CENTRO UNIVERSITÁRIO FAMETRO - UNIFAMETRO
GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

JÉSSICA RODRIGUES DA COSTA

UTILIZAÇÃO DE TESTES ESTRUTURAIS NO APRIMORAMENTO DA QUALIDADE DE SOFTWARE

FORTALEZA - CE
2021

JÉSSICA RODRIGUES DA COSTA

UTILIZAÇÃO DE TESTES ESTRUTURAIS NO APRIMORAMENTO DA QUALIDADE DE SOFTWARE

Trabalho apresentado ao Curso Superior de Graduação em Sistemas de Informação da Universidade Metropolitana da Grande Fortaleza (UNIFAMETRO) como requisito para aprovação na disciplina Trabalho de Conclusão de Curso, sob a orientação do Prof. Izequiel Pereira de Norões.

FORTALEZA - CE
2021

JÉSSICA RODRIGUES DA COSTA

UTILIZAÇÃO DE TESTES ESTRUTURAIS NO APRIMORAMENTO DA QUALIDADE DE SOFTWARE

Trabalho apresentado ao Curso Superior de Graduação em Sistemas de Informação da Universidade Metropolitana da Grande Fortaleza (UNIFAMETRO) como requisito para aprovação na disciplina Trabalho de Conclusão de Curso, sob a orientação do Prof. Izequiel Pereira de Norões.

BANCA EXAMINADORA

Prof. MSc Eduardo Julião Máximo

Membro – Centro Universitário Unifametro (UNIFAMETRO)

Prof. MSc Fábio Henrique Souza

Membro – Centro Universitário Unifametro (UNIFAMETRO)

Prof. MSc Izequiel Pereira de Norões

Orientador – Centro Universitário Unifametro (UNIFAMETRO)

FORTALEZA - CE
2021

LISTA DE ILUSTRAÇÕES E TABELA

SUMÁRIO

UTILIZAÇÃO DE TESTES ESTRUTURAIS NO APRIMORAMENTO DA QUALIDADE DE SOFTWARE

RESUMO

Este trabalho busca apresentar o processo de implementação de testes estruturais no aprimoramento da qualidade dos softwares nas empresas. A busca por qualidade e desempenho tem sido um dos fatores a serem considerados no processo de implementação. Diante disto o ato de testar um sistema se tornou uma das etapas mais importantes da engenharia de software por poder estar presente em quase todas as etapas da construção de um projeto de software, com isto tentando minimizar erros, falhas e retrabalhos. Tais medidas de utilização de testes estruturais quando usadas de forma errônea podem trazer consequências no custo, na qualidade do projeto e até em malefícios para a empresa, como perdas de investimentos, custos absurdos na manutenção e entre outros. Diante disto, na utilização de testes estruturais devemos construir os testes de forma a garantir que a codificação feita no sistema não apresente falhas e erros de execução, além do mais o uso de métricas, padrões e principalmente de ferramentas contribuem para uma boa entrega e construção do sistema. Tais procedimentos de testes podem trazer benefícios desde uma redução de custos, diminuição de retrabalho e de correções de erros, na segurança e estabilidade do sistema.

Palavras-chave: Teste. Software. Qualidade. Implementação. Evolução.

USE OF STRUCTURAL TESTS IN IMPROVING SOFTWARE QUALITY

ABSTRACT

This work seeks to present the process of implementing structural tests to improve software quality in companies. The search for quality and performance has been one of the factors to be considered in the implementation process. Therefore, the act of testing a system has become one of the most important stages of software engineering as it can be present in almost all stages of the construction of a software project, thus trying to minimize errors, failures and rework. Such measures of use of structural tests, when misused, may bring consequences on the cost, on the project quality, and even harm to the company, such as investment losses, enormous maintenance costs and others. Therefore, in the use of structural tests, we must build the tests to ensure that the code made in the system does not present flaws and execution errors, in addition, the use of metrics, patterns, and especially tools contribute to good delivery and construction of the system. Such testing procedures may bring benefits from a reduction in costs, reduction of rework and error corrections, in the security and stability of the system.

Keywords: Test. Software. Quality. Implementation. Evolution.

1 INTRODUÇÃO

O processo de evolução dos testes se deu na necessidade de qualidade e da preocupação com o aumento de informações e a complexidade com que esses dados estão sendo armazenados. Segundo Alice H. Tamashiro, “No início, os testes de software eram realizados dentro do processo de desenvolvimento pelo próprio desenvolvedor realizando os atualmente chamados Testes Unitários”. (GERSHON et al., 2009).

Com o advento dos sistemas e de sua complexidade em executar tarefas e até mesmo testes, muitos programas eram entregues com falhas e bugs, fatos que ao longo prazo eram refletidos na utilização do sistema de forma negativa para as empresas que entregavam tal programa defeituoso e com sua qualidade comprometida. No decorrer da evolução da Engenharia de Software, a necessidade de validar que uma aplicação atende corretamente as demandas que ele se propôs se tornou recorrente, visto que um sistema que está constantemente gerando falhas, trará além de problemas uma visão de qualidade comprometida para a empresa, levando ao pensamento de que será mais viável não ter o sistema, do que ter um que se torna inviável sua utilização. Diante desse cenário a busca por qualidade se tornou algo essencial e assim a necessidade de “prever” e simular falhas antes da entrega de um sistema se tornou decisivo no sucesso do software e no custo a longo prazo, conseguindo corrigir falhas antes de uma entrega ou mesmo antes do usuário perceber tal problema.

Neste trabalho será apresentado e exposto de forma teórica a respeito do processo de implementação dos testes e como isto impacta na qualidade de um software. Sendo abordado de forma prática a construção e execução dos testes estruturais por meio de ferramentas. Sendo apresentados as técnicas de testes estruturais, métricas, ferramentas e os benefícios que eles proporcionam. Além do mais, como um mau desenvolvimento de testes podem prejudicar na construção do sistema e ser muitas vezes crítico para as empresas. O trabalho fará uso de exemplos descritos e implementados na linguagem de programação ***Java*** e suas bibliotecas e ferramentas como: framework JUnit, Spring, Asserts, Mockito etc.

2 TESTE DE SOFTWARE

2.1 CONCEITOS

Em meados dos anos de 1979, Glenford Myers em sua obra *“The Art of Software Testing”* introduziu o conceito que teste “É o processo de executar um programa com o objetivo único de encontrar defeitos”. Foi entre os anos 80 e 90 que os testes de softwares começaram a ganhar força devido aos resultados positivos em diversas empresas, logo essas mesmas empresas começaram a investir em ferramentas e meios para que os custos de correções de defeitos diminuíssem. A partir dessa visão e da necessidade, diversas áreas surgiram, sendo áreas com um propósito maior em testes, qualidade, desempenho, segurança entre outras. Já para William Hetzel em 1988, conceitualiza o teste como sendo qualquer atividade que tenha como foco a qualidade e a avaliação de um programa. (CRAIG, JASKIEL, 2002).

Segundo Renata Eliza e Vivian Lagares (DEVMEDIA, 2012), “Um Processo de Teste de Software tem como objetivo estruturar as etapas, as atividades, os artefatos, os papéis e as responsabilidades do teste, permitindo organização e controle de todo o ciclo do teste, minimizando os riscos e agregando valor ao software”. Além de tudo, segundo Hetzel (1987), “teste é um processo de aquisição de confiança no fato de que um programa ou sistema faz o que se espera dele”. Ou seja, o processo de testes reflete se o sistema atende ao que foi proposto durante a etapa de levantamento de requisitos.

Conforme explica BASTOS et al. (2007, p.11), “os testes eram efetuados pelos próprios desenvolvedores de software, cobrindo aquilo que hoje chamamos de testes unitários e testes de integração.” Durante um certo tempo os testes ficavam de responsabilidade dos desenvolvedores sendo apenas uma etapa do processo de desenvolvimento de software, onde a qualidade do software ficava encarregada apenas dos desenvolvedores testando suas aplicações após o processo de codificação.

O Teste de Software usa de comparação entre saídas reais e saídas esperadas, diante dessa comparação os testes conseguem simular saídas que em um determinado fluxo de execução do sistema podem ocorrer sucesso ou erro. Entretanto apesar da criação de testes deve ter em mente que não é possível testar todas as possibilidades de entradas e saídas. Na figura 1 apresenta um método que deverá retornar o preço de um produto a partir das entradas que esse método pode ter, logo temos:

- ***preco e margem***: Por serem valores do tipo “double” eles terão ***2^64*** possibilidades cada um de entradas possíveis;
- ***vista***: Por ser um valor do tipo “boolean” ele terá ***2*** possibilidades de entradas possíveis.

Figura 1 - Método que retorna o valor do preço de um produto.

```
1  public double getPreco (double preco, double margem, boolean vista) {
2      // implementação
3  }
```

Fonte: (Próprio autor, 2021).

