UNIVERSIDADE FEDERAL RURAL DO SEMI-ÁRIDO
PRÓ-REITORIA DE GRADUAÇÃO
CENTRO MULTIDISCIPLINAR DE PAU DOS FERROS
BACHARELADO EM INTERDISCIPLINAR DA TECNOLOGIA DA INFORMAÇÃO

ASSUNAUENY RODRIGUES DE OLIVEIRA

PROJETO ARQUITETURAL DE UMA FERRAMENTA PARA GERAÇÃO AUTOMATIZADA DE CASOS DE TESTE

PAU DOS FERROS
2019

ASSUNAUENY RODRIGUES DE OLIVEIRA

PROJETO ARQUITETURAL DE UMA FERRAMENTA PARA GERAÇÃO AUTOMATIZADA DE CASOS DE TESTE

.

Monografia apresentada a Universidade Federal Rural do Semi-Árido como requisito para obtenção do título de Bacharel em Tecnologia da Informação.

Orientador: Álvaro Alvares de Carvalho César Sobrinho, Prof. Dr.
Co-orientador: Reudismam Rolim de Sousa, Prof. Dr.

PAU DOS FERROS

2019

©Todos os direitos estão reservados à Universidade Federal Rural do Semi-Árido.O conteúdo desta obra é de inteira responsabilidade do (a) autor (a), sendo o mesmo, passível de sanções administrativas ou penais, caso sejam infringidas as leis que regulamentam a Propriedade Intelectual, respectivamente, Patentes: Lei nº 9.279/1996, e Direitos Autorais: Lei nº 9.610/1998. O conteúdo desta obra tornar-se-á de domínio público após a data de defesa e homologação da sua respectiva ata, exceto as pesquisas que estejam vinculas ao processo de patenteamento. Esta investigação será base literária para novas pesquisas, desde que a obra e seu (a) respectivo (a) autor (a) seja devidamente citado e mencionado os seus créditos bibliográficos.

Ficha catalográfica elaborada pelo Sistema de Bibliotecas
da Universidade Federal Rural do Semi-Árido, com os dados fornecidos pelo(a) autor(a)

O48p

Oliveira, Assunaueny Rodrigues de.
Projeto arquitetural de uma ferramenta para geração automatizada de casos de teste / Assunaueny Rodrigues de Oliveira. - 2019.
66 f. : il.

Orientador: Álvaro Alvares de Carvalho César Sobrinho Sobrinho.
Coorientador: Reudismam Rolim de Sousa Sousa.
Monografia (graduação) - Universidade Federal Rural do Semi-árido, Curso de Tecnologia da Informação, 2019.

1. Teste de software. 2. Automatizado. 3. Arquitetura de software. 4. Caso de teste. I. Sobrinho, Álvaro Alvares de Carvalho César Sobrinho , orient. II. Sousa, Reudismam Rolim de Sousa, co-orient. III. Título.

O serviço de Geração Automática de Ficha Catalográfica para Trabalhos de Conclusão de Curso (TCC´s) foi desenvolvido pelo Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo (USP) e gentilmente cedido para o Sistema de Bibliotecas da Universidade Federal Rural do Semi-Árido (SISBI-UFERSA), sendo customizado pela Superintendência de Tecnologia da Informação e Comunicação (SUTIC) sob orientação dos bibliotecários da instituição para ser adaptado às necessidades dos alunos dos Cursos de Graduação e Programas de Pós-Graduação da Universidade.

ASSUNAUENY RODRIGUES DE OLIVEIRA

PROJETO ARQUITETURAL DE UMA FERRAMENTA PARA GERAÇÃO AUTOMATIZADA DE CASOS DE TESTE

Monografia apresentada a Universidade Federal Rural do Semi-Árido como requisito para obtenção do título de Bacharel em Tecnologia da Informação.

Defendida em: 20 / 03 / 2019 .

BANCA EXAMINADORA

Álvaro Alvares de C. S. Sobrinho

Prof. Dr. Álvaro Alvares de Carvalho César Sobrinho (UFERSA)

Presidente

Lenardo Chaves e Silva

Prof. Dr. Lenardo Chaves e Silva (UFERSA)

Membro Examinador

Helder Fernando de Araujo Oliveira

Prof. Dr. Helder Fernando de Araújo Oliveira (UFERSA)

Membro Examinador

A minha mãe Maria Nazarene, ao meu pai Francisco Sueldo e a minha irmã Suênia Rodrigues, eu dedico este trabalho.

AGRADECIMENTOS

Agradeço primeiramente a Deus por me permitir o dom da vida e por proporcionar vários momentos felizes e tristes.

Agradeço aos meus pais, Maria Nazarene e Francisco Sueldo por sempre dividirem esse sonho comigo e sempre me guiarem no melhor caminho.

Agradeço a minha irmã Suênia Rodrigues por sempre acreditar no meu potencial.

Agradeço ao meu orientador, Álvaro Alvares, pela disponibilidade e paciência para me ajudar.

Agradeço ao meu co-orientador, Reudismam Rolim, pela disponibilidade e paciência para me ajudar.

Agradeço a todos os meus amigos e colegas que sempre me deram forças para lutar pelos meus sonhos e objetivos.

RESUMO

Neste trabalho é abordado a geração automatizada de casos de teste. O foco está nas necessidades de testadores de software, como forma de amenizar os custos gerados com testes manuais e permitir uma maior qualidade na definição de conjuntos de testes de software. Neste contexto é desenvolvido o projeto arquitetural de uma ferramenta focada na automatização da geração de casos de teste para aplicações desenvolvidas com diferentes paradigmas de programação. Portanto, para desenvolver essa arquitetura foram definidos requisitos funcionais e não funcionais. A partir desses requisitos e do atributo de qualidade, modificabilidade, foi possível a aplicação de uma técnica para definição de arquitetura conhecida por *Attribute-Driven Design* (ADD). A técnica possibilitou que a arquitetura fosse desenvolvida, porém, foi necessário aplicar uma técnica de avaliação para verificar o nível de conformidade da arquitetura com requisitos e o atributo de qualidade. A técnica de avaliação utilizada foi a *Architecture Tradeoff Analysis Method* (ATAM). Com a avaliação, não foram encontrados conflitos entre a arquitetura e o objetivo final.

Palavras-chaves: Teste de software automatizado. Arquitetura de software. Caso de teste.

ABSTRACT

In this work the automated generation of test cases is approached. The focus is on the needs of software testers, as a way to mitigate the costs generated with manual testing and to allow a higher quality in the set of tests for software. In this context the architectural design of a tool focused on the automation of the generation of test cases for applications developed with different programming paradigms is developed. Therefore, functional and non-functional requirements were defined to develop this architecture. From these requirements and the attribute of quality, modifiability, it was possible to apply a technique for architecture definition known as Attribute-Driven Design (ADD). The technique allowed the architecture to be developed, however, it was necessary to apply an evaluation technique to verify the level of conformity of the architecture with requirements and the attribute of quality. The evaluation technique used was the Architecture Tradeoff Analysis Method (ATAM). With the evaluation, no conflicts were found between the architecture and the final goal.

Keywords: Automated Software Testing. Software architecture. Test case.

LISTA DE FIGURAS

FIGURA 1 - EXEMPLO DE GERAÇÃO DE GFC 20

FIGURA 2 - DIAGRAMAS DE CASOS DE USO 32

FIGURA 3 - INSERÇÃO DE CRITÉRIO CUSTOMIZADO IMPERATIVO 33

FIGURA 4 - INSERÇÃO DE CRITÉRIO CUSTOMIZADO ORIENTADO A OBJETOS ... 33

FIGURA 5 - INSERÇÃO DE NOVO PARADIGMA DE TESTE 34

FIGURA 6 - INTEGRAÇÃO DE PLUGIN COM A FERRAMENTA 34

FIGURA 7 - MODIFICAÇÃO DE INTERFACE DE USUÁRIO 35

FIGURA 8 - MÓDULOS MAIS EXTERNOS 36

FIGURA 9 - PADRÃO ARQUITETURAL 37

FIGURA 10 - LISTA DE SUNMÓDULOS DO MÓDULO GERADOR 38

FIGURA 11 - LISTA DE SUBMÓDULOS DO MÓDULO PLATAFORMA 39

FIGURA 12 - LISTA DE SUBMÓDULOS DO MÓDULO INTERFACE DE USUÁRIO ... 40

FIGURA 13 - LISTA DE COMPOSIÇÃO GERAL DO MÓDULO GERADOR 41

FIGURA 14 - LISTA DE COMPOSIÇÃO GERAL DO MÓDULO PLATAFORMA 43

FIGURA 15 - LISTA DE COMPOSIÇÃO GERAL DO MÓDULO INTERFACE DE USUÁRIO 45

FIGURA 16 - VISÃO DE USO ENTRE SUBMÓDULOS 46

FIGURA 17 - VISÃO EM CAMADAS 47

FIGURA 18 - DIAGRAMA DE CLASSES E SUAS INTERFACES 48

FIGURA 19 - DIAGRAMADE IMPLANTAÇÃO 49

FIGURA 20 - DIAGRAMA DE ESTADO PARA GERAR CASOS DE TESTE 51

FIGURA 21 - DIAGRAMA DE ESTADO PARA INSERIR CRITÉRIO CUSTOMIZADO 52

FIGURA 22 - DIAGRAMA DE ESTADO PARA INSERIR NOVO PARADIGMA DE TESTE 53

FIGURA 23 - FLUXO QUE REPRESENTA A GERAÇÃO DE CASOS DE TESTE 58

FIGURA 24 – CRITERIO INTRAMÉTODO 59

FIGURA 25 - GFC GERADO COM BASE EM UMA UNIDADE DE POO 59

FIGURA 26 - CRITÉRIO TODAS-ARESTAS 61

FIGURA 27 - GFC GERADO COM BASE EM NOVA UNIDADE DO POO 61

FIGURA 28 - INSERINDO CRITÉRIO CUSTOMIZADO NO POO 63

FIGURA 29 - INSERINDO CRITERIO CUSTOMIZADO DO PARADIGMA IMPERATIVO 64

LISTA DE TABELAS

TABELA 1 – DESCRIÇÃO DOS PAPÉIS DO ATAM.....24
TABELA 2 – CASOS DE TESTE POO.....60
TABELA 3 – CASOS DE TESTE DO PARADIGMA IMPERATIVO.....63

