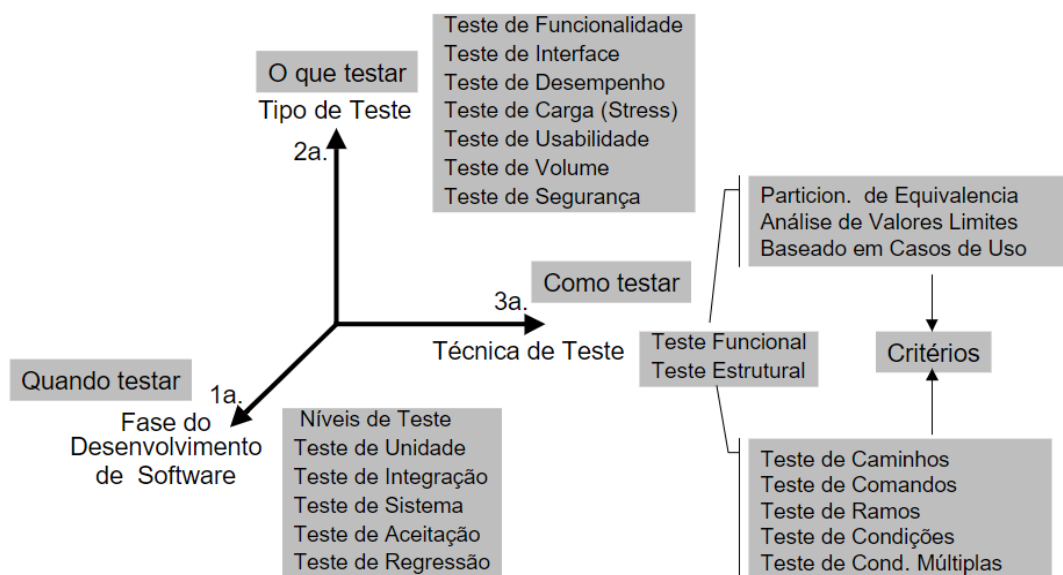
Crespo *et al.* (2004) aponta que o teste a ser aplicado depende da fase do desenvolvimento do *software*, sendo os seguintes níveis:

- Teste de unidade: abrangendo a codificação dos módulos;
- Teste de integração: compreendendo a etapa de integração dos módulos do sistema;

- Teste de sistema: verificando o atendimento aos requisitos funcionais e não funcionais do sistema;
- Teste de aceitação: onde é verificado a aceitação do sistema pelo usuário;
- Teste de regressão: que deve ser aplicado na fase de manutenção do sistema.

Crespo (2004) ainda aponta a existência de duas técnicas de teste: o teste estrutural que visa identificar defeitos na estrutura interna do código do *software*, e o teste funcional que tem por objetivo garantir que os requisitos do *software* foram plenamente alcançados. E, por fim, apresenta os tipos de teste que visam testar características específicas do *software*, sendo eles: teste de funcionalidade, teste de interface, teste de desempenho, teste de carga, teste de usabilidade, teste de volume e teste de segurança. A Figura 2 apresenta a ligação entre os níveis de teste, os tipos de teste e os critérios que podem ser adotados ao se definir a estratégia de teste.

Figura 2 - Relação entre os níveis, tipos e técnicas de teste

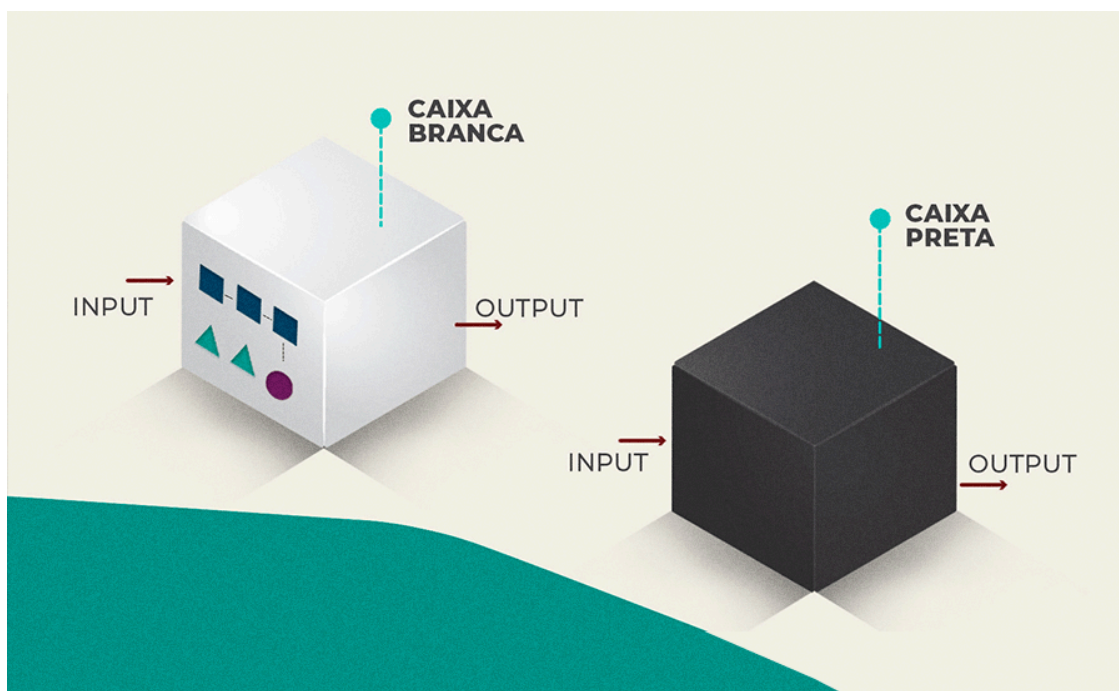


Fonte: Crespo, 2004.

As técnicas de teste podem ainda ser descritas como teste de caixa branca e teste de caixa preta, sendo análogos ao teste estrutural e teste funcional, respectivamente (MYERS; SANDLER; BADGETT, 2011). A Figura 3 ilustra de forma didática o funcionamento dessas técnicas de teste mencionadas, onde o teste de caixa branca permite verificar, através dos valores iniciais de *input*, o funcionamento

do código fonte e o valor esperado de *output*; enquanto no teste de caixa preta se tem apenas o conhecimento do valor de *output* esperado a partir do valor de *input*, sem ter conhecimento do funcionamento do código fonte.

Figura 3 - Técnicas de teste: caixa preta e caixa branca.



Fonte: Base2, 2023.

Fowler (2006) ainda destaca sobre a importância do teste de *software* não como sendo um custo adicional inerente ao projeto, mas sim um investimento estratégico que garante a sustentabilidade, a competitividade e a reputação do produto e da equipe de desenvolvimento dentro do mercado, e ainda defendendo a integração contínua dos mesmos em todas as fases do desenvolvimento.

2.2 Testes automatizados

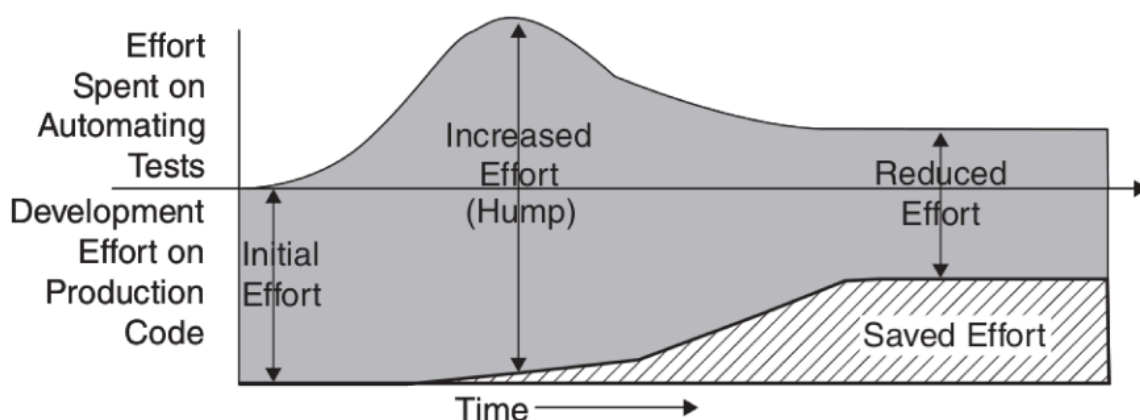
Meszaros (2007) define o teste automatizado como qualquer interação com o SUT (*system under test* ou sistema sob teste) feita sem qualquer intervenção manual, sendo eles ainda autoverificáveis e repetíveis, ou seja, quando os resultados esperados estão corretos e produzem o mesmo resultado sempre que repetidos. Barbosa e Torres (2011) ainda definem a automação de testes como a transcrição de testes manuais em linhas de código que podem ser lidos e

executados por *softwares*. Bartié (2002) ressalta que a automação pode simular atividades manuais, de forma a evitar o contato humano durante a execução dos testes, não exigindo mão de obra especializada.

Testes automatizados ajudam a controlar a qualidade do *software* de forma eficaz e apresentam um baixo custo ao longo do tempo (BERNARDO; KON, 2008). Outra vantagem dos testes automatizados, em comparação com os testes manuais, está na rapidez de sua execução e na consistência de seus resultados, garantindo uma integração contínua de código em produção (PRESSMAN; MAXIM, 2021). Outra característica positiva dos testes automatizados está no fato de que testes automatizados possuem precisão e consistências elevadas, eliminando a variabilidade associada à execução manual, e também permitem execução frequente de testes de regressão, o que garante que a correção de *bugs* ou o desenvolvimento de código novo não introduz falhas em partes já existentes do código fonte do sistema (MYERS; SANDLER; BADGETT, 2011).

Chiavegatto *et al.* (2013) apontam como principal desvantagem o alto custo para implantação inicial, uma vez que demanda de um tempo maior para elaboração dos códigos de teste e necessita de mão de obra qualificada. Nesse âmbito, a elaboração de testes automatizados traz retornos, conforme a Figura 4, onde é possível reduzir esforços para manutenção de código após a implantação de testes automatizados.

Figura 4 - Gastos com desenvolvimento de código de software através do tempo



Fonte: Meszaros, 2007.

2.3 Teste de unidade

O teste de unidade, como mencionado anteriormente, é um dos níveis de teste de *software*. Também conhecido como teste unitário, é o teste da menor unidade a ser testada dentro de um *software* (módulos ou componentes) e normalmente é executado pelo próprio desenvolvedor (SILVA, 2008). Sendo assim, pode-se dizer que é o primeiro nível de teste de *software*, uma vez que as unidades do produto são testadas isoladamente (MEDEIROS, 2020).

Seu objetivo visa testar as menores unidades do sistema, tais como métodos, funções, procedimentos e classes, limitando-se a lógica interna do código e onde espera-se encontrar com maior facilidade falhas de programação (JÚNIOR; ANDRADE; DELAMARO, 2016). Bernardo e Kon (2008) apontam o teste de unidade, para a maioria das situações, como um dos níveis de teste de maior importância, uma vez que garante que o código encapsulado por uma unidade produz o efeito esperado, além dos erros serem facilmente localizados e reparados.

Pela sua característica de verificar apenas uma classe ou método, sua implementação costuma ser fácil e comumente aplicada pelos próprios desenvolvedores, que são os que mais tiram proveito de sua aplicação (NAIK; TRIPATHY, 2008). Além disso, Meszaros (2007) diz que os testes unitários podem ajudar o desenvolvedor a entender a definição de pronto (*definition of ready*) adotado por equipes de planejamento. McConnell (2004) compara os testes de unidade como uma rede de segurança, já que tem por finalidade detectar defeitos precocemente e que garante uma economia significativa para o projeto, uma vez que se torna mais econômico corrigir falhas em fases iniciais do projeto do que falhas em fases posteriores, ou até mesmo em produção.

Dentre as qualidades de um teste unitário, Meszaros (2007) relata que devem ser independentes e não depender de outros testes ou do estado global do sistema e devem ser determinísticos e sempre apresentar os mesmos resultados quando os mesmos dados de entradas são aplicados. A presença de testes de unidades dentro de sistema costuma deixar a equipe de desenvolvedores mais confiante ao realizar mudanças no código, já que problemas durante a sua execução podem ser facilmente detectados, gerando mais confiabilidade, reduzindo a quantidade de bugs e facilitando a refatoração e compreensão do código (FOWLER, 2006).

divide as estratégias de configuração inicial em 3 padrões: configuração local, configuração implícita e configuração delegada.

A estratégia de configuração local ocorre quando as configurações do teste estão presentes dentro do próprio método de teste e acaba ocorrendo em testes iniciais ou em cenários muito específicos em que houve a decisão de se duplicar a configuração do teste. Uma das principais consequências dessa estratégia é a duplicação de código, pois os métodos de teste que podem utilizar a mesma configuração inicial devem incluir a configuração inicial dentro do seu próprio método de teste. A Figura 6 apresenta um exemplo da configuração local de teste, onde é possível ver as configurações iniciais do teste nas linhas 4 e 5.

Figura 6 - Exemplo de configuração local

A screenshot of a code editor with a dark background and light-colored text. At the top left, there are three colored circles (red, yellow, green) representing window control buttons. The code is written in Java and shows a class named 'FuncionarioTest' with a single test method 'criarFuncionarioTest'. The method contains two lines of configuration: setting a salary and creating a funcionario object. The lines are numbered 1 through 8 on the left side.

```
1 public class FuncionarioTest {
2     @Test
3     public void criarFuncionarioTest() {
4         float salario = 1500.00;
5         Funcionario funcionario = new Funcionario.criarFuncionario(salario);
6         Assertions.assertEquals(funcionario.getSalario, salario);
7     }
8 }
```

Fonte: Elaborado pelo autor.

Na estratégia de configuração implícita a configuração inicial de teste fica localizada em um método especial, mas ainda dentro da mesma classe de teste, e assim essas configurações podem ser reutilizadas em todos os métodos de teste localizados na mesma classe. A dificuldade nesta estratégia está em agrupar um número grande de testes que utilizem exatamente a mesma configuração inicial, isso devido à particularidade que cada teste pode ter. A Figura 7 apresenta um exemplo da configuração implícita, onde é apresentado o método *setUp* na linha 6 que representa a configuração inicial da classe de teste que é reutilizada por todos os métodos de teste presentes nesta classe.

Na estratégia de configuração delegada a configuração inicial de teste costuma ficar em um arquivo separado das classes de teste e, assim, pode ser compartilhado por mais de uma classe de teste. Nesta estratégia a principal dificuldade está na dificuldade de compreensão do resultado de alguns métodos de

Figura 7 - Exemplo de configuração implícita

```

1 public class FuncionarioTest {
2
3     float salario;
4     Funcionario funcionario;
5
6     @BeforeEach
7     public void setUp() {
8         this.salario = 1500.00;
9         this.funcionario = new Funcionario.criarFuncionario(this.salario);
10    }
11
12    @Test
13    public void criarFuncionarioTest() {
14        Assertions.assertEquals(this.funcionario.getSalario, this.salario);
15    }
16 }

```

Fonte: Elaborado pelo autor.

teste, uma vez que a configuração inicial fica em uma classe separada. A Figura 8 apresenta um exemplo de uso da configuração delegada, onde a classe *FuncionarioAuxiliar* exibida na linha 1 representa uma classe de configuração que pode ser utilizada por qualquer classe de teste.

Figura 8 - Exemplo de configuração delegada

```

1 public class FuncionarioAuxiliar {
2     public Funcionario setUpFuncionario() {
3         return new Funcionario.criarFuncionario(1500.00);
4     }
5 }
6
7
8 public class FuncionarioTest {
9     @Test
10    public void FuncionarioTest() {
11        Funcionario funcionario = new FuncionarioAuxiliar.setUpFuncionario();
12        float salario = 1500.00;
13
14        Assertions.assertEquals(funcionario.getSalario, salario);
15    }
16 }

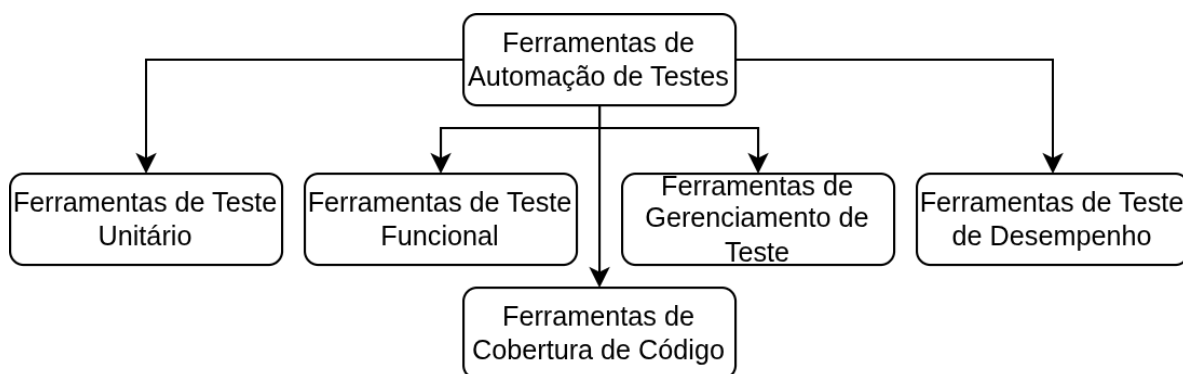
```

Fonte: Elaborado pelo autor.

2.5 Ferramentas de automação de testes

Silva (2025) aponta que as ferramentas e *frameworks* de teste automatizados possuem um papel fundamental na etapa de garantia de qualidade do *software*, uma vez que permitem a execução dos testes de forma rápida, eficiente, confiável e repetível. Em seu estudo, Umar e Zhanfang (2019) reuniu algumas ferramentas disponíveis no mercado e as classificou em diferentes categorias, conforme apresentado na Figura 9.

Figura 9 - Categorias de ferramentas de automação de testes



Fonte: Adaptado de Umar e Zhanfang, 2019.

A seguir são apresentadas as definições de cada categoria, juntamente com os *frameworks* que a elas pertencem e que foram destacados por Umar e Zhanfang (2019) em seu estudo:

- Ferramentas de Teste Unitário: ferramentas que garantem o funcionamento de pequenas unidades específicas do sistema, atuando nas partes mais fundamentais do código. *Frameworks* destacados: JUnit, NUnit, JMockit, PHPUnit;
- Ferramentas de Teste Funcional: ferramentas que atuam na verificação do sistema com relação aos requisitos de usuário, atuando em especificações de componentes. *Frameworks* destacados: Selenium, HP QuickTest Professional, TestComplete, Ranorex, Watir, Tricentis Tosca Testsuite, Test Studio;
- Ferramentas de Cobertura de Código: Ferramentas responsáveis por verificar trechos de código que foram abrangidos por teste automatizados, resultando

em um percentual de trechos não cobertos. *Frameworks* destacados: Coverage, CodeCover, EMMA, PITest, Atlassian Clover;

- Ferramentas de Gerenciamento de Teste: Ferramentas responsáveis pela criação de casos de teste, planos e estratégias. *Frameworks* destacadas são: Test Manager, Test Link, TETware, Test Environment Toolkit (TET), QA Complete;
- Ferramentas de Teste de Desempenho: Ferramentas responsáveis por avaliar o desempenho do sistema sob diferentes condições de carga. *Frameworks* destacados: JMeter, Rational Performance Tester, HP LoadRunner, Silk Performer

2.6 JUnit 5

JUnit é um *framework* para realização de testes automatizados em linguagem de programação Java, facilitando a criação de testes unitários através do seu relatório direto ao final de cada método de teste, sendo possível verificar se o sistema está se comportando como o esperado ou se há falhas ou erros (FERREIRA, 2017).

Uma das principais inovações do JUnit 5 em relação às versões anteriores é sua nova arquitetura modular, onde o *framework* é dividido em três subprojetos: JUnit Platform, que fornece a base para a execução de testes em diferentes ambientes; JUnit Jupiter, que contém a nova API de testes e extensões; e JUnit Vintage, que garante compatibilidade com versões anteriores do JUnit (JUNIT, 2025). Essa estrutura favorece a integração com outras ferramentas de testes, como Maven e Gradle, e IDEs, como Eclipse e IntelliJ IDEA.

O JUnit 5 utiliza notações intuitivas, por exemplo, `@Test`, `@BeforeEach`, `@AfterEach`, `@DisplayName` e `@Disabled`. Essas anotações oferecem maior controle sobre o ciclo de vida dos testes e permitem uma documentação mais legível do que está sendo verificado. Além disso, há suporte para testes parametrizados por meio da anotação `@ParameterizedTest`, o que permite executar um mesmo teste com diferentes entradas, promovendo reutilização e clareza no código de teste (GARCIA, 2017).

3 TRABALHOS RELACIONADOS

O presente capítulo tem por objetivo apresentar trabalhos presentes na literatura correlatos ao tema proposto neste trabalho. Serão abordados trabalhos que foram encontrados através de uma pesquisa na base de dados do Scopus². Com a finalidade de se obter resultados mais atualizados e refinados, a pesquisa foi realizada no dia 26/06/2025 e para a busca foi utilizada a seguinte string:

("unit test" OR "unit testing" OR "unit-test") AND ("test fixture" OR "test method" OR "test suite" OR "setup execution") AND ("reuse" OR "reusing" OR "reducing" OR "reduce").

A pesquisa obteve um total de 58 resultados, listados no Apêndice A. A partir disso então, uma nova seleção dos trabalhos se deu através da leitura do título de cada resultado e os que apresentavam uma proximidade maior com o tema deste trabalho foram selecionados para leitura do resumo. Assim foram selecionados 9 trabalhos apresentados a seguir na Tabela 1.

Tabela 1 - Trabalhos relacionados selecionados após leitura do título

Título	Autor(es)	Ano
E2E test execution optimization for web application based on state reuse	Ruilian Zhao, Shukai Zhang, Zhifan Zhu, Ying Shang, Weiwei Wang	2025
ATSM: A coverage-based framework and a tool for test suite minimization	Alekseev, Y., Onischuck, M., Zorin, A., Chernyi, V., Iliyn, E., Itsykson, V.	2024
SolUnit: A framework for reducing execution time of smart contract unit tests	Hallan Medeiros, John Mylopoulos, Patrícia Vilain, Hans-Arno Jacobsen	2020
Reuse of fixture setup between test classes	Lucas Silva, Patrícia Vilain	2017
Multi-objective Regression Test Suite Minimisation for Mockito	Andrew J. Turner, David R. White & John H. Drake	2016
Unit test virtualization with VMVM	Jonathan Bell, Gail Kaiser	2014
Reusing and Composing Tests with Traits	Stéphane Ducasse, Damien Pollet, Alexandre Bergel & Damien Cassou	2009
A Unit-Test Framework for Database Applications	Claus A. Christensen; Steen Gundersborg; Kristian de Linde; Kristian Torp	2006
GridUnit: Software testing on the grid	Alexandre Duarte, Walfredo Cirne, Francisco Brasileiro, Patrícia Machado	2006

Fonte: Elaborado pelo autor.