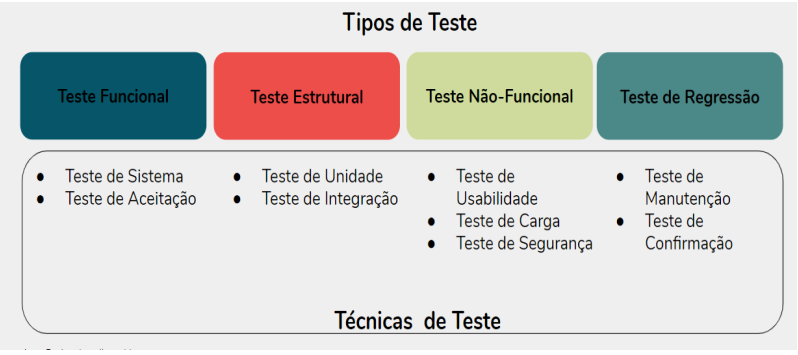
Logo, se quisermos testar “todas” as entradas possíveis no método getPreco (figura 1) seu nível de complexidade será grande pois os valores possíveis podem ser: ***2^64 * 2^64 * 2***. Tornando inviável cobrir todas as possibilidades de teste.

2.2 TIPOS DE TESTES

No processo de testes foram criadas técnicas para auxiliar e padronizar a construção e o desenvolvimentos dos mesmos, ajudando a escolher conjuntos de dados com a proposta de diminuir números de casos de testes e mostrar uma maior probabilidade de existência de erros. Para MACORATTI (2005, p.2), dessas técnicas podem se destacar duas principais, estruturais (ou teste caixa branca) e funcionais (ou teste caixa preta).

Segundo ROCHA (2007, p.1), os testes podem ser divididos em quatro tipos: os testes funcionais (caixa-preta), os testes estruturais (caixa-branca), os testes não-funcionais e os testes de regressão. Podemos ver isso em uma tipificação em resumo na figura 2 e na classificação a seguir. Além dos tipos de testes, existem as técnicas de testes onde em cada tipo de teste há técnicas específicas para sua utilização.

Figura 2 - Relação entre tipos e técnicas de teste.



Fonte: Modificado de Rocha (2007, p.1).

- **Testes Funcionais -** Testes funcionais (também conhecidos como testes de caixa preta) o testador não precisa ter conhecimento da implementação e codificação do sistema, são informados os dados de entrada e assim verificado se as saídas são o esperado. Ou seja, o principal objetivo dos testes funcionais são os resultados das execuções a partir das entradas.
- **Testes Estruturais -** Testes estruturais (também conhecidos como testes de caixa branca) diferente dos funcionais, o testador tem que ter conhecimento da implementação e codificação e se os diferentes caminhos estão sendo testados. Seu principal objetivo é testar especificações técnicas e de codificação como decisões lógicas, variáveis estáticas, dinâmicas entre outras.
- **Testes Não-Funcionais -** Testes não-funcionais não se limitam ao nível do teste podem ser realizados em qualquer nível. Tendo como objetivo fundamental testar os aspectos gerais e referente às restrições do sistema, aspectos esses que tentam verificar a qualidade do sistema.
- **Testes de Regressão -** Testes de regressão ocorrem quando o sistema sofre atualizações e mudanças bruscas ou consideráveis que de alguma forma podem afetar o sistema ou gerar bugs. Sua rotina consiste em reexecutar todos os testes funcionais, isto se o sistema for pequeno. Caso se for um sistema grande reexecutar os testes referente aquele escopo modificado e as partes referente a ele, Tendo como objetivo verificar se tais mudanças como um novo bloco de teste ou correção não afetou partes que estavam funcionando do software.

3 UTILIZAÇÃO DE TESTES ESTRUTURAIS

Sobre testes estruturais podemos dizer que:

> Teste de Caixa Branca é uma técnica de teste de software em que a estrutura interna, o design e a codificação do software são testados para verificar o fluxo de entrada e saída e para melhorar o design, a usabilidade e a segurança. No teste de caixa branca, o código é visível para os testadores, por isso também é chamado de teste de caixa branca, teste de caixa aberta, teste de caixa transparente, teste baseado em código e teste de caixa de vidro.(HAMILTON, 2021, p.1, tradução nossa).

Os programadores e testadores têm o conhecimento total das técnicas sobre a estrutura do programa e estão concentrados principalmente na lógica interna e estrutura do código. A escolha de criar casos de testes se torna algo fundamental na implementação do software por centralizar as partes essenciais do que se deve testar.

Tais testes são usados principalmente para detectar erros na lógica de negócio, no código e ao depurar o código com o propósito de encontrar erros tipográficos e de implementações. **Seu principal objetivo é testar especificações técnicas e de codificação como decisões lógicas, variáveis estáticas, dinâmicas entre outras.**

Os testes estruturais em muitos casos são construídos e desenvolvidos pelos próprios programadores e testadores com o intuito de validar a estrutura do código e se suas funções e métodos estão implementados de forma correta, preocupando-se ainda de como o código está se comportando. São níveis de testes estruturais os unitários e os integrados, além de serem usados com o propósito da validação interna, podem ser utilizados para análise de requisitos, projetos e casos de testes. Testes de estrutura são baseados em fluxo de controle, de dados e de complexidade.

3.1 CRITÉRIOS DOS TESTES ESTRUTURAIS

3.1.1 Fluxo de Controle

Para Myers (2004), os critérios de fluxo de controle mais relevantes são, “Todos-Nós”, “Todas-Arestas” e “Todos-Caminhos”. Utilizam-se de informações relacionadas a decisões de caminhos.

A. **Todos-Nós -** Diz que a execução do programa deve passar pelo menos uma vez a cada nó.
B. **Todas-Arestas -** Diz que cada grafo (desvio do fluxo) deve ser executado no mínimo uma vez.
C. **Todos-Caminhos -** Diz que todos os caminhos do programa devem ser executados.

3.1.2 Fluxo de Dados

Critérios de fluxo de dados utilizam-se das informações dos dados para então definir qual os casos de testes e os caminhos de execução dos testes. O teste de fluxo de dados analisa o ciclo de vida de uma parte específica dos dados e usa-se do fluxo de controle para explorar comportamentos irracionais que ocorrem nos dados. Para o fluxo de dados existem duas abordagens, são:

A. **Por Rapps e Weyuker (1982, 1985)** - Tal proposta pode ser apresentada nos três principais critérios:
    a. **Todas-Definições/Todas-Defs** - Diz que para cada definição de variável deve ser executada pelo menos uma vez.
    b. **Todos-Usos** - Diz que todas as associações entre as variáveis e seus usos, sejam executadas pelos casos de teste onde sejam um caminho em que a variável não é redefinida.
    c. **Todos-DU-Caminhos** - Diz que todas as associações entre as variáveis e seus usos sejam executados por todos os caminhos, onde devem ser livres de definições e de laços que cubram a associação.
B. **Por Maldonado (1991)** - Propôs o critério de **potenciais de usos/Potenciais-Usos**, onde devem ser executadas todas as possíveis mudanças de estado em programa no teste.

3.1.3 Fluxo de Complexidade

Critérios de fluxo de complexidade faz-se uso de informações sobre a complexidade do programa para então serem definidos os requisitos dos testes. Critério mais conhecido é o de McCabe (MCCABE, 1976), onde utiliza a complexidade ciclomática do grafo para derivar tais requisitos de teste. Para DELAMARO, MALDONADO e JINO este critério requer um conjunto de caminhos lineares que sejam independentes do grafo que é executado.

3.2 ESCRITA DE TESTES ESTRUTURAIS

3.2.1 Padrões de Teste

Assim como existem padrões de escrita de códigos e de projetos, os testes também possuem formas de escrita, uma delas é o padrão ***Triple A*** ou ***AAA*** *(Arrange-Act-Assert)* que é um dos mais utilizados no desenvolvimento de testes no sistema e quando utilizado a técnica

de desenvolvimento ***TDD*** *(Test-driven development)*. O padrão consiste em ser elaborado três passos (DARDE, 2020, p.1-2):

- **Arrange (Cenário) -** Consiste na configuração e na preparação dos objetos, variáveis, dados e de pré-requisitos para o teste.
- **Act (Execução) -** Consiste em executar o método ou função a ser testado utilizando o que foi preparado na etapa anterior (arrange).
- **Assert (Verificação) -** Consiste em validar ou verificar se o resultado do teste é o esperado.

Na figura 3, apresenta o teste ***get_by_id_test*** usando o padrão AAA.

Figura 3 - Teste usando o padrão AAA.

```
@Test
@DisplayName("Deve obter um livro por id")
public void get_by_id_test(){
    //cenario (arrange)
    Long id = 11;
    Book book = createValidBook();
    book.setId(id);
    when(repository.findById(id)).thenReturn(Optional.of(book));

    //execucao (act)
    Optional<Book> foundBook = service.getById(id);

    //verificacoes (assert)
    assertThat(foundBook.isPresent()).isTrue();
    assertThat(foundBook.get().getId()).isEqualTo(id);
    assertThat(foundBook.get().getAuthor()).isEqualTo(book.getAuthor());
    assertThat(foundBook.get().getIsbn()).isEqualTo(book.getIsbn());
    assertThat(foundBook.get().getTitle()).isEqualTo(book.getTitle());
}
```

Fonte: (Próprio autor, 2021).

Deve ser salientado que os testes (unitários e integrados) não devem ser validados apenas no cenários "positivos", devem ser criados cenários de teste que cubram desde testes que retorne um valor positivo ou de exceção. Os *testes de sucesso* retornam o valor que o sistema deve fazer, já os *testes de erro* são escritos de tal forma a verificar as exceções do fluxo (figuras 4).