LISTA DE ABREVIATURAS E SIGLAS

ADD *Attribute-Driven Design*

ATAM Architecture Tradeoff Analysis Method

CFG *Control Flow Graph*

GFC Grafo de Fluxo de Controle

POO Paradigma Orientado a Objetos

SysML *Systems Modeling Language*

UML Unified Modeling Language

SUMÁRIO

1. INTRODUÇÃO 15

1.1. Objetivos 17

1.1.1. Objetivo geral 17

1.1.2. Objetivos específicos 17

1.2. Metodologia 17

2. REFERENCIAL TEÓRICO 19

2.1 Teste de software 19

2.1.1 Paradigma Imperativo 20

2.1.2 Paradigma Orientado a Objetos 21

2.1.3 *Framework* JUnit 5 22

2.2 Linguagem de Modelagem Unificada 22

2.3 Plugin Java 23

2.4 Arquitetura de software 23

3. TRABALHOS RELACIONADOS 27

3.1 Geração de Grafo de Fluxo de Controle 27

3.2 Geração de casos de teste 28

4. PROJETO ARQUITETURAL DE UMA FERRAMENTA PARA GERAÇÃO AUTOMATIZADA DE CASOS DE TESTE 29

4.1 Casos de uso 29

4.2 Requisitos de modificabilidade 32

4.2.1 Inserir critério de teste customizado com paradigma imperativo 32

4.2.2 Inserir critério de teste customizado com paradigma orientado a objetos 33

4.2.3 Inserir novo paradigma de teste 33

4.2.4 Integrar Plugin com a ferramenta 34

4.2.5 Modificar interface de usuário 34

4.3 Projeto 35

4.3.1 Visão de decomposição de módulos 35
4.3.2 Visão de uso entre submódulos 45
4.3.3 Visão em camadas 46
4.3.4 Visão de estrutura em classes 47
4.3.5 Visão de estrutura de alocação 49
4.4 Exemplos de comportamentos de módulos do sistema 50
5. AVALIAÇÃO DA ARQUITETURA APLICANDO A TÉCNICA ATAM 55
5.1 Avaliação dos requisitos 56
5.1.1 Imperativo customizado 56
5.1.2 POO customizado 57
5.1.3 Novo paradigma 57
5.1.4 Integrar Plugin 57
5.1.5 Interface de usuário 57
5.2 Simulação de uso de uma ferramenta derivada a partir da arquitetura proposta 58
5.2.1 Geração de casos de testes com POO 58
5.2.2 Geração de casos de testes usando critério do paradigma imperativo 60
5.2.3 Inserindo critério customizado no POO 62
5.2.4 Inserindo critério customizado do paradigma imperativo 63
6. CONCLUSÃO 65
REFERÊNCIAS 66

1. INTRODUÇÃO

O uso de softwares tem se tornado cada vez mais comum em diferentes domínios (*e.g.* na saúde). Durante a utilização desses sistemas, os usuários esperam não encontrar erros. Nesse sentido, a demanda por softwares corretos é crescente. O efeito causado por erros em sistemas de software pode variar de sistema para sistema, desde a insatisfação do usuário, para sistemas de propósito gerais, até elevados danos financeiros e físicos, para sistemas críticos.

A verificação de software busca maximizar a remoção de defeitos em sistemas de software e começou a ser buscada quando as empresas desenvolvedoras passaram a enfatizar o desenvolvimento de produtos com qualidade (GARCIA, 2017). Para minimizar o número de erros em sistemas de software, técnicas relacionadas com a atividade de verificação são comumente utilizadas. Com isso, a satisfação do cliente ganhou maior importância e o objetivo das organizações desenvolvedoras de software passou a ser criar produtos que atendessem as expectativas dos usuários (ACKERMAN et al., 2016). Nesse sentido, gerou-se uma preocupação, não só em atender às necessidades dos usuários, mas também com o desenvolvimento de sistemas livres de defeitos. Para atender essa demanda, técnicas, tais como a análise estática de códigos e testes de software passaram a ser comuns ao longo do desenvolvimento do software.

O principal propósito com o teste de software é encontrar defeitos, ou seja, imperfeições ou inconsistências inseridas no código durante o processo de desenvolvimento, que foram geradas por engano humano. Um defeito pode levar o sistema a uma situação de erro, que representa um estado inconsistente no software. Quando os erros causam resultados inesperados, tem-se uma falha. Uma falha pode ter diversas implicações, por exemplo, uma falha em um sistema de propósito geral, tal como um processador de texto, pode resultar na insatisfação do usuário. Para os sistemas críticos, tal como um sistema de piloto automático de uma aeronave, falhas no software podem gerar danos a integridade e podem também gerar danos a integridade física de usuários, além de danos ao meio ambiente.

O teste de software está inserido em uma área maior, a verificação & validação (V&V) (GARCIA, 2017). Na verificação é realizado um processo de análise comparativa entre o software em desenvolvimento e os requisitos documentados, ou seja, a verificação analisa se o sistema foi desenvolvido corretamente. Por outro lado, validação está relacionada com um conjunto de tarefas responsáveis por analisar se o software desenvolvido pode ser rastreado segundo os requisitos do usuário. Ou seja, a validação analisa se o produto desenvolvido foi o correto, de acordo com as requisições dos usuários (PRESSMAM, 2011). Portanto, a

verificação e a validação podem ser utilizadas como formas de analisar a qualidade de um software de acordo com os seus requisitos. Nesse trabalho, o foco está em sistemas construídos com o Paradigma Orientado a Objetos (POO). Além disso, dentre os diversos tipos de teste de software, o foco está nos testes de unidade.

No paradigma de desenvolvimento de software orientado a objetos, a definição de unidade se altera comparado ao paradigma imperativo. No paradigma imperativo uma função/procedimento pode ser considerada como unidade, enquanto que, no POO, o encapsulamento acompanha a definição de classes e objetos. Isso indica que cada classe e cada instância de uma classe engloba atributos e as operações que manipulam dados, o qual no paradigma imperativo os dados são manipulados por meio de varáveis e funções. Ou seja, não existe a ideia de encapsulamento. Uma classe que contém elementos internos é habitualmente o eixo do teste de unidade. Como uma classe pode ser composta por um conjunto de distintas operações, e uma operação específica pode compor parte de um conjunto de distintas classes, a estratégia utilizada para teste de unidade precisa ser alterada comparada ao paradigma imperativo (PRESSMAM, 2011).

Segundo Bernardo e Kon (2008), o teste de unidade é um dos fatores mais importantes durante as atividades de teste, pois verifica a conformidade dos algoritmos com as tarefas designadas. Além disso, facilita a compreensão de defeitos, visto que a unidade de teste é descrita em partes específicas do código.

Aplicado o teste de unidade, pode-se definir um outro tipo de teste, conhecido por teste de integração. A integração no contexto de POO ocorre quando a interação entre unidades, ou seja, classes, é verificada. Já no paradigma imperativo a integração pode ser verificada por meio da comunicação entre funções. Isto é, o fato de analisar resultados gerados pela comunicação direta ou indireta entre unidades, origina-se no teste de integração (GALLAGHOR; OFFUTT; CINCOTTA, 2006).

Durante a atividade de teste, testadores definem um conjunto de entradas e saídas desejadas para que o software em desenvolvimento seja verificado (conjunto denominado de casos de teste). Neste contexto, técnicas para geração de casos de teste são aplicadas com o propósito de definir o conjunto de casos de testes a serem analisados (SOMMERVILLE, 2011). Portanto, com o foco em reduzir a quantidade de trabalho e reduzir as chances de enganos humanos durante as tarefas de teste, surgiram os testes automatizados.

Em paradigmas distintos de programação, a tarefa de geração de um conjunto de teste depende da utilização de critérios, onde os mesmos são aplicados com o objetivo de selecionar testes essenciais. Esses critérios são geralmente definidos com base no fluxo de teste a ser

analisado, por exemplo, fluxo de controle e fluxo de dados. Quando critérios são aplicados e um conjunto de casos de teste é gerado, o teste de software pode ser automatizado por meio da combinação entre os casos de teste e a utilização de ferramentas que realizam testes (LEAL, 2009). O foco neste trabalho é desenvolver uma arquitetura de uma ferramenta de geração automatizada de casos de teste.

A arquitetura de um software é o passo inicial para a geração de um projeto. A arquitetura pode ser composta por um conjunto de propriedades demandadas pelo sistema, além de prover visão sobre elementos de software, propriedades externas e relacionamento entre componentes. Pode-se definir a arquitetura de um software com base em várias propriedades, tais como, confiança, disponibilidade, compatibilidade com plataformas, utilização de memórias, uso de rede, segurança, usabilidade e interoperabilidade (BASS; CLEMENTS; KAZMAN, 2003).

Portanto, o processo de desenvolvimento de arquitetura de um software contribui para que uma aplicação seja expandida e detalhada da melhor forma possível. Isso é útil para que o seu desenvolvimento possua uma menor complexidade, dado que todos os aspectos, como, componentes, relacionamentos, algoritmos e propriedades do software possam ser definidos no processo de desenvolvimento arquitetural.

1.1. Objetivos

Para a realização deste trabalho são definidos objetivos gerais e específicos. Estes objetivos são descritos a seguir.

1.1.1. Objetivo geral

- Auxiliar codificadores no desenvolvimento de ferramentas para a geração automatizada de casos de testes em sistemas computacionais.