² <https://www.scopus.com/pages/home?display=basic#basic>

Após a leitura dos resumos, foram selecionados 5 trabalhos que apresentavam ferramentas propostas para redução do tempo de execução de testes automatizados, e assim foram selecionados os seguintes trabalhos: Silva e Vilain (2017), apresentando a ferramenta Estoria com a reutilização do configuração implícita da classe de teste; Medeiros *et al.* (2020), apresentando o *framework* SolUnit que propõe a redução do tempo de execução de testes de *smart contract*; Duarte *et al.* (2006), com a ferramenta GridUnit que utiliza a paralelização como parte da sua proposta; Christensen *et al.* (2006), com a ferramenta Unit-Test Framework for Database, destinada a otimizar a execução de testes com banco de dados; Zhao *et al.* (2025) apresentando uma abordagem para acelerar a execução de testes E2E.

3.1 Estoria

A ferramenta Estoria (SILVA; VILAIN, 2017) trabalha com a reutilização de código de configuração dentro da classe de teste. Essa reutilização ocorre através da utilização da notação @Safe, marcando os métodos de teste que podem reutilizar o código de configuração de teste sem gerar dependência entre os métodos de teste e sem causar estados inconsistentes entre as classes.

A ferramenta foi implementada como sendo uma extensão do framework JUnit, em linguagem Java. Sua implementação utiliza do princípio que o testador é o responsável pela identificação de quais métodos do código fonte são seguros e quais métodos não são seguros. Dessa forma, durante a implementação das classes de teste, o testador já deve indicar quais testes são seguros e quais são não seguros utilizando a notação @Safe. Através da utilização da notação durante a implementação dos testes, a extensão criada pelos autores garante que os testes a serem executados primeiros serão os testes seguros, que são aqueles que não irão alterar o SUT, e, em seguida, os testes não seguros, que fazem a alteração do SUT.

Os autores ainda apresentam resultados de experimentos onde a ferramenta Estoria foi aplicada, sendo possível verificar a sua eficácia com uma redução de oito vezes do tempo de execução dos testes automatizados.

Os autores discutem que, embora a abordagem tenha se mostrado eficaz nos experimentos aplicados, sua aplicação pode ser limitada em casos onde as

configurações de teste são altamente específicas ou complexas. Nesses casos, a reutilização de fixtures pode exigir adaptações adicionais para acomodar as particularidades de cada teste. Além disso, a introdução de classes auxiliares de configuração pode aumentar a complexidade inicial do projeto, exigindo um planejamento cuidadoso.

A ferramenta Estoria faz a reutilização da configuração implícita da classe de teste de forma manual através da inclusão da notação `@Safe` por parte do desenvolvedor, o que difere da proposta atual, que irá fazer a reutilização da configuração implícita de forma automatizada.

3.2 SolUnit

SolUnit (MEDEIROS, 2020) é um *framework* para testes de unidade de *smart contracts* implantados na plataforma Ethereum e linguagem de programação Java como uma extensão do *framework* JUnit e tem como objetivo melhorar o desempenho dos testes de smart contracts através da utilização da implantação do contrato e das configurações do teste, sem gerar uma dependência entre os testes de unidade. Para isso, é utilizado o reuso das configurações implícitas de teste e da implantação do contrato de forma automatizada, através de uma análise estática do código fonte.

A proposta é dividida em dois passos, sendo o primeiro passo a identificação dos métodos de testes que não geram novas transações na *blockchain*, considerados métodos seguros, e o segundo passo a reutilização da implantação do contrato e reutilização da execução da configuração de teste implícita para cada método de teste definido como seguro.

No primeiro passo, a identificação dos métodos de teste considerados seguros ocorre através da análise do código fonte de todos os *smart contracts* a serem testados. Nesta análise, são identificados os métodos do contrato que geram novas transações na *blockchain*, chamados de métodos não seguros, e os métodos que não geram novas transações na *blockchain*, chamados de métodos seguros. Em seguida é analisado o código fonte de todas as classes de testes que serão executadas com o objetivo de identificar os testes que chamam algum método não

seguro, classificados como testes não seguros, e os testes que chamam apenas métodos seguros, classificados como testes seguros.

Já no segundo passo, os testes de todas as classes de testes que foram anteriormente classificados em seguros são executados, sendo primeiro executada a implantação do *smart contract* e a configuração de teste implícita uma única vez. Em seguida são executados todos os testes seguros, sem a necessidade de executar a implantação do contrato e a configuração do teste novamente. Após executados todos os métodos de teste seguro, são executados os testes classificados como não seguros, incluindo a implantação do contrato e a configuração implícita de teste para cada teste, exceto para o primeiro teste não seguro que acabará utilizando a implantação do contrato e a configuração utilizados nos testes seguros, uma vez que esses testes não geraram novas transações na *blockchain*.

Para verificar a validade do SolUnit, foram realizados experimentos utilizando 5 projetos diferentes, executando os testes com e sem o *framework* e utilizando uma *blockchain* emulada e uma *blockchain* Ethereum real. Para os testes executados em uma *blockchain* emulada, foi observada uma redução de 65,97% do tempo utilizando o *framework* no primeiro projeto. Já para o segundo projeto foi observado uma redução de 34,71%. O terceiro projeto apresentou uma redução de 52,60%, o quarto projeto apresentou uma redução de 50,35% e o quinto reduziu 68,85%

Já para os testes realizados em uma *blockchain* Ethereum real, o primeiro projeto apresentou uma queda de 65,08%. O segundo projeto teve uma queda de 46,07%. O terceiro projeto teve uma redução de 68,17% do tempo, o quarto projeto reduziu 49,99% e por fim o quinto projeto reduziu em 38,08% do tempo.

O SolUnit utiliza a mesma abordagem à qual esse trabalho tem como base, porém sua aplicação ocorre somente em *smart contracts* dentro de uma *blockchain*, diferindo da proposta deste trabalho que tem por objetivo reduzir o tempo de execução de testes automatizados de projetos não desenvolvidos em *blockchain*.

3.3 GridUnit

O GridUnit (DUARTE; *et al*, 2006) é uma extensão do *framework* JUnit capaz de distribuir automaticamente a execução de testes automatizados em um *grid*

computacional. Sua ideia central é realizar a paralelização da execução dos testes automatizados e, assim, reduzir o tempo necessário para concluí-los. Seu projeto se dá de forma a operar com o mínimo de intervenção manual do usuário, o que facilitaria sua integração em fluxos de trabalho já existentes.

Em experimentos realizados pelos autores, foi possível constatar uma redução no tempo de execução de teste em até 70 vezes, destacando o potencial para otimizar o tempo despendido em testes pela ferramenta. Além do mais, os autores destacam que o usuário não precisa do conhecimento aprofundado sobre computação distribuída para se beneficiar da ferramenta, uma vez que a ferramenta não exige modificações no código fonte da aplicação que está sendo testada.

Contudo, vale destacar que a ferramenta citada explora o paralelismo para reduzir o tempo despendido em testes automatizados, sendo necessário então ter uma infraestrutura que suporte a execução múltiplas de testes, premissa essa diferente da proposta apresentada na ferramenta neste trabalho, que não explora o paralelismo para execução dos testes automatizados.

3.4 Unit-Test Framework for Database

Unit-Test Framework for Database (CHRISTENSEN; *et al.* 2006) aborda a complexidade sobre teste desenvolvidos em aplicações que interagem com banco de dados, pois com a contínua mudança das regras de negócio, a estrutura do banco de dados da aplicação acaba sendo constantemente atualizada e, em alguns cenários específicos, até alterada. Sendo assim, a ferramenta acaba focando nos testes automatizados do código fonte da aplicação que acessa o banco de dados. A principal dificuldade está no fato de que o resultado de um teste que armazena dados em um banco de dados é intrinsecamente ligado ao estado desse banco de dados, ou seja, o estado inicial do banco de dados acaba influenciando diretamente no resultado de um teste e o estado do banco ao final do teste pode afetar os resultados dos demais testes a serem executados em sequência.

Com isso em mente, a abordagem proposta pela ferramenta permite que testes unitários reutilizem dados que foram inseridos por outros testes de maneira estruturada, o que acaba reduzindo a necessidade de recriar o estado do banco de dados a cada execução de teste. Na extensão são declaradas dependências entre

as classes de teste, e essa dependência deve refletir a mesma dependência existente entre as entidades do banco de dados. Assim, os testes das dependências de determinada classe são executados de forma recursiva antes da execução do teste da própria classe. Também deve ser configurada uma classe de teste uma para cada tabela do banco de dados que será testado e também será necessário definir teste para as tabelas que forem chaves estrangeiras das tabelas testadas.

A ferramenta desenvolvida foi aplicada em um projeto de médio porte, obtendo resultados satisfatórios, onde houve um aumento de 40% na velocidade de execução dos testes e uma redução estimada de 25% no número de linhas de código de teste desenvolvidas.

A proposta dos autores de Unit-Test Framework for Database tem como foco exclusivo na diminuição do tempo de execução de testes automatizados de sistemas que possuem interação com banco de dados, apresentando assim características específicas desse tipo de teste, o que não é abordado na proposta deste atual trabalho.

3.5 E2E test execution optimization for web application based on state reuse

O trabalho de Zhao *et al.* (2025) aborda a otimização do processo de execução de testes E2E (teste de ponta a ponta) para aplicações *web*. Os autores visam aprimorar a eficiência por meio de uma abordagem que foca na reutilização de estados do sistema.

O trabalho descreve um *framework* para otimização se baseando na reutilização de estados, sendo esta abordagem crucial para reduzir o tempo de execução dos testes E2E, testes esses que podem ser demorados e custosos, especialmente em aplicações *web* complexas onde o estado da aplicação pode mudar significativamente entre os testes.

O trabalho explora a representação de *scripts* de teste utilizando uma estrutura de árvore de prefixo comum para organizar as mesmas operações entre os scripts, sendo essa representação fundamental para a técnica de reutilização de estados, permitindo que o *framework* identifique e gerencie os pontos onde o estado

pode ser reutilizado entre diferentes cenários de teste, e assim evitar a necessidade de reconfigurar o ambiente para cada execução. Sob a orientação da árvore de prefixos comum, o estado reutilizável ideal é identificado e duplicado para maximizar a utilização dos estados da página acionados pelas mesmas operações, reduzindo assim o tempo geral de execução do conjunto de testes.

Um aspecto importante investigado pelos autores é a relação entre a proporção de operações reutilizadas e não reutilizadas e a taxa de aceleração na execução dos testes. Essa análise quantifica o impacto da reutilização de estado na performance geral do processo de teste, demonstrando como a maior proporção de reutilização pode levar a ganhos significativos de tempo.

Os autores ainda sugerem implicitamente que as otimizações propostas não se concentram na redução do tempo de execução dos casos de teste individuais em si, mas sim na eficiência do processo de execução da suíte completa de testes através da eliminação de setups redundantes.

Para verificar a eficácia do método proposto, foram realizados experimentos em 347 testes E2E de oito aplicativos *web* de código aberto, apresentando uma redução de tempo de execução dos testes entre 52% e 68%.

O foco do trabalho *E2E test execution optimization for web application based on state reuse* está exclusivamente em teste E2E, ou seja, testes que visam verificar se as funcionalidades do sistema estão de acordo com os requisitos especificados para o seu funcionamento, ocorrendo em fases mais posteriores do desenvolvimentos. Sendo assim, acaba diferindo da proposta deste trabalho que tem como objetivo o foco em testes unitários, que ocorre nas partes iniciais do desenvolvimento.

4 PROPOSTA

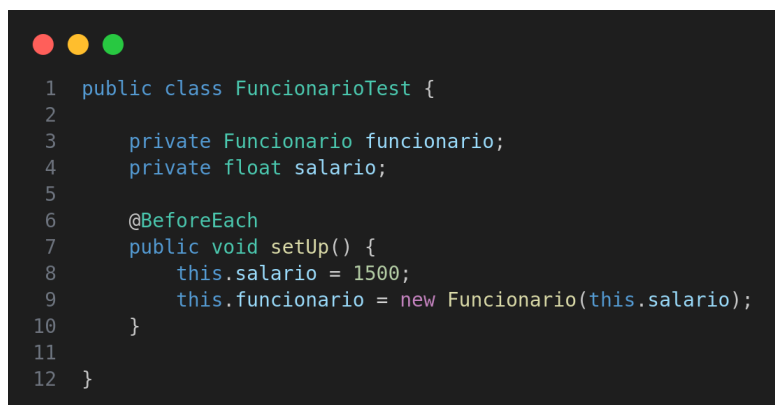
O presente capítulo tem por objetivo detalhar a proposta e as etapas de implementação do trabalho.

4.1 - Visão geral da proposta

O objetivo geral deste trabalho é a redução do tempo de execução de testes automatizados de projetos desenvolvidos em linguagem de programação Java e com testes automatizados utilizando o *framework* de teste JUnit 5, atual versão do *framework* de teste baseado em linguagem Java 8 ou superior (JUNIT, 2025). A proposta parte do princípio de que cada método de teste, dentro de uma classe de teste, é executado de forma independente e não de forma sequencial, possibilitando alterar a ordem de execução dos métodos de teste dentro de cada classe.

A proposta consiste na construção de uma ferramenta de extensão do JUnit, que reusa a execução do setup inicial dentro de cada classe de teste, comumente representado pela notação `@BeforeEach`. A Figura 10 ilustra a configuração implícita de teste dentro de uma classe de teste, seguido da notação `@BeforeEach`, que deve ser executada antes do início de cada método de teste.

Figura 10 - Configuração implícita de teste



```
1 public class FuncionarioTest {
2
3     private Funcionario funcionario;
4     private float salario;
5
6     @BeforeEach
7     public void setUp() {
8         this.salario = 1500;
9         this.funcionario = new Funcionario(this.salario);
10    }
11
12 }
```

Fonte: Elaborado pelo autor.

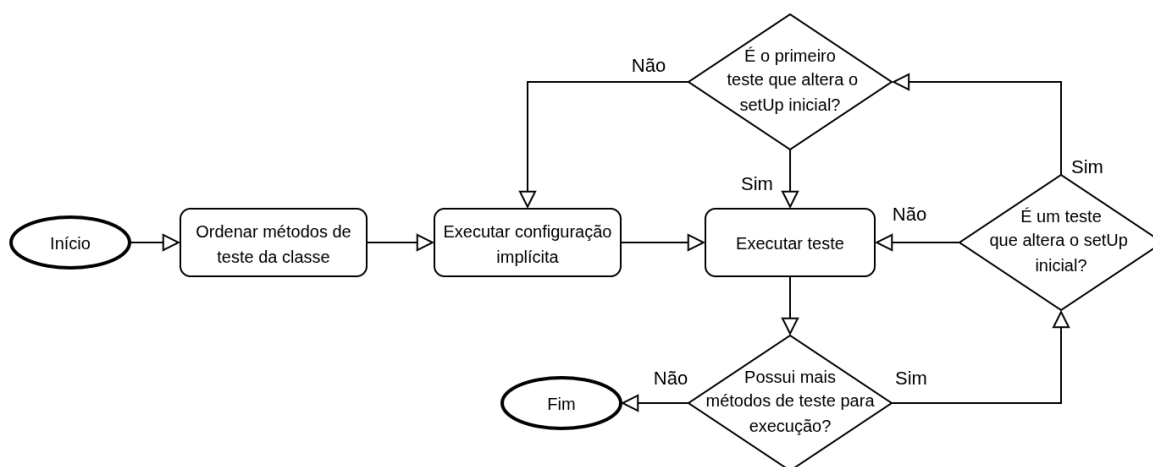
O reuso das configurações implícitas que são executadas antes de cada método de teste de cada classe de teste é possível quando um método de teste não faz alteração dos atributos da classe de teste, possibilitando a redução do tempo de

execução da classe de teste. Contudo, uma vez que um teste altere o estado dessas variáveis de classe, a reutilização dessa configuração torna-se inviável, já que fere o princípio da independência de testes (MESZAROS; SMITH; ANDREA, 2003).

Na Figura 10 é possível identificar as variáveis de classe “funcionario” e “salario”, que são inicializadas com o valor inicial dentro da configuração implícita de teste através do método *setUp*. Essa configuração implícita é executada antes do início de cada método de teste, mas caso o primeiro método de teste não faça alteração de nenhuma dessas duas variáveis, a configuração implícita não precisa ser executada novamente antes do início do próximo método de teste. Por outro lado, caso algumas dessas duas variáveis tenham seu valor alterado, a configuração implícita precisa ser executada antes do próximo método de teste ser executado.

Com isso, a execução da configuração implícita da classe de teste precisa ser executada uma vez e logo em seguida devem ser executados todos os métodos de teste que não fazem a alteração dos valores das variáveis contidas dentro desse *setUp*. Com a finalização da execução de todos os métodos de teste que não alteram as variáveis do *setUp* inicial, inicia-se a execução do primeiro método de teste que faz alguma alteração nas variáveis dentro do *setUp*, seguindo da execução do *setUp*, seguido do próximo método de teste que faz alguma alteração das variáveis do *setUp* e assim até a finalização de todos os métodos de teste. A Figura 11 ilustra o fluxograma da reutilização da configuração implícita proposta.

Figura 11 - Fluxograma de reutilização da configuração implícita



Fonte: Elaborado pelo autor.

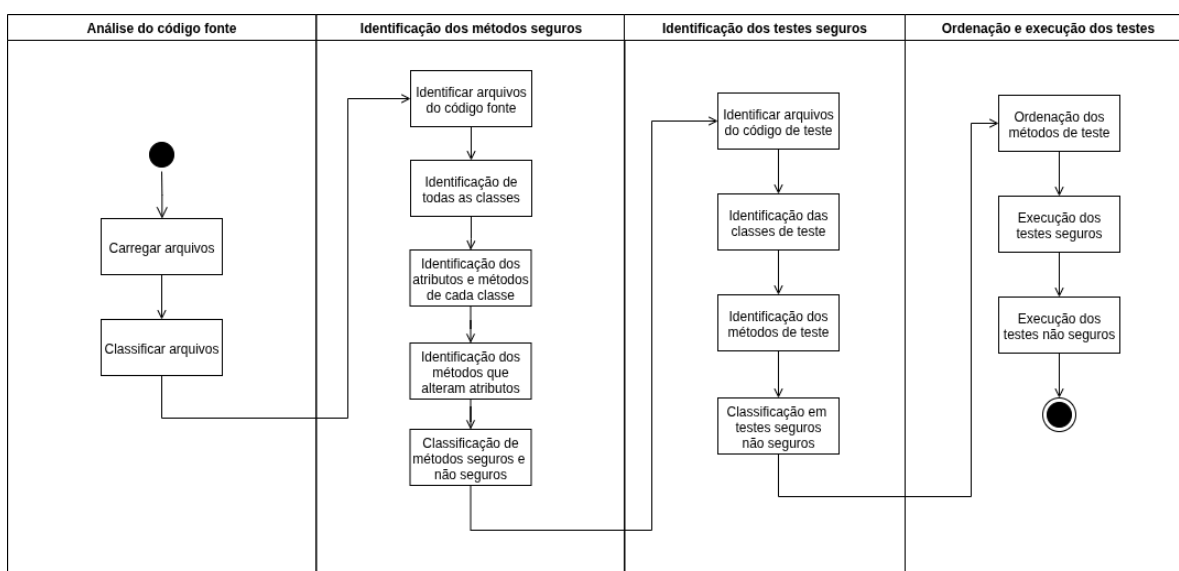
A proposta deste trabalho compreende a identificação desses métodos que alteram o valor do *setUp* de forma automática e com isso ordenar a execução dos

métodos de testes de cada classe de teste, e assim reduzir o tempo de execução dos testes de um projeto.

4.2 - Etapas da Proposta

A ferramenta proposta realiza quatro etapas durante a sua execução conforme ilustrado na Figura 12: (1) análise do código fonte do projeto; (2) identificação dos métodos que alteram o estado dos atributos dentro do projeto; (3) classificação dos métodos de testes em seguros e não seguros; e (4) ordenação e execução dos métodos de testes.

Figura 12 - Diagrama de atividades da proposta



Fonte: Elaborado pelo autor.

4.2.1 Análise do código fonte

A primeira etapa tem por objetivo carregar todos os arquivos do código fonte do projeto e identificar quais arquivos compõem o código principal do projeto e quais arquivos compõem o código de teste do projeto.

4.2.2 Identificação dos métodos seguros

A segunda etapa busca identificar todos os atributos e métodos presentes em cada classe do código principal do projeto. Estes métodos são classificados como métodos seguros e métodos não seguros. Os **métodos seguros** são aqueles que não irão alterar os atributos das classes sendo testadas, enquanto os métodos **não seguros** são aqueles que podem realizar alguma alteração nos atributos destas classes. A ideia de classificação de métodos em seguros e não seguros foi proposta por Silva e Vilain (2021)

A Figura 13 ajuda a ilustrar como os métodos não seguros são identificados. A classe *Funcionario* possui um atributo chamado *salario* e os métodos *getSalario* e *setSalario*, onde o método *getSalario* retorna o valor atual do *salario* do *Funcionario*, e assim sendo classificado como um método seguro, enquanto o método *setSalario* altera o valor atual do *salario* do *funcionario*, logo sendo classificado como um método não seguro.

Figura 13 - Exemplo da classe Funcionario

The image shows a code editor window with a dark background and syntax-highlighted Java code. The code defines a class named 'Funcionario' with a private attribute 'salario' of type 'float'. It includes a constructor 'Funcionario(float salario)' that initializes 'this.salario' with the passed value. There are two public methods: 'getSalario()' which returns 'this.salario', and 'setSalario(float novoSalario)' which assigns 'novoSalario' to 'this.salario'. The code is numbered from 1 to 17 on the left side of the editor.

```
1  public class Funcionario {
2
3      private float salario;
4
5      public Funcionario (float salario) {
6          this.salario = salario;
7      }
8
9      public float getSalario() {
10         return this.salario;
11     }
12
13     public void setSalario(float novoSalario) {
14         this.salario = novoSalario;
15     }
16
17 }
```

Fonte: Elaborado pelo autor.

4.2.3 Identificação dos testes seguros

Na terceira etapa é realizada a análise estática do código de teste do projeto. O objetivo desta etapa é localizar e classificar todos os métodos de teste, presentes em todas as classes de teste. Estes métodos de teste são representados pela notação *@Test* e estes métodos são classificados em métodos de teste seguros e não seguros. Essa classificação é realizada com base nos métodos seguros e não seguros identificados na segunda etapa da ferramenta. Um **método de teste seguro** apresenta em seu código a utilização de apenas métodos seguros do código principal, conforme classificado na segunda etapa da ferramenta (Seção 4.2.2). Caso o método de teste apresente em seu código um método não seguro, conforme explicado na segunda etapa da ferramenta (Seção 4.2.2), esse método de teste é classificado como um **método de teste não seguro**.

A Figura 14 ilustra essa classificação de métodos de teste seguros, onde a classe de teste *FuncionarioTest* apresenta dois métodos de teste. O método *deveRetornarSalarioCorretoDoFuncionario* utiliza apenas o método seguro *getSalario*, logo esse método de teste é classificado como seguro pois não altera o estado atual do SUT. Por outro lado, o método *deveAlterarCorretamenteSalarioFuncionario* utiliza o método seguro *getSalario* e também o método não seguro *setSalario*, sendo classificado, então, como não seguro, pois altera o estado do SUT.

4.2.4 Ordenação e execução dos métodos de testes

A última etapa da ferramenta ordena a execução dos métodos de teste dentro de cada classe de teste, executando primeiro os métodos de teste seguros e em seguida os métodos de teste não seguros, conforme classificados na terceira etapa da ferramenta (Seção 4.2.3). É nesta etapa que ocorre o reuso das configurações implícitas de cada classe de teste.