Figura 4 - Teste de erro.

```
@Test
@DisplayName("Deve retornar vazio ao obter um livro por id quando ele nao existe na base.")
public void book_not_found_by_id_test(){
    //cenario (arrange)
    Long id = 11;
    when(repository.findById(id)).thenReturn(Optional.empty());

    //execucao (act)
    Optional<Book> book = service.getById(id);

    //verificacoes (assert)
    assertThat(book.isPresent()).isFalse();
}
```

Fonte: (Próprio autor, 2021).

Além do padrão *Arrange-Act-Assert*, existe outro padrão ***GWT*** *(Given-When-Then)* conhecido principalmente quando se usa a técnica de desenvolvimento ágil ***BDD*** *(Behavior Driven Development)*, o padrão se assemelha ao *Arrange-Act-Assert* tendo consigo também três etapas que são (SYKES, 2019, p.2):

- **Given -** Se assemelha ao *Arrange*: define o estado da aplicação fornecendo seu contexto.
- **When -** Se assemelha ao *Act*: chamada do método a ser testado.

- **Then -** Se assemelha ao *Assert*: verifica o comportamento ou resultado do teste.

Na figura 5, apresenta o teste ***get_by_id_test*** usando o padrão AAA.

Figura 5 - Teste usando o padrão GWT.

```
@Test
@DisplayName("Deve obter informacoes de um livro")
public void get_book_details_test() throws Exception{
    //cenario (given)
    Long id = 11;

    Book book = Book.builder()
                .id(id)
                .title(createNewBook().getTitle())
                .author(createNewBook().getAuthor())
                .isbn(createNewBook().getIsbn())
                .build();

    BDDMockito.given(service.getById(id)).willReturn(Optional.of(book));

    //execulcao (when)
    MockHttpServletRequestBuilder request = MockMvcRequestBuilders
            .get(BOOK_API.concat("/" + id))
            .accept(MediaType.APPLICATION_JSON);

    //verificação (then)
    mvc
            .perform(request)
            .andExpect(status().isOk())
            .andExpect(jsonPath("id").value(id) )
            .andExpect(jsonPath("title").value(createNewBook().getTitle()) )
            .andExpect(jsonPath("author").value(createNewBook().getAuthor()) )
            .andExpect(jsonPath("isbn").value(createNewBook().getIsbn()) );

}
```

Fonte: (Próprio autor, 2021).

3.2.2 Tecnologias de Teste

Kent Beck criou o conceito de ***xUnit*** que consiste em um pacote de programas e moldes de classes de teste que servem para organizar e executar os testes. Kent Beck fez sua primeira implementação do xUnit na linguagem de programação *SmallTalk*, onde ficou conhecida como *SUnit*. (SOARES, 2007). Todas as implementações do xUnit nas linguagem de programação recebem um prefixo diferente tais como em:

- Java - JUnit;
- PHP - PHPUnit;
- JavaScript - JSUnit;
- Python - PyUnit;
- .Net - NUnit.

Em algumas implementações as plataformas possuem suas próprias especificações e diferenças que atendem a necessidade da linguagem e sua evolução.

Na construção dos testes estruturais os programadores fazem uso de ferramentas e frameworks que ajudam na execução dos testes seja no uso de bibliotecas que simulam o valor dos objetos e execuções como os ***mocks*** e de ***asserções*** que são utilizadas para validar se o teste está fazendo e executando o que se espera. Dentre essas tecnologias usam-se os ***spies*** e ***stubs*** para auxiliar na criação e validação dos testes.

- **Asserções -** A biblioteca de ***assert*** é usada para verificar as demais características a ser testada. Os assert´s podem verificar igualdades, diferenças e resultados de erro. A biblioteca de *assert* possui alguns modificadores como (NUNIT, 2018, p.1):
    - **Assert.Is** - Testa se o objeto é o mesmo que o valor esperado, se assemelha a ***x==y***.

- **Assert.Not -** Testa se o objeto não é o mesmo que o valor esperado, se assemelha a ***x!=y***.
- **Assert.True** - Testa se o valor na condição é verdadeiro.
- **Assert.False -** Testa se o valor na condição é falso.
- **Assert.Null -** Testa se o valor especificado é nulo.
- **Assert.NotNull -** Testa se o valor especificado não é nulo.
- **Assert.IsEmpty -** Testa se a String, Coleção ou IEnumerable é vazia.
- **Assert.IsNotEmpty -** Testa se a String, Coleção ou IEnumerable não está vazia.
- **Assert.AreEqual -** Testa se dois argumentos são iguais.
- **Assert.AreNotEqual -** Testa se dois argumentos não são iguais.
- **Assert.Contains -** Testa se o objeto está contido numa coleção.

Além destes, a biblioteca assert possui outros modificadores.

Os Mocks, Stubs e Spies são conhecidos como objetos de teste (*test doubles*), que em linhas gerais são usados para representar ou simular algo. (JAVATPOINT, 2021, p.2-3).

- **Mocks -** Objetos mock ou simulados, são considerados como tudo que é usado para representar um elemento real, uma "falsificação". São objetos que armazenam chamadas de métodos. Sua principal função é fornecer o controle total sobre o comportamento dos objetos. Mocks são criados por meios de bibliotecas ou estrutura de mocking como Mockito, JMock e EasyMock.
- **Stubs -** São objetos cujo possuem comportamento fixo e usados para fornecer respostas durante a execução dos testes. O stub se assemelha ao mock por ser uma representação real porém com o número mínimo de métodos para executar o teste. São usados principalmente quando não se quer usar objetos com dados reais.
- **Spies -** São conhecidos como objetos parcialmente fictícios. Spy assim como o mock é um objeto que grava suas interações com outros objetos. Muito usado quando se quer testar *callbacks* ou testar certos métodos de uma classe grande que contém vários métodos.

3.3 TÉCNICAS

3.3.1 Testes Unitários

A respeito do teste unitário, pode se dizer que:

> O teste de unidade verifica a menor parte testável do software - que é chamada de unidade. Neste tipo de teste, essa unidade é testada de forma isolada para garantir que tenha o comportamento esperado. Como esse teste é focado em um trecho específico do software, os erros são encontrados facilmente, diminuindo o tempo gasto com depuração. (GUEDES, 2019, p.3).

Testes Unitários se caracterizam por:

A. Normalmente usarem o padrão de estrutura *Arrange-Act-Assert*;
B. Não devem ser complexos;
C. Preocupa-se apenas com a unidade testada;
D. Não fazem uso de chamadas de banco de dados, de API´s (*Application Programming Interface*) externas ou recursos.

Na figura 6, mostra a classe *"Book"* nela possui um método *"wicthLoans"* que recebe como parâmetro uma lista de objetos da classe *"Loan"* e atribui essa lista a lista de *"Loan"*

que contém na estrutura da classe, ou seja o método relaciona uma lista de empréstimos a um livro (relação de *"**OneToMany**"* entre a classe Book e Loans).

Figura 6 - Classe "Book" e o método "wicthLoans".

```java
public class Book {

    @Id
    @Column
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column
    private String title;

    @Column
    private String author;

    @Column
    private String isbn;

    @OneToMany(mappedBy = "book")
    private List<Loan> loans;

    public Book withLoans(List<Loan> loans){
        this.loans = loans;
        return this;
    }
}
```

Fonte: (Próprio autor, 2021).

Na figura 7, mostra um exemplo de teste unitário da classe *"Book"* feito na linguagem de programação Java e com o framework JUnit. O teste *"should_loans"*, assim como descrita na anotação ***@DisplayName,*** *"Deve relacionar uma lista de empréstimos a um livro."* No teste é criado dois objetos, um sendo do tipo *Book* e outro *Loans*, no objeto *Book* não foi setado o valor do atributo *"loans"* justamente para testar o método *wicthLoans.* Além do mais é criado uma lista de *Loan* e adicionado esse objeto do mesmo tipo, logo após preparar o **cenário** do teste seguirá para a etapa de **execução** onde será *"chamado"* o método *wicthLoans* e passado como parâmetro a lista de *Loan.* Após as etapas de cenário e execução, temos a etapa de **verificação**, onde será utilizado o ***assertEquals*** com o objetivo de verificar se o atributo *loans* do objeto do tipo *Book* é "igual" a lista de *Loan* declarada na etapa de cenário.

Figura 7 - Teste unitário do método "wicthLoans".

```java
    @Test
    @DisplayName("Deve relacionar uma lista de empréstimos a um livro.")
    public void should_loans() {
        //cenario
        Book book = mockBook();
        List<Loan> loans = new ArrayList<>();
        loans.add(mockLoan());

        //execucao
        book.withLoans(loans);

        //verificacao
        assertEquals(loans, book.getLoans());
    }

    private Book mockBook() {
        return Book.builder().isbn("123").author("Jessi").title("As aventuras").build();
    }

    private Loan mockLoan(){
        return Loan.builder().id(11).customer("jessica").loanDate(LocalDate.now()).build();
    }
```

Fonte: (Próprio autor, 2021).

3.3.2 Testes Integrados

A respeito do teste integrado, pode se dizer que:

> O teste de integração verifica se uma unidade tem o comportamento esperado quando funciona de maneira integrada a outros elementos de software, como chamada de serviços, APIs e banco de dados. Aqui, a unidade de software em si não é avaliada, mas sim a sua integração com outras unidades. (GUEDES, 2019, p. 4).