1.1.2. Objetivos específicos

- Desenvolver uma arquitetura de ferramenta para geração automatizada de casos de teste.
- Avaliar a arquitetura proposta por meio de técnica ATAM.

1.2. Metodologia

Para o desenvolvimento da arquitetura deste trabalho, inicialmente realizou-se um levantamento bibliográfica sobre teste de software orientado a objetos, linguagem de

programação JAVA, linguagem de modelagem unificada e criação de plugins para a ferramenta Eclipse. O levantamento bibliográfico foi realizado utilizando o Google Acadêmico[1], Scielo[2], IEEE Xplore[3] e ACM Library[4]. Por outro lado, na modelagem da arquitetura utilizou-se a UML.

1 https://scholar.google.com.br/
2 http://www.scielo.org/php/index.php
3 http://ieeexplore.ieee.org/Xplore/home.jsp
4 http://dl.acm.org/

2. REFERENCIAL TEÓRICO

Neste capítulo são apresentados conceitos fundamentais necessários para o desenvolvimento deste trabalho.

2.1 Teste de software

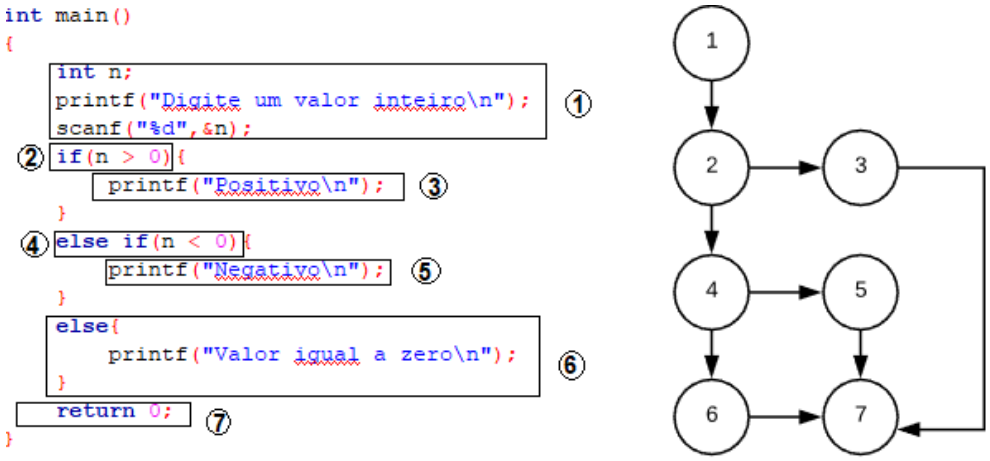
O foco no teste é identificar a maior quantidade possível de defeitos com um empenho gerenciável durante um espaço de tempo realístico (PRESSMAM, 2011). Com isso, o teste é utilizado para encontrar problemas em aplicações que provavelmente não estão desenvolvendo as suas funcionalidades da forma a qual foram especificadas.

O teste é uma das atividades de engenharia de software que tem como propósito elevar a produtividade e oferecer clareza da qualidade de softwares em conjunto com outras atividades durante o processo de desenvolvimento de software. Para Korel (1990), o teste pode ser visto como uma forma de averiguar se o software está em conformidade com as funções no qual foram propostas pelo cliente solicitante do produto.

Segundo Garcia (2017), não existe classificação genérica para os tipos de testes existentes. Ele define que os testes podem ser classificados em três eixos, nível de teste, método e tipo. Para o autor, os níveis de testes podem ser definidos entre teste de unidade (testa-se cada unidade particular do sistema), teste de integração (testa-se interação entre unidades), teste de sistema (testa-se o sistema por completo, ou seja, o funcionamento total) e teste de aceitação (teste feito com a participação do usuário final do software). Já os métodos, podem ser caixa branca (teste aplicado na parte interna do software, isto é, no seu código fonte), caixa-preta (verifica saídas do sistema, dado um conjunto de entradas) e não funcional (são testes que verificam atributos internos do sistema, como exemplo, eficiência). Além disso, define duas maneiras nas quais os testes podem ser realizados: manuais ou automatizados. Alguns testes são baseados no Grafo de Fluxo de Controle (GFC) do programa.

Segundo Kapoor e Dhavale (2016), o GFC é uma estrutura capaz de distinguir todos os possíveis caminhos que podem ser executados por um programa computacional. Na figura 1 é apresentado um exemplo de um grafo de fluxo de controle gerado a partir de um simples programa de computador. Instruções sequenciais são consideradas em um único nó, porém quando encontra-se instruções de desvios, novos nós são criados para representar os possíveis caminhos desses desvios.

FIGURA 1 - EXEMPLO DE GERAÇÃO DE GFC



Fonte: (Autoria Própria)

Na figura 1 é mostrado que o GFC foi gerado a partir de um simples código que verifica se um número é positivo, negativo ou zero. Portanto, inicialmente foram definidos os nós responsáveis por representarem cada instrução. Ou seja, o primeiro nó foi definido a partir de uma instrução que representa a declaração de uma variável "n" até que uma instrução de desvio é encontrada, pois a instrução de desvio pode seguir caminhos não sequenciais. Quando a instrução de desvio é encontrada, novos nós são definidos, inclusive, o da instrução, isto é, o nó 2. Esse nó pode seguir caminhos diferentes de acordo com a veracidade ou não da condição. Se a condição for verdadeira, o fluxo segue para o nó 3 e finaliza a execução no nó 7, caso seja falsa, o fluxo de execução segue para o nó 4 que também é uma instrução condicional e segue a mesma ideia do nó 2. Se a condição for verdadeira, o fluxo segue para o nó 5 e finaliza a execução no nó 7, caso seja falso, segue para o nó 6, que consequentemente segue para o nó 7.

2.1.1 Paradigma Imperativo

O paradigma imperativo é uma metodologia de programação que possui peculiaridades totalmente diferentes de outros paradigmas, ou seja, ele trata as instruções de forma sequencial, onde dados são manipulados de acordo com ordem de organização das instruções (FRAME e COFFEY, 2014). A programação imperativa define elementos conhecidos por "variáveis" que são responsáveis pelo o armazenamento de dados. Com isso, define funções/procedimentos que tem como objetivo manipular dados armazenado. Uma das principais características do paradigma imperativo é a forte dependência entre procedimentos e variáveis. Para realizar testes no paradigma imperativo, critérios de teste podem ser adotados.

Segundo Delamaro (2014), existem vários critérios de teste no paradigma imperativo. Porém, para definir quais critérios de teste utilizar em uma aplicação, inicialmente deve ser analisado qual tipo de fluxo será utilizado: fluxo de controle, fluxo de dados, etc. Visando o fluxo de controle, pode-se definir três principais critérios, sendo eles, Todos-Nós, Todas-Arestas e Todos-Caminhos. O critério Todos-Nós define que a execução do programa percorra no mínimo uma vez cada vértice do grafo de fluxo de controle. O critério Todas-Arestas define que cada desvio que compõe o programa seja executado ao menos uma vez. O critérios Todos-Caminhos define que todos os possíveis caminhos existentes no programa sejam executados.

2.1.2 Paradigma Orientado a Objetos

O POO surgiu como forma de suprir algumas necessidades pertencentes ao paradigma imperativo. No paradigma imperativo um dos principais problemas é uma forte dependência entre procedimentos/funções e dados. Portanto, com o POO, surgiu a ideia de agrupar em uma entidade chamada de classe (entidade responsável por armazenar métodos e atributos referentes a um objeto) os atributos (entidade responsável por armazenar características de um objeto) e métodos (ações que o objeto pode exercer quando solicitado) que realizam operações sobre dados. Isso contribui para que o problema da forte dependência seja amenizado (DELAMARO et al., 2016).

Segundo Virius e Pecinovský (2014), o termo objeto refere-se a uma entidade composta por atributos (entidade responsável por armazenar dados de um objeto) e um conjunto de operações responsáveis pela manipulação desses dados. A atividade de teste de software no POO pode ser baseada em critérios de teste.

Os critérios de testes utilizados no POO foram desenvolvidos com base nos critérios existentes no paradigma imperativo. Por exemplo, os critérios de teste baseados em fluxo de dados são ampliados para o teste de classe (HARROLD e ROTHERMEL, 1994). Estes critérios podem ser utilizados com o objetivo de testar tanto métodos individuais quanto métodos que interagem entre si dentro de uma mesma classe.

O teste de fluxo de dados no POO pode ser aplicado com base na criação de três tipos de grafos: grafos de chamada de classes, grafo de fluxo de controle de classes e o grafo de fluxo de controle encapsulado. Os testes podem ser divididos em três níveis: intramétodo (métodos testados individualmente), intermétodos (métodos públicos testados em conjunto com métodos de uma mesma classe) e intraclasse (testando interação de métodos públicos quando são chamados em diferentes sequências). Esses são os três níveis que permitem a realização do teste de fluxo de dados no POO (HARROLD e ROTHERMEL, 1994).

A aplicação de critérios de geração de casos de teste em um dado software é a melhor maneira de selecionar dados para serem testados, dado que aplicando critérios podemos selecionar somente os casos de teste que realmente necessitam de avaliação. Com isso, o problema de teste exaustivo pode ser resolvido. Existem *frameworks* que possibilitam a execução de casos de teste. Ou seja, quando um caso de teste é gerado, podemos analisar os resultados apresentados pelo software automatizando esse teste com a utilização de *frameworks*, como por exemplo o JUnit5 (GÁRCIA, 2017).