A execução dos métodos de teste ocorre na seguinte sequência:

1. Primeiro ocorre uma execução do *setUp* inicial da classe de teste e, em seguida, são executados todos os métodos de teste seguros desta classe de teste.

Figura 14 - Exemplo da classe FuncionarioTest

```
1  @ExtendWith(SunitRunner.class)
2  @TestMethodOrder(SunitRunner.class)
3  public class FuncionarioTest {
4
5      private static Funcionario funcionario;
6      private static float salario;
7
8      @BeforeEach
9      public void setUp() {
10         this.salario = 1500;
11         this.funcionario = new Funcionario(this.salario);
12     }
13
14     @Test
15     public void deveRetornarSalarioCorretoDoFuncionario() {
16         Assertions.assertEquals(this.funcionario.getSalario(), this.salario);
17     }
18
19     @Test
20     public void deveAlterarCorretamenteSalarioFuncionario() {
21         float novoSalario = 2000;
22         this.funcionario.setSalario(novoSalario);
23         Assertions.assertEquals(this.funcionario.getSalario(), novoSalario);
24     }
25 }
26 }
```

Fonte: Elaborado pelo autor.

2. Em seguida é executado o primeiro método de teste não seguro; para esta execução ainda não é necessário executar novamente o *setUp* inicial, uma vez que o SUT ainda não foi alterado.
3. Após a execução do primeiro teste não seguro, é executado o *setUp* inicial da classe de teste novamente e o próximo método de teste não seguro. Esta sequência é repetida até que todos os testes não seguros sejam executados.

Desta forma, a presente proposta permite a diminuição do tempo de execução dos testes automatizados por não executar sempre o *setUp* inicial de cada classe de teste antes de cada método de teste, uma vez que testes não seguros não alteram o estado inicial da configuração implícita de teste.

5 IMPLEMENTAÇÃO

O presente capítulo tem por objetivo detalhar o desenvolvimento da ferramenta proposta por este trabalho, o SUnit³. A ferramenta foi desenvolvida como uma extensão do *framework* de automação de testes JUnit 5, podendo ser encontrada no repositório do GitHub (<https://github.com/engdough/tcc-gilmar>).

Vale destacar que o SUnit foi adaptado do *framework* SolUnit (MEDEIROS, 2020), que é um *framework* que estende o JUnit 4 e tem por finalidade a redução do tempo de execução de testes de *smart contracts* em *blockchains*. Foi reaproveitada a estrutura em geral do projeto, como nome de classes, arquitetura dos arquivos, utilização da biblioteca do JavaParser e a estratégia para leitura dos arquivos e análise estática do código. O SolUnit utilizava a versão 4 do JUnit e para o SUnit foi realizado o *upgrade* de versão, e utilizado a versão 5 do *framework*.

O SUnit foi desenvolvido na linguagem de programação Java e projetado para atuar na redução do tempo de execução de testes unitários. Para estender o JUnit, a ferramenta utiliza as notações básicas do *framework* durante o ciclo de vida da execução de cada teste: `@Test` indicando todos os métodos de teste, `@BeforeEach` indicando as configurações implícitas que devem ser executadas antes da execução de cada método de teste, `@ExtendWith` indicando que a classe de teste irá estender o SUnit, e `@TestMethodOrder` indicando que a ordenação dos métodos de teste irá seguir a ordenação proposta pelo SUnit.

Além disso, apesar da ferramenta realizar a identificação automática de testes seguros, manteve-se a notação `@Safe` apresentada por Silva e Vilain(2017) e adotada por Medeiros (2020) no SolUnit, que classifica manualmente o método de teste em seguro e pode realizar o reuso das configurações implícitas de teste sem comprometer a execução de todos os testes dentro da classe. Vale ressaltar que o sistema não faz o uso automático da notação e cabe ao desenvolvedor aplicar seu uso, caso ele tenha absoluta certeza que o método de teste em si é seguro e pode fazer o reuso das configurações implícitas de teste.

³ O termo SUnit é uma abreviação de SetupUnit.

5.1 Diagrama de Classes

A Figura 15 apresenta o diagrama de classes do SUnit. A classe *SunitRunner* é responsável por estender o framework JUnit, pela execução dos testes, por instanciar a classe *SafeParser*, por interceptar a ordenação dos métodos de teste através da interface *MethodOrderer*, e também por interceptar a interface *InvocationIntereptor* que executa todas as configurações implícitas de teste através do uso da notação *@BeforeEach* e decidir se haverá execução das configurações implícitas de cada teste.

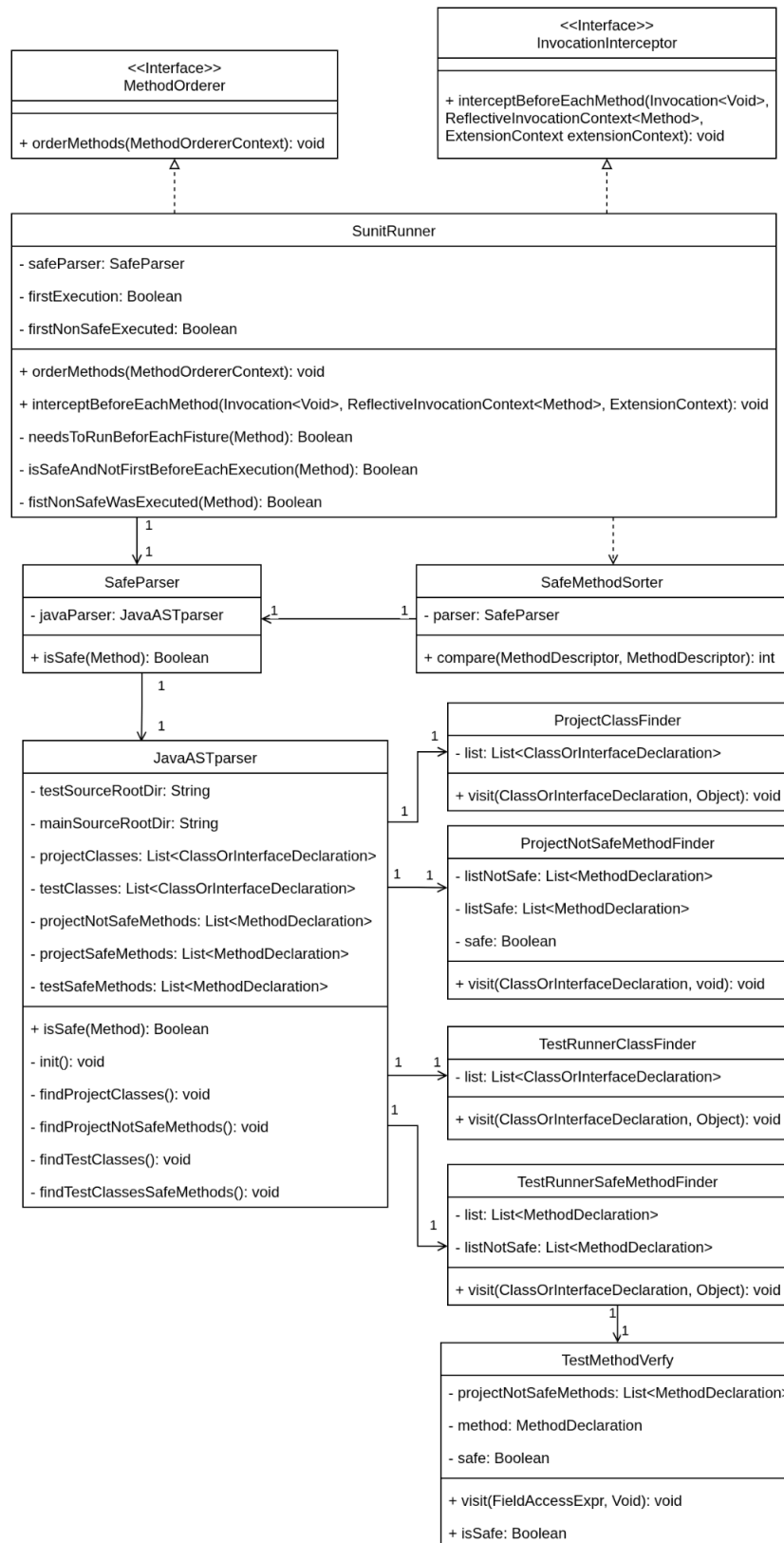
A classe *SafeParser* consome a classe *JavaASTparser* instanciada anteriormente, além de ser utilizada pela classe *SafeMethodSorter*. A classe *SafeParser* é a responsável por injetar o endereço do diretório da classe principal e do diretório de teste do projeto em uso. Já a classe *SafeMethodSorter* é a classe responsável pela ordenação dos métodos de testes, onde os métodos de testes seguros são executados primeiro, seguidos dos métodos de teste classificados como não seguros.

Por fim, a classe *JavaASTparser* é a responsável por fazer a leitura estática dos arquivos do código principal através da classe *ProjectClassFinder*, leitura estática do código de teste através da classe *TestRunnerClassFinder*, da classificação dos métodos das classes em seguros e não seguros através da classe *ProjectNotSafeMethodFinder* e da classificação dos métodos de teste em seguros e não seguros através da classe *TestRunnerSafeMethodFinder*.

5.2 - Análise do código fonte

Conforme indicado no Capítulo 4 da proposta, a primeira etapa da implementação tem por objetivo identificar os arquivos do código fonte do projeto a ser testado. Essa etapa começa a partir da injeção dos endereços dos diretórios da classe principal e da classe de teste do projeto pela classe *SafeParser*. Estes endereços são obtidos por meio de um arquivo de configuração denominado “*sunit.properties*”. A Figura 16 ilustra um exemplo do conteúdo do arquivo de configuração, onde na primeira linha de código é informado o endereço do diretório

Figura 15 - Diagrama de classes do SUnit



Fonte: Elaborado pelo autor.

dos arquivos do código principal do projeto e na segunda linha é informado o endereço do diretório dos arquivos do código de teste.

Figura 16 - Arquivo de configuração “*sunit.properties*”



```
1  main.src.dir=src/main/java
2  test.src.dir=src/test/java
```

Fonte: Elaborado pelo autor.

É importante que os endereços dos diretórios de teste e principal informados no arquivo de configuração estejam corretos, pois a partir disso a classe *SafeParser* consegue obter todos os arquivos Java presentes nos diretórios informados e ler todo o código presente dentro desses arquivos. A partir desse ponto começa a identificação dos métodos das classes e métodos de testes em seguros e não seguros.

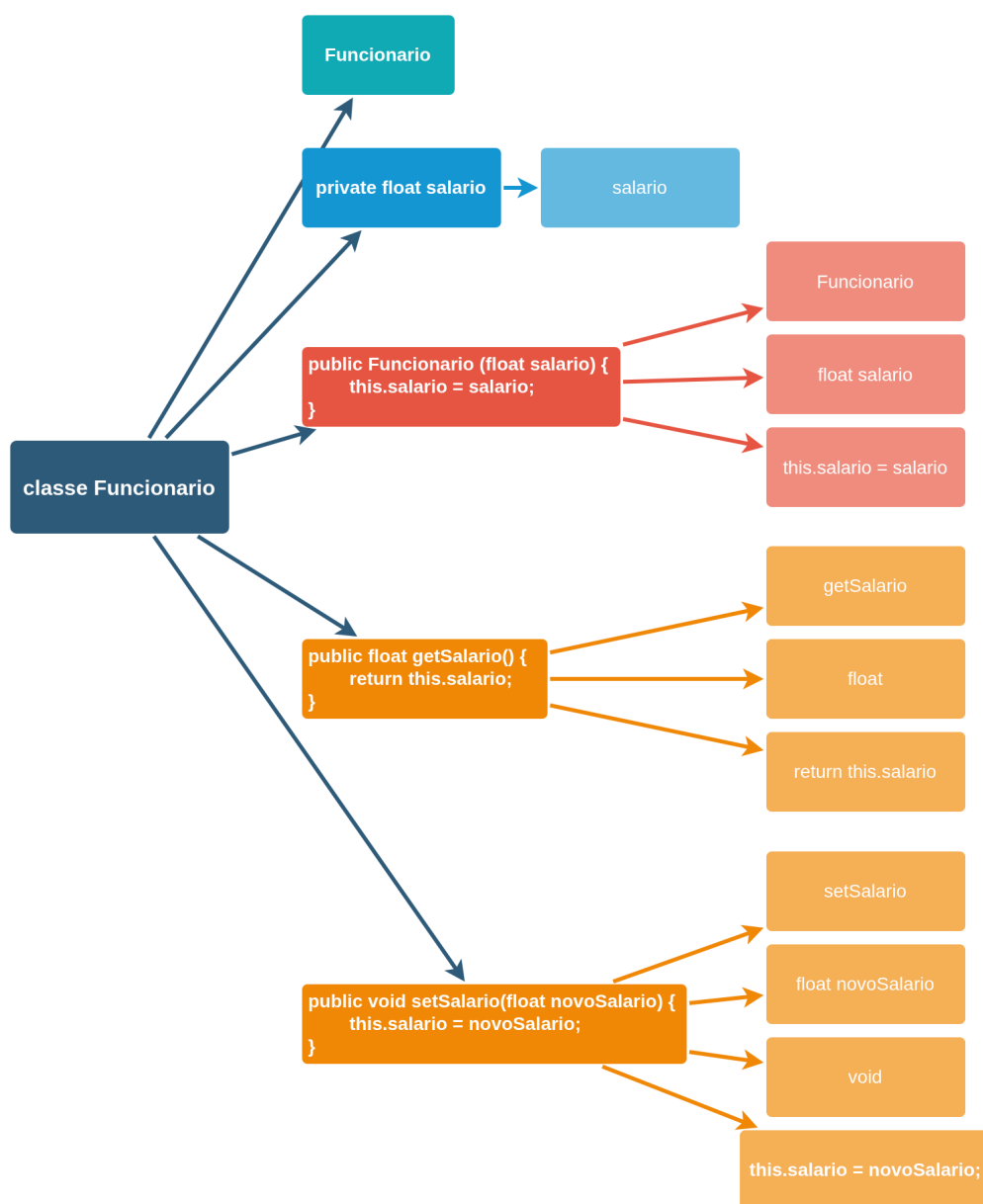
5.3 - Identificação dos métodos das classes seguros e não seguros

Nesta etapa é realizada a análise do código principal do projeto, onde se tem por finalidade a identificação dos métodos de cada classe que fazem alteração do valor dos atributos das classes a que pertencem, conforme mencionado anteriormente. Para fazer a análise estática foi utilizada a biblioteca de código aberto *JavaParser*, que possibilita analisar código em linguagem de programação Java em uma árvore de sintaxe abstrata (COX, 2024).

Primeiramente são identificados todos os arquivos localizados no diretório do código principal do projeto que foi indicado dentro do arquivo de configuração. Em seguida é feita uma filtragem desses arquivos, onde são utilizados apenas os arquivos que possuam a extensão .java. Por fim, dos arquivos filtrados, são selecionados apenas aqueles que representam uma classe ou interface em Java para ser realizada a análise e classificação dos métodos em seguros e não seguros. Essa identificação dos arquivos da classe principal é realizada pelo método *findProjectClasses* da classe *JavaASTparser*.

Após carregar os arquivos da classe principal do projeto é feita a classificação dos métodos de todas as classes carregadas em métodos seguros e não seguros. A classificação dos métodos é possível com o auxílio da biblioteca *JavaParser*, que transforma todos os arquivos da classe principal que foram selecionados em uma árvore de sintaxe abstrata. Um exemplo desta árvore é visível na Figura 17, que representa a árvore de sintaxe abstrata gerada a partir do exemplo da classe *Funcionario* mostrada na Figura 13. O método *findProjectNotSafeMethods* da classe *JavaASTparser* é o responsável por realizar essa classificação.

Figura 17 - Árvore de sintaxe abstrata da classe *Funcionario*



Fonte: Elaborado pelo autor.

Ainda na Figura 17, é possível verificar que a árvore gerada da classe *Funcionario* gerou 5 nodos, sendo o primeiro representando o nome da classe, o segundo representando o atributo da classe, o terceiro indicando o construtor da classe e os dois últimos folhas representando os métodos *getter* e *setter* da classe. Cada nodo gerado pela árvore da classe também pode gerar folhas filhas de acordo com suas características e cada classe pode gerar inúmeros nodos, de acordo com o número de atributos, construtores e métodos presentes nela.

A partir da árvore formada, os métodos são classificados em seguros ou não seguros verificando se nos nodos métodos existe alguma função que altera o valor do nodo atributo; caso haja alteração do valor do nodo atributo o método é considerado não seguro. A Figura 18 ilustra o trecho de código da classe responsável por realizar a classificação dos métodos em seguros e não seguros. A classe *ProjectNotSafeMethodFinder* presente na Figura 18 é chamada pelo método *findProjectNotSafeMethods* da classe *JavaASTparser*.

Na linha 13 da Figura 18 pode-se verificar que, a partir da entrada de uma classe ou interface Java, é possível obter os métodos e atributos presentes na classe através dos métodos *getMethods* e *getFields*, respectivamente, ambos da biblioteca *JavaParser*. A verificação ocorre na linha 19, onde é verificado se o método possui em seu corpo de texto o nome do atributo seguido do símbolo de igualdade.

Contudo, além de verificar se os métodos alteram os atributos da própria classe, é necessário verificar se os métodos não utilizam métodos de outras classes que foram considerados não seguros. Na Figura 18 os métodos declarados como não seguros anteriormente são armazenados na lista *listNotSafe* e através da iteração da linha 25 é verificado se o método em análise não utiliza em seu corpo nenhum outro método da lista *listNotSafe*. Portanto, um método é considerado não seguro se ele altera o valor do atributo da própria classe e/ou utiliza métodos de outras classes considerados não seguros.

Por fim, caso o método sendo analisado seja não seguro, é necessário analisar se os métodos anteriores já classificados como seguros não utilizam o método em questão; caso utilizem, os mesmos passam a ser classificados como métodos não seguros. Esta verificação é realizada na linha 32 da Figura 18, onde os métodos declarados como seguros estão presentes na lista *listSafe*.

Figura 18 - Classificação dos métodos de classe

```

1  public class ProjectNotSafeMethodFinder extends VoidVisitorAdapter<Void> {
2
3      List<MethodDeclaration> listNotSafe;
4      List<MethodDeclaration> listSafe;
5      boolean safe;
6
7      public ProjectNotSafeMethodFinder(List<MethodDeclaration> listNotSafe, List<MethodDeclaration> listSafe) {
8          this.listNotSafe = listNotSafe;
9          this.listSafe = listSafe;
10     }
11
12     @Override
13     public void visit(ClassOrInterfaceDeclaration classe, Void arg) {
14         super.visit(classe, arg);
15         this.safe = true;
16
17         classe.getMethods().stream().forEach(method -> {
18             classe.getFields().stream().forEach(field -> {
19                 if (method.toString().contains(field.getChildNodes().get(0).toString() + "=")) {
20                     this.safe = false;
21                 }
22             });
23
24             this.listNotSafe.stream().forEach(methodNotSafe -> {
25                 if (method.toString().contains(".") + methodNotSafe.getNameAsString())) {
26                     this.safe = false;
27                 }
28             });
29
30             if (!safe) {
31                 this.listSafe.stream().forEach(methodSafe -> {
32                     if (methodSafe.toString().contains(".") + method.getNameAsString())) {
33                         this.listNotSafe.add(methodSafe);
34                     }
35                 });
36                 this.listNotSafe.add(method);
37             } else {
38                 this.listSafe.add(method);
39             }
40
41             this.safe = true;
42         });
43     }
44 }

```

Fonte: Elaborado pelo autor.

Os métodos que não utilizam nenhum método previamente classificado como não seguro, são classificados como seguros e adicionados na lista *listSafe*. Contudo, caso o método utilize algum método que ainda não tenha sido classificado e posteriormente ele venha a ser classificado como não seguro, esse método já adicionado à lista de métodos seguros vai ser reclassificado como não seguro. Isso é garantido através da iteração que ocorre na linha 31 da Figura 18. Outro fato importante está na utilização de métodos proveniente de APIs externas, *frameworks*, extensões e bibliotecas externas, nesse caso a ferramenta não consegue garantir que esses métodos externos não causam a alteração do estado das variáveis de classe. Nesse caso, o métodos que utilizam esses métodos que não podem ser

classificados em seguros e não seguros deverão ser classificados como não seguros impreterivelmente e caso o desenvolvedor tenha segurança em classificar esse método como seguro, o método de teste que utilize esse método pode ser classificado manualmente como seguro através da notação *@Safe*.

5.4 - Identificação dos testes seguros e não seguros

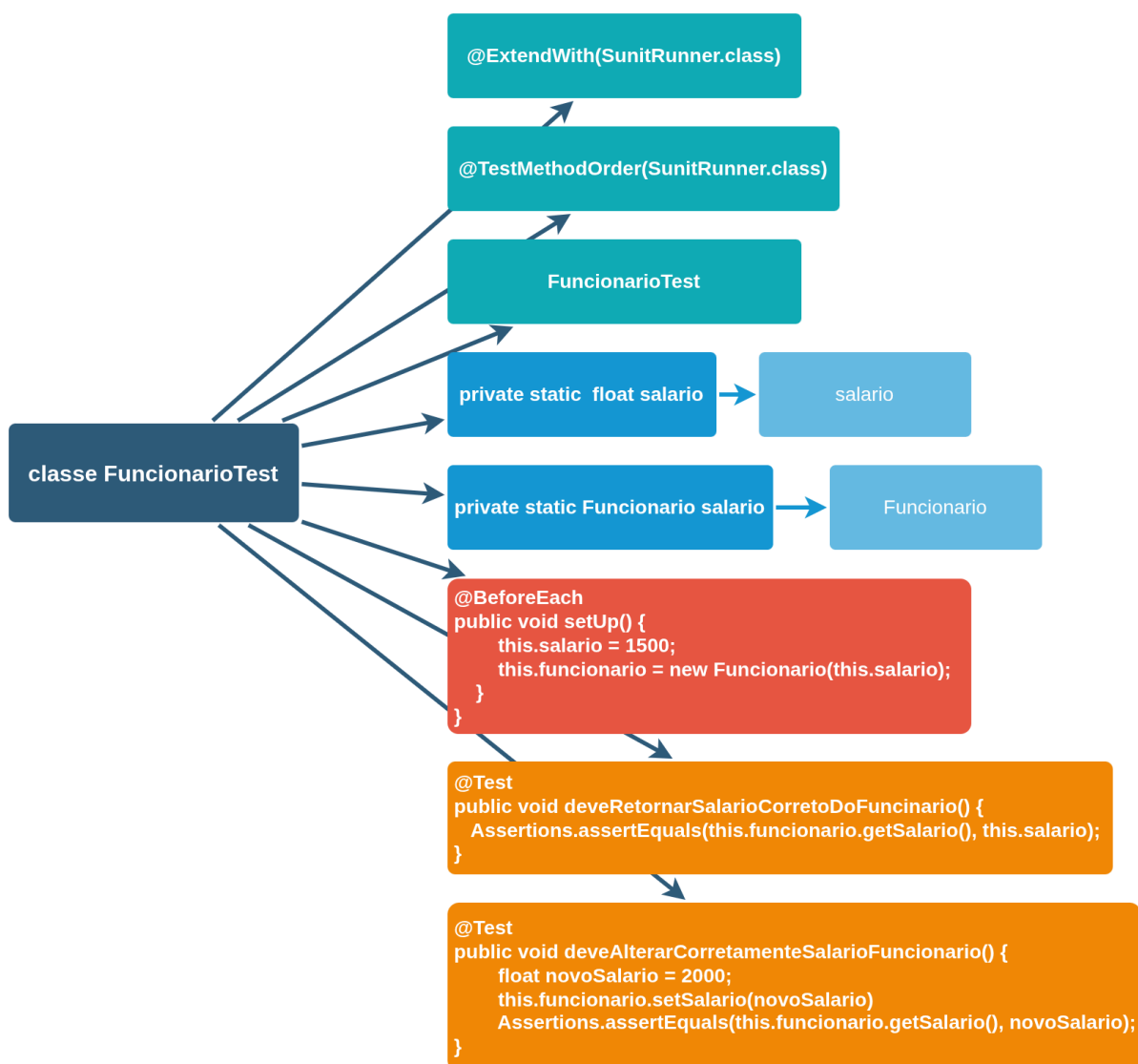
Analogamente à etapa anterior, a classificação dos testes em seguros e não seguros ocorre através da análise do código de teste do projeto. São identificados todos os arquivos localizados no diretório do código de teste do projeto indicado no arquivo de configuração. Novamente é feita a filtragem desses arquivos, onde são verificados apenas os arquivos que possuam a extensão *.java*. Por fim, dos arquivos filtrados, são selecionados apenas aqueles que apresentam em seu código a notação *@ExtendWith("SunitRunner.class")*, pois são as classes que utilizarão a ferramenta deste trabalho e que terão seus testes classificados como testes seguros ou testes não seguros. Essa identificação dos arquivos da classe principal é realizada pelo método *findTestClasses* da classe *JavaASTparser*.