Testes Integrados se caracterizam por:

A. Normalmente usarem o padrão de estrutura *Given-When-Then*;
B. Preocupa-se com a interação entre as unidades;
C. Fazem uso de chamadas de banco de dados, de API´s (*Application Programming Interface*) externas ou recursos.

Na figura 8 mostra uma ***interface*** *"BookRepository"* que estende a interface ***JpaRepository***, que contém o método *"save"* já estruturado que recebe como parâmetro um objeto do tipo *"Book"*.

Figura 8 - Interface "BookRepository".

```
2
3  public interface BookRepository extends JpaRepository<Book,Long> {
4
5  }
6
```

Fonte: (Próprio autor, 2021).

Na figura 9, mostra um exemplo de teste integrado da classe *"BookRepositoryTest"* feito na linguagem de programação Java e os framework JUnit e Spring. O teste *"save_book_test"*, assim como descrita na anotação ***@DisplayName,*** *"Deve salvar um livro."* No teste na etapa de **cenário** é criado um objeto do tipo *Book,* já na etapa de de **execução** também é criado um objeto do tipo *Book*, entretanto esse objeto é criado a partir do retorno do método *save* que está recebendo como parâmetro o objeto *Book*, ou seja na etapa de execução o objeto que será criado terá que conter os mesmos valores que o objeto criado na etapa de cenário o que indicaria que o objeto foi ***persistido*** na base de dados, seguindo para etapa de **verificação**, onde será utilizado o ***assertThat*** com o objetivo de verificar se o atributo ***id*** do objeto *"saveBook"* não está nulo.

Figura 9 - Teste de integração do método "save_book_test".

```
10     @Test
11     @DisplayName("Deve salvar um livro.")
12     public void save_book_Test(){
13         //cenario
14         Book book = createNewBook("123");
15
16         //execucao
17         Book savedBook = repository.save(book);
18
19         //verificacao
20         assertThat(savedBook.getId()).isNotNull();
21     }
22
23
24     public static Book createNewBook(String isbn) {
25         return Book.builder().title("Aventuras").author("Jessica").isbn(isbn).build();
26     }
```

Fonte: (Próprio autor, 2021).

3.4 MÉTRICAS E FERRAMENTAS

Na utilização e criação de testes podem ser escritos sendo baseados em métricas. As medidas em Engenharia de software são chamadas de métricas, que são processos em que números ou símbolos são atribuídos com o objetivo de caracterizar as regras do sistema.

De acordo com FERNANDES (1995, p.81)

> As métricas de software podem ser definidas como métodos de determinar quantitativamente a extensão em que o projeto, o processo e o produto de software têm certos atributos. Isto inclui a fórmula para determinar o valor da métrica como também sua forma de apresentação e as diretrizes de utilização e interpretação dos resultados obtidos no contexto do ambiente de desenvolvimento de software.

Seu principal objetivo é de realizarmos medições ao decorrer do desenvolvimento de um software e obter níveis de qualidade maiores, considerando o projeto, o processo e o produto, visando à satisfação dos envolvidos e a um custo compatível. (FERNANDES, 1995, p.25).

Neste tópico, será apresentado duas métricas e a sua abordagem técnica por meio de ferramentas de análise e cobertura, tendo como base um projeto desenvolvido pelo autor na linguagem de programação Java com uso do Spring. As demonstrações e exemplos foram retiradas do projeto.

3.4.1 Métricas

3.4.1.1 Complexidade Ciclomática

A complexidade ciclomática foi primeiro apresentada por Thomas J. McCabe, em 1976, onde consiste-se de uma métrica de software que fornece um grau quantitativo da dificuldade lógica de um sistema e de um determinado módulo, seja classes ou uma função. Essa métrica consistem em medir os diferentes fluxo de execução que um código pode ter, tendo como principal foco as estruturas que são compostas de quantos ***if´s-then-else***, ***while´s***, ***for´s***, ***switch´s***, entre outros que existem na estrutura do código.

Caso o trecho do código não possua instruções *if* ou *switch* no código fonte, a complexidade será ***1***, pois para ser coberto todo o código será necessário apenas um caminho.

Sua importância se dá pelo fato de a partir dela ser possível identificar a quantidade mínima de testes, ou seja o número de casos de teste que precisam para cobrir todo o código que deverá ser criado no sistema para que se verifiquem todos os fluxos possíveis do código.

- **Parâmetros aceitáveis para complexidade ciclomática:**

McCabe apresenta valores base de referência para a complexidade, onde:

A. **Risco Baixo:** Complexidade de 1 a 10.
B. **Risco Moderado:** Complexidade de 11 a 20.
C. **Risco Elevado:** Complexidade de 21 a 50.
D. **Risco Altíssimo:** Complexidade de 51 a N.

Tais valores servem de referência e não como regra para medir a complexidade no sistema. Caso o sistema venha a medir riscos elevados ou altíssimos não quer dizer que o mesmo não possa ser reestruturado.

3.4.1.2 Cobertura de Testes

A métrica de cobertura de testes tem como propósito responder o questionamento de "*O quanto o teste é completo*". A cobertura se baseia comumente em medidas que cubram os requisitos do software e do código fonte. A cobertura de teste pode ser baseada em dois fatores, em requisitos ou em código. Baseados em requisitos consistem em medidas de integridade relacionadas a um requisito e com verificações de casos de uso, são suficientes para produzir medidas quantificáveis da integridade do teste, onde será analisada a porcentagem com que os requisitos dos testes foram verificados. Já baseadas em código, consistem em critérios de design e em implementações do código ou execuções de todas as linhas do código, sua estratégia de teste será formulada na quantidade de código que foi executada pelos testes. (RUP-VC, 2006)

Sua importância se dá pelo fato de ajudar na qualidade do código e a identificar as partes do código que não foram testadas, onde indiretamente ajuda na qualidade do software, além disso, auxilia a verificar se o sistema contém bugs quando comparado a um que não possui uma boa cobertura.

- **Porcentagens para cobertura de testes:**

Para que haja uma boa utilização da cobertura deve haver um número base a ser seguido na cobertura. Recomenda-se que os testes tenham uma cobertura entre 50% a 85%, Entretanto tais porcentagens não devem ser tratadas como ideal para uso da cobertura de testes e sim como base de metrificação. A porcentagem da cobertura varia desde a linguagem de programação usada ao seu requisito.

Além dessas métricas elas por si próprias colaboram para criação de outras. A própria cobertura de teste faz parte da métrica de ***Cobertura***, onde fazem parte dessa métrica a *cobertura de código*, *cobertura de linhas*, *cobertura de ramificações* entre outras. Logo, tais métricas acabam sendo calculadas e analisadas junto com outras e a necessidade de apresentá-las em relatórios se dá na escolha da definição do projeto ou da ferramenta utilizada para gerar tais relatórios.

3.4.2 Ferramentas

Em sua maioria as ferramentas de análise de testes estruturais cobrem as métricas de cobertura de teste, de complexidade ciclomática e entre outras métricas de estruturação de código, como variáveis repetidas, linhas de código por método, quantidade de métodos, padrões de nomenclatura, critérios de segurança e entre outros.

3.4.2.1. SonarQube

SonarQube é uma ferramenta de qualidade de código de multilinguagem (suporta várias linguagem de programação), fazendo uso de revisão de código com o propósito de detectar bugs, falhas, vulnerabilidades e outros fatores a respeito do código-fonte, podendo ser integrado no fluxo do trabalho para efetuar a inspeção do código em todas as suas ramificações e solicitações de ***pull´s***. A SonarQube se trata de um scanner que pode ser executado por meio de sua construção ou como parte da integração contínua (CL). A ferramenta faz uma varredura sempre que é acionado o processo de construção. (SONARQUBE, 2021, p.1). Para que possa ser utilizada a ferramenta é necessário possuir a *SonarQube* e o *SonarScanner*, como o próprio nome sugere o sonarscanner irá

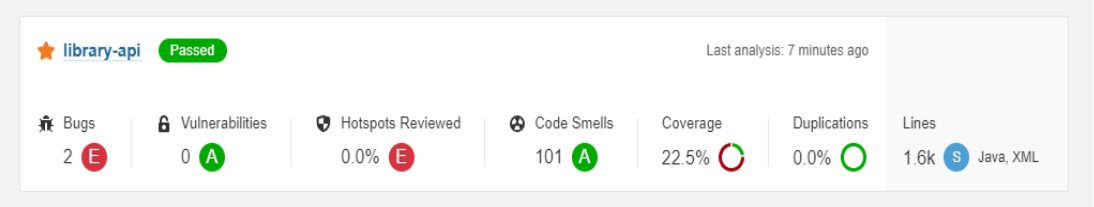
escanear/analisará o projeto onde tal análise poderá ser visualizada pelo sonarqube, sua instalação comparada às demais ferramentas apresentadas se torna mais complexa.