2.1.3 *Framework* JUnit 5

O JUnit 5 é um *framework* projetado para realizar teste de unidade. Porém pode ser utilizado para outros tipos de teste específicos, como o teste de integração. O JUnit foi projetado com base em três componentes conhecidos por *Jupter, Platform e Vintage*. O *Jupter* garante a extensão do *framework* JUnit 5 oferecendo diferentes tipos de testes. A *Platform* é um mecanismo responsável pela fundação dos testes e o *Vintage* é o mecanismo responsável pela execução desses testes (GÁRCIA, 2017). Esse *framework* é um plugin Java no qual permite a execução de casos de teste por meio de um ambiente de desenvolvimento, como, por exemplo, o Eclipse.

Como afirma Bechtold et al. (2018), por meio do *Jupter*, o JUnit disponibiliza diferentes tipos de testes denotados por anotações, tais como @Test (indica o teste em um dado método de uma classe), @ParametizedTest (indica um método com teste parametrizado). Com o *Jupter*, é possível realizar vários tipos de testes tanto em métodos, como em classes que são desenvolvidos com a linguagem de programação Java.

Portanto, o JUnit5 é uma maneira de automatizar testes, visto que, com um conjunto de casos de testes gerados a partir da aplicação de critérios, pode-se testar o software em busca de analisar resultados e encontrar defeitos durante a execução dos elementos que compõem o sistema.

2.2 Linguagem de Modelagem Unificada

De acordo com Faisal, Nazir e Babar (2018), a linguagem de modelagem unificada (*Unified Modeling Language - UML*), oferece um mecanismo padrão a ser utilizado na modelagem de softwares desenvolvidos utilizando o paradigma orientado a objetos. Ou seja, a UML é uma linguagem visual que permite aos profissionais de desenvolvimento de software modelar características dos sistemas, tais como, requisitos, comportamentos e estruturas lógicas.

A UML pode ser utilizada para modelar o comportamento e a estrutura do software, isto é, aplica-se a UML com o objetivo de obter noções de projetista do sistema em alto nível (VISWANATHAN e SAMUEL, 2016).

2.3 Plugin Java

O *plugin* para o ambiente de desenvolvimento Eclipse é uma chave que permite a integração de ferramentas transparentes e extensível para o Eclipse. Com os *plugins* pode-se desenvolver ferramentas que desempenham tarefas em conjunto com o ambiente de desenvolvimento (GALLARDO, 2002).

O acompanhamento do plugin é acrescido de um arquivo plugin.xml, responsável por definir a integração entre o plugin e a ferramenta Eclipse. Esse arquivo valida elementos como, informações simples (como nome) e dados significativos para a integração com o Eclipse (extensões, pontos de extensão). Geralmente, a integração entre os elementos, plugin e Eclipse, geram um conjunto de classes Java.

2.4 Arquitetura de software

A arquitetura de software é um mecanismo, que, segundo Bass, Clements e Kazman (2003), permite que uma organização seja capaz de atingir os seus objetivos almejados. Isto é, durante o desenvolvimento do projeto arquitetural são definidos propriedades e componentes do software, que possibilitam análises e planejamentos definidos por meio dos objetivos almejados em um produto final. A definição da arquitetura de um software pode ser baseada em técnicas que mapeiam as etapas a serem desenvolvidas de acordo com os objetivos de uma organização. Por exemplo, a técnica ADD define passos a serem seguidos para que uma arquitetura desenvolvida seja capaz de atender todos os cenários derivados a partir do atributo de qualidade almejado. Ou seja, dado um ou mais atributos de qualidade, são definimos cenários nos quais descrevem ações necessárias para que esse atributo seja alcançado (BASS; CLEMENTS; KAZMAN, 2003). Seguindo a técnica ADD, os atributos de qualidade em conjunto com os requistos não funcionais são utilizados como guia arquitetônico para que a arquitetura seja projetada. Os passos definidos no ADD são os seguintes:

- Definir guias arquitetônicos;
- Definir padrão arquitetural com base nos guias arquitetônicos;
- Instanciar módulos atribuindo funcionalidades dos casos de uso e representa-los utilizando diversão visões;
- Definir interface de módulos filhos;

- Validar todo o processo com os casos de uso e cenários de qualidade almejado continuar a decomposição de módulos filhos.

O projeto da arquitetura no desenvolvimento de um software é uma das tarefas mais importantes, porém, é necessário que análises sejam realizadas na arquitetura desenvolvida para que os problemas ou conflitos sejam identificados ainda na parte de projeto (CARVALHO et al., 2015). Portanto, métodos de avaliação podem ser utilizado para que a arquitetura seja analisada. Por exemplo, no método ATAM são definidos papéis e uma sequência de passos a serem realizados para que uma arquitetura seja analisada com base em cenários derivados a partir de atributos de qualidade. Baseado em Bass, Clements e Kazman (2003), os papéis definidos pelo ATAM são os seguintes:

- **Equipe de avaliação:**

Geralmente constituída por 3 a 5 pessoas, onde cada participante da equipe é responsável por tarefas específicas. Esses membros devem possuir conhecimento sobre a arquitetura desenvolvida. A Tabela 1 exibe os papeis de membro da equipe.

TABELA 1 - DESCRIÇÃO DE PAPÉIS DO ATAM

Papel	**Descrição**
Líder de equipe	Configura a avaliação, gerencia com o cliente para garantir que as exigências do mesmo sejam atendidas.
Líder de avaliação	Coloca a avaliação em execução e prioriza os cenários a serem analisados na arquitetura.
Escriba de cenários	Escreve cenários em folha de papel ou em lousa a sua descrição.
Escriba do ocorrido	Atua em equipamento eletrônico, por exemplo, computador. É responsável por encontrar problemas que motivam cada cenário e desenvolve uma solução para cada cenário quando aplicado a arquitetura, gerando um folheto com a descrição de cenários para cada membro participante da avaliação.

Controlador de tempo	Responsável por guiar o líder de avaliação a permanecer no cronograma, auxiliando a controlar o tempo dedicado a avaliação de cada cenário.
Observador do processo	Provê observações de melhorias após o processo de avaliação, pode gerar sugestões de melhoria durante o processo de avaliação e no final gera relatórios sobre o processo de avaliação e define pontos a serem melhorados.
Catalisador do processo	Auxilia o líder de avaliação no acompanhamento e execução das etapas de avaliação.
Questionador	Responsável por relatar questões arquitetônicas que não foram abordadas pelas partes interessadas.

Fonte: (Autoria Própria)

- **Tomadores de decisão do projeto:**

Essa equipe é responsável por tomar decisões no projeto, geralmente composta por um gerente de projeto, cliente, arquiteto e promotor de avaliação.

- **Partes interessadas da arquitetura:**

São as partes interessadas que para realizarem as suas tarefas dependem da arquitetura, como, por exemplo, Desenvolvedores, testadores, integradores, engenheiros de desempenho e usuário.

Definidos os papéis, o foco agora é descrever somente o nome de cada passo necessário para que a técnica ATAM seja aplicada. Os passos são aplicados em duas fases, onde na fase 1 são aplicados os passos de 1 a 6, e na fase 2 o restante dos passos são executados. Abaixo são listados esses passos.

- Apresentar o ATAM;
- Dirigir negócios atuais;
- Apresentação da arquitetura atual;
- Identificar abordagens arquitetônicas;
- Gerar árvore do utilitário de atributos de qualidade;
- Analisar abordagens arquiteturais;
- Brainstorm e priorizar cenários;
- Analisar abordagens arquitetônicas;
- Apresentação de resultados.

3. TRABALHOS RELACIONADOS

Neste capítulo são abordados alguns trabalhos relacionados ao tema explanado e apresentado neste documento. A seguir serão apresentados alguns estudos sobre a geração automatizada de casos de teste.

3.1 Geração de Grafo de Fluxo de Controle

No estudo de Son, Park e Kim (2016), a geração do grafo de fluxo de controle contribui para a definição de um conjunto de testes mais complexos a serem realizados em uma aplicação. Nesse estudo, eles propõem a extensão do Grafo de Fluxo de controle (*Control Flow Graph* - GFC) original para um novo grafo chamado grafo de fluxo de controle de múltipla condição, no qual, tem como objetivo trabalhar com condições múltiplas em testes baseados em modelos, possibilitando a geração de casos de testes de forma automatizada. Segundo os autores, o grafo de fluxo de controle original possui limitações ao expressar informações referentes a condições múltiplas.

Kapoor e Dhavale (2016), propuseram a utilização do GFC para gerar códigos de operações e em conjunto com a técnica de separação bi-normal detectarem indícios de *malwares* em executáveis de códigos fontes de forma automática. Ou seja, dado que o GFC busca retratar todos os caminhos possíveis de execução de um programa de software, a contribuição com a detecção de *malwares* é relevante, pois, dado a sequência de caminhos que podem ser executados no software, torna-se mais fácil a análise mais detalhada do funcionamento de cada um desses caminhos, possibilitando detectar quais desses estão mais vulneráveis.

Alimucaj (2009), criou um plugin para geração de diagramas de fluxo de controle, com o objetivo de permitir ao usuário realizar medidas de aprendizado. Ou seja, a ferramenta possibilita que o usuário gere grafos de fluxo de controle de unidades, nas quais essas unidades são os métodos que compõem cada classe. Segundo o autor, a escolha por desenvolver um plugin para a ferramenta Eclipse, se deu devido ao mesmo ser muito robusto, difundido e também permitir ampliação de uma forma não necessariamente tão complexa. Em busca de tornar o plugin mais eficiente, Alimucaj (2009), buscou conhecer ferramentas que realizam cobertura de código e fossem integradas ao eclipse. Com isso, identificou o CodeCover, que segundo Alimucaj (2009), é uma ferramenta *open source* que é capaz de aplicar técnicas de cobertura de códigos, como, por exemplo, *cover branches, conditions, loops e statements.*

3.2 Geração de casos de teste

No trabalho de Panichella, Kifetew e Tonella (2018), a geração automatizada de casos de teste é vista como uma forma de maximizar o escopo de teste do software de acordo com as peculiaridades das técnicas de teste escolhidas. Isto é, gerar casos de teste de forma automatizada contribui com a criação de um conjunto de testes que podem ser utilizados para verificar saídas do sistema. Dentre os diferentes métodos de geração de casos de teste, existem por exemplo a geração de casos de teste por meio de algoritmos.