Após todos os arquivos de teste serem carregados é iniciada a classificação dos métodos de teste. É utilizada novamente a biblioteca *JavaParser* que transforma todos os arquivos em árvores de sintaxe abstrata. A Figura 19 apresenta a árvore de sintaxe abstrata da classe *FuncionarioTest* gerada a partir do exemplo da Figura 14. Na Figura 19 é possível ver que a classe de teste *FuncionarioTest* gerou 8 nodos, sendo os dois primeiros representando as notações utilizadas na implementação deste trabalho, o terceiro representando o nome da classe de teste, o quarto e o quinto nodos representando os atributos utilizados nesta classe de teste, o sexto nodo representa a configuração implícita de teste, e os dois últimos representando os métodos de teste implementados dentro da classe. O método *findTestClassesSafeMethods* da classe *JavaASTparser* é o responsável por realizar essa classificação.

A partir da árvore formada, os métodos de teste são classificados em seguros ou não seguros através da verificação se nas folhas com os métodos de teste existe a utilização de ao menos um dos métodos classificado como não seguro, conforme explicado na Seção 5.3. Assim, caso tenha ao menos um método não seguro, o

teste é classificado como não seguro, caso contrário, o método de teste é classificado como seguro. Essa classificação pode ser visualizada no trecho de código da Figura 20. A classe *TestMethodVerify* presente na Figura 20 é chamada pelo método *findTestClassesSafeMethods* da classe *JavaASTparser*.

Figura 19 - Árvore de sintaxe abstrata da classe *FuncionarioTest*



Fonte: Elaborado pelo autor.

A classificação dos métodos de teste em seguros e não seguros é feita para cada método de teste e ocorre na classe *TestMethodVerify* através da verificação se o método de teste possui algum método não seguro, conforme classificado na etapa 5.4. Na Figura 20, os métodos não seguros estão armazenados na lista *projectNotSafeMethods* e a verificação se o método de teste possui algum desses

métodos ocorre na linha 27. Contudo, o método ainda verifica se as variáveis presentes na classe de teste e que compõem o SUT a ser testado também não são modificadas. As variáveis são verificadas de forma iterativa através da linha 21, onde todas as variáveis da classe de teste são extraídas pelo método *findCompilationUnit*, da biblioteca do *JavaParser*, e a verificação se ocorreu alteração nos seus valores realizada na linha 28.

Figura 20 - Classificação dos métodos de teste

```

1  class TestMethodVerify extends VoidVisitorAdapter<Void> {
2
3      List<MethodDeclaration> projectNotSafeMethods;
4      MethodDeclaration md;
5      boolean projectAccess;
6      boolean safe;
7
8      public TestMethodVerify(MethodDeclaration md) {
9          this.md = md;
10         this.safe = true;
11     }
12
13     public void setProjectNotSafeMethods(List<MethodDeclaration> projectNotSafeMethods) {
14         this.projectNotSafeMethods = projectNotSafeMethods;
15     }
16
17     @Override
18     public void visit(FieldAccessExpr field, Void arg) {
19         super.visit(field, arg);
20
21         field.findCompilationUnit().get().findAll(FieldDeclaration.class)
22             .stream().forEach(a -> {
23             if (a.isStatic()) {
24                 String fieldName = a.getVariable(0).getNameAsString();
25                 String testBody = md.getBody().get().toString();
26                 if (this.projectNotSafeMethods.stream().forEach(b -> {
27                     if (testBody.contains(fieldName + " = ") ||
28                         testBody.contains(fieldName + " + " + b.getNameAsString())) {
29                         this.safe = false;
30                     }
31                 }));
32             }
33         });
34     }
35
36     public boolean isSafe() {
37         return safe;
38     }
39
40 }

```

Fonte: Elaborado pelo autor.

5.5 - Ordenação e execução dos métodos de teste

Com todos os métodos de teste classificados em seguros e não seguros, dá-se início à etapa de ordenação e execução dos métodos de teste. Esta etapa é possível através da utilização da notação `@TestMethodOrder` que possibilita interceptar a ordenação padrão do JUnit 5 e implementar a ordenação desejada. No SUnit, a ordenação ocorre na classe *SafeMethodSorter*, que ordena os métodos de teste de forma a serem executados primeiro os métodos de teste seguros e em seguida os métodos de teste não seguros. A ordenação dos métodos de teste ocorre isoladamente em cada classe de teste.

A Figura 21 exibe o trecho de código onde ocorre a interceptação do método de ordenação do JUnit 5 e o método que realiza a ordenação dos métodos de testes. A linha 1 da Figura 21 exibe a interceptação do método *orderMethods* do JUnit, onde o método é sobrescrito e ocorre a ordenação através da classe *SafeMethodSorter()*. Já a linha 7 da Figura 21 exibe a classe de ordenação em si, que se assemelha ao método de ordenação *Bubble Sort*, onde o método passa par a par pelos métodos de teste dentro da classe de teste e faz a alteração de suas posições caso o método *m1* seja não seguro e o método *m2* seguro, conforme linha 15. O método passa por cada par de métodos de teste dentro da classe de teste, até que não haja mais alteração de posição nos métodos de teste.

Já a Figura 22 exibe o trecho de código onde ocorre a interceptação do método que decide se ocorrerá ou não a execução da configuração implícita de teste representada pela notação `@BeforeEach`. A linha 1 da Figura 22 exibe a interceptação do método *interceptBeforeEachMethod* do JUnit e a configuração implícita de teste é executada através do comando *invocatin.proceed()* da linha 8 e não executada pelo comando *invocation.skip()* da linha 16.

Na Figura 22 ainda é possível ver que a execução da configuração implícita de teste ocorre em três cenários conforme a condição da linha 7 que executa o método da linha 19. O primeiro cenário ocorre quando o primeiro método de teste a ser executado é um método seguro, e assim não houve nenhuma execução da configuração implícita de teste, conforme condição da linha 20 que chama o método

da linha 36, onde é verificado que a variável *firstExecution* é verdadeira e alterando a variável para falso em seguida na linha 21.

Figura 21 - Ordenação dos métodos de teste



```

1  @Override
2  public void orderMethods(MethodOrdererContext context) {
3      Collections.sort(context.getMethodDescriptors(), new SafeMethodSorter());
4  }
5
6
7  public class SafeMethodSorter implements Comparator<MethodDescriptor> {
8
9      SafeParser parser;
10
11     public SafeMethodSorter() {
12         this.parser = new SafeParser();
13     }
14
15     public int compare(MethodDescriptor m1, MethodDescriptor m2) {
16         boolean m1Safe = parser.isSafe(m1.getMethod());
17         boolean m2Safe = parser.isSafe(m2.getMethod());
18
19         if (m1Safe && !m2Safe) {
20             return -1;
21         }
22         else if (!m1Safe && m2Safe) {
23             return 1;
24         }
25         return 0;
26     }
27 }

```

Fonte: Elaborado pelo autor.

Já o segundo cenário ocorre quando é a execução do segundo método de teste não seguro, uma vez que o primeiro método de teste não seguro não precisa executar a configuração implícita de teste já que a configuração foi executado com os métodos de testes seguros, conforme condição da linha 25 que verifica que a variável *firstNonSafeExecuted* é verdadeira, conforme alterada para true após a execução do primeiro método não seguro na linha 13.

Já o último cenário ocorre quando não há nenhum método seguro de teste na classe, e assim precisa ser sempre executada a configuração implícita de teste conforme condição da linha 29, onde a variável *firstExecution* será sempre verdadeira.

Resumidamente a execução dos métodos de teste ocorre da seguinte forma:

- Primeiro ocorre a execução das configurações implícitas da classe de teste, representado pela notação *@BeforeEach*;

- Em seguida ocorre a execução de todos os métodos de teste considerados seguros dentro da classe;
- Finalizando os métodos seguros, ocorre a execução do primeiro método de teste não seguro;
- Por fim, ocorre a execução da configuração implícita de teste novamente seguida de um teste não seguro sucessivamente até o último método de teste não seguro.

Figura 22 - Trecho de código que define a execução da configuração implícita

```

1  @Override
2  public void interceptBeforeEachMethod
3  (Invocation<Void> invocation, ReflectiveInvocationContext<Method> reflectiveInvocationContext,
4  ExtensionContext extensionContext) throws Throwable {
5      Method method = extensionContext.getRequiredTestMethod();
6
7      if (this.needsToRunBeforeEachFixture(method)) {
8          invocation.proceed();
9          return;
10     }
11
12     if (!safeParser.isSafe(method)) {
13         this.firstNonSafeExecuted = true;
14     }
15
16     invocation.skip();
17 }
18
19 private boolean needsToRunBeforeEachFixture(Method actualMethod) {
20     if (isSafeAndNotFirstBeforeEachExecution(actualMethod)) {
21         this.firstExecution = false;
22         return true;
23     }
24
25     if (firstNonSafeWasExecuted(actualMethod)) {
26         return true;
27     }
28
29     if ( this.firstExecution ) {
30         return true;
31     }
32
33     return false;
34 }
35
36 private boolean isSafeAndNotFirstBeforeEachExecution(Method actualMethod) {
37     return (this.safeParser.isSafe(actualMethod) && this.firstExecution );
38 }
39
40 private boolean firstNonSafeWasExecuted(Method actualMethod) {
41     return (!this.safeParser.isSafe(actualMethod) && this.firstNonSafeExecuted );
42 }

```

Fonte: Elaborado pelo autor.

Conforme já mencionado, na primeira execução de um método de teste não seguro não é necessário realizar a configuração implícita de teste devido ao fato de todos os teste anteriores terem sido seguros, garantindo, dessa forma, que as configurações implícitas necessárias para execução de cada teste não tiveram alteração.

5.6 - Utilização do SUnit

Para se utilizar o SUnit é necessário que o projeto preencha os seguintes requisitos:

- Código fonte do projeto deve ser desenvolvido em linguagem Java 17;
- Classes de teste deve ser desenvolvida utilizando o *Framework* JUnit 5;
- Caso o projeto esteja utilizando a biblioteca do javaparser-core, ela deve estar na versão 3.8 ou menor;
- O diretório de teste deve incluir, além do código de desenvolvimento de teste, um diretório definido como *resources* contendo o arquivo *sunit.properties* conforme detalhado na seção 5.2 deste trabalho.

A Figura 23 ilustra como o diretório de teste é definido utilizando a IDE IntelliJ IDEA 2025.1.2. Na imagem é exibido um *print* do janela *Project Structure* do projeto, que pode ser acessado através do menu *File>Project Structure*. Após selecionar a aba *Modules* na lateral esquerda da janela, deve-se selecionar o módulo do projeto na coluna ao meio da janela e em seguida selecionar a aba *Sources* na lateral direita da janela, onde irão aparecer todos os diretórios dentro do módulo do projeto selecionado. Ao expandir o diretório de teste e selecionar o diretório de resources é possível marcá-lo como diretório de *resource* no campo *Mark as* logo acima dos diretórios, como apresentado na Figura 23.

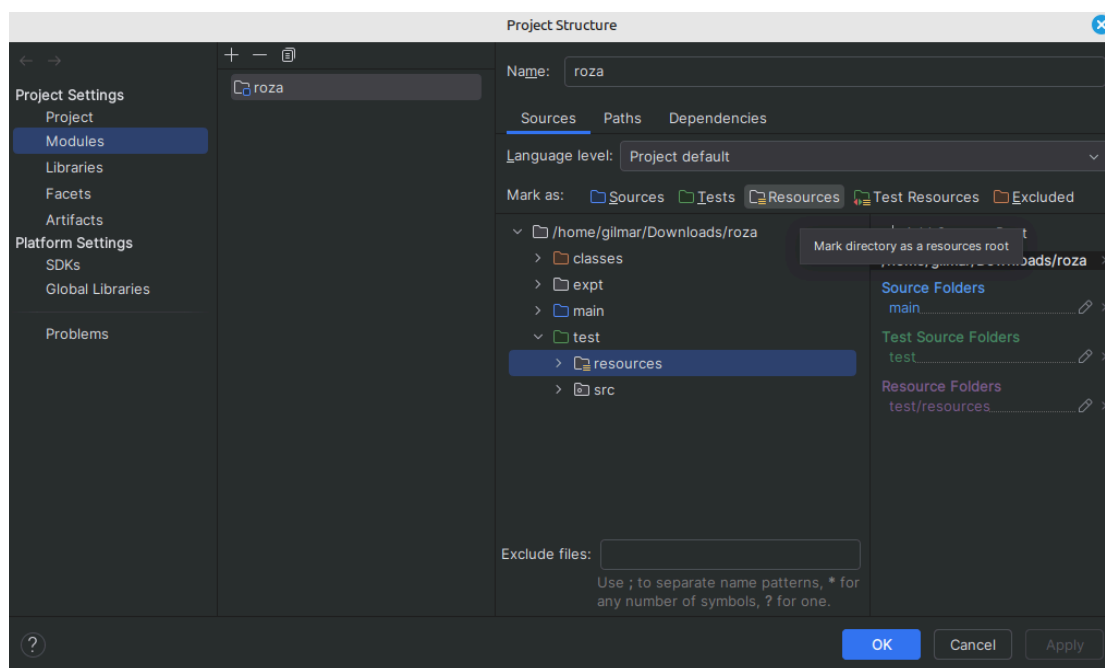
Em seguida, deve-se realizar o *download* do SUnit.jar⁴ e adicioná-lo ao projeto como uma biblioteca java. A Figura 24 exhibe como adicionar a biblioteca java no projeto: ainda na janela de *Project Structure*, selecionar a aba *Dependencies* na lateral direita da janela, em seguida clicar no botão de “+” e selecionar a opção

⁴ https://drive.google.com/drive/folders/1GQfnrlpVnKQeC4NFB4ynmD_O6f8P97dh?usp=sharing

Library>Java. No menu que se abra, deve-se escolher o arquivo baixado e assim adicionar ao projeto.

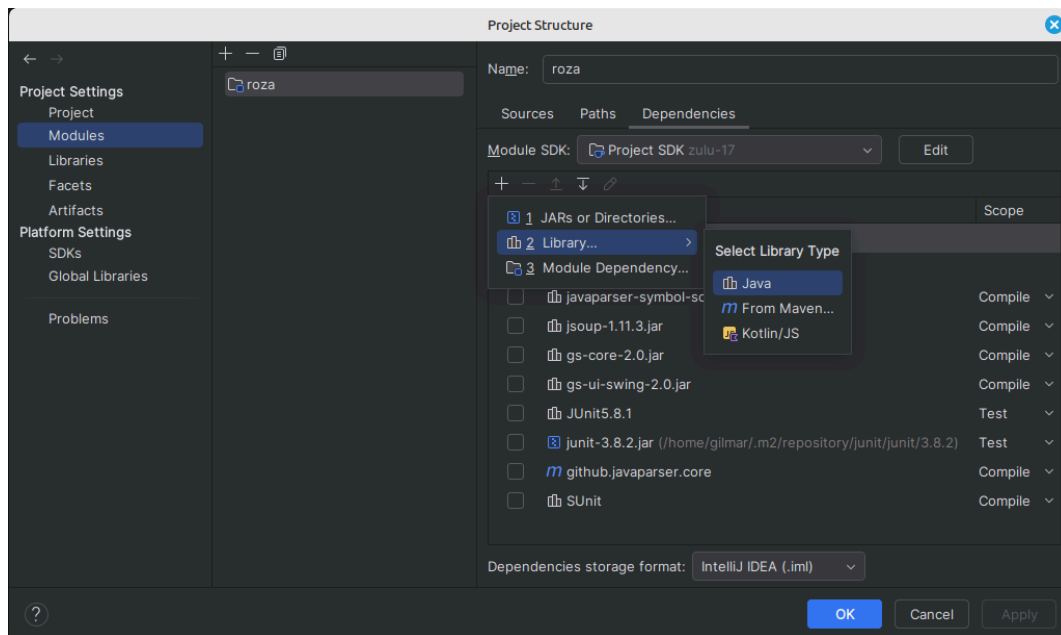
Após adicionado o SUnit no projeto, inicia-se o desenvolvimento das classes de teste em que se deseja utilizar a ferramenta. A classe de teste deve ser desenvolvida normalmente com a utilização do JUnit 5, utilizando as notações nativas do *framework* importantes para utilização do SUnit: `@Test` e `@BeforeEach`. Para o uso do SUnit, a classe de teste deve utilizar as notação `@ExtendWith` e `@TestMethodOrder` antes da declaração da classe, bem como realizar a importação da biblioteca do SUnit para a classe. Na Figura 25 é possível ver o uso das notações `@ExtendWith` e `@TestMethodOrder` nas linhas 9 e 10, além da importação da biblioteca na linha 6 e 7. Além disso, é possível utilizar a notação `@Safe`, linha 22 da Figura 25, como proposto por Silva e Vilain (2017), que marca o método de teste como seguro de forma manual.

Figura 23 - Definindo o diretório *resource* de teste



Fonte: Elaborado pelo autor.

Figura 24 - Inserindo o SUnit no projeto



Fonte: Elaborado pelo autor.

Figura 25 - Utilização do SUnit

```

1 import org.junit.jupiter.api.BeforeEach;
2 import org.junit.jupiter.api.Test;
3 import org.junit.jupiter.api.TestMethodOrder;
4 import org.junit.jupiter.api.extension.ExtendWith;
5
6 import sunit.annotations.Safe;
7 import sunit.runner.SUnitRunner;
8
9 @ExtendWith(SUnitRunner.class)
10 @TestMethodOrder(SUnitRunner.class)
11 public class FuncionarioTest {
12
13     private static Funcionario funcionario;
14     private static float salario;
15
16     @BeforeEach
17     public void setUp() {
18         this.salario = 1500;
19         this.funcionario = new Funcionario(this.salario);
20     }
21
22     @Safe
23     @Test
24     public void deveRetornarSalarioCorretoDoFuncionario() {
25         Assertions.assertEquals(this.funcionario.getSalario(), this.salario);
26     }
27 }

```

Fonte: Elaborado pelo autor.

6 EXPERIMENTOS

O presente capítulo tem por objetivo apresentar os experimentos realizados com a finalidade de validar a ferramenta desenvolvida. Para isso, foi desenvolvido um pequeno projeto para verificar se a ferramenta classifica corretamente os métodos das classes em seguros e não seguros e os métodos de testes em seguros e não seguros.

Além disso, foi selecionado um projeto disponível no GitHub, o Roza⁵, para verificar se o tempo de execução dos testes automatizados é de fato reduzido. Roza apresenta em seu repositório uma coleção de 81 classes de testes, totalizando 391 métodos de testes automatizados, desenvolvido em linguagem de programação Java e com o *framework* de teste JUnit. Esse projeto foi selecionado por apresentar uma grande lista de testes automatizados, desenvolvidos em linguagem Java e com o *Framework* JUnit e com código aberto. Não foram encontrados mais projetos que cumprissem esse pré-requisito a tempo para serem adicionados como experimentos neste trabalho.

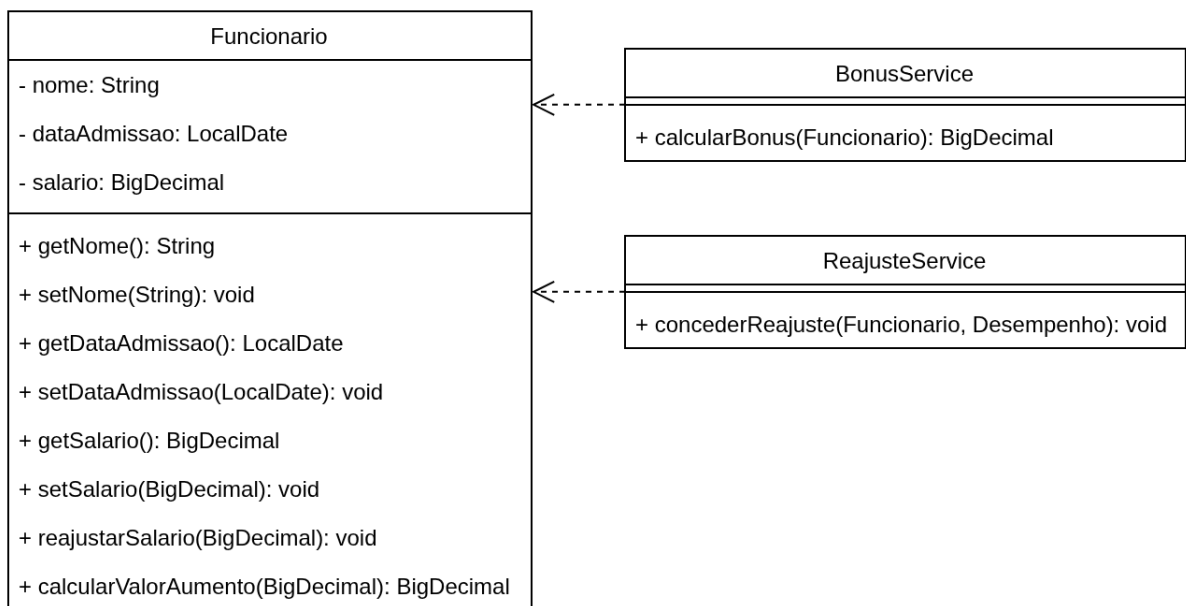
6.1 - Validação da classificação dos métodos de classe e testes

Para fazer a validação da classificação dos métodos das classes foram criadas três classes: *Funcionario*, *ReajusteService* e *BonusService*. O diagrama de classes deste projeto está ilustrado na Figura 26. A classe *Funcionario* representa um funcionário e apresenta os seguintes atributos: nome, data de admissão e salário. Esta classe também apresenta os métodos *getters* e *setters* para esses atributos, além de mais dois métodos: o método *calcularValorAumento*, que faz o cálculo do valor do aumento do funcionário conforme o valor percentual de entrada, e o método *reajustarSalario*, que faz o reajuste do salário do funcionário também recebendo um valor percentual de entrada. Já a classe *ReajusteService* realiza o reajuste de salário do funcionário com base no seu desempenho, enquanto a classe *BonusService* realiza o cálculo do valor do bônus que o funcionário deverá receber.