A SonarQube oferece recursos como (SONARQUBE, 2021, p.2):

- **SonarLint:** Trata-se de um produto complementar que serve para detectar e corrigir problemas antes que cheguem a solicitação de pull;
- **Quality Gate:** Trata-se de um recurso onde é possível verificar se o projeto está pronto para ir para produção;
- **Clean as You Code:** Trata-se de uma abordagem para a qualidade do código onde elimina vários desafios que vem ao utilizado abordagens tradicionais;
- **Issues:** Se trata de problemas em trechos de código, bugs, vulnerabilidades, falhas de manutenção entre outros fatores que podem ser considerados como problemas ao se analisar o projeto;
- **Security Hotspots:** Trata-se da detecção de partes do código ou projeto que são sensíveis à segurança onde precisam ser revisados.

A SonarQube permite ter uma visão resumida da análise dos projetos, na figura 10 a ferramenta exibe alguns dados, como a porcentagem da cobertura, bugs, linhas, vulnerabilidades dentre outros que serve como uma análise rápida.

Figura 10 - Visão geral resumida da análise do projeto library-api.

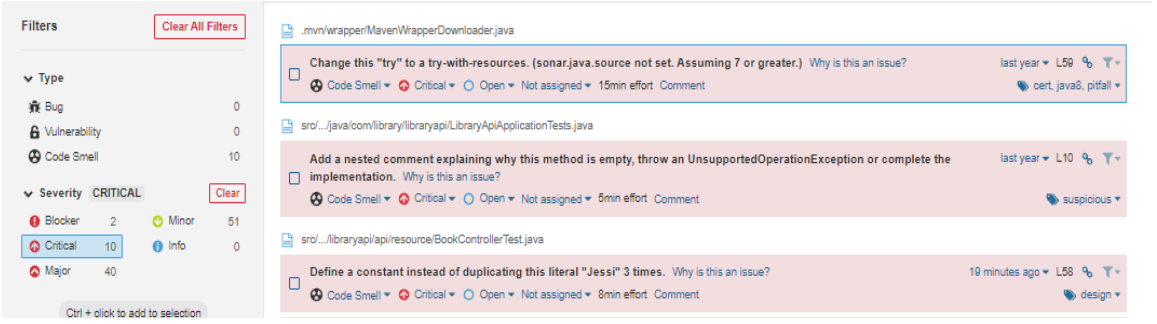


Fonte: (Próprio autor, 2021).

Em *Issues* ou Problemas é possível identificar problemas relacionados à estrutura do código e do teste como apresenta na figura 11. Além do mais, a própria ferramenta apresenta sugestões de melhorias e uma explicação do porque seria um problema, tal justificativa pode ser encontrada em *Why is this an issue?* (Por que isso é um problema?). Os problemas podem ser listados por meio do seu tipo (Type), sendo eles: *Bug*, *Vulnerability* e *Code Smell*.

Na figura 11 apresenta os problemas com gravidade crítica encontrados no projeto e ao clicar no problema é possível visualizar em qual parte do código se encontra e sua sugestão de melhoria ou ajuste. A sugestão que a ferramenta apresenta para o usuário em um dos problemas (figura 12) é de definir uma constante ao em vez de setar o valor "Jessi" duplicar nos testes.

Figura 11 - Issues do projeto library-api.



Fonte: (Próprio autor, 2021).

Figura 12 - Análise da Issue do projeto library-api.

Fonte: (Próprio autor, 2021).

A SonarQube apresenta em *Code* ou Código (figura 13) a porcentagem de cobertura organizada em pacotes e arquivos. Além da cobertura é apresentado a quantidade de linhas, de bugs, de vulnerabilidades, de pontos de segurança e de *code smells* que podem ser visualizados a nível de pacote ou arquivo.

Figura 13 - Módulo de Código na análise do projeto library-api.

	Lines of Code	Bugs	Vulnerabilities	Code Smells	Security Hotspots	Coverage	Duplications
main/java/com/library/libraryapi	637	0	0	11	0	45.4%	0.0%
api	289	0	0	9	0	43.3%	0.0%
config	35	0	0	0	0	100%	0.0%
exception	6	0	0	0	0	100%	0.0%
model	94	0	0	0	0	30.9%	0.0%
service	192	0	0	0	0	67.7%	0.0%
LibraryApiApplication.java	21	0	0	2	0	50.0%	0.0%

Fonte: (Próprio autor, 2021).

Algo que difere é o padrão de cores usada na análise de cobertura (figura 14), onde na visualização do arquivo não possui a cor amarelo para representar trechos que não foram cobertos totalmente, ao invés do amarelo a sonarqube usa a cor vermelha com traços em rosa representando que nem todos os caminhos naquele trecho de código foram cobertos. Já para representar trechos cobertos totalmente e não cobertos são representados pelas cores verde e vermelho.

Figura 14 - Análise da Sonarqube em código-fonte.

Fonte: (Próprio autor, 2021).

3.4.2.3. EclEmma

EclEmma é um ferramenta que tem como métrica e foco principal a cobertura de código e de teste em Java. O EclEmma se trata de uma ferramenta para o ambiente de desenvolvimento integrado (IDE) ***Eclipse*** que está disponível sob a licença pública eclipse. A ferramenta se trata de um plugin para a IDE que pode ser encontrado pelo site ou pode ser

encontrado por meio do ***Cliente Eclipse Marketplace***, bastando apenas pesquisar pela ferramenta e instalar na IDE. (ECLEMMA, 2017, p.1).

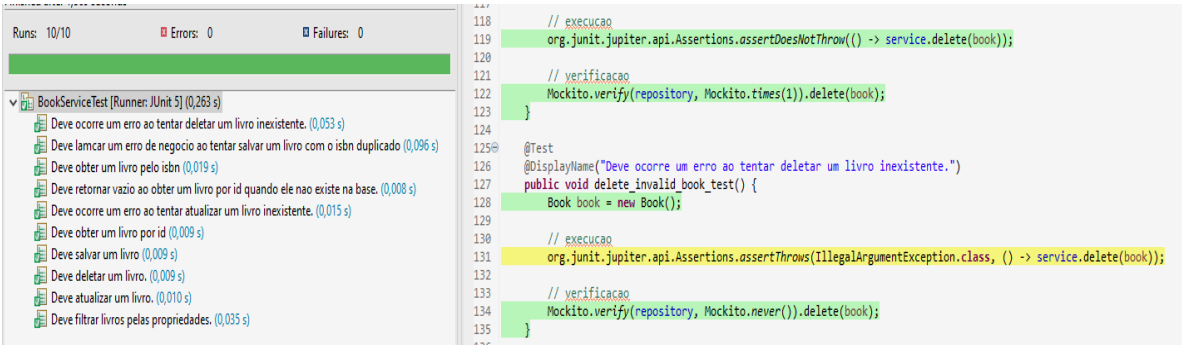
EclEmma traz a análise de cobertura diretamente para o ambiente de trabalho do eclipse que traz benefícios como (ECLEMMA, 2017, p.1):

- **Ciclo de desenvolvimento e teste rápido:** Lançamentos de execuções como os testes no *JUnit*, podem ser analisados diretamente quando ocorrem a execução da cobertura;
- **Análises ricas de cobertura:** O resultado da cobertura é imediatamente resumido e destacado na IDE;
- **Não invasivo:** Não requer a modificação do projeto ao realizar a configuração da ferramenta na IDE.

Após a instalação da ferramenta é adicionado um módulo de *coverage As* (cobertura) que funciona exatamente como os módulos de *run As* (executar) e *Debug As* (depurar).

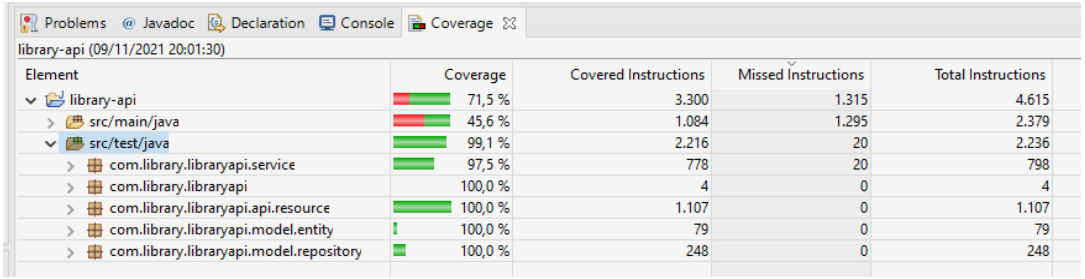
Na figura 15 é apresentada a visão geral após a execução da cobertura, seja ela em um teste ou todos os testes. As informações da cobertura ficam automaticamente disponíveis na IDE, como ilustra na figura 16, que mostra a porcentagem total da cobertura por arquivo e a quantidade de instruções cobertas, instruções falhas e o número total de instruções que possuem no projeto ou em cada arquivo. Com o resultado da cobertura é possível listar o resumo dos testes no projeto, permitindo o detalhamento até o nível do método como apresentado na figura 15, o resultado também pode ser visto no editor de código, onde as cores das linhas no código se destacam seja por meio das cores verde (totalmente coberta), amarelo (parcialmente coberto) e vermelho (não coberto).

Figura 15 - Visão geral do EclEmma.



Fonte: (Próprio autor, 2021).

Figura 16 - Análise da cobertura no EclEmma.



Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
library-api	71,5 %	3.300	1.315	4.615
src/main/java	45,6 %	1.084	1.295	2.379
src/test/java	99,1 %	2.216	20	2.236
com.library.libraryapi.service	97,5 %	778	20	798
com.library.libraryapi	100,0 %	4	0	4
com.library.libraryapi.api.resource	100,0 %	1.107	0	1.107
com.library.libraryapi.model.entity	100,0 %	79	0	79
com.library.libraryapi.model.repository	100,0 %	248	0	248

Fonte: (Próprio autor, 2021).

Apesar da ferramenta ser utilizada diretamente no eclipse, nela é possível importar e exportar relatórios de análise de cobertura. Na importação é permitido arquivos de dados de execução no ***JaCoCo*** (*.exec) e na exportação, o relatório podem ser em formatos de ***HTML***,

XML, ***CSV*** ou como arquivos de dados de execução do ***JaCoCo*** (*.exec). (ECLEMMA, 2017, p.2).

O EclEmma desde a versão 2.0 é baseado na ferramenta ***JaCoCo*** sua integração tem como foco o suporte ao desenvolvimento individual do programador de modo a ser altamente interativo. Apesar do EclEmma desde a versão 2.0 se basear na JaCoCo, originalmente foi inspirado tecnicamente na biblioteca ***EMMA***, que foi desenvolvida por Vlad Roubtsov. (ECLEMMA, 2017, p.2).

3.4.2.4. JaCoCo

JaCoCo ou Java Code Coverage é um ferramenta que permite a geração de relatórios de análise de códigos feitos em ***Java***. Tais relatórios servem para encontrar trechos do código não cobertos durante a execução dos testes. (ICHI.PRO, 2021, p. 9). JaCoCo é uma biblioteca de cobertura e análise de código e testes criada pela equipe de desenvolvedores do ***EclEmma***, com base e experiência na utilização do uso e integração da biblioteca já existente. (ECLEMMA, 2017, p.1).

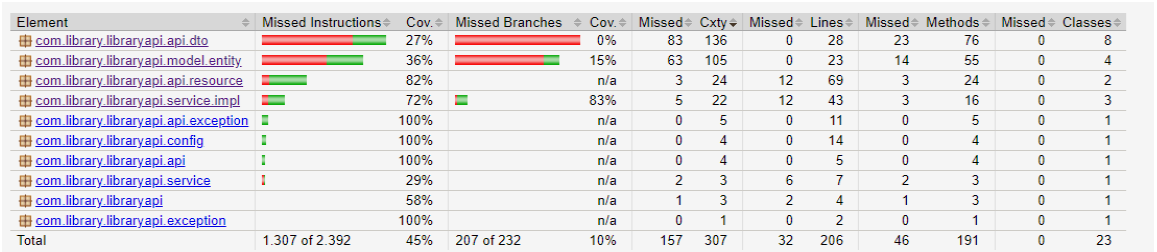
A ferramenta pode ser "instalada" por meio do ***Apache Maven ou Maven*** no arquivo ***POM*** e executado por meio do codeBuild ou executando o comando ***mvn jacoco:report*** do JaCoCo, assim que for executado o comando será gerado o relatório da análise da JaCoCo. A própria IDE informa o caminho do relatório. (BAELDUNG, 2021, p.4).

A ferramenta fornece principalmente três métricas importantes, sendo (BAELDUNG, 2021, p.5):

1. **Cobertura de linhas:** Reflete a quantidade de código que foi exercida com base no número de instruções de código de *byte* chamadas pelos testes;
2. **Cobertura de ramificações:** Reflete a porcentagem de ramificações exercidas no código (if/else e switch);
3. **Complexidade ciclomática:** Reflete a complexidade do código, ou seja fornece o número de caminhos necessários para cobrir todos os caminhos/ramificações possíveis.

Visualmente a estética da análise da JaCoCo se assemelha ao EclEmma como apresenta na figura 17, entretanto se diferem do tipo de análise que a ferramenta proporciona a começar que ela não traz somente a cobertura de testes, mas também a complexidade ciclomática do projeto entre outras métricas.

Figura 17 - Visão geral da análise na JaCoCo.



Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.library.libraryapi.api.dto		27%		0%	83	136	0	28	23	76	0	8
com.library.libraryapi.model.entity		36%		15%	63	105	0	23	14	55	0	4
com.library.libraryapi.api.resource		82%		n/a	3	24	12	69	3	24	0	2
com.library.libraryapi.service.impl		72%		83%	5	22	12	43	3	16	0	3
com.library.libraryapi.api.exception		100%		n/a	0	5	0	11	0	5	0	1
com.library.libraryapi.config		100%		n/a	0	4	0	14	0	4	0	1
com.library.libraryapi.api		100%		n/a	0	4	0	5	0	4	0	1
com.library.libraryapi.service		29%		n/a	2	3	6	7	2	3	0	1
com.library.libraryapi		58%		n/a	1	3	2	4	1	3	0	1
com.library.libraryapi.exception		100%		n/a	0	1	0	2	0	1	0	1
Total	1.307 of 2.392	45%	207 of 232	10%	157	307	32	206	46	191	0	23

Fonte: (Próprio autor, 2021).

Na figura 17 apresenta o relatório com a análise do projeto. No relatório são apresentadas as colunas (ICHI.PRO, 2021, p.12):

- **Element:** Reflete aos pacotes no projeto;
- **Missed Instructions e Cov:** Reflete a medição e percentual do número de instruções de código que foram cobertos nos testes;

- **Missed Branches e Cov:** Reflete a medição e percentual do número de caminhos/ramificações de código que foram cobertos nos testes;
- **Missed e Cxty:** Reflete ao número de complexidade ciclomática presente no projeto, seja a nível de pacote ou arquivo;
- **Missed e Lines:** Reflete o número de linhas no código que possui cobertura e que não possuem cobertura;
- **Missed e Methods:** Reflete o número de métodos no código que possui cobertura e que não possuem cobertura;
- **Missed e Classes:** Reflete o número de classes no código que possui cobertura e que não possuem cobertura.

O relatório ajuda a analisar visualmente a cobertura usando a dinâmica de cores (figuras 18 e 19) assim como outras ferramentas já apresentadas, onde (BAELDUNG, 2021, p.5):

- **Diamante vermelho:** Nenhum dos ramos foram exercitados durante o teste;
- **Diamante amarelo:** Parcialmente dos ramos foram exercitados durante o teste;
- **Diamante verde:** Todos os ramos foram exercitados durante o teste.

Figura 18 - Análise da JaCoCo com diamantes verdes e vermelhos.

```
44.  @Override
45.  public Page<Loan> find(LoanFilterDTO filterDTO, Pageable pageable) {
46.      return repository.findByBookIsbnOrCustomer(filterDTO.getIsbn(),filterDTO.getCustomer(),pageable);
47.  }
48.
49.  @Override
50.  public Page<Loan> getLoansByBook(Book book, Pageable pageable) {
51.      return repository.findByBook(book, pageable);
52.  }
53.
54.  @Override
55.  public List<Loan> getAllLateLoans() {
56.      final Integer loanDays = 4;
57.      LocalDate threeDaysAgo = LocalDate.now().minusDays(loanDays);
58.      return repository.findByLoanDateLessThanAndNotRereturned(threeDaysAgo);
59.  }
```

Fonte: (Próprio autor, 2021).

Figura 19 - Análise da JaCoCo com diamantes verdes e amarelos.

```
37.     @Override
38.     public void delete(Book book) {
39.         if(book == null || book.getId() == null){
40.             throw new IllegalArgumentException("Book id cant be null.");
41.         }
42.         this.repository.delete(book);
43.     }
44.
45.     @Override
46.     public Book update(Book book) {
47.         if(book == null || book.getId() == null){
48.             throw new IllegalArgumentException("Book id cant be null.");
49.         }
50.         return this.repository.save(book);
51.     }
```

Fonte: (Próprio autor, 2021).

Tabela 1 - Análise comparativa das ferramentas.

	FERRAMENTAS		
CARACTERÍSTICAS RELEVANTES	SONARQUBE	ECLEMMA	JACOCO
Pode ser usado para várias linguagens de programação (Geral)	Sim	Não	Não
Voltada/Criada para linguagens de programação o específica	Não	Sim	Sim

Informa a cobertura (código e teste) no relatório	Sim	Sim	Sim
Informa a complexidade ciclomática no relatório	Sim	Não	Sim
Configuração e Instalação no projeto	Complexa	Simples	Simples
Possui relatórios dinâmicos com cores	Sim	Sim	Sim
Uso geral da ferramenta	Qualidade do código	Cobertura de código	Cobertura de código

Fonte: (Próprio autor, 2021).

Diante do que foi apresentado sobre as ferramentas, podemos verificar que algumas delas são mais complexas, robustas e específicas que outras, entretanto não é porque a ferramenta possui diversas funcionalidades ou ausência que será ideal para o projeto. A ferramenta ideal para o projeto deve ser pensada na sua utilização e na necessidade com que se queira obter as informações.

3.5 VANTAGENS E DESVANTAGENS DOS TESTES ESTRUTURAIS

Ao decorrer do desenvolvimento do software e a implementação dos testes é possível identificar vantagens ao usá-los. Entretanto, o uso de testes podem trazer desvantagens dependendo do ponto de vista em relação ao seu uso. (HAMILTON, 2021, p.7).