Em sua pesquisa, Yin (2017) percebeu que os algoritmos existentes que utilizavam o diagrama de atividade de linguagem de modelagem de sistema (*Systems Modeling Language* – SysML) para a geração automatizada de casos de teste possuíam limitações ao serem utilizados em modelos de diagramas de atividades não estruturados. Portanto, propôs a criação de um algoritmo de geração automatizada de casos teste baseado diagramas de atividades não estruturados. Com isso, a ideia do autor foi, a partir de um diagrama de atividade não estruturado, transformá-lo em uma expressão intermediária, denominada pelo autor de IBM, onde após essa transformação é possível que os casos de teste já possam ser gerados de forma automatizada.

Manral (2015) afirma em seu trabalho que a geração de casos de teste com qualidade influencia na diminuição de custos relacionado ao tempo necessário para testar aplicações. Portanto, gerou modelos de teste de Redes de Petri de Alto Nível com base em modelos informais de UML. Ou seja, dado o modelo em UML, eram geradas as Redes de Petri de Alto Nível tornando-se os casos de teste, permitindo a simulação do sistema em alto nível.

Portanto, realizado esse estudo sobre trabalhos relacionados. É notório que na pesquisa não foi encontrado trabalhos que buscam a geração de casos de teste de forma automatizada para testar aplicações de software. Com isso, nesse trabalho é desenvolvida uma arquitetura que busca facilitar o desenvolvimento de ferramentas com objetivo de gerar casos de teste de forma automatizada para aplicações de software.

4. PROJETO ARQUITETURAL DE UMA FERRAMENTA PARA GERAÇÃO AUTOMATIZADA DE CASOS DE TESTE

Neste capítulo é apresentado o projeto arquitetural para o desenvolvimento da ferramenta construído com base em conceitos definidos pelo método ADD. Esse método baseia-se no processo de decomposição com base nos atributos de qualidade pretendidos pelo software. O atributo de qualidade almejado pela ferramenta em questão é o de modificabilidade. Deseja-se desenvolver uma ferramenta que ofereça suporte a extensões de suas funcionalidades, além de prover maior agilidade de implementação e redução de custos. Diz-se que um software provê modificabilidade quando suas funcionalidades são distribuídas em módulos e quando algum deles é modificado, somente o menor número de módulos possível é atingido (BASS; CLEMENTS; KAZMAN, 2003). Os requisitos definidos na arquitetura foram embasados em revisão literária e também na arquitetura do framework JUnit5 que dispõe de uma arquitetura definida em módulos.

4.1 Casos de uso

Nos próximos parágrafos são definidos casos de usos que representam a interação entre o desenvolvedor e a ferramenta proposta.

Caso de uso 01: *Gerar GFC*

Este caso de uso permite ao usuário gerar o grafo de fluxo de controle através de uma unidade de entrada por meio de uma anotação. Ou seja, dado um método ou função anotada com a Tag @GFC, o grafo de fluxo de controle referente a unidade é gerado.

Caso de uso 02: *Gerar casos de teste com critério Todos-Nós*

Este caso de uso permite ao usuário gerar casos de teste para uma unidade com base no paradigma imperativo aplicando o critério de teste Todos-Nós. Para gerar os casos de teste, o usuário deve anotar a unidade com a Tag @TodosNos. Quando a ferramenta é executada, é gerada uma estrutura de dados que representa o GFC da unidade. O GFC é percorrido e são gerados caminhos de teste onde todos os nós criados na representação do GFC devem ser percorridos pelo menos uma vez gerando os caminhos de teste. Com isso, definidos os caminhos de teste, é gerado um conjunto de dados de teste, separados por cada casos de teste que cobrem todos os caminhos definidos.

Caso de uso 03: *Gerar casos de Teste com critério Todas-Arestas*

Este caso de uso permite ao usuário gerar casos de teste para uma unidade com base no paradigma imperativo aplicando o critério de teste Todas-Arestas. Para gerar os casos de teste, o usuário deve anotar a unidade com a Tag @TodasArestas. Quando a ferramenta é executada, é gerada uma estrutura de dados que representa o GFC da unidade. O GFC é percorrido e são gerados caminhos de teste onde todas as arestas criadas na representação do GFC devem ser percorridas pelo menos uma vez gerando os caminhos de teste. Contudo, definidos os caminhos de teste, é gerado um conjunto de dados de teste, separados por casos de testes que cobrem todos os caminhos definidos.

Caso de uso 04: *Gerar casos de teste com criterio Todos-Caminhos*

Este caso de uso permite ao usuário gerar casos de teste para uma unidade com base no paradigma imperativo aplicando o critério de teste Todas-Caminhos. Para gerar os casos de teste, o usuário deve anotar a unidade com a Tag @TodosCaminhos. Quando a ferramenta é executada, é gerada uma estrutura de dados que representa o GFC da unidade. O GFC é percorrido e são gerados caminhos de teste onde todos os caminhos criados na representação do GFC devem ser percorridos pelo menos uma vez gerando os caminhos de teste. Com isso, definidos os caminhos de teste, é gerado um conjunto de dados de teste, separados por casos de testes que cobrem todos os caminhos definidos.

Caso de uso 05: *Gerar casos de teste imperativo customizado*

Este caso de uso permite ao usuário desenvolver um novo critério de teste voltado ao paradigma imperativo. Para gerar os casos de teste, o usuário deve instanciar uma classe abstrata residente no módulo de teste que armazena informações do paradigma imperativo. Um classe é implementada para definir como o critério funcionará, ou seja, como serão gerados os caminhos de teste por meio do GFC, definidos também a Tag referente ao teste customizado. Portanto, a ideia desse caso de uso é que o usuário customize a geração dos seus casos de teste para esse paradigma de acordo com a sua necessidade.

Caso de uso 06: *Gerar casos de teste com criterio Intramétodo*

Este caso de uso permite ao usuário gerar casos de teste para uma unidade com base no paradigma orientado a objetos aplicando o nível de teste Intramétodo. Ou seja, esse caso de uso permite que métodos de códigos desenvolvidos com o paradigma Orientado a Objetos possam ser testados individualmente. Para que o teste seja possível, o GFC do método é gerado e o usuário pode escolher aplicar critérios de teste como o Todos-Nós e Todas-Arestas. Isto é, tanto

o critério Todos-Nós quanto o Todas-Arestas serão dispostos em duas versões, onde em uma delas considera o tratamento de exceções e a outra não.

Caso de uso 07: *Gerar casos de teste com criterio Intermétodo*

Este caso de uso permite ao usuário gerar casos de teste para uma unidade com base no paradigma orientado a objetos aplicando o nível de teste Intermétodo. Esse caso de uso permite ao usuário testar a iteração entre métodos que residem em uma única classe. Ou seja, é permitido ao usuário analisar as invocações entre os métodos residentes em uma mesma classe.

Caso de uso 08: *Gerar caso de teste OO customizado*

Esse caso de uso permite ao usuário desenvolver um novo critério de teste voltado ao paradigma Orientado a Objetos. Para gerar os casos de teste, o usuário deve instanciar uma classe abstrata residente no módulo de teste que armazena informações doPOO. Uma classe para definir como o critério funcionará, ou seja, como serão gerados os caminhos de teste por meio do GFC, definidos também a Tag referente ao teste customizado. Portanto, a ideia desse caso de uso é que o usuário customize a geração dos seus casos de teste para esse paradigma de acordo com a sua necessidade.

Caso de uso 09: *Executar teste via linha de comando*

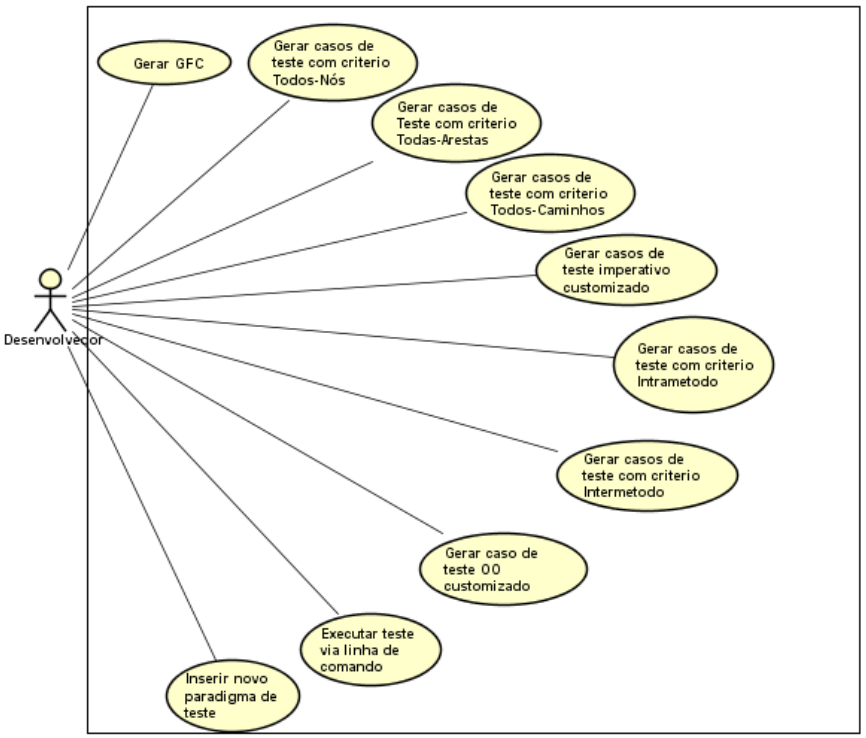
Esse caso de uso permite que o usuário utilize a aplicação por meio de um interpretador de comandos. Ou seja, quando o usuário não necessita utilizar um ambiente de desenvolvimento para executar o software, ele pode executar a mesma via comandos. Dado um arquivo na qual as unidades estejam anotadas com os tipos de testes a serem analisados, a linha de comandos permitirá a execução da ferramenta.