⁵ <https://github.com/lucasPereira/roza>

Para avaliar a classificação dos métodos de teste em seguro e não seguro, foi realizada a implementação de duas classes de teste, sendo elas: *FuncionarioTest* e *ReajusteServiceTest*. A classe *FuncionarioTest* apresenta um total de onze métodos de teste desenvolvidos, sendo cinco deles implementados como seguros e seis deles implementados como não seguros. Já a classe *ReajusteServiceTest* apresenta um total de três métodos de teste, sendo todos eles implementados como testes não seguros.

Figura 26 - Diagrama de classe do projeto usado para validar o SUnit



Fonte: Elaborado pelo autor.

A seguir, a Figura 27 apresenta os resultados obtidos pela ferramenta SUnit no projeto de teste. A ferramenta conseguiu identificar os cinco métodos de classes que eram de fato não seguros e os cinco métodos de teste que eram considerados seguros.

Figura 27 - Resultados obtidos pelo SUnit no projeto criado pelo autor

```

Run All in teste (1) x
11 ms 14 tests passed 14 tests total, 11 ms
FuncionarioTest 11 ms
  ✓ deveRetornarNomeCorretoDoFuncionario()
  ✓ bonusDeveriaSer10PorCentoDoSalario()
  ✓ deveRetornarSalarioCorretoDoFuncionario()
  ✓ deveRetornarDataAdmissaoCorretoDoFuncionario()
  ✓ bonusDeveriaSerDezPorCentoParaSalarioDeExata
  ✓ bonusDeveriaSer50ParaSalarioDeExatamento()
  ✓ deveAlterarNomeCorretamenteDoFuncionario()
  ✓ bonusDeveriaSerZeroPorCentoParaSalarioDeExatu
  ✓ deveAlterarDataAdmissaoCorretamenteDoFuncio
  ✓ bonusDeveriaSerZeroParaFuncionarioComSalario
  ✓ deveAlterarSalarioCorretamenteDoFuncionario()
ReajusteServiceTest
  ✓ reajusteDeveriaSerDeQuinzePorcentoQuandoDes
  ✓ reajusteDeveriaSerDeVintePorcentoQuandoDeser
  ✓ reajusteDeveriaSerDeTresPorcentoQuandoDesen

/home/gilmar/.sdkman/candidates/java/17.0.14-zulu/bin/java ...
[main] INFO sunit.parser.code.ast.JavaASTParser - Initializing
[main] INFO sunit.parser.code.ast.JavaASTParser - Metodos não seguro encontrados: 5
[main] INFO sunit.parser.code.ast.JavaASTParser - Method not safe found: [Funcionario].setName
[main] INFO sunit.parser.code.ast.JavaASTParser - Method not safe found: [Funcionario].setDataAdmissao
[main] INFO sunit.parser.code.ast.JavaASTParser - Method not safe found: [Funcionario].setSalario
[main] INFO sunit.parser.code.ast.JavaASTParser - Method not safe found: [ReajusteService].concederReajuste
[main] INFO sunit.parser.code.ast.JavaASTParser - Method not safe found: [Funcionario].reajustarSalario
[main] INFO sunit.parser.code.ast.JavaASTParser - Metodos de teste seguro encontrados: 5
[main] INFO sunit.parser.code.ast.JavaASTParser - Safe test found: [FuncionarioTest].deveRetornarNomeCorretoDoFuncionario()
[main] INFO sunit.parser.code.ast.JavaASTParser - Safe test found: [FuncionarioTest].deveRetornarSalarioCorretoDoFuncionario()
[main] INFO sunit.parser.code.ast.JavaASTParser - Safe test found: [FuncionarioTest].deveRetornarDataAdmissaoCorretoDoFuncionario()
[main] INFO sunit.parser.code.ast.JavaASTParser - Safe test found: [FuncionarioTest].bonusDeveriaSer10PorCentoDoSalario()
[main] INFO sunit.parser.code.ast.JavaASTParser - Safe test found: [FuncionarioTest].bonusDeveriaSerDezPorCentoParaSalarioDeExatamento()
[main] INFO sunit.parser.code.ast.JavaASTParser - Safe test found: [FuncionarioTest].bonusDeveriaSer50ParaSalarioDeExatamento()
[main] INFO sunit.parser.code.ast.JavaASTParser - Safe test found: [FuncionarioTest].deveAlterarNomeCorretamenteDoFuncionario()
[main] INFO sunit.parser.code.ast.JavaASTParser - Safe test found: [FuncionarioTest].bonusDeveriaSerZeroPorCentoParaSalarioDeExatamento()
[main] INFO sunit.parser.code.ast.JavaASTParser - Safe test found: [FuncionarioTest].deveAlterarDataAdmissaoCorretamenteDoFuncionario()
[main] INFO sunit.parser.code.ast.JavaASTParser - Safe test found: [FuncionarioTest].bonusDeveriaSerZeroParaFuncionarioComSalario()
[main] INFO sunit.parser.code.ast.JavaASTParser - Safe test found: [FuncionarioTest].deveAlterarSalarioCorretamenteDoFuncionario()
[main] INFO sunit.parser.code.ast.JavaASTParser - Safe test found: [ReajusteServiceTest].reajusteDeveriaSerDeQuinzePorcentoQuandoDeser()
[main] INFO sunit.parser.code.ast.JavaASTParser - Safe test found: [ReajusteServiceTest].reajusteDeveriaSerDeVintePorcentoQuandoDeser()
[main] INFO sunit.parser.code.ast.JavaASTParser - Safe test found: [ReajusteServiceTest].reajusteDeveriaSerDeTresPorcentoQuandoDeser()
Process finished with exit code 0

```

Fonte: Elaborado pelo autor.

6.2 - Validação do SUnit em um projeto real

Neste experimento foi utilizado o Roza para aplicação da ferramenta deste trabalho. O Roza é um *framework* de refatoração automatizada de classes de teste e de redistribuição de testes em classes através da utilização de métricas de similaridade de código e agrupamento. O *framework* foi desenvolvido utilizando a abordagem TDD (Test Driven Development) e possui um conjunto de 81 classes de testes com 391 métodos de teste.

Para validação do SUnit em um projeto real, foi realizada a execução de todos os testes do Roza em duas etapas, sendo a primeira sem a utilização da ferramenta SUnit, ou seja, sem a reutilização da configuração implícita de teste e a segunda utilizando a ferramenta, a fim de poder comparar os resultados. Na primeira etapa não houve nenhuma classificação feita nos métodos de teste de forma manual, utilizando a notação `@Safe`. Para realização do experimento foi utilizada uma máquina com sistema operacional Linux Mint 22 Cinnamon, processador Intel Core Ultra 9 185H x 16, 32 GB de memória RAM, OpenJDK 17.0.14 e IntelliJ IDEA 2025.1.2.

Para validação do SUnit em um projeto real, cada uma das duas etapas (sem a utilização do SUnit e com a utilização do SUnit) foi executada 10 vezes com a finalidade de verificar se há diferença estatística significativa no tempo de execução.

Dessa forma, poderemos afirmar que a diferença observada nos tempos de execução é real e não por acaso, uma vez que cada execução da suíte de teste não apresenta sempre o mesmo tempo de execução, devido a flutuações do ambiente de teste, como por exemplo outros processos em execução. A Tabela 2 exibe os valores obtidos em cada uma das 10 rodadas de teste de cada etapa, sendo a Etapa 1 realizada sem a utilização do SUnit e a Etapa 2 com a utilização do SUnit.

Após executar todas as rodadas da suíte de testes do Roza em cada etapa, foi realizada uma análise estatística para aceitar ou não a hipótese nula, onde a hipótese nula indica que não há diferença significativa no tempo de execução entre as duas estratégias. Para isso foi utilizado o teste estatístico t , devido à pequena amostra e por não se conhecer a variância da população. Com a aplicação do teste t foi obtido um p -valor, onde um resultado pequeno para esse valor indica que há uma diferença estatística de significância entre as duas estratégias. E para determinar o nível de significância de 5% (0,05), ou seja, se encontrarmos um p -valor que é menor que 0,05 rejeitamos a hipótese nula, caso contrário, aceitamos a hipótese nula.

Tabela 2 - Tempo de execução da suíte de teste do Roza

	Etapa 1 (ms)	Etapa 2 (ms)
Rodada 1	221	141
Rodada 2	234	109
Rodada 3	213	100
Rodada 4	207	97
Rodada 5	197	99
Rodada 6	198	127
Rodada 7	223	122
Rodada 8	208	119
Rodada 9	206	108
Rodada 10	206	108
Média	211,30	113,00
Desvio Padrão	11,62	14,08
Variância	135,12	198,22
p-valor	0,0000000000280	

Fonte: Elaborado pelo autor.

Através da Tabela 2, observamos que o p -valor encontrado é igual a 0.00000000000280, sendo esse valor menor que 0.05 e com isso rejeitamos a

hipótese nula, ou seja, há um valor estatístico significativo com a utilização do SUnit para redução do tempo de execução de testes automatizados.

Por fim, considerando a média dos tempos de execução das etapas 1 e 2 na Tabela 2, podemos dizer que a execução da suíte de testes do Roza sem a utilização do SUnit foi de 211,3 ms e com a utilização do SUnit esse tempo caiu para 113 ms. Sendo assim, o tempo de execução da suíte de teste do Roza utilizando o SUnit reduziu 46,52% em relação ao tempo de execução sem o SUnit.

7 CONCLUSÃO

Neste trabalho foi proposto o SUnit, uma ferramenta para reutilização da execução da configuração implícita de testes automatizados, com a finalidade de reduzir o tempo de execução dos mesmos e ainda garantir a eficácia inerente aos testes de *softwares*.

A proposta foi implementada através da análise estática do código principal e do código de teste do projeto ao qual se deseja aplicar a ferramenta, permitindo a reutilização de execução das configurações implícitas caso os métodos de teste não realizem mudança no seu estado inicial. O SUnit é capaz de analisar todos os métodos contidos em todas as classes do projeto e todos os métodos de teste em todo o código de teste do projeto, e com isso realizar a classificação de tais métodos em seguros e não seguros.

O SUnit também teve sua abordagem validada por meio de dois experimentos. O primeiro validou sua classificação dos métodos de classe e dos métodos de teste em seguros e não seguros. O segundo experimento testou a capacidade da ferramenta em reduzir o tempo de execução dos testes automatizados, utilizando a abordagem estatística *teste-t*, resultando em uma redução de tempo de execução próximo a 47%.

Como recomendação para trabalhos futuros, recomenda-se o projeto e desenvolvimento de uma nova versão do SUnit incluindo o suporte à paralelização de testes, uma vez que o *framework* JUnit 5 apresenta essa funcionalidade em seu escopo.

REFERÊNCIAS

BARBOSA, Felipe; TORRES, Isabelle. O teste de software no mercado de trabalho. **Tecnologias em Projeção**, v. 2, n. 1, 2011.

BARTIÉ, Alexandre. **Garantia da qualidade de software: adquirindo maturidade organizacional**. Gulf Professional Publishing, 2002.

BASE2. **Conceitos: testes de caixa branca e caixa preta**. 2023. Disponível em: <<https://base2.com.br/blog/conceitos-testes-de-caixa-branca-e-caixa-preta/>>. Acesso em: 25 de mai. de 2025.

BERNARDO, Paulo Cheque; KON, Fabio. A importância dos testes automatizados. **Engenharia de Software Magazine**, v. 1, n. 3, p. 54-57, 2008.

CHIAVEGATTO, Rafael et al. Especificação e Automação Colaborativas de Testes utilizando a técnica BDD. In: **Simpósio Brasileiro de Qualidade de Software (SBQS)**. SBC, 2013. p. 334-341.

CHRISTENSEN, Claus A. et al. A unit-test framework for database applications. In: **2006 10th International Database Engineering and Applications Symposium (IDEAS'06)**. IEEE, 2006. p. 11-20.

COX, Graham. **Introduction to JavaParser**. Baeldung, 12 mai. 2024. Disponível em: <<https://www.baeldung.com/javaparser>>. Acesso: 23 de mai. de 2025.

CRESPO, Adalberto Nobiato et al. Uma metodologia para teste de Software no Contexto da Melhoria de Processo. In: **Anais do III Simpósio Brasileiro de Qualidade de Software**. SBC, 2004. p. 204-218.

DELAMARO, Marcio; JINO, Mario; MALDONADO, Jose. **Introdução ao teste de software**. Elsevier Brasil, 2013.

DIAS NETO, Arilo Claudio. Introdução a teste de Software. **Engenharia de Software Magazine**, v. 1, p. 22, 2007.

DUARTE, Alexandre et al. Gridunit: software testing on the grid. In: **Proceedings of the 28th international conference on Software engineering**. 2006. p. 779-782.

EMERY, Dale H. **Writing maintainable automated acceptance tests**. In: Agile Testing Workshop, Agile Development Practices, Orlando, Florida. 2009.

FERREIRA, Gustavo Antonio Santos. ESTUDO DE FRAMEWORKS DE TESTES UNITÁRIOS EM JAVA. 2017.

FEWSTER, Mark; GRAHAM, Dorothy. **Software test automation**. Reading: Addison-Wesley, 1999.

FORSGREN, Nicole; HUMBLE, Jez; KIM, Gene. **Accelerate: The science of lean software and devops: Building and scaling high performing technology organizations**. IT Revolution, 2018.

FOWLER, M. **Test Pyramid**. 2012. Disponível em: <<https://martinfowler.com/bliki/TestPyramid.html>> . Acesso em: 25 de mai. de 2025.

FOWLER, M. **Continuous Integration**. In: **Refactoring: Improving the Design of Existing Code**. Addison-Wesley, 2006.

GAO, Yi et al. Automated Unit Test Refactoring. **Proc. ACM Softw. Eng.**, [S. l.], v. 2, n. FSE, art. FSE033, 21 p., 2025.

GARCIA, Boni. **Mastering Software Testing with JUnit 5: Comprehensive guide to develop high quality Java applications**. Packt Publishing Ltd, 2017.

GREILER, Michaela et al. **Strategies for avoiding text fixture smells during software evolution**. In: Proceedings of the 10th Working Conference on Mining Software Repositories. IEEE Press, 2013. p. 387-396

HUMBLE, Jez; FARLEY, David. **Continuous delivery: reliable software releases through build, test, and deployment automation**. Pearson Education, 2010.

JÚNIOR, Misael; ANDRADE, Stevão; DELAMARO, Márcio. **Automatização de teste de software com ênfase em teste de unidade**, 2016.

JUNIT. JUnit 5. Disponível em: <<https://junit.org/junit5/docs/current/user-guide/>>. Acesso em: 16 de mar. de 2025.

MARTIN, Robert C. **Clean code: a handbook of agile software craftsmanship**. Pearson Education, 2009.

MCCONNELL, Steve. Code Complete. 2. ed. Redmond, WA: Microsoft Press, 2004.

MEDEIROS, Hallan et al. Solunit: A framework for reducing execution time of smart contract unit tests. In: **Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering**. 2019. p. 264-273.

MEDEIROS, H. **SolUnit: um framework de testes de unidade para smart contracts**. 2020. 93 f. Dissertação (Mestrado em Ciências da Computação) - UFSC, Florianópolis, 2020.

MESZAROS, Gerard; SMITH, Shaun M.; ANDREA, Jennitta. The test automation manifesto. In: **Conference on extreme programming and agile methods**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. p. 73-81.

MESZAROS, Gerard. **xUnit test patterns: Refactoring test code**. Pearson Education, 2007.

MYERS, Glenford J.; SANDLER, Corey; BADGETT, Tom. **The art of software testing**. John Wiley & Sons, 2011.

NAIK, K; TRIPATHY, P. **Software testing and quality assurance: theory and practice. [S.1.]**: John Wiley & Sons, 2008.

PRESSMAN, Roger S.; MAXIM, Bruce R. **Engenharia de software**. 9. ed. McGraw Hill Brasil, 2021.

SILVA, Carla Lizandra. Investigando o uso de práticas modernas de testes de software na indústria e na academia. 2025.

SILVA, Davi Augusto Gadêlha. **Evolunit: geração e evolução de testes de unidade em java utilizando algoritmos genéticos**. 2008. Dissertação de Mestrado. Universidade Federal de Pernambuco.

SILVA, Lucas Pereira; VILAIN, Patrícia. Reuse of Fixture Setup between Test Classes. In: **SEKE**. 2017. p. 224-229.

SILVA, L. P.; VILAIN, P. Estória: A framework for code and execution reuse between test classes. **Journal of Computer Science**, v. 17, p. 90–111, 2021.

SOMMERVILLE, Ian. Engenharia de software, 9a. **São Palo, SP, Brasil**, 2011.

TECNISYS, **A Pirâmide de Automação de Testes**. Disponível em: <<https://www.tecnisys.com.br/a-piramide-de-automacao-de-testes/>> . Acesso em: 25 de mai. de 2025

TSAI, Wei-tek et al. **Scenario-based object-oriented testing framework**. In: Third International Conference on Quality Software, 2003. Proceedings. IEEE, 2003. p. 410-417.79

UMAR, Mubarak Albarka; ZHANFANG, Chen. A study of automated software testing: Automation tools and frameworks. **International Journal of Computer Science Engineering (IJCSE)**, v. 6, n. 217-225, p. 47-48, 2019.

VASCONCELOS, Alexandre Marcos Lins et al. Introdução à Engenharia de Software e à Qualidade de Software. **Minas Gerais: Universidade Federal de Lavras**, 2006.

ZHAO, Ruilian et al. E2E test execution optimization for web application based on state reuse. **Journal of Software: Evolution and Process**, v. 37, n. 1, p. e2714, 2025.

APÊNDICE A - RESULTADO DA BUSCA DE TRABALHOS RELACIONADOS

Título	Autor(es)	Ano
LLM-enhanced evolutionary test generation for untyped languages	Ruofan Yang, Xianghua Xu & Ran Wang	2025
E2E test execution optimization for web application based on state reuse	Ruilian Zhao, Shukai Zhang, Zhifan Zhu, Ying Shang, Weiwei Wang	2025
ATSM: A coverage-based framework and a tool for test suite minimization	Alekseev, Y., Onischuck, M., Zorin, A., Chernyi, V., Iliyn, E., Itsykson, V.	2024
Regression test selection in test-driven development	Zohreh Mafi & Seyed-Hassan Mirian-Hosseiniabadi	2024
Coverage Goal Selector for Combining Multiple Criteria in Search-Based Unit Test Generation	Zhichao Zhou; Yuming Zhou; Chunrong Fang; Zhenyu Chen; Xiapu Luo; Jingzhu He	2024
Dynamic Economic Dispatch Problem in Hybrid Wind Based Power Systems Using Quasi Oppositional Based Chaotic Honey Badger Algorithm	Mandal, B., Roy, P.K.	2024
Probe Log: Visualizing the Control Flow of Babylonian Programming	Eva Krebs, Patrick Rein, Joana Bergsiek, Lina Urban, Robert Hirschfeld	2023
Research on Low-Cost IMU Testing Methods	Yajing Guo, Fan Yang, Jing Chen, Binyan Liang & Shuxuan Liu	2023
A Framework for the Gamification of GUI Testing	Riccardo Coppola, Luca Ardito, Tommaso Fulcini, Giacomo Garaccione, Marco Torchiano & Maurizio Morisio	2023
Selectively Combining Multiple Coverage Goals in Search-Based Unit Test Generation	Zhichao Zhou, Yuming Zhou, Chunrong Fang, Zhenyu Chen, Yutian Tang	2023
Generating Unit Tests for Documentation	Mathieu Nassif; Alexa Hernandez; Ashvitha Sridharan; Martin P. Robillard	2022
DScribe: Co-generating Unit Tests and Documentation	Alexa Hernandez; Mathieu Nassif; Martin P. Robillard	2022
Testing Software in Production Environments with Data from the Field	Luca Gazzola; Leonardo Mariani; Matteo Orrù; Mauro Pezzè; Martin Tappler	2022
Object-Oriented Test Case Generation Using Teaching Learning-Based Optimization (TLBO) Algorithm	Ohood Al-Masri; Wedad Abdulraqueeb Al-Sorori	2022
Testing with Fewer Resources: An Adaptive Approach to Performance-Aware Test Case Generation	Giovanni Grano; Christoph Laaber; Annibale Panichella; Sebastiano Panichella	2019
Developer's Perspectives on Unit Test Cases Understandability	Novi Setiani; Ridi Ferdiana; Rudy Hartanto	2021
An Empirical Analysis of UI-Based Flaky Tests	Alan Romano; Zihe Song; Sampath Grandhi; Wei Yang; Weihang Wang	2021
HTR-1.2 solver: Hypersonic Task-based Research solver version 1.2	Mario Di Renzo, Sergio Pirozzoli	2021
Compressing Automatically Generated Unit Test Suites Through Test Parameterization	Aidin Azamnouri & Samad Paydar	2021
A unified framework for declarative debugging and testing	Rafael Caballero, Enrique Martin-Martin, , Adrián Riesco, Salvador Tamarit	2021

A Method for Finding Missing Unit Tests	Daniel Gaston; James Clause	2020
SolUnit: A framework for reducing execution time of smart contract unit tests	Hallan Medeiros, John Mylopoulos, Patrícia Vilai, Hans-Arno Jacobsen	2020
On the Effectiveness of Manual and Automatic Unit Test Generation: Ten Years Later	Domenico Serra; Giovanni Grano; Fabio Palomba; Filomena Ferrucci; Harald C. Gall; Alberto Bacchelli	2019
Assessing Diffusion and Perception of Test Smells in Scala Projects	Jonas De Bleser; Dario Di Nucci; Coen De Roover	2019
8th International Conference on Fundamentals of Software Engineering, FSEN 2019	-	2019
Speeding up mutation testing via the cloud: lessons learned for further optimisations	Sten Vercammen, Serge Demeyer, Markus Borg, Sigrid Eldh	2018
Practical Amplification of Condition/Decision Test Coverage by Combinatorial Testing	Artur Andrzejak; Thomas Bach	2018
Tests from traces: automated unit test extraction for R	Filip Křikava, Jan Vitek	2018
Regression Testing of Database Applications Under an Incremental Software Development Setting	Raúl H. Rosero; Omar S. Gómez; Glen Rodríguez	2017
Hybrid Data Approach for Selecting Effective test Cases During the Regression Testing	M. Mohan and Tarun Shrimali	2017
Reuse of fixture setup between test classes	Lucas Silva, Patrícia Vilain	2017
9th International Symposium on Search-Based Software Engineering, SSBSE 2017	-	2017
Memory mutation testing	Fan Wu, Jay Nanavati, Mark Harman, Yue Jia, Jens Krinke	2017
FSX: a tool for fine-grained incremental unit test generation for C/C++ programs	Hiroaki Yoshida, Susumu Tokumoto, Mukul R. Prasad, Indradeep Ghosh, Tadahiro Uehara	2016
FSX: Fine-grained incremental unit test generation for C/C++ programs	Hiroaki Yoshida, Susumu Tokumoto, Mukul R. Prasad, Indradeep Ghosh, Tadahiro Uehara	2016
Multi-objective Regression Test Suite Minimisation for Mockito	Andrew J. Turner, David R. White & John H. Drake	2016
International Workshop on Emerging Trends in Software Metrics, WETSoM	-	2015
Test case selection using software complexity and volume metrics	Shadi Banitaan, Kevin Daimi, Yujun Wang, Mohammed Akour	2015
Unit test virtualization with VMVM	Jonathan Bell, Gail Kaiser	2014
VMVM: unit test virtualization for Java	Jonathan Bell, Gail Kaiser	2014
Optimizing unit test execution in large software programs using dependency analysis	Taesoo Kim, Ramesh Chandra, Nickolai Zeldovich	2013
Finding bugs by isolating unit tests	Kivanç Muşlu, Bilge Soran, Jochen Wuttke	2011
Development of test automation framework for testing avionics systems	Ashutosh Kumar Jha	2010
Parameterized unit testing: theory and practice	Nikolai Tillmann, Jonathan de Halleux, Tao Xie	2010
The impact of Test-First programming on branch coverage and mutation score indicator of unit tests: An experiment	Lech Madeyski	2010