- **Algumas de suas vantagens são:**
  - Otimização do código onde é possível encontrar erros ocultos;
  - Seus casos de testes podem ser facilmente otimizados;
  - Todos os caminhos do código geralmente serão cobertos, entretanto o teste ficará mais complexo;
  - O teste pode começar desde o início do SDLC (*Software Development Life Cycle*), mesmo se a GUI (*Graphical User Interface*) não estiver disponível.

- **Algumas de suas desvantagens são:**
  - Os testes podem ser bastante caro e complexo para empresa ou provedora;
  - Alguns desenvolvedores que executam os casos de teste não costumam gostar;
  - Testes não detalhados podem levar a erros de produção;
  - Os testes requerem uma compreensão detalhada de programação e de implementação;
  - Por os testes consumirem tempo e esforço, aplicações grandes podem ocupar um tempo maior para serem cobertos completamente.

3.6 CUSTOS E BENEFÍCIOS DO USO DE TESTES

3.6.1 Custos

Como mencionado no tópico anterior (*Vantagens e Desvantagens dos Testes Estruturais*) o ato de criar testes exige um certo custo, seja para criá-lo, mantê-lo ou corrigi-lo. Segundo Myers (2011), 50% do custo total da construção do software é gasto com atividades relacionadas ao teste. Entretanto, a porcentagem varia de acordo com o escopo e custo de cada projeto.

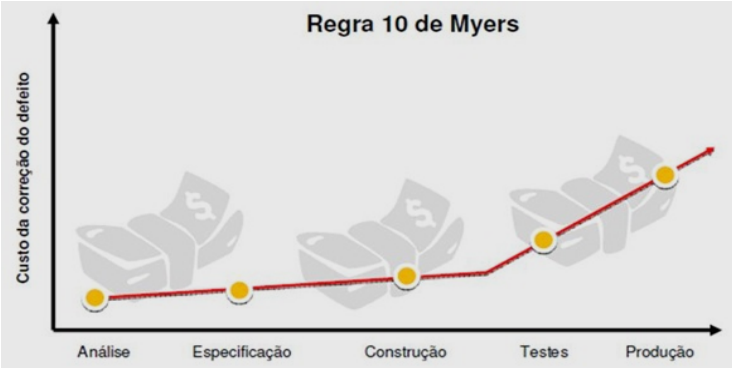
Os testes consomem recursos principalmente com:

- **Profissionais -** Devem ser adequados e qualificados para as atividades sendo eles: desenvolvedores, analistas de testes, testadores, etc;
- **Equipamentos -** Seja máquinas ou plataformas que permitam a criação de ambientes de teste que simula ambientes de produção;
- **Ferramentas -** Como visto nos tópicos anteriores, a criação de testes requer o uso de ferramentas que ajudem em sua construção ou que auxiliem em sua análise.

Myers propôs a **"Regra 10 de Myers"**, onde ele diz que o custo para corrigir defeitos aumenta em *10x* para cada etapa em que o projeto de software avança. O custo aumentará mais caso o erro for identificado pelo cliente, do que um erro detectado pela equipe de desenvolvimento ou testadores. O custo da correção tende a ser maior quando-se descoberto mais tarde (MYERS, 2004).

Como expresso na figura 10, que ilustra a variação do custo à medida que os testes são implementados no processo de criação de um projeto, o custo de encontrar defeitos na etapa de análise são mais baratos do que encontrar na etapa de produção, ou seja, testes bem escritos na etapa de construção (etapa que ocorre o desenvolvimento do software) podem ocasionar em uma entrega com menos defeitos mesmo que no início tenha gastado mais do que projetos que focam em *teste* nas etapas de testes e produção.

Figura 20 - Regra 10 de Myers.



Fonte: Modificado de Galitezi (2012, p.1).

Softwares com defeitos a nível de produção, não ocasionam somente a bugs e falhas, mas podem gerar incômodos e transtorno, exemplo disto é o ocorrido com a GOL.

De acordo com CAMPOS (2010, n.p.)

> A Agência Nacional de Aviação Civil (Anac) informou nesta terça-feira que a companhia aérea Gol disse que um problema no software para planejamento de escala da tripulação gerou dados incorretos que resultaram no "planejamento inadequado da malha aérea e da jornada de trabalho dos tripulantes". Hoje, a Gol foi convocada pela agência a apresentar um plano de ação para atender os passageiros de voos cancelados ou atrasados. A companhia operava cerca de 70% dos voos atrasados ontem em todo o País. De acordo com a Anac, a Gol afirmou que o problema no sistema aconteceu em julho, durante um upgrade no programa. "Por essa razão, foi adotada novamente a configuração de escala do mês anterior", disse à agência, em nota: "O sistema era utilizado há três meses, segundo a companhia, e com o conhecimento da Anac.

Além deste, há cenários onde tais defeitos podem acarretar a acidentes gravíssimos, como em softwares de Hospitais, de Aeroportos, de Bancos, de construção, entre outros, ou seja, “*testes podem salvar vidas*".

3.6.2 Benefícios

A criação de testes em um projeto seja ela na etapa de construção do software ou em outra não influencia somente na qualidade do sistema, eles trazem benefícios como (BAUMGARTNER, 2021, p.2-3; FRAGA, 2017, p.4-13):

A. Avaliar as funcionalidades, assim alinhando as expectativas sobre o produto;
B. Localizar defeitos (de codificação, funcionalidade, design, etc) antes da entrega para o cliente;
C. Sistema estável, livre de defeitos que impedem sua usabilidade;
D. Melhora a manutenção do sistema;
E. Reduz os custos a longo prazo;
F. Segurança dos dados;
G. Trabalhos colaborativos entre os desenvolvedores se torna mais seguro;
H. Garante a entrega de um software funcional e com qualidade.

4 METODOLOGIA

O presente trabalho é de finalidade teórica com natureza descritiva, tendo como objetivo apresentar as informações coletadas e abordadas a respeito do uso de testes estruturais no processo de criação de um projeto de software. Foi realizada uma pesquisa bibliográfica em artigos, revistas e blogs em português e inglês, utilizando a ferramenta de busca *Google Acadêmico*. A busca e a análise dos dados se deu de modo secundário por meio dos termos como: “teste de software", “teste de caixa branca”, “teste integrados”, "métricas de teste”, "gerência de projeto”, "arquitetura de softwares” e “ferramentas de teste”, presentes nos artigos, documentações, revistas e blogs encontrados.

Sua metodologia se deu por meio das duas técnicas sendo elas a coleta de dados e a análise de dados, onde na coleta de dados foi possível fazer diversas pesquisas bibliográficas escrita por autores como: MYERS, HETZEL, BASTOS, MACORATTI, MCCABE, MALDONADO, entre outros, que contribuíram de forma significativa para diversos conceitos sobre engenharia e teste de software. Na análise de dados foi utilizado a forma qualitativa, onde foi apresentado técnicas e ferramentas do uso de testes estruturais para medir, analisar e construir o desempenho dos testes no projeto e como tais medidas se relacionam na qualidade e desempenho dos softwares nas empresas.

5 CONSIDERAÇÕES FINAIS E TRABALHOS FUTUROS

Os testes estruturais como visto ao longo do desenvolvimento do trabalho são de suma importância tanto para qualidade do software como para a codificação do projeto. O presente estudo teve como objetivo apresentar a utilização de testes unitários e integrados no processo de codificação no desenvolvimento do software, além do mais, foi apresentado conceitos que englobam desde sua utilização, etapas, características e em ferramentas de análise de teste. Tais ferramentas são baseadas em métricas, sendo as mais utilizadas em testes estruturais a cobertura de testes e a complexidade ciclomática, entretanto todas as ferramentas apresentadas consistem em ferramentas baseadas na cobertura de código. Para o projeto *library-api* que foi utilizado como base e criação dos testes e na instalação das ferramentas, a ferramenta que apresenta um bom desempenho para o projeto é a JaCoCo. Apesar do

EclEmma trazer uma vantagem significativa por apresentar o relatório direto na IDE a ferramenta fica exclusiva do uso do desenvolvedor instalar ou não, por outro lado como a instalação da JaCoCo pode ser feita pelo Maven no arquivo *POM* todos os desenvolvedores que trabalharam naquele projeto terão a ferramenta. Já a SonarQube traz consigo diversas funcionalidades e não está reclusa apenas a uma linguagem de programação, entretanto por se tratar de uma ferramenta robusta, para o projeto *library-api* a ferramenta não será aproveitado o máximo dos recursos dela. Embora um projeto contenha testes e ferramentas não é garantia que o projeto irá se desenvolver em um software sem falhas, bugs ou não possuir retrabalhos após a sua entrega. Para que seja desenvolvido um bom projeto é necessário analisar quais serão as métricas, ferramentas, padrões e casos de teste entre outros fatores para construção do projeto, além disso, se tais fatores forem trabalhados de forma errônea podem trazem desde incômodos para o usuário final a acidentes gravíssimos podendo acarretar a grandes custos para a empresa.

Concluindo-se então, que teste de software não somente testes estruturais se tornou um dos fatores cruciais para vida útil do sistema, sendo responsáveis por dizer se o sistema foi bem construído e se importa com o resultado final. No projeto não se deve pensado em testes somente na etapa de teste, pois como foi apresentado quanto mais tarde são implementados os testes mais caros ficam para serem implementados o que não seria bom para a empresa, logo chegando à conclusão de que testes não são só utilizados para garantir qualidade mas também para garantir sua permanência no mercado.