Caso de uso 10: *Inserir novo paradigma de teste*

Esse caso de uso permite ao usuário inserir um novo paradigma de teste a ser utilizado na ferramenta. Ou seja, quando o usuário necessitar inserir um novo paradigma de teste, ele pode fazer isso por meio da implementação de um módulo definido na arquitetura, onde esse módulo disponibilizará de classes abstratas com métodos padrões que podem ser implementadas de acordo com as necessidades do usuário.

O diagrama mostrado na Figura 2 retrata todos os casos de uso descritos acima, permitindo uma visão mais abstrata da iteração do usuário com as funcionalidades oferecidas na ferramenta.

FIGURA 2 - DIAGRAMAS DE CASOS DE USO



Fonte: (Autoria Própria)

4.2 Requisitos de modificabilidade

Baseado na técnica ADD, os atributos de qualidade podem ser alcançados por meio da elicitação de requisitos não funcionais. Segundo Bass, Clements e Kazman (2003), para definir o cenário do requisito não funcional, pode-se basear nos seguintes aspectos: Fonte (atuador que pode ser um usuário, sistema, etc.), Estímulo (condição que o sistema deve considerar), Artefato (sistema por completo ou alguma parte dele), Ambiente (condição em que o sistema encontra-se ao ser requisitado), Resposta (Atividade realizada após o estimulo) e Medida de resposta (define quando a resposta ocorre). Buscando atingir um nível elevado de modificabilidade, requisitos essenciais são definidos nas próximas seções.

4.2.1 Inserir critério de teste customizado com paradigma imperativo

FIGURA 3 - INSERÇÃO DE CRITÉRIO CUSTOMIZADO IMPERATIVO

Fonte: (Autoria Própria)

No cenário da Figura 3 é definido que um desenvolvedor deve ser capaz de inserir um critério de teste customizado utilizando o paradigma imperativo em até uma hora. A fonte de estímulo desse requisito é o desenvolvedor. Esse requisito representa um cenário em que ele decide inserir um novo critério de teste no paradigma imperativo. Para isso, ele desenvolve códigos fontes para integrar a ferramenta em tempo de projeto. A resposta da ferramenta é aceitar as modificações sem sofrer efeitos colaterais, isto é, não deve apresentar problemas após essa ação ser realizada.

4.2.2 Inserir critério de teste customizado com paradigma orientado a objetos

FIGURA 4 - INSERÇÃO DE CRITÉRIO CUSTOMIZADO ORIENTADO A OBJETOS

Fonte: (Autoria Própria)

No cenário descrito na Figura 4 define que um desenvolvedor deve ser capaz de inserir um critério de teste customizado utilizando o paradigma orientado a objetos em até uma hora. A fonte de estímulo desse requisito é o desenvolvedor. O cenário representa a ação em que ele decide inserir um novo critério de teste no paradigma orientado a objetos, com isso, ele desenvolve códigos fontes para integrar a ferramenta em tempo de projeto. A resposta da ferramenta é aceitar as modificações sem sofrer efeitos colaterais, ou seja, não deve apresentar problemas por essa ação ser realizada.

4.2.3 Inserir novo paradigma de teste

FIGURA 5 - INSERÇÃO DE NOVO PARADIGMA DE TESTE

Fonte: (Autoria Própria)

No cenário da Figura 5 define que um desenvolvedor deve ser capaz de inserir um novo paradigma de teste em até um dia. A fonte de estímulo desse requisito é o desenvolvedor, ocorre quando ele decide inserir um novo paradigma de teste a ser utilizado na ferramenta, com isso, ele desenvolve códigos fontes para integrar a ferramenta em tempo de projeto. A resposta da ferramenta é aceitar as modificações sem sofrer efeitos colaterais, isto é, não deve apresentar problemas por essa ação ser realizada.

4.2.4 Integrar Plugin com a ferramenta

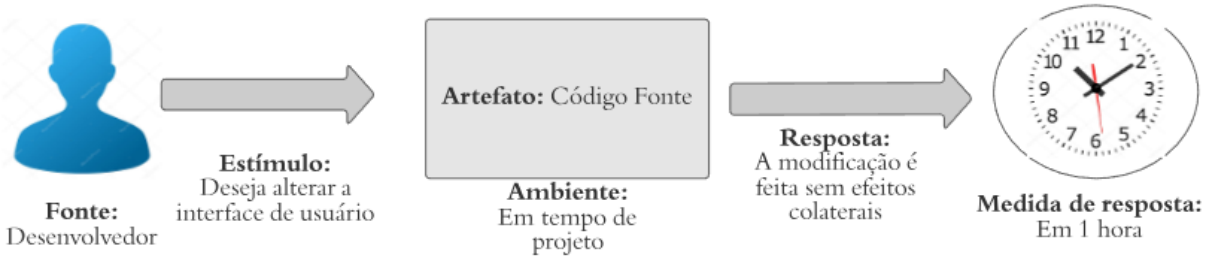
FIGURA 6 - INTEGRAÇÃO DE PLUGIN COM A FERRAMENTA

Fonte: (Autoria Própria)

No cenário abordado na Figura 6 define que um desenvolvedor deve ser capaz de integrar um plugin a ferramenta em no máximo uma hora. A fonte de estímulo desse requisito é o desenvolvedor, onde o cenário representa a ação em que ele decide integrar um plugin que possibilite a exibição de dados gráficos ao usuário. Portanto, ele desenvolve códigos fontes característicos desse plugin para integrar a ferramenta em tempo de projeto. A resposta da ferramenta aceitar as modificações sem sofrer efeitos colaterais, ou seja, não deve apresentar problemas por essa ação ser realizada.

4.2.5 Modificar interface de usuário

FIGURA 7 - MODIFICAÇÃO DE INTERFACE DE USUÁRIO



Fonte: (Autoria Própria)

No cenário descrito na Figura 7 define que um desenvolvedor deve ser capaz de realizar alterações na interface de usuário em 1 hora. A fonte de estímulo desse requisito é o desenvolvedor quando deseja realizar modificações na interface de usuário da ferramenta. Com isso, ele desenvolve ou modifica códigos fontes integráveis a ferramenta em tempo de projeto. A resposta da ferramenta aceitar as modificações sem sofrer efeitos colaterais, ou seja, não deve apresentar problemas por essa ação ser realizada.

4.3 Projeto

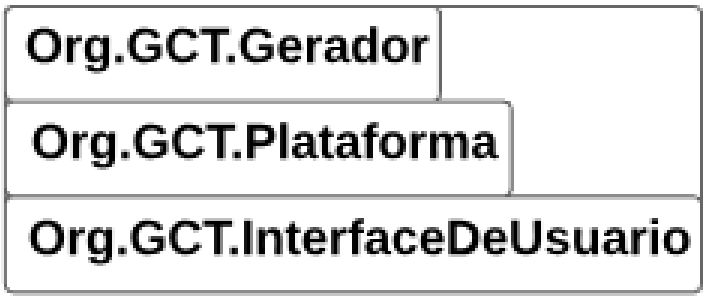
Atributos de qualidade na técnica ADD são definidos como guias arquitetônicos (do Inglês *Architectural Drivers*). Portanto, como o atributo objetivado pela arquitetura é o de modificabilidade, a arquitetura deve ser baseada em módulos, onde cada módulo possui papéis específicos, permitindo que alterações particulares não comprometam o funcionamento da ferramenta. Com o objetivo de alcançar o atributo de qualidade definido acima, foram utilizadas as táticas de coerência semântica (do Inglês *Semantic Coherence*) e encapsulamento de informações (do Inglês *information Hiding*).

4.3.1 Visão de decomposição de módulos

Tendo em vista o atributo de modificabilidade e as táticas de coerência semântica e encapsulamento de informações, foi utilizada a estrutura arquitetural de decomposição em módulos. A estrutura arquitetural de decomposição em módulos pode ser definida com o objetivo de facilitar a compreensão de quais submódulos são responsáveis pela composição de um módulo geral. Ou seja, divide-se o módulo maior em todos os seus componentes ao ponto de simplificar a compreensão do papel de cada submódulo encapsulado. A arquitetura da ferramenta proposta foi definida em três módulos, nos quais são compostos por submódulos que são responsáveis por funcionalidades essenciais do sistema. A decomposição em módulos foi derivada considerando os passos definidos para a aplicação do ADD. Portanto, a

decomposição foi iniciada pelo sistema por completo, considerando restrições, requisitos funcionais e atributos de qualidade. Esses módulos são exibidos na Figura 8. Após definir a lista de módulos mais externos, pode-se descrever as responsabilidades atreladas a cada um desses módulos.