Reusing and Composing Tests with Traits	Stéphane Ducasse, Damien Pollet, Alexandre Bergel & Damien Cassou	2009
Testing of GIS components at 1000 kV rated voltage	Walter Holaus, Urs Krüsi, Diego Sologuren, Uwe Riechert, Markus Keller	2008
Impact of pair programming on thoroughness and fault detection effectiveness of unit test suites	Lech Madeyski	2008
Application of a Quantitative Carrier Test to Evaluate Microbicides against Mycobacteria	Susan V Springthorpe , Syed A Sattar	2007
A Unit-Test Framework for Database Applications	Claus A. Christensen; Steen Gundersborg; Kristian de Linde; Kristian Torp	2006
Detecting Redundant Unit Tests for AspectJ Programs	Tao Xie; Jianjun Zhao; Darko Marinov; David Notkin	2006
GridUnit: Software testing on the grid	Alexandre Duarte, Walfredo Cirne, Francisco Brasileiro, Patrícia Machado	2006
Large-scale software unit testing on the grid	Yaohang Li, Tao Dong, Xinyu Zhang, Yong-Duan Song, Xiaohong Yuan	2006
Tool-assisted unit-test generation and selection based on operational abstractions	Tao Xie & David Notkin	2006
Special report: Clean fuels: Multivariable step testing for MPC projects reduces crude unit testing time	Alex Kalafatis, Kalpesh Patel, Michael Harmse, Qingsheng Zheng, Mark Craik	2006
A Case for Efficient Solution Enumeration	Sarfraz Khurshid, Darko Marinov, Ilya Shlyakhter & Daniel Jackson	2004
Tool-assisted unit test selection based on operational violations	Tao Xie; D. Notkin	2003
An integrated test approach for avionics system development	Braid D.M., Johnson C.W., Schillinger G.D.	2001

APÊNDICE B - MUDANÇA DO TEMA DESTE TRABALHO

Este apêndice tem como objetivo apresentar a justificativa para alteração do tema deste trabalho, que inicialmente iria tratar sobre a inclusão de uma nova funcionalidade a ferramenta SolUnit, que otimiza o tempo de execução de testes automatizados em *blockchain*. A nova funcionalidade que foi proposta visava a ordenação dos testes automatizados implantados dentro de um *smart contract* pelo tempo de execução de cada teste, de forma crescente. Essa proposta tinha a finalidade de executar mais testes nos momentos iniciais da execução da suíte de teste, já que os testes mais rápidos seriam os primeiros da fila de teste a serem executados, e assim possibilitar a detecção de testes com problemas ou falhas de forma mais rápida. Dessa forma, o trabalho de desenvolvimento em cima da resolução dessas falhas se daria mais cedo.

A primeira alteração na ferramenta foi buscar uma forma de armazenar o tempo de execução de cada método de teste dentro de uma classe de teste. A primeira abordagem utilizada foi a utilização de serialização de objetos, que possibilita a conversão de objetos Java em uma sequência de *bytes* para ser salva em disco. Quando os testes fossem rodados novamente, esse arquivo poderia ser desserializado e convertidos em objetos Java novamente para manipulação e utilização na ferramenta.

Definida a forma de armazenamento dos tempos de execução de cada teste, se dava início a fase da proposta em que eram ordenados os métodos de teste conforme seu tempo de execução. A ferramenta partia do princípio proposto pelo SolUnit de reutilizar a configuração implícita de teste, sendo assim os testes seguros seriam realizados antes dos testes não seguros.

Dessa forma, a alteração da ferramenta focaria no método de ordenação do SolUnit, sendo ordenado todos os métodos de teste seguros primeiro de forma crescente e em seguida todos os métodos não seguros, também de forma crescentes. Então tínhamos dois cenários a tratar, testes que já haviam sido executados anteriormente e era conhecido o tempo de execução e teste que não haviam sido executados nenhuma vez ainda, e portanto com o tempo de execução desconhecido.

Sendo assim, decidiu-se então executar primeiro os testes com tempo de execução conhecidos e em seguida os testes com tempo de execução desconhecidos e por ordem crescente de tempo de execução. A nova proposta de inclusão ao SolUnit ainda realizava a atualização dos tempos de execução de cada método de teste que já se conhecia o tempo de execução.

Contudo, após as primeiras implementações e execuções de testes da nova funcionalidade, constatou-se que tal premissa não se concretizava e que não era possível garantir que o teste com falha seria sempre encontrado sempre antes, caso os testes fossem ordenados pelo tempo de execução. Em testes realizados, a identificação dos testes com falha ocorria sempre em um tempo maior utilizando a nova funcionalidade implementada no SolUnit em relação ao uso do framework sem a nova funcionalidade.

Foram realizados quatro experimentos, cada um deles realizado em duas etapas. A primeira etapa consistia em realizar os testes normalmente sem aplicar a nova funcionalidade de ordenação dos testes por tempo. Já a segunda etapa consistia em realizar os testes com ordenação por tempo de execução dos testes, sendo realizados os testes com tempo de execução mais rápidos primeiro e indo até o teste com tempo de execução mais lento.

Para isso era forçado o erro em um teste aleatoriamente e verificado em quanto tempo a falha seria verificada. Os erros forçados foram obtidos através de pequenas alterações no *smart contract* utilizado para teste: alterações de condições de métodos *if* que eram *true* para utilizar resultados que eram *false*, alteração em condições *for* onde a variável que era incrementada positivamente através de *i++* para ser incrementada negativamente como sendo *i--*. Os resultados serão expostos a seguir:

1. Na primeira rodada de teste sem ordenação o erro foi identificado com 3,407s e com ordenação foi identificado o erro com 3,68s;
2. Na segunda rodada de teste sem ordenação o erro foi identificado com 35,43s e com ordenação foi identificado o erro com 39,857s;
3. Na terceira rodada de teste sem ordenação o erro foi identificado com 41,938s e com ordenação foi identificado o erro com 46,938s;
4. Na quarta rodada de teste sem ordenação o erro foi identificado com 15,746s e com ordenação foi identificado o erro com 10,902s.

Com esse resultados obtidos, apenas na quarta rodada a hipótese de que testes ordenados pelo tempo de execução seria possível de se identificar erros de forma mais rápida e na três primeiras rodadas essa hipótese não se confirmou.

Com isso em mente, tal ideia foi abandonada e seguiu-se com a ideia de implementação da abordagem utilizada no SolUnit em uma ferramenta que pudesse ser utilizada em outros projetos que não fossem em *blockchain*.

APÊNDICE C - ARTIGO

SUnit: Uma ferramenta para redução do tempo de execução de testes automatizados com JUnit5

Gilmar D. B. de Souza¹, Patrícia Vilain¹, Lucas P. da Silva¹

¹Departamento de Informática e Estatística – Universidade Federal de Santa Catarina (UFSC)

gilmar500@gmail.com, patricia.vilain@ufsc.br, lucas.pereira@ufsc.br

Abstract. *Currently, automated tests are of great importance in the software quality production pipeline, providing a reduction in maintenance costs and increasing the delivery capability of an agile team. This work aims to implement an extension of the JUnit 5 framework, called SUnit, with the objective of reducing the time spent during the execution of automated tests, through the approach of reusing the implicit setup of each test, whenever possible, without violating the principle of test independence. The reuse of each test's implicit setup is achieved by identifying tests that do not alter the initial state of this setup, called safe tests. The reduction in time spent during test execution is possible because the implicit setup included in a test class is executed only once before the execution of all safe tests. An experiment was conducted to verify the tool's effectiveness in a real project, with a test suite presenting 391 test methods; and the result showed a reduction of approximately 46% in test execution time.*

Resumo. *Atualmente, testes automatizados se fazem de grande importância na linha de produção de software de qualidade, proporcionando uma redução nos custos de manutenção e aumentando a capacidade de entrega de uma equipe ágil. Este trabalho visa implementar uma extensão do framework JUnit 5, chamado SUnit, com o objetivo de reduzir o tempo gasto durante a execução de testes automatizados, através da abordagem de reutilização da configuração implícita de cada teste, sempre que possível, sem ferir o princípio de independência dos testes. A reutilização da configuração implícita de cada teste é feita através da identificação dos testes que não alteram o estado inicial dessa configuração, chamados de testes seguros. A redução no tempo gasto durante a execução dos testes é possível porque a configuração implícita incluída em uma classe de teste é executada uma única vez antes da execução de todos os testes seguros. Foi realizado um experimento para verificar a eficácia da ferramenta em um projeto real, com uma suíte de teste apresentando 391 métodos de teste; e o resultado mostrou uma redução de aproximadamente 46% no tempo de execução dos testes.*

1. Introdução

É comum que grandes empresas adotem um processo de qualidade rigoroso na etapa final de desenvolvimento de software e esse controle de qualidade pode ser realizado por meio de testes manuais, onde a execução desses testes precisa ser rápida e efetiva. Contudo, com a manutenção e evolução contínua do código, a execução de um vasto conjunto de testes manuais torna-se progressivamente cansativa e dispendiosa [Bernardo e Kon 2008].

Para enfrentar esse desafio, garantindo a efetividade e reduzindo o tempo gasto na etapa final do desenvolvimento de software são introduzidos os testes automatizados, que podem diminuir em até 80% do tempo gasto durante a execução dos testes. [Fewster e Graham 1999]. Contudo, à medida que o sistema cresce, e com isso a quantidade de testes automatizados desenvolvidos escala consideravelmente, o tempo de execução dos mesmos tende a aumentar significativamente, impactando diretamente na eficiência das

equipes de desenvolvimento. Emery (2009) ainda destaca que um dos principais motivos para que equipes de desenvolvimento deixem de lado a inserção de testes automatizados está no fato de que testes tendem a se tornar frágeis e de alto custo.

Dessa forma, torna-se cada vez mais necessário buscar estratégias que possam otimizar o tempo de execução de testes automatizados e ainda manter a confiabilidade do processo de validação do software. Com isso em mente, este trabalho apresenta uma proposta de diminuição do tempo despendido durante a execução de testes automatizados.

1.1. Problema

Um dos principais obstáculos atuais das equipes de desenvolvimento é a execução excessivamente longa de testes automatizados, impactando diretamente na eficácia dos ciclos de integração contínua. Nesses ambientes, onde ocorre a busca por entrega contínua e rápida de valor, o tempo do feedback é fundamental. Segundo Humble e Farley (2010), um dos princípios fundamentais da entrega contínua é obter feedback rápido a partir da execução de testes automatizados, permitindo que falhas sejam detectadas e corrigidas rapidamente.

Além disso, processos de testes automatizados ineficientes podem impactar negativamente no tempo total de entrega de um projeto, o que aumenta o lead time do projeto e reduz a capacidade das equipes ágeis em responder rapidamente a mudanças nos requisitos de negócio [Forsgren, Humble e Kim 2018].

1.2. Objetivo Geral

O objetivo geral deste trabalho é propor a redução do tempo de execução de testes automatizados utilizando o framework JUnit através da reutilização das configurações implícitas de cada classe de teste quando possível, ocorrendo de forma automatizada e sem acarretar na alteração do resultado esperado durante a execução de cada teste.

1.3. Objetivos Específicos

- Identificação dos testes automatizados que podem reutilizar as configurações implícitas da classe de teste;
- Projeto de uma ferramenta que reutiliza a configuração implícita de testes de unidade;
- Desenvolvimento da ferramenta proposta que estende JUnit 5.

2. Fundamentação Teórica

2.1. Testes de software

O teste de software tem por objetivo verificar se o produto que foi projetado está funcionando corretamente, revelando falhas que devem ser resolvidas pelo time de desenvolvimento antes de entrar em na fase de produção [Dias Neto 2007]. As atividades de teste de software não devem se restringir apenas ao produto final, elas devem ser executadas desde a sua concepção para garantir que o software está sendo construído de acordo com o especificado [Delamaro, Jino e Maldonado 2013].

Vasconcelos et al. (2006) ressalta que a fase de teste não garante que o software está livre de falhas, pois só é possível garantir com eles que o software sob teste contém erros. Quando os testes não encontram erros haverá sempre duas possibilidades: a qualidade do software é aceitável ou os testes foram inadequados. Dias Neto (2007) por sua vez, afirma que em virtude da sua natureza de revelar falhas em softwares, os testes de software são de natureza destrutiva e não construtiva, onde a confiança em um produto é sustentada pela revelação de suas falhas antes da entrega final. Além disso, testes de software não ajudam apenas na detecção de falhas antes da entrega do produto final, mas também contribui para a melhoria contínua da qualidade do código do software [Sommerville 2011].

Fowler (2006) ainda destaca sobre a importância do teste de software não como sendo um custo adicional inerente ao projeto, mas sim um investimento estratégico que garante a sustentabilidade, a competitividade e a reputação do produto e da equipe de desenvolvimento dentro do mercado, e ainda defendendo a integração contínua dos mesmos em todas as fases do desenvolvimento.

2.2. Testes automatizados

Meszaros (2007) define o teste automatizado como qualquer interação com o SUT (system under test ou sistema sob teste) feita sem qualquer intervenção manual, sendo eles ainda autoverificáveis e repetíveis, ou seja, quando os resultados esperados estão corretos e produzem o mesmo resultado sempre que repetidos. Barbosa e Torres (2011) ainda definem a automação de testes como a transcrição de testes manuais em linhas de código que podem ser lidos e executados por softwares. Bartié (2002) ressalta que a automação pode simular atividades manuais, de forma a evitar o contato humano durante a execução dos testes, não exigindo mão de obra especializada.

Testes automatizados ajudam a controlar a qualidade do software de forma eficaz e apresentam um baixo custo ao longo do tempo [Bernardo e Kon 2008]. Outra vantagem dos testes automatizados, em comparação com os testes manuais, está na rapidez de sua execução e na consistência de seus resultados, garantindo uma integração contínua de código em produção [Pressman e Maxim 2021]. Outra característica positiva dos testes automatizados está no fato de que testes automatizados possuem precisão e consistências elevadas, eliminando a variabilidade associada à execução manual, e também permitem execução frequente de testes de regressão, o que garante que a correção de bugs ou o desenvolvimento de código novo não introduz falhas em partes já existentes do código fonte do sistema [Myers, Sandler e Badgett 2011].

2.3. Testes de unidade

Também conhecido como teste unitário, é o teste da menor unidade a ser testada dentro de um software (módulos ou componentes) e normalmente é executado pelo próprio desenvolvedor [Silva 2008]. Sendo assim, pode-se dizer que é o primeiro nível de teste de software, uma vez que as unidades do produto são testadas isoladamente [Medeiros 2020]. Seu objetivo visa testar as menores unidades do sistema, tais como métodos, funções, procedimentos e classes, limitando-se a lógica interna do código e onde espera-se encontrar com maior facilidade falhas de programação [Júnior, Andrade e Delamaro 2016]. Bernardo e Kon (2008) apontam o teste de unidade, para a maioria das

situações, como um dos níveis de teste de maior importância, uma vez que garante que o código encapsulado por uma unidade produz o efeito esperado, além dos erros serem facilmente localizados e reparados.

Pela sua característica de verificar apenas uma classe ou método, sua implementação costuma ser fácil e comumente aplicada pelos próprios desenvolvedores, que são os que mais tiram proveito de sua aplicação [Naik e Tripathy 2008]. Dentre as qualidades de um teste unitário, Meszaros (2007) relata que devem ser independentes e não depender de outros testes ou do estado global do sistema e devem ser determinísticos e sempre apresentar os mesmos resultados quando os mesmos dados de entradas são aplicados. A presença de testes de unidades dentro de sistema costuma deixar a equipe de desenvolvedores mais confiante ao realizar mudanças no código, já que problemas durante a sua execução podem ser facilmente detectados, gerando mais confiabilidade, reduzindo a quantidade de bugs e facilitando a refatoração e compreensão do código [Fowler 2006].

2.3. Configuração de testes

Greiler et al. (2013) afirmam que um dos requisitos para o funcionamento de um teste automatizado é a configuração de todas as dependências para colocar o SUT no estado desejado para execução dos testes. Para isso, Meszaros (2007) divide as estratégias de configuração inicial em 3 padrões: configuração local, configuração implícita e configuração delegada.

A estratégia de configuração local ocorre quando as configurações do teste estão presentes dentro do próprio método de teste e acaba ocorrendo em testes iniciais ou em cenários muito específicos em que houve a decisão de se duplicar a configuração do teste. Uma das principais consequências dessa estratégia é a duplicação de código, pois os métodos de teste que podem utilizar a mesma configuração inicial devem incluir a configuração inicial dentro do seu próprio método de teste.

Na estratégia de configuração implícita a configuração inicial de teste fica localizada em um método especial, mas ainda dentro da mesma classe de teste, e assim essas configurações podem ser reutilizadas em todos os métodos de teste localizados na mesma classe. A dificuldade nesta estratégia está em agrupar um número grande de testes que utilizem exatamente a mesma configuração inicial, isso devido à particularidade que cada teste pode ter.

Na estratégia de configuração delegada a configuração inicial de teste costuma ficar em um arquivo separado das classes de teste e, assim, pode ser compartilhado por mais de uma classe de teste. Nesta estratégia a principal dificuldade está na dificuldade de compreensão do resultado de alguns métodos de teste, uma vez que a configuração inicial fica em uma classe separada.

3. Trabalhos relacionados

O presente capítulo tem por objetivo apresentar trabalhos presentes na literatura correlatos ao tema proposto neste trabalho. Serão abordados trabalhos que foram encontrados através de uma pesquisa na base de dados do Scopus. Com a finalidade de

se obter resultados mais atualizados e refinados, a pesquisa foi realizada no dia 26/06/2025 e para a busca foi utilizada a seguinte string:

("unit test" OR "unit testing" OR "unit-test") AND ("test fixture" OR "test method" OR "test suite" OR "setup execution") AND ("reuse" OR "reusing" OR "reducing" OR "reduce").

A pesquisa obteve um total de 58 resultados, listados no Apêndice A. A partir disso então, uma nova seleção dos trabalhos se deu através da leitura do título de cada resultado e os que apresentavam uma proximidade maior com o tema deste trabalho foram selecionados para leitura do resumo. Assim foram selecionados 9 trabalhos apresentados a seguir na Tabela 1.

Tabela 1. Trabalhos relacionados selecionados após leitura do título

Título	Autor(es)	Ano
E2E test execution optimization for web application based on state reuse	Ruilian Zhao, Shukai Zhang, Zhifan Zhu, Ying Shang, Weiwei Wang	2025
ATSM: A coverage-based framework and a tool for test suite minimization	Alekseev, Y., Onischuck, M., Zorin, A., Chernyi, V., Ilyin, E., Itsykson, V.	2024
SolUnit: A framework for reducing execution time of smart contract unit tests	Hallan Medeiros, John Mylopoulos, Patrícia Vilain, Hans-Arno Jacobsen	2020
Reuse of fixture setup between test classes	Lucas Silva, Patrícia Vilain	2017
Multi-objective Regression Test Suite Minimisation for Mockito	Andrew J. Turner, David R. White & John H. Drake	2016
Unit test virtualization with VMVM	Jonathan Bell, Gail Kaiser	2014
Reusing and Composing Tests with Traits	Stéphane Ducasse, Damien Pollet, Alexandre Bergel & Damien Cassou	2009
A Unit-Test Framework for Database Applications	Claus A. Christensen; Steen Gundersborg; Kristian de Linde; Kristian Torp	2006
GridUnit: Software testing on the grid	Alexandre Duarte, Walfredo Cirne, Francisco Brasileiro, Patrícia Machado	2006

Após a leitura dos resumos, foram selecionados 5 trabalhos que apresentavam ferramentas propostas para redução do tempo de execução de testes automatizados, e assim foram selecionados os seguintes trabalhos: Silva e Vilain (2017), apresentando a ferramenta Estoria com a reutilização do configuração implícita da classe de teste; Medeiros et al. (2020), apresentando o framework SolUnit que propõe a redução do tempo de execução de testes de smart contract; Duarte et al. (2006), com a ferramenta GridUnit que utiliza a paralelização como parte da sua proposta; Christensen et al. (2006), com a ferramenta Unit-Test Framework for Database, destinada a otimizar a execução de testes com banco de dados; Zhao et al. (2025) apresentando uma abordagem para acelerar a execução de testes E2E.

3.1. Estoria

A ferramenta Estoria [Silva e Vilain 2017] é uma extensão do JUnit em Java que otimiza a execução de testes automatizados ao reutilizar o código de configuração implícita dentro das classes de teste. Isso é feito através da anotação @Safe, onde o testador marca os métodos que não alteram o estado inicial do Sistema Sob Teste (SUT),

garantindo que esses "testes seguros" sejam executados primeiro sem criar dependências ou inconsistências, seguidos pelos "testes não seguros". Experimentos demonstraram que Estoria pode reduzir o tempo de execução dos testes automatizados em até oito vezes. Contudo, a eficácia da ferramenta pode ser limitada em configurações complexas, exigindo adaptações adicionais ou aumentando a complexidade inicial do projeto com classes auxiliares de configuração. A principal diferença para outras abordagens é a dependência da marcação manual pelo desenvolvedor via @Safe para a reutilização da configuração implícita.

3.2. SolUnit

SolUnit [Medeiros 2020] é um framework em Java que estende o JUnit para otimizar o desempenho de testes de unidade em smart contracts na plataforma Ethereum, utilizando reuso automatizado da implantação do contrato e configurações implícitas de teste. O processo envolve a análise estática do código-fonte para classificar métodos de contrato e métodos de teste como "seguros" (não geram transações blockchain) ou "não seguros". Em seguida, os testes seguros são executados primeiro com uma única implantação de contrato e configuração, aproveitando o reuso, e só então os testes não seguros são executados individualmente. Experimentos com cinco projetos mostraram uma redução significativa no tempo de execução, tanto em blockchains emuladas (reduções entre 34,71% e 68,85%) quanto em uma blockchain Ethereum real (reduções entre 38,08% e 68,17%). Apesar de similar em abordagem de reuso, o SolUnit foca exclusivamente em ambientes de blockchain, diferentemente da proposta deste trabalho, que visa a redução do tempo de execução de testes automatizados em projetos que não utilizam blockchain.