Em relação a trabalhos futuros o estudo apresenta ferramentas de análise de testes que foram integradas a um projeto simples que consiste em um API de livraria, sendo um dos primeiros passos aplicar a usabilidade dessas ou de outras ferramentas em projetos e cenários reais, com o objetivo de analisar o desempenho dessas ferramentas.

REFERÊNCIAS BIBLIOGRÁFICAS

ABDALA, Daniel D. Testes de Software. **Central de conteúdos,** Uberlândia, v. 1, n. 1, p. 2, nov. 2011. Disponível em: <http://www.facom.ufu.br/~abdala/DAS5312/Testes.pdf>. Acesso em: 19 ago. 2021.

BAELDUNG. **Intro to JaCoCo.** Disponível em: <https://www.baeldung.com/jacoco>. Acesso em: 13 nov. 2021.

BASTOS, Aderson; RIOS, Emerson; CRISTALLI, Ricardo; MOREIRA, Trayahú. **Base de conhecimento em teste de Software**. São Paulo: Martins, 2007.

BAUMGARTNER, Cristiano. **Conheça 5 benefícios dos testes automatizados de software.** Disponível em: <https://testingcompany.com.br/blog/conheca-5-beneficios-dos-testes-automatizados-de-software>. Acesso em: 22 out. 2021.

CAMPOS, Vagner. **Passageiros aguardam para fazer check-in no aeroporto de Guarulhos.** Disponível em: <http://noticias.terra.com.br/brasil/gol-diz-a-anac-que-falha-em-software-causou-erro-em-escalas,6dea4bc92690b310VgnCLD200000bbcceb0aRCRD.html>. Acesso em: 02 nov. 2021.

COUTINHO, Pedro Henrique Mannato. **Pós-Graduação Teste de Software**. Primeira edição. Itaparica: ESAB – Escola Superior Aberta do Brasil, 2011.

CRAIG, Rick D.; JASKIEL, Stefan P. **Systematic Software Testing.** Artech House 2002.

DARDE, Pablo Rodrigo. **O Padrão Triple A (Arrange, Act, Assert).** Disponível em: <https://medium.com/@pablodarde/o-padr%C3%A3o-triple-a-arrange-act-assert-741e2a94cf88>. Acesso em: 21 out. 2021.

DELAMARO, M. E; MALDONADO, J. C; JINO, M. **Introdução ao teste de software.** 4.ed. Rio de Janeiro: Elsevier, 2007.

DEVMEDIA. **Processo de Teste de Software**. Disponível em: <https://www.devmedia.com.br/processo-de-teste-de-software/23795>. Acesso em: 17 ago. 2021.

NUNIT. **Classic Model.** Disponível em: <https://docs.nunit.org/articles/nunit/writing-tests/assertions/assertion-models/classic.html>. Acesso em: 28 out. 2021.

ECLEMMA. **Java Code Coverage for Eclipse.** Disponível em: <https://www.eclemma.org/>. Acesso em: 12 nov. 2021.

ECLEMMA. **JaCoCo Java Code Coverage Library.** Disponível em: <https://www.jacoco.org/jacoco/index.html>. Acesso em: 31 out. 2021.

FERNANDES, Aguinaldo A. **Gerência de software através de métricas: garantindo a qualidade do projeto, processo e produto**. São Paulo: Atlas, 1995.

FRAGA, Daiane Azevedo De. **Motivos para considerar o teste de software indispensável.** Disponível em: <https://blog.umbler.com/br/qualidade-de-software-1-7-motivos-para-considerar-o-teste-de-software-indispensavel/>. Acesso em: 22 out. 2021.

GALITEZI, Thiago. **Regra 10 de Myers.** 11 de fevereiro de 2012. ll. color. Disponível em: <http://www.galitezi.com.br/2012/02/teste-de-software-e-o-custo-da-nao.html>. Acesso em: 13 nov. 2021.

GERSHON, Shmuel *et al*. **Visões sobre Teste de Software**: Diferentes perspectivas da comunidade de teste brasileira discutidas no DFTestes. Disponível em: <http://sucesurs.org.br/sites/default/files/2019-04/Vis%C3%B5es-sobre-Teste-de-Software.pdf>. Acesso em: 21 out. 2021.

GUEDES, Marylene. **Por que escrever testes automatizados?.** Disponível em: <https://www.treinaweb.com.br/blog/por-que-escrever-testes-automatizados>. Acesso em: 21 out. 2021.

HAMILTON, Thomas. **What is WHITE Box Testing? Techniques, Example & Types.** Disponível em: <https://www.guru99.com/white-box-testing.html>. Acesso em: 31 out. 2021.

HETZEL, William. **Guia completo ao teste de software.** Rio de Janeiro: Campus, 1987.

ICHI.PRO. **Melhore a cobertura e a qualidade do código Java com testes de unidade e JaCoCo.** Disponível em: <https://ichi.pro/pt/melhore-a-cobertura-e-a-qualidade-do-codigo-java-com-testes-de-unidade-e-jacoco-197097730553581>. Acesso em: 13 nov. 2021.

JAVATPOINT. **Mock vs. Stub vs. Spy.** Disponível em: <https://www.javatpoint.com/mock-vs-stub-vs-spy>. Acesso em: 28 out. 2021.

JAVATPOINT. **White Box Testing.** Disponível em: <https://www.javatpoint.com/white-box-testing>. Acesso em: 31 out. 2021.

KOSSOSKI, Clayton. Proposta de um método de teste para processos de desenvolvimento de software usando o Paradigma Orientado a Notificações. **Programa de Pós-graduação em Engenharia Elétrica e Informática Industrial, Universidade Tecnológica Federal do Paraná.** Curitiba, v. 1, n. 1, p. 52-67, ago. 2015. <http://repositorio.utfpr.edu.br/jspui/bitstream/1/1405/1/CT_CPGEI_M_Kossoski%2cClayton_2015.pdf>. Acesso em: 2 out. 2021.

MACORATTI, José Carlos. **Testes em desenvolvimento de Software – você precisa disto?**. Disponível em: <http://www.macoratti.net/tst_sw1.htm>. Acesso em: 13 set. 2021.

MALDONADO, J. C. **Critérios potenciais usos: Uma contribuição ao teste estrutural de software.** Faculdade de Engenharia Elétrica, UNICAMP. 1991.

MYERS, Glenford J. **The Art of Software Testing.** 3.ed. Editora: John Wiley & Sons Inc, 2011.

NIDHRA, Srinivas; DONDETI, Jagruthi. BLACK BOX AND WHITE BOX TESTING TECHNIQUES – A LITERATURE REVIEW. **International Journal of Embedded Systems and Applications (IJESA),** v. 2, n. 2, p. 38-47, Jun 2012. Disponível em: <https://www.researchgate.net/profile/S-Nidhra/publication/276198111_Black_Box_and_White_Box_Testing_Techniques_-_A_Literature_Review/links/570e313f08ae2b772e46aa40/Black-Box-and-White-Box-Testing-Techniques-A-Literature-Review.pdf>. Acesso em: 22 out. 2021.

ROCHA, Anne Caroline. **Principais tipos de Testes de Software**. Disponível em: <http://gtsw.blogspot.com/2007/10/tipos-de-testes-de-software.html>. Acesso em: 20 out. 2021.

ROCHA, Anne Caroline. **Relação entre tipos e técnicas de teste.** 1 de outubro de 2007. ll. color. Disponível em: <http://gtsw.blogspot.com/2007/10/tipos-de-testes-de-software.html>. Acesso em: 20 out. 2021.

RUP-VC. **Principais Medidas de Teste.** Disponível em: <https://www.cin.ufpe.br/~gta/rup-vc/core.base_rup/guidances/concepts/key_measures_of_test_62253EE4.html?nodeId=e7d4942b>. Acesso em: 02 nov. 2021.

SOARES, Diogo Castro Veloso. **Testes de unidade com JUnit.** 2007. Disponível em: <https://www.devmedia.com.br/testes-de-unidade-com-junit/4637>. Acesso em:22 out. 2021.

SOARES, João Paulo. **Principais técnicas de testes estruturais.** Disponível em: <https://www.treinaweb.com.br/blog/principais-tecnicas-de-testes-estruturais>. Acesso em: 22 out. 2021.

SOFTWARETESTINGHELP. **Code Coverage Tutorial: Branch, Statement, Function Coverage.** Disponível em: <https://www.softwaretestinghelp.com/code-coverage-tutorial/>. Acesso em: 02 nov. 2021.

SONARQUBE. **SonarQube Documentation.** Disponível em: <https://docs.sonarqube.org/latest>. Acesso em: 14 nov. 2021.

SYKES, Alistair. **Given When Then - Our Testing Approach: Why we write tests and how we go about them.** Disponível em: <https://proandroiddev.com/given-when-then-our-testing-approach-c9087b291c36>. Acesso em: 21 out. 2021.

TEDESCO, Kennedy. **Complexidade ciclomática, análise estática e refatoração.** Disponível em: <https://www.treinaweb.com.br/blog/complexidade-ciclomatica-analise-estatica-e-refatoracao>. Acesso em: 22 out. 2021.

VALENTE, Marco Tulio. **Engenharia de Software Moderna:** Princípios e Práticas para Desenvolvimento de Software com Produtividade.