FIGURA 8 - MÓDULOS MAIS EXTERNOS



Fonte: (Autoria Própria)

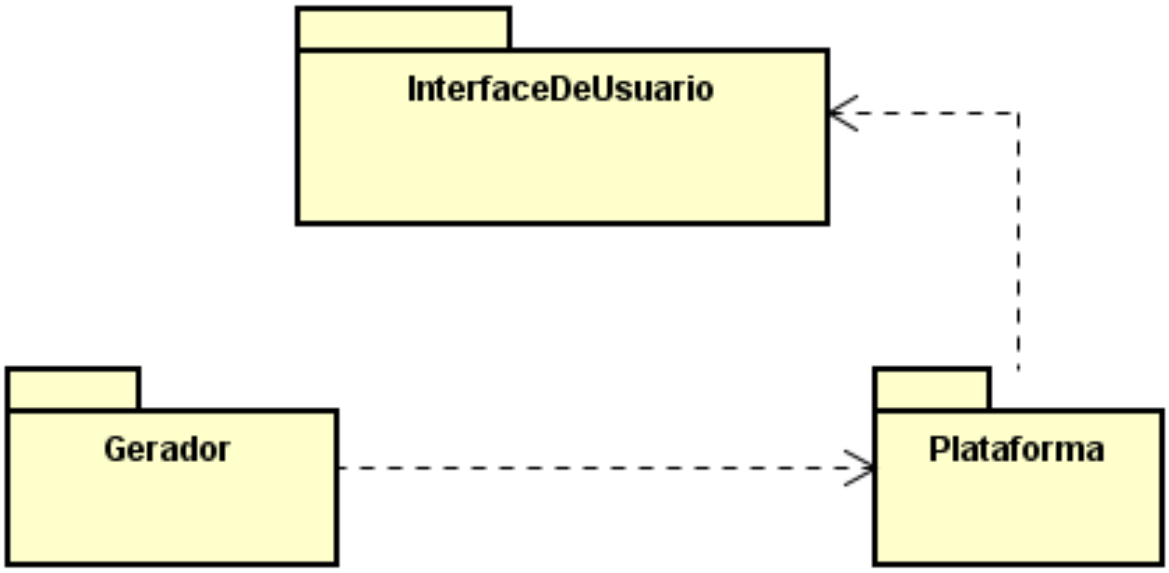
O módulo Gerador é uma visão mais externa de todos os submódulos responsáveis pela geração de casos de teste do sistema. Ou seja, todas as informações necessárias para que sejam gerados casos de teste a partir de um paradigma de programação e critérios definidos, estão encapsuladas em submódulos que compõem esse módulo mais externo denominado Gerador.

O módulo Plataforma é uma visão mais externa de todos os submódulos responsáveis pela execução da ferramenta. Ou seja, todas as informações necessárias para que a ferramenta seja posta em execução tanto via linha de comando quanto via plugin estão encapsuladas em submódulos que compõem esse módulo mais externo denominado Plataforma.

O módulo InterfaceDeUsuário é uma visão mais externa de todos os submódulos responsáveis pela exibição gráfica de grafos de fluxo de controle e casos de teste gerados pela ferramenta. Essa exibição é estimulada através de um plugin gerado em um ambiente de desenvolvimento.

Seguindo a técnica ADD, após a descrição dos módulos mais externos da arquitetura, gera-se um padrão arquitetural que possibilita ilustrar as dependências entre esses módulos. Esse padrão pode ser visto na Figura 9.

FIGURA 9 - PADRÃO ARQUITETURAL



Fonte: (Autoria Própria)

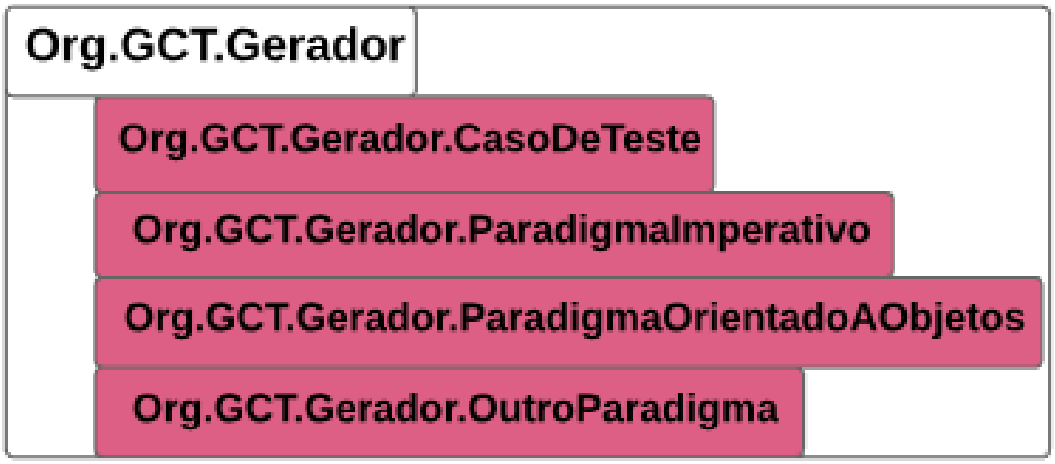
Na Figura 9 é abordada uma visão mais ampla da dependência entre os módulos mais externos. Pode-se perceber que o módulo Gerador possui dependência com o módulo Plataforma. Essa dependência ocorre porque o módulo Gerador necessita de artefatos existentes no módulo Plataforma para que os testes sejam efetuados. A dependência entre Plataforma e InterfaceDeUsuario ocorre porque o módulo Plataforma depende de artefatos presentes no módulo InterfaceDeUsuario para que seja possível exibir interações gráficas ao usuário. Maiores detalhes são relatados à medida que os módulos vão sendo decompostos.

A ideia de decomposição em módulos gera vários benefícios à arquitetura. No exemplo da Figura 9 percebe-se que em cada módulo são definidos papéis específicos. Ou seja, no módulo InterfaceDeUsuario, por exemplo, somente são encapsuladas informações de interface de usuário. Alterações nesse módulo não gera problemas no funcionamento dos demais, sendo isso um dos benefícios principais da decomposição em módulos. Os módulos foram definidos de forma a garantir a satisfação dos requisitos não funcionais definidos nas Figuras 3,4,5,6 e 7. Por exemplo, para atender a demanda dos requisitos atrelados a inserção de critérios e paradigmas de testes customizados foi necessário a definição do módulo Gerador. Por outro lado, para cumprir o requisito de teste vinculado a execução da ferramenta via console foi definido o módulo Plataforma. Por fim, para os requisitos atrelados à interface de usuário foi definido o módulo InterfaceDeUsuario.

Com base na técnica ADD, deve-se seguir os passos de decomposição até chegar em um nível de implementação. Com isso, para que esse objetivo seja atingido, passos de decomposição devem ser descritos na arquitetura proposta. O próximo passo é decompor os módulos mais externos definidos anteriormente.

O primeiro módulo a ser decomposto foi o módulo Gerador, onde na Figura 10 são listados todos os submódulos que o compõem. Após definir essa lista, pode-se definir as responsabilidades de cada submódulos.

FIGURA 10 - LISTA DE SUNMÓDULOS DO MÓDULO GERADOR



Fonte: (Autoria Própria)

O submódulo CasoDeTeste é responsável por armazenar todos os submódulos relacionados com a criação de casos de teste. Este possui dependências com os submódulos ParadigmaImperativo, ParadigmaOrientadoAObjetos e OutroParadigma. Essas dependências são necessárias para que o submódulo CasoDeTeste tenha acesso às características de cada critério de teste específico. Esse submódulo também possui relação de dependência com módulos presentes no módulo Plataforma. Como por exemplo, com o submódulo Motor, onde isso permite que os casos de testes sejam executados. Além disso, possui dependência com o submódulo Comuns para terem acesso a utilidades comuns entre todos os critérios de teste da ferramenta, além de possuir relação com o submódulo Anotacao para ter acesso às anotações referentes a cada critério de teste.

O submódulo ParadigmaImperativo é responsável por armazenar todos os submódulos necessários para que os critérios de teste referentes ao paradigma imperativo sejam aplicados.

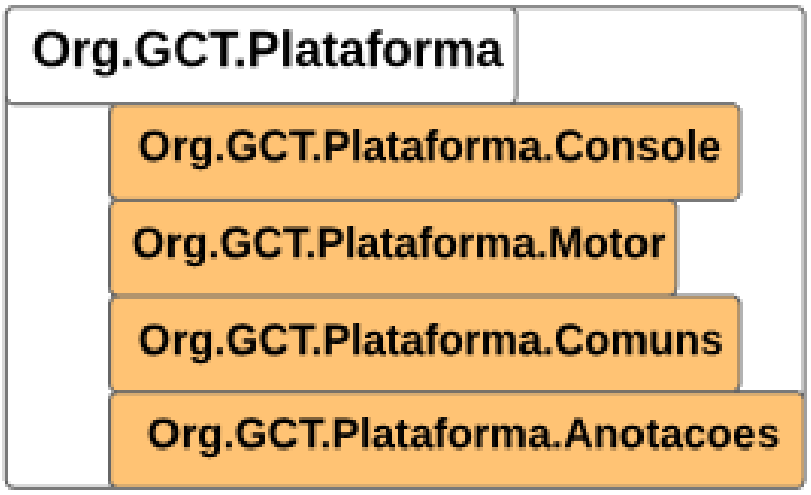
Nesse submódulo existem elementos mais internos com características peculiares ao aos testes voltados ao paradigma imperativo.

O submódulo ParadigmaOrientadoAObjetos é responsável por armazenar todos os elementos necessários para que os critérios de teste referentes ao POO sejam aplicados. Ele é composto por itens que possuem características peculiares aos testes voltados ao POO.

O submódulo OutroParadigma é responsável por permitir ao usuário inserir um novo paradigma de teste na ferramenta. Nesse submódulo existem elementos mais internos que permitem ao usuário inserir critérios de teste específico para um outro paradigma diferente dos paradigmas Imperativo e Orientado a Objeto.

O segundo módulo a ser decomposto é o módulo Plataforma, no qual são listados na Figura 11 todos os submódulos presentes no mesmo.

FIGURA 11 - LISTA DE SUBMÓDULOS DO MÓDULO PLATAFORMA



Fonte: (Autoria Própria)

O submódulo Console é responsável por permitir que a ferramenta proposta na arquitetura descrita seja capaz de ser executada via console ou via linha de comando. Além disso, esse submódulo encapsula submódulos que permitem ao usuário as impressões de GFC gráfico e casos de teste gerados. Mas para que a execução seja permitida, esse submódulo possui uma dependência com o elemento Motor, que é responsável pela execução da ferramenta. Com o objetivo de prover interface gráfica ao usuário, é necessário uma relação de dependência entre o Console e o IntefaceDeUsuario.