3.3. GridUnit

O GridUnit [Duarte et al. 2006] é uma extensão do framework JUnit projetada para distribuir automaticamente a execução de testes automatizados em uma grade computacional. O objetivo central é paralelizar a execução dos testes para reduzir significativamente o tempo de conclusão, operando com mínima intervenção manual do usuário para facilitar a integração em fluxos de trabalho existentes. Experimentos demonstraram uma redução de até 70 vezes no tempo de execução, otimizando o processo de testes, e sem exigir conhecimento aprofundado em computação distribuída ou modificações no código-fonte da aplicação. No entanto, é importante notar que o GridUnit depende de uma infraestrutura que suporte o paralelismo e a execução múltipla de testes, o que o diferencia de propostas que não exploram o paralelismo para a execução de testes automatizados.

3.4. Unit-Test Framework for Database

O framework Unit-Test Framework for Database [Christensen et al. 2006] foi desenvolvido para lidar com a complexidade dos testes em aplicações que interagem com bancos de dados, onde as constantes alterações nas regras de negócio impactam diretamente a estrutura do banco e, consequentemente, o estado dos dados, que pode influenciar os resultados dos testes subsequentes. A ferramenta foca nos testes automatizados do código-fonte da aplicação que acessa o banco de dados, permitindo a

reutilização estruturada de dados inseridos por outros testes, o que elimina a necessidade de recriar o estado do banco a cada execução. A abordagem exige a declaração de dependências entre as classes de teste, refletindo as dependências das entidades do banco de dados, e executa os testes de forma recursiva com base nessas dependências. Aplicado em um projeto de médio porte, o framework demonstrou um aumento de 40% na velocidade de execução e uma redução estimada de 25% no código de teste, sendo uma solução específica para sistemas com interação com banco de dados, diferentemente do escopo do presente trabalho.

3.5. E2E test execution optimization for web application based on state reuse

O trabalho de Zhao et al. (2025) apresenta uma otimização para testes E2E em aplicações web focadas na reutilização de estados do sistema. Essa abordagem é crucial para reduzir o tempo de execução de testes E2E, que são frequentemente demorados, especialmente em aplicações web complexas. O framework proposto utiliza uma estrutura de árvore de prefixo para identificar e gerenciar a reutilização de estados entre diferentes cenários de teste, eliminando a necessidade de reconfigurar o ambiente para cada execução e maximizando a utilização de estados de página acionados por operações comuns, o que otimiza o tempo total. A pesquisa quantifica o impacto da reutilização, demonstrando que uma maior proporção de operações reutilizadas pode gerar ganhos significativos de tempo. Os autores indicam que as otimizações visam a eficiência da suíte de testes completa, eliminando setups redundantes, em vez de reduzir o tempo de testes individuais. Experimentos com 347 testes E2E em oito aplicações web open source mostraram reduções de tempo de execução entre 52% e 68%. Contudo, o foco exclusivo em testes E2E em fases posteriores do desenvolvimento diferencia-o da proposta deste trabalho, que se concentra em testes unitários nas fases iniciais.

4. Proposta

4.1. Visão geral da proposta

O objetivo deste trabalho é reduzir o tempo de execução de testes automatizados em projetos Java que utilizam JUnit 5, baseando-se no princípio de que métodos de teste são independentes e podem ter sua ordem de execução alterada. A proposta envolve a criação de uma extensão para o JUnit que reutiliza a execução do `setUp` inicial (`@BeforeEach`) de cada classe de teste. Isso será feito através da construção de uma extensão para o JUnit que reusa a execução da configuração inicial (`@BeforeEach`) de cada classe de teste. O reuso dessas configurações é viável quando um método de teste não altera os atributos da classe, evitando assim a violação do princípio de independência dos testes.

A execução da configuração implícita (`setUp`) de uma classe de teste ocorrerá uma vez. Em seguida, todos os métodos de teste que não modificam as variáveis inicializadas nesse `setUp` serão executados. Após essa fase, será executado o primeiro método de teste que altera as variáveis, então precedido pela execução do `setUp` novamente e por outro método de teste que altera as variáveis do `setUp` inicial, e assim sucessivamente até finalizar todos os métodos de teste. A figura 1 ilustra o fluxograma da proposta. A proposta deste trabalho se concentra em identificar automaticamente

esses métodos que alteram o estado do setUp e, com base nisso, ordenar a execução dos testes dentro de cada classe, visando a redução do tempo total de execução do projeto.

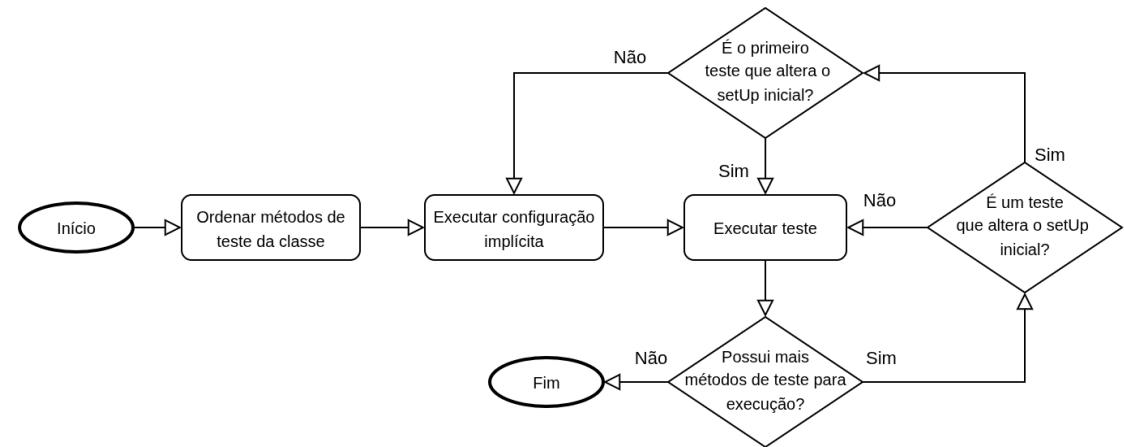


Figura 1. Fluxograma de reutilização da configuração implícita

4.2. Etapas da proposta

A ferramenta proposta realiza quatro etapas durante a sua execução conforme ilustrado na Figura 2: (1) análise do código fonte do projeto; (2) identificação dos métodos que alteram o estado dos atributos dentro do projeto; (3) classificação dos métodos de testes em seguros e não seguros; e (4) ordenação e execução dos métodos de testes.

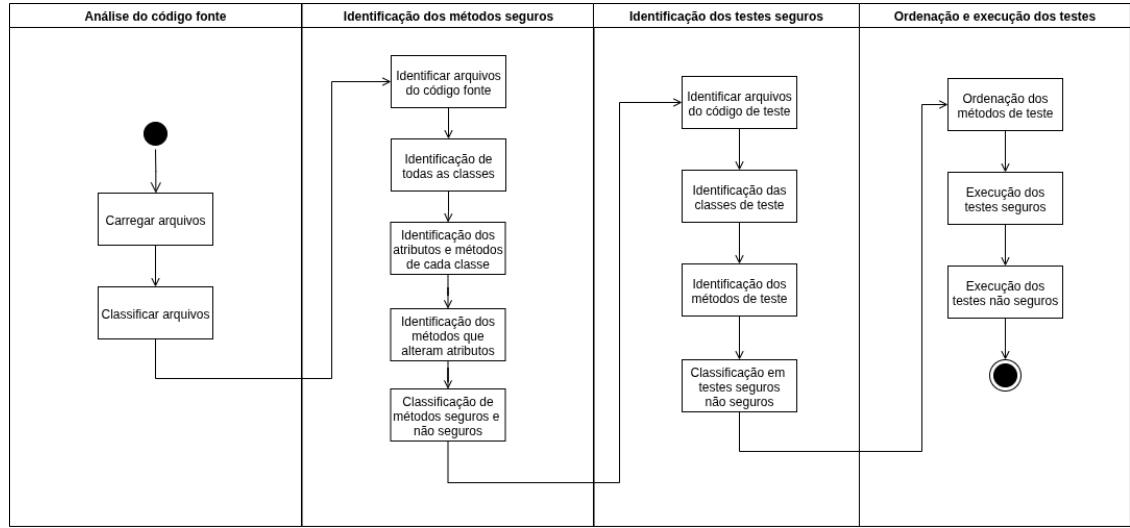


Figura 2. Diagrama de atividades da proposta

4.2.1. Análise do código fonte

A primeira etapa tem por objetivo carregar todos os arquivos do código fonte do projeto e identificar quais arquivos compõem o código principal do projeto e quais arquivos compõem o código de teste do projeto.

4.2.2. Identificação dos métodos seguros

A segunda etapa busca identificar todos os atributos e métodos presentes em cada classe do código principal do projeto. Estes métodos são classificados como métodos seguros e métodos não seguros. Os métodos seguros são aqueles que não irão alterar os atributos das classes sendo testadas, enquanto os métodos não seguros são aqueles que podem realizar alguma alteração nos atributos destas classes. A ideia de classificação de métodos em seguros e não seguros foi proposta por Silva e Vilain (2021)

A Figura 3 ajuda a ilustrar como os métodos não seguros são identificados. A classe Funcionario possui um atributo chamado salario e os métodos getSalario e setSalario, onde o método getSalario retorna o valor atual do salario do Funcionario, e assim sendo classificado como um método seguro, enquanto o método setSalario altera o valor atual do salario do funcionario, logo sendo classificado como um método não seguro.



```
1 public class Funcionario {
2
3     private float salario;
4
5     public Funcionario (float salario) {
6         this.salario = salario;
7     }
8
9     public float getSalario() {
10         return this.salario;
11     }
12
13     public void setSalario(float novoSalario) {
14         this.salario = novoSalario;
15     }
16
17 }
```

Figura 3. Exemplo da classe Funcionario

4.2.3. Identificação dos testes seguros

Na terceira etapa é realizada a análise estática do código de teste do projeto. O objetivo desta etapa é localizar e classificar todos os métodos de teste, presentes em todas as classes de teste. Estes métodos de teste são representados pela notação @Test e estes métodos são classificados em métodos de teste seguros e não seguros. Essa classificação é realizada com base nos métodos seguros e não seguros identificados na segunda etapa da ferramenta. Um método de teste seguro apresenta em seu código a utilização de apenas métodos seguros do código principal, conforme classificado na segunda etapa da ferramenta (Seção 4.2.2). Caso o método de teste apresente em seu código um método não seguro, conforme explicado na segunda etapa da ferramenta (Seção 4.2.2), esse método de teste é classificado como um método de teste não seguro.

A Figura 4 ilustra essa classificação de métodos de teste seguros, onde a classe de teste FuncionarioTest apresenta dois métodos de teste. O método deveRetornarSalarioCorretoDoFuncionario utiliza apenas o método seguro getSalario, logo esse método de teste é classificado como seguro pois não altera o estado atual do SUT. Por outro lado, o método deveAlterarCorretamenteSalarioFuncionario utiliza o

método seguro `getSalario` e também o método não seguro `setSalario`, sendo classificado, então, como não seguro, pois altera o estado do SUT.

```
1 @ExtendWith(SunitRunner.class)
2 @TestMethodOrder(SunitRunner.class)
3 public class FuncionarioTest {
4
5     private static Funcionario funcionario;
6     private static float salario;
7
8     @BeforeEach
9     public void setUp() {
10         this.salario = 1500;
11         this.funcionario = new Funcionario(this.salario);
12     }
13
14     @Test
15     public void deveRetornarSalarioCorretoDoFuncionario() {
16         Assertions.assertEquals(this.funcionario.getSalario(), this.salario);
17     }
18
19     @Test
20     public void deveAlterarCorretamenteSalarioFuncionario() {
21         float novoSalario = 2000;
22         this.funcionario.setSalario(novoSalario);
23         Assertions.assertEquals(this.funcionario.getSalario(), novoSalario);
24     }
25
26 }
```

Figura 4. Exemplo da classe FuncionarioTest

4.2.4. Ordenação e execução dos métodos de testes

A última etapa da ferramenta ordena a execução dos métodos de teste dentro de cada classe de teste, executando primeiro os métodos de teste seguros e em seguida os métodos de teste não seguros, conforme classificados na terceira etapa da ferramenta (Seção 4.2.3). É nesta etapa que ocorre o reuso das configurações implícitas de cada classe de teste.

A execução dos métodos de teste ocorre na seguinte sequência:

1. Primeiro ocorre uma execução do `setUp` inicial da classe de teste e, em seguida, são executados todos os métodos de teste seguros desta classe de teste.
2. Em seguida é executado o primeiro método de teste não seguro; para esta execução ainda não é necessário executar novamente o `setUp` inicial, uma vez que o SUT ainda não foi alterado.
3. Após a execução do primeiro teste não seguro, é executado o `setUp` inicial da classe de teste novamente e o próximo método de teste não seguro. Esta sequência é repetida até que todos os testes não seguros sejam executados.

Desta forma, a presente proposta permite a diminuição do tempo de execução dos testes automatizados por não executar sempre o `setUp` inicial de cada classe de teste antes de cada método de teste, uma vez que testes não seguros não alteram o estado inicial da configuração implícita de teste.

5. Implementação

A ferramenta SUnit foi desenvolvida como uma extensão do framework de automação de testes JUnit 5 e pode ser encontrada no repositório GitHub (<https://github.com/engdough/tcc-gilmar>). É importante destacar que o SUnit foi

adaptado do SolUnit, um framework que estende o JUnit 4 e tem como objetivo reduzir o tempo de execução de testes de smart contracts em blockchains. Foi reaproveitada a estrutura em geral do projeto, como nome de classes, arquitetura dos arquivos, utilização da biblioteca do JavaParser e a estratégia para leitura dos arquivos e análise estática do código. O SolUnit utilizava a versão 4 do JUnit e para o SUnit foi realizado o upgrade de versão, e utilizado a versão 5 do framework.

O SUnit foi desenvolvido na linguagem de programação Java e projetado para reduzir o tempo de execução de testes unitários. Para estender o JUnit, a ferramenta utiliza as anotações básicas do framework durante o ciclo de vida de execução de cada teste: `@Test` para indicar todos os métodos de teste, `@BeforeEach` para as configurações implícitas que devem ser executadas antes de cada método de teste, `@ExtendWith` para indicar que a classe de teste estenderá o SUnit, e `@TestMethodOrder` para indicar que a ordenação dos métodos de teste seguirá a proposta do SUnit. Além disso, embora a ferramenta permita a identificação automática de testes seguros, a anotação `@Safe` apresentada por Silva e Vilain (2017) e adotada por Medeiros (2020) foi mantida para indicar manualmente que o método de teste é seguro e pode reutilizar as configurações implícitas do teste sem comprometer sua execução.

5.1. Diagrama de Classes

A Figura 5 apresenta o diagrama de classes da ferramenta SUnit. A classe `SunitRunner` é responsável por estender o framework JUnit, pela execução de testes, por instanciar a classe `SafeParser`, por interceptar a ordenação dos métodos de teste através da interface `MethodOrderer`, e também por interceptar a interface `InvocationInterceptor`, que executa todas as configurações implícitas de teste através do uso da anotação `@BeforeEach` e decide se as configurações implícitas serão executadas para cada teste.

A classe `SafeParser` consome a classe `JavaASTparser` previamente instanciada e também é utilizada pela classe `SafeMethodSorter`. A classe `SafeParser` é responsável por injetar o caminho do diretório da classe principal e o diretório de testes do projeto em uso. A classe `SafeMethodSorter` é responsável pela ordenação dos métodos de teste, onde os métodos de teste seguros são executados primeiro, seguidos pelos métodos de teste classificados como não seguros.

Finalmente, a classe `JavaASTparser` é responsável pela leitura dos arquivos estáticos do código principal através da classe `ProjectClassFinder`, pela leitura do código de teste através da classe `TestRunnerClassFinder`, pela classificação de métodos como seguros e não seguros através da classe `ProjectNotSafeMethodFinder`, e pela classificação dos métodos de teste como seguros e não seguros através da classe `TestRunnerSafeMethodFinder`.

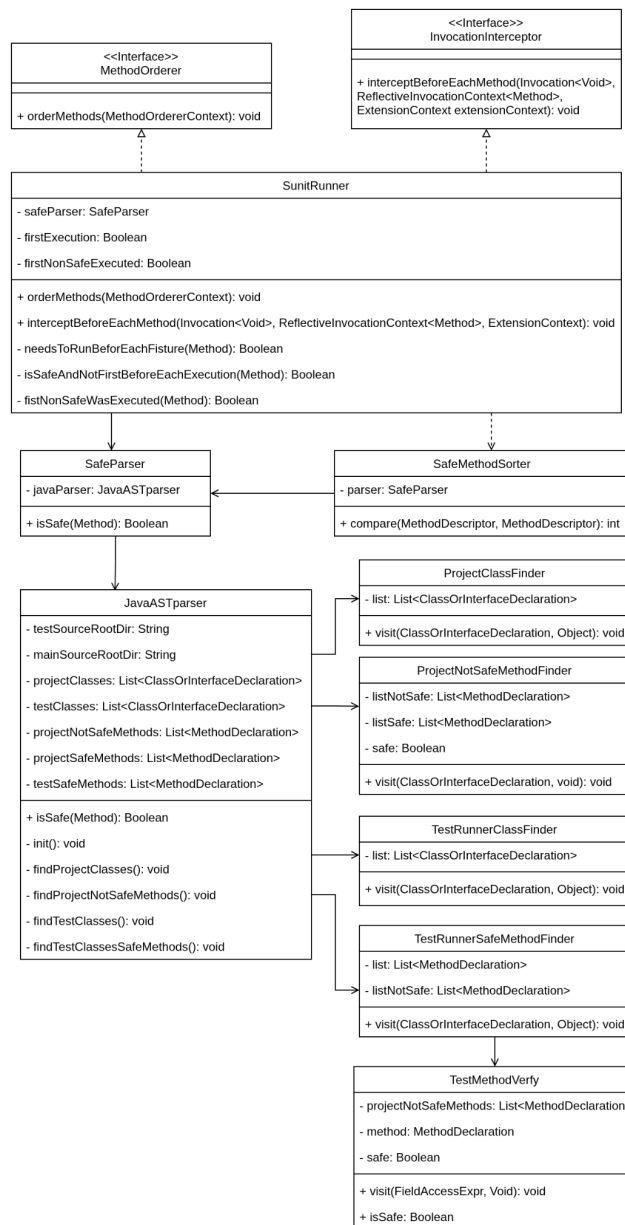


Figura 5. Diagrama de classes do SUnit

5.2. Análise do código fonte

Conforme indicado na Seção 4, a primeira fase de implementação visa identificar os arquivos de código-fonte do projeto a ser testado. Esta fase começa com a injeção dos caminhos de diretório das classes principais e das classes de teste do projeto pela classe `SafeParser`. Esses caminhos são obtidos através de um arquivo de configuração chamado "sunit.properties". A Figura 6 ilustra um exemplo do conteúdo do arquivo de configuração, onde na primeira linha de código é especificado o caminho do diretório dos arquivos de código principal do projeto, e na segunda linha é especificado o caminho do diretório dos arquivos de código de teste.



```
1 main.src.dir=src/main/java
2 test.src.dir=src/test/java
```

Figura 6. Arquivo de configuração “sunit.properties”

É importante que os caminhos dos diretórios de teste e principal informados no arquivo de configuração estejam corretos, uma vez que a classe SafeParser obtém todos os arquivos Java dos diretórios especificados neste arquivo de configuração.

5.3. Identificação dos métodos das classes seguros e não seguros

Nesta fase, é realizada a análise do código principal do projeto, com o propósito de identificar os métodos de cada classe que modificam o valor dos atributos das classes às quais pertencem, conforme mencionado anteriormente. Primeiramente, todos os arquivos localizados no diretório do código principal do projeto, indicado no arquivo de configuração, são identificados. Em seguida, os arquivos são filtrados, sendo utilizados apenas aqueles com a extensão .java. Por fim, dos arquivos filtrados, apenas aqueles que representam uma classe ou interface em Java são selecionados para realizar a análise e classificação dos métodos como seguros e não seguros. Essa identificação dos arquivos da classe principal é realizada pelo método findProjectClasses da classe JavaASTparser.

Após o carregamento dos arquivos das classes principais do projeto, os métodos de todas as classes carregadas são classificados em métodos seguros e não seguros. A classificação dos métodos é possível com a ajuda da biblioteca JavaParser, que transforma todos os arquivos de classe principal selecionados em uma Árvore de Sintaxe Abstrata (AST). Um exemplo dessa árvore é mostrado na Figura 7, que representa a AST gerada a partir do exemplo da classe Funcionario mostrada na Figura 3. O método findProjectNotSafeMethods da classe JavaASTparser é responsável por realizar essa classificação.

Também na Figura 7, é possível observar que a árvore gerada para a classe Funcionario produziu cinco nós: o primeiro representando o nome da classe, o segundo representando o atributo da classe, o terceiro indicando o construtor da classe, e os dois últimos nós folha representando os métodos getter e setter da classe. Cada nó gerado pela árvore de classe pode ter nós filhos de acordo com suas características, e cada classe pode gerar vários nós dependendo do número de atributos, construtores e métodos que contém.

A partir da árvore formada, os métodos são classificados como seguros ou não seguros verificando se os nós dos métodos contêm alguma função que modifica o valor do nó de atributo; se houver uma modificação do valor de um nó de atributo, o método é considerado não seguro.

Além de verificar se os métodos alteram os atributos da própria classe, é necessário verificar se os métodos utilizam outros métodos de classes diferentes que foram previamente considerados não seguros. Adicionalmente, se o método que está sendo analisado é não seguro, é preciso verificar se métodos seguros previamente

classificados estão utilizando o método em questão; se sim, eles devem ser reclassificados como métodos não seguros. Portanto, um método é considerado não seguro se ele modifica o valor do atributo de sua própria classe e/ou utiliza métodos de outras classes considerados não seguros.

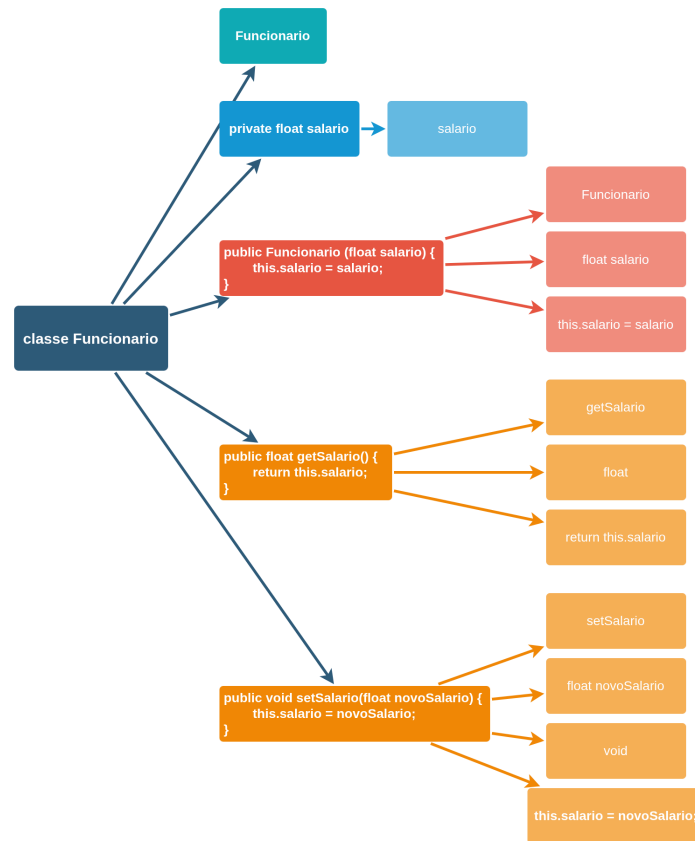


Figura 7. Árvore de sintaxe abstrata da classe Funcionario

5.4. Identificação dos testes seguros e não seguros

Analogamente à fase anterior, a classificação dos testes como seguros e não seguros é feita através da análise do código de teste do projeto. Todos os arquivos localizados no diretório do código de teste do projeto, conforme indicado no arquivo de configuração, são identificados. Mais uma vez, esses arquivos são filtrados, e somente os arquivos com a extensão .java são considerados. Finalmente, dos arquivos filtrados, apenas aqueles que contêm a anotação @ExtendWith("SunitRunner.class") em seu código são selecionados. Essa identificação dos arquivos de classe de teste é realizada pelo método findTestClasses da classe JavaASTparser.

Após o carregamento de todos os arquivos de teste, inicia-se a classificação dos métodos de teste. A biblioteca JavaParser é utilizada novamente para transformar todos os arquivos em ASTs. A Figura 8 mostra a AST gerada a partir do exemplo da Figura 4. É possível observar que a classe de teste FuncionarioTest gerou oito nós: os dois primeiros representam as anotações usadas na implementação deste trabalho, o terceiro representa o nome da classe de teste, o quarto e o quinto nós representam os atributos usados nesta classe de teste, o sexto nó representa a configuração de teste implícita, e os

dois últimos nós representam os métodos de teste implementados dentro da classe. O método `findTestClassesSafeMethods` da classe `JavaASTparser` é responsável por realizar essa classificação.

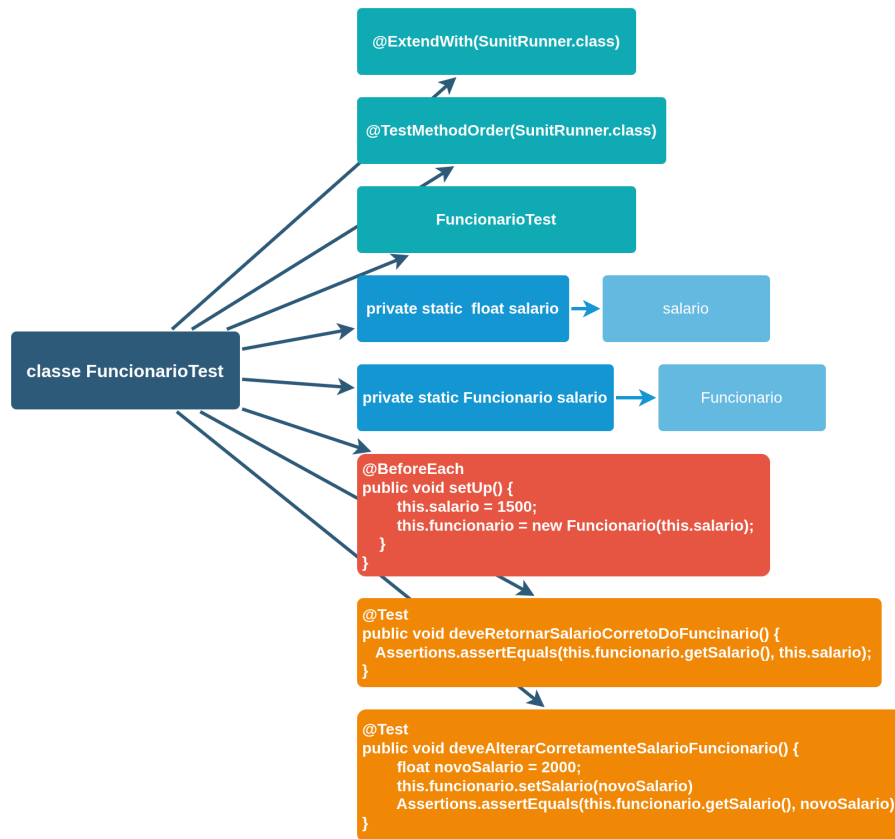


Figura 7. Árvore de sintaxe abstrata da classe FuncionarioTest

A partir da árvore gerada, os métodos de teste são classificados como seguros ou não seguros verificando se os nós que contêm os métodos de teste usam pelo menos um dos métodos classificados como não seguros. Assim, se houver pelo menos um método de produção não seguro, o teste é classificado como não seguro; caso contrário, o método de teste é classificado como seguro. A classe `TestMethodVerify` é chamada pelo método `findTestClassesSafeMethods` da classe `JavaASTparser`. A classificação dos métodos de teste como seguros e não seguros é realizada para cada método de teste e é executada na classe `TestMethodVerify` verificando se o método de teste chama algum método não seguro. Além disso, o método também verifica se as variáveis presentes na classe de teste e os atributos das classes a serem testadas são modificados.

5.5. Ordenação e execução dos métodos de teste

Depois que todos os métodos de teste são classificados como seguros e não seguros, a fase de ordenação e execução começa. Esta fase é possível através do uso da anotação `@TestMethodOrder`, que permite a interceptação da ordenação padrão do JUnit 5 e a implementação da ordem desejada. No SUnit, a ordenação ocorre na classe `SafeMethodSorter`, que ordena os métodos de teste de forma que os métodos de teste seguros sejam executados primeiro, seguidos pelos métodos de teste não seguros. A

ordenação dos métodos de teste ocorre de forma independente dentro de cada classe de teste.

A Figura 8 mostra o trecho de código onde ocorre a interceptação do método de ordenação do JUnit 5, juntamente com o método que realiza a ordenação dos métodos de teste. A linha 1 da Figura 8 mostra a interceptação do método `orderMethods` no JUnit, onde o método é sobrescrito e a ordenação é realizada usando a classe `SafeMethodSorter()`. A linha 7 da Figura 8 mostra a própria classe de ordenação, que se assemelha ao algoritmo Bubble Sort, onde o método itera em pares pelos métodos de teste dentro da classe de teste e troca suas posições se o método `m1` for não seguro e o método `m2` for seguro, conforme mostrado na linha 15. O método percorre cada par de métodos de teste dentro da classe de teste até que não ocorram mais mudanças de posição.



```
1  @Override
2  public void orderMethods(MethodOrdererContext context) {
3      Collections.sort(context.getMethodDescriptors(), new SafeMethodSorter());
4  }
5
6
7  public class SafeMethodSorter implements Comparator<MethodDescriptor> {
8
9      SafeParser parser;
10
11      public SafeMethodSorter() {
12          this.parser = new SafeParser();
13      }
14
15      public int compare(MethodDescriptor m1, MethodDescriptor m2) {
16          boolean m1Safe = parser.isSafe(m1.getMethod());
17          boolean m2Safe = parser.isSafe(m2.getMethod());
18
19          if (m1Safe && !m2Safe) {
20              return -1;
21          }
22          else if (!m1Safe && m2Safe) {
23              return 1;
24          }
25          return 0;
26      }
27  }
```

Figura 8. Ordenação dos métodos de teste

A Figura 9 mostra o trecho de código onde ocorre a interceptação do método que decide se a configuração de teste implícita, representada pela anotação `@BeforeEach`, será ou não executada. A linha 1 da Figura 9 mostra a interceptação do método `interceptBeforeEachMethod` do JUnit, e a configuração de teste implícita é executada através do comando `invocation.proceed()` na linha 8 e ignorada através do comando `invocation.skip()` na linha 16.

Na Figura 9, também é possível observar que a execução da configuração de teste implícita ocorre em três cenários, de acordo com a condição na linha 7, que aciona o método na linha 19. O primeiro cenário ocorre quando o primeiro método de teste a ser executado é um método seguro, o que significa que nenhuma configuração de teste implícita ainda foi executada, conforme verificado pela condição na linha 20, que chama o método na linha 36. Lá, a variável `firstExecution` é encontrada como verdadeira e é então alterada para falsa na linha 21.

```

1  @Override
2  public void interceptBeforeEachMethod
3  (Invocation<Void> invocation, ReflectiveInvocationContext<Method> reflectiveInvocationContext,
4  ExtensionContext extensionContext) throws Throwable {
5      Method method = extensionContext.getRequiredTestMethod();
6
7      if (this.needsToRunBeforeEachFixture(method)) {
8          invocation.proceed();
9          return;
10     }
11
12     if (!safeParser.isSafe(method)) {
13         this.firstNonSafeExecuted = true;
14     }
15
16     invocation.skip();
17 }
18
19 private boolean needsToRunBeforeEachFixture(Method actualMethod) {
20     if (isSafeAndNotFirstBeforeEachExecution(actualMethod)) {
21         this.firstExecution = false;
22         return true;
23     }
24
25     if (firstNonSafeWasExecuted(actualMethod)) {
26         return true;
27     }
28
29     if ( this.firstExecution ) {
30         return true;
31     }
32
33     return false;
34 }

```

Figura 9. Trecho de código que define a execução da configuração implícita

O segundo cenário ocorre durante a execução do segundo método de teste não seguro, uma vez que o primeiro método de teste não seguro não precisa executar a configuração de teste implícita, dado que ela já foi executada com os métodos de teste seguros. Isso é tratado pela condição na linha 25, que verifica se a variável `firstNonSafeExecuted` é verdadeira, conforme definido como verdadeiro após a execução do primeiro método não seguro na linha 13.

O cenário final ocorre quando não há métodos de teste seguros na classe, portanto, a configuração de teste implícita deve ser sempre executada. Isso é gerenciado pela condição na linha 29, onde a variável `firstExecution` será sempre verdadeira.

6. Experimentos

Para avaliar a ferramenta, foi desenvolvido um pequeno projeto para verificar se a ferramenta classifica corretamente os métodos das classes em seguros e não seguros e os métodos de testes em seguros e não seguros. Além disso, foi selecionado um projeto disponível no GitHub, o Roza, para verificar se o tempo de execução dos testes automatizados é de fato reduzido. Roza apresenta em seu repositório uma coleção de 81 classes de testes, totalizando 391 métodos de testes automatizados, desenvolvido em linguagem de programação Java e com o framework de teste JUnit.

6.1. Validação da classificação dos métodos de classe e testes

Para fazer a validação da classificação dos métodos das classes foram criadas três classes: `Funcionario`, `ReajusteService` e `BonusService`. Esse projeto apresenta um total de 10 métodos, sendo 5 deles seguros e 5 não seguros. O diagrama de classes deste projeto está ilustrado na Figura 10. Para avaliar a classificação dos métodos de teste em

seguro e não seguro, foi realizada a implementação de duas classes de teste, sendo elas: `FuncionarioTest` e `ReajusteServiceTest`. A classe `FuncionarioTest` apresenta um total de onze métodos de teste desenvolvidos, sendo cinco deles implementados como seguros e seis deles implementados como não seguros. Já a classe `ReajusteServiceTest` apresenta um total de três métodos de teste, sendo todos eles implementados como testes não seguros.

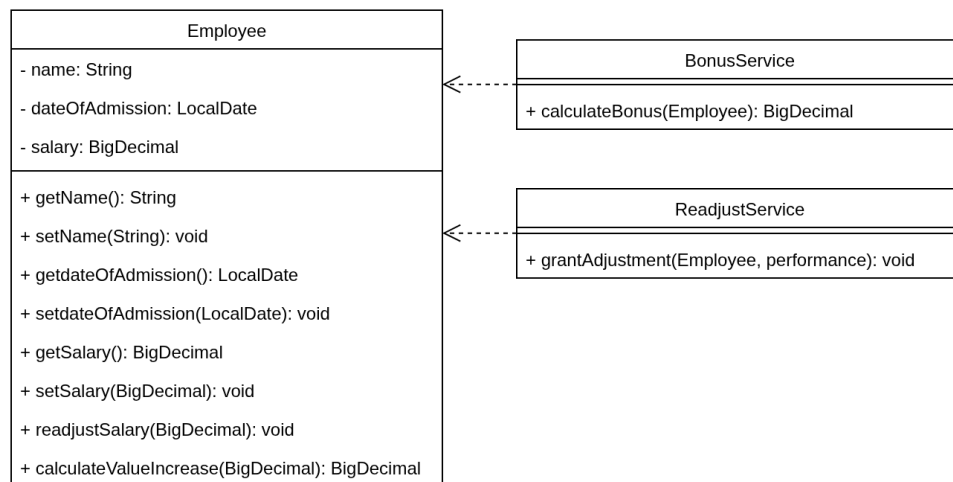


Figura 10. Diagrama de classe do projeto usado para validar o SUnit

A Figura 11 apresenta os resultados obtidos pela ferramenta SUnit no projeto de teste. A ferramenta conseguiu identificar os cinco métodos de classes que eram de fato não seguros e os cinco métodos de teste que eram considerados seguros.

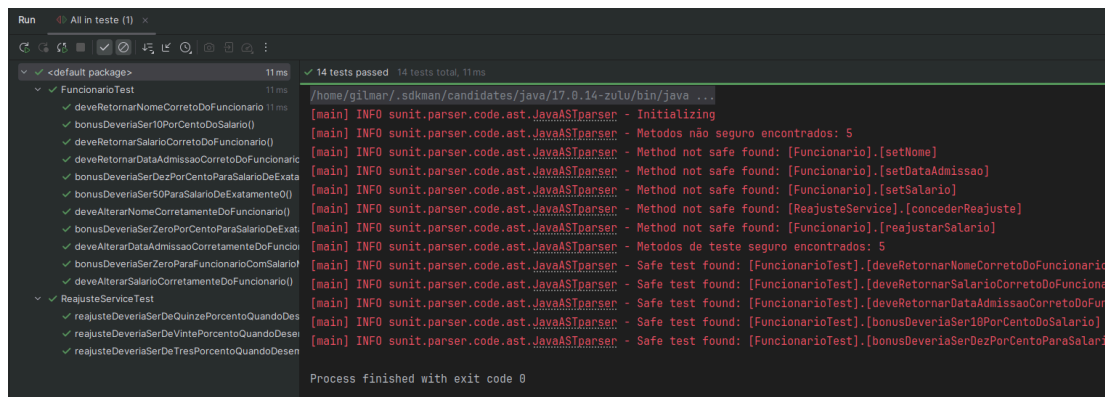


Figura 11. Resultados obtidos pelo SUnit no projeto criado pelo autor

6.2. Validação do SUnit em um projeto real

Neste experimento foi utilizado o Roza para aplicação da ferramenta deste trabalho. O Roza é um framework de refatoração automatizada de classes de teste e de redistribuição de testes em classes através da utilização de métricas de similaridade de código e agrupamento. O framework foi desenvolvido utilizando a abordagem TDD (Test Driven Development) e possui um conjunto de 81 classes de testes com 391 métodos de teste.

Neste experimento foi utilizado o Roza para aplicação da ferramenta deste trabalho. O Roza é um framework de refatoração automatizada de classes de teste e de redistribuição de testes em classes através da utilização de métricas de similaridade de código e agrupamento. O framework foi desenvolvido utilizando a abordagem TDD (Test Driven Development) e possui um conjunto de 81 classes de testes com 391 métodos de teste.

Para validação do SUnit em um projeto real, foi realizada a execução de todos os testes do Roza em duas etapas, sendo a primeira sem a utilização da ferramenta SUnit, ou seja, sem a reutilização da configuração implícita de de teste e a segunda utilizando a ferramenta, a fim de poder comparar os resultados. Na primeira etapa não houve nenhuma classificação feita nos métodos de teste de forma manual, utilizando a notação @Safe. Para realização do experimento foi utilizada uma máquina com sistema operacional Linux Mint 22 Cinnamon, processador Intel Core Ultra 9 185H x 16, 32 GB de memória RAM, OpenJDK 17.0.14 e IntelliJ IDEA 2025.1.2.

Para validação do SUnit em um projeto real, cada uma das duas etapas (sem a utilização do SUnit e com a utilização do SUnit) foi executada 10 vezes com a finalidade de verificar se há diferença estatística significativa no tempo de execução. Dessa forma, poderemos afirmar que a diferença observada nos tempos de execução é real e não por acaso, uma vez que cada execução da suíte de teste não apresenta sempre o mesmo tempo de execução, devido a flutuações do ambiente de teste, como por exemplo outros processos em execução. A Tabela 2 exibe os valores obtidos em cada uma das 10 rodadas de teste de cada etapa, sendo a Etapa 1 realizada sem a utilização do SUnit e a Etapa 2 com a utilização do SUnit.

Tabela 2. Tempo de execução da suíte de teste do Roza

	Etapa 1 (ms)	Etapa 2 (ms)
Rodada 1	221	141
Rodada 2	234	109
Rodada 3	213	100
Rodada 4	207	97
Rodada 5	197	99
Rodada 6	198	127
Rodada 7	223	122
Rodada 8	208	119
Rodada 9	206	108
Rodada 10	206	108
Média	211,30	113,00
Desvio Padrão	11,62	14,08
Variância	135,12	198,22
p-valor	0,000000000000280	

Após executar todas as rodadas da suíte de testes do Roza em cada etapa, foi realizada uma análise estatística para aceitar ou não a hipótese nula, onde a hipótese nula indica que não há diferença significativa no tempo de execução entre as duas

estratégias. Para isso foi utilizado o teste estatístico t, devido à pequena amostra e por não se conhecer a variância da população. Com a aplicação do teste t foi obtido um p-valor, onde um resultado pequeno para esse valor indica que há uma diferença estatística de significância entre as duas estratégias. E para determinar o nível de significância de 5% (0,05), ou seja, se encontrarmos um p-valor que é menor que 0,05 rejeitamos a hipótese nula, caso contrário, aceitamos a hipótese nula.

Através da Tabela 2, observamos que o p-valor encontrado é igual a 0.000000000000280, sendo esse valor menor que 0.05 e com isso rejeitamos a hipótese nula, ou seja, há um valor estatístico significativo com a utilização do SUnit para redução do tempo de execução de testes automatizados.

Por fim, considerando a média dos tempos de execução das etapas 1 e 2 na Tabela 2, podemos dizer que a execução da suíte de testes do Roza sem a utilização do SUnit foi de 211,3 ms e com a utilização do SUnit esse tempo caiu para 113 ms. Sendo assim, o tempo de execução da suíte de teste do Roza utilizando o SUnit reduziu 46,52% em relação ao tempo de execução sem o SUnit.

7. Conclusão

Neste trabalho foi proposto o SUnit, uma ferramenta para reutilização da execução da configuração implícita de testes automatizados, com a finalidade de reduzir o tempo de execução dos mesmos e ainda garantir a eficácia inerente aos testes de softwares.

A proposta foi implementada através da análise estática do código principal e do código de teste do projeto ao qual se deseja aplicar a ferramenta, permitindo a reutilização de execução das configurações implícitas caso os métodos de teste não realizem mudança no seu estado inicial. O SUnit é capaz de analisar todos os métodos contidos em todas as classes do projeto e todos os métodos de teste em todo o código de teste do projeto, e com isso realizar a classificação de tais métodos em seguros e não seguros.

O SUnit também teve sua abordagem validada por meio de dois experimentos. O primeiro validou sua classificação dos métodos de classe e dos métodos de teste em seguros e não seguros. O segundo experimento testou a capacidade da ferramenta em reduzir o tempo de execução dos testes automatizados, utilizando a abordagem estatística teste-t, resultando em uma redução de tempo de execução próximo a 47%.

Como recomendação para trabalhos futuros, recomenda-se o projeto e desenvolvimento de uma nova versão do SUnit incluindo o suporte à paralelização de testes, uma vez que o framework JUnit 5 apresenta essa funcionalidade em seu escopo.

Referências

- Barbosa, Felipe; Torres, Isabelle. O teste de software no mercado de trabalho. *Tecnologias em Projeção*, v. 2, n. 1, 2011.
- Bartié, Alexandre. *Garantia da qualidade de software: adquirindo maturidade organizacional*. Gulf Professional Publishing, 2002.
- Bernardo, Paulo Cheque; Kon, Fabio. A importância dos testes automatizados. *Engenharia de Software Magazine*, v. 1, n. 3, p. 54-57, 2008.

- Christensen, Claus A. et al. A unit-test framework for database applications. In: 2006 10th International Database Engineering and Applications Symposium (IDEAS'06). IEEE, 2006. p. 11-20.
- Delamaro, Marcio; Jino, Mario; Maldonado, Jose. Introdução ao teste de software. Elsevier Brasil, 2013.
- Dias Neto, Arilo Claudio. Introdução a teste de Software. Engenharia de Software Magazine, v. 1, p. 22, 2007.
- Duarte, Alexandre et al. Gridunit: software testing on the grid. In: Proceedings of the 28th international conference on Software engineering. 2006. p. 779-782.
- Emery, Dale H. Writing maintainable automated acceptance tests. In: Agile Testing Workshop, Agile Development Practices, Orlando, Florida. 2009.
- Fewster, Mark; Graham, Dorothy. Software test automation. Reading: Addison-Wesley, 1999.
- Forsgren, Nicole; Humble Jez; Kim, Gene. Accelerate: The science of lean software and devops: Building and scaling high performing technology organizations. IT Revolution, 2018.
- Fowler, M. Continuous Integration. In: Refactoring: Improving the Design of Existing Code. Addison-Wesley, 2006.
- Greiler, Michaela et al. Strategies for avoiding text fixture smells during software evolution. In: Proceedings of the 10th Working Conference on Mining Software Repositories. IEEE Press, 2013. p. 387-396
- Humble, Jez; FARLEY, David. Continuous delivery: reliable software releases through build, test, and deployment automation. Pearson Education, 2010.
- Júnior, Misael; Andrade, Stevão; Delamaro, Márcio. Automatização de teste de software com ênfase em teste de unidade, 2016.
- Medeiros, H. SolUnit: um framework de testes de unidade para smart contracts. 2020. 93 f. Dissertação (Mestrado em Ciências da Computação) - UFSC, Florianópolis, 2020.
- Meszaros, Gerard. xUnit test patterns: Refactoring test code. Pearson Education, 2007.
- Myers, Glenford J.; Sandler, Corey; Badgett, Tom. The art of software testing. John Wiley & Sons, 2011.
- Naik, K; Tripathy, P. Software testing and quality assurance: theory and practice. [S.l.]: John Wiley & Sons, 2008.
- Pressman, Roger S.; Maxim, Bruce R. Engenharia de software. 9. ed. McGraw Hill Brasil, 2021.
- Silva, Davi Augusto Gadêlha. Evolunit: geração e evolução de testes de unidade em java utilizando algoritmos genéticos. 2008. Dissertação de Mestrado. Universidade Federal de Pernambuco.

Silva, Lucas Pereira; Vilain, Patrícia. Reuse of Fixture Setup between Test Classes. In: SEKE. 2017. p. 224-229.

Sommerville, Ian. Engenharia de software, 9a. São Palo, SP, Brasil, 2011.

Vasconcelos, Alexandre Marcos Lins et al. Introdução à Engenharia de Software e à Qualidade de Software. Minas Gerais: Universidade Federal de Lavras, 2006.

Zhao, Ruilian et al. E2E test execution optimization for web application based on state reuse. Journal of Software: Evolution and Process, v. 37, n. 1, p. e2714, 2025.