O submódulo Motor é responsável por permitir a execução dos critérios de teste para que os casos de teste sejam gerados. Esse submódulo encapsula todos os membros responsáveis por selecionar casos de teste a serem gerados por meio de uma unidade anotada.

O submódulo Comuns armazena informações padrões utilitárias por elementos da arquitetura. Ou seja, dados compartilhados com todos os tipos de teste gerados pela ferramenta, devem ser encapsuladas em módulos que compõem esse submódulo.

O submódulo Anotacao é responsável por armazenar informações sobre anotações que podem ser utilizadas para que casos de teste sejam gerador. Ou seja, nesse submódulos são encapsulados elementos responsáveis por gerenciar anotações referentes aos paradigmas Imperativo, Orientado a Objetos e um possível paradigma inserido pelo usuário.

O terceiro módulo a ser decomposto é o módulo Interface de Usuario, onde a Figura 12 lista todos os submódulos presentes no mesmo.

FIGURA 12 - LISTA DE SUBMÓDULOS DO MÓDULO INTERFACE DE USUÁRIO

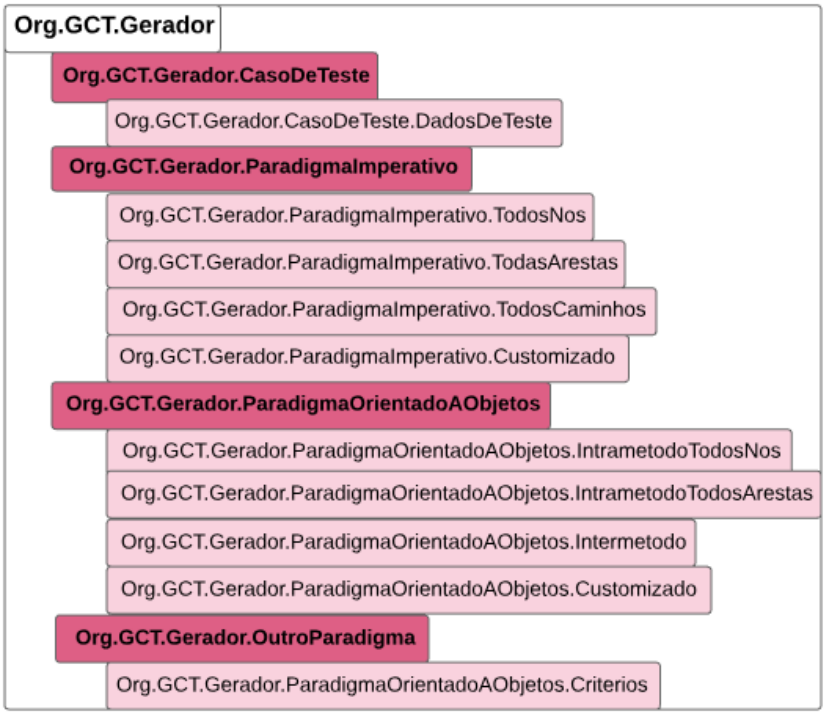
Fonte: (Autoria Própria)

O submódulo Apresentacoes armazena toda a estrutura necessária para prover interface gráfica ao usuário. Ou seja, essa exibição é permitida através de dados do que serão manipulados pelo submódulo Console que está encapsulado no módulo mais externo denominado Plataforma. Essa dependência possibilita que o GFC e os casos de teste sejam exibidos aos usuários.

Seguindo com as definições da técnica ADD, precisa-se decompor todos os submódulos encapsulados em toda a hierarquia que compõe os módulos mais externos, ou seja, é necessário desenvolver mais um passo para que sejam identificados os elementos presentes em cada submódulo.

Portanto, os primeiros submódulos a serem decompostos estão presentes nos submódulos que compõem o módulo Gerador. Com isso, a Figura 13 exibe a lista de toda a composição desses elementos.

FIGURA 13 - LISTA DE COMPOSIÇÃO GERAL DO MÓDULO GERADOR



Fonte: (Autoria Própria)

Dentro do módulo Gerador existe um submódulo identificado por CasoDeTeste. Esse elemento é composto pelo membro DadosDeTeste. O módulo DadosDeTeste armazena classes necessárias para que os dados de teste sejam criados a partir da aplicação de critérios de teste. Ou seja, nesse módulo existem todas as classes que monitoram os dados de teste e como eles foram gerados.

O segundo submódulo presente no módulo Gerador é o ParadigmaImperativo. Esse submódulo é composto pelos seguintes elementos: TodosNos, TodasArestas, TodosCaminhos e Customizados.

O submódulo TodosNos armazena classes responsáveis por gerenciar características do critério Todos-Nós baseado em fluxo de controle com foco no paradigma Imperativo. Essas classes encapsulam informações de como o grafo deve ser percorrido para que os caminhos sejam gerados, e com isso os dados de teste sejam derivados.

O submódulo TodosArestas mantém classes que objetivam administrar características pertencentes ao critério Todos-Arestas baseado em fluxo de controle com foco no paradigma imperativo. Essas classes encapsulam informações de como o grafo deve ser percorrido para que os caminhos sejam gerados, e com isso os dados de teste sejam derivados.

O submódulo TodosCaminhos encapsula classes que gerenciam propriedades particulares ao critério Todos-Arestas baseado em fluxo de controle com foco no paradigma Imperativo. Essas classes encapsulam informações de como o grafo deve ser percorrido para que os caminhos sejam gerados, e com isso os dados de teste sejam derivados.

O submódulo Customizado é composto por classes que definem padrões para que um novo critério de teste voltado ao paradigma imperativo seja desenvolvido pelo usuário. A ideia principal desse submódulo é atender ao requisito de modificabilidade definido na Figura 3. Para isso, deve existir uma classe abstrata que disponibilize métodos necessários para que um novo critério seja inserido.

O terceiro submódulo presente no módulo Gerador é o Paradigma.OrientadoAObejtos. Esse submódulo é composta pelos seguinte submódulos: IntrametodoTodosNos, IntrametodoTodasArestas, Intermetodo e Customizado.

O submódulo IntrametodoTodosNos encapsula classes capazes de gerarem casos de testes a partir de um método. Esse elemento encapsula classes que permitem gerar casos de teste por meio de um GFC gerado com base no fluxo de execução de um método, garantindo que todos os nós presentes nesse grafo foram percorridos ao menos uma vez. Porém, essas classes armazenam características de duas formas diferentes para gerar os dados de teste, onde em uma das formas considera o tratamento de exceção listado no grafo e outra não. Nesse submódulo existem classes que armazenam características do critério Todos-Nos considerando as exceções, como também existem classes que armazenam as características do critério para ser aplicado sem considerar as exceções.

O submódulo IntrametodoTodasArestas tem como papel armazenar classes que possibilitam a geração de casos de testes também a partir de um GFC. As classes armazenadas nesse elemento garantem a geração de casos de teste com o critério Todas-Arestas de duas formas, onde em uma forma o tratamento de exceção é considerado e em outra não.

O submódulo Intermétodo encapsula classes necessárias para que o teste de interação entre os métodos públicos de uma classe seja realizado. Ou seja, nesse contexto a unidade tratada é a classe.

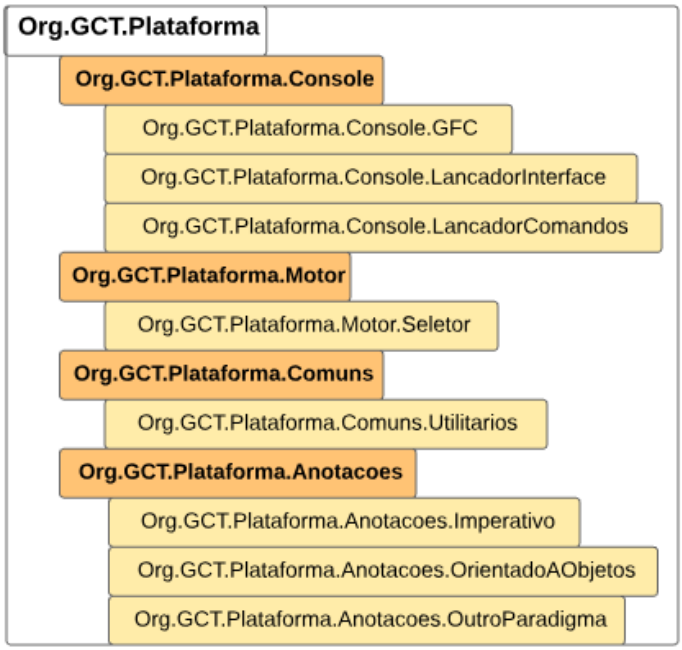
O submódulo Customizado encapsula classes necessárias para que um novo critério de teste voltado ao paradigma Orientado a Objetos seja inserido por um desenvolvedor. A ideia desse submódulo é atender ao requisito definido na Figura 4. Para isso, dentro desse elemento

deve existir uma classe abstrata que disponibilize métodos necessários para que critérios relacionados a um novo paradigma seja inserido.

O quarto elemento pertencente no módulo Gerador é o OutroParadigma descrito em passos anteriores. A composição desse submódulo é feita somente pelo membro Criterio. O submódulo Criterio encapsula uma classe abstrata contendo todos os métodos necessários para que todos os critérios referentes ao paradigma sejam inseridos, dado que a realização de teste voltados a um paradigma específico depende de um conjunto de critérios de teste. Ou seja, essas classes são responsáveis por manter critérios e padrões que definem os passos que devem ser seguidos para que o novo paradigma seja concretizado e obtenha sucesso no seu funcionamento. Portanto, esse submódulo foi definido com o objetivo de provê o requisito definido na Figura 5.

Com isso, toda a hierarquia de submódulos que está encapsula no módulo Gerador foi descrita. Portanto, o passo seguinte tem como objetivo descrever todos os elementos que estão encapsulados nos submódulos do módulo Plataforma. Os quais são listados na Figura 14.

FIGURA 14 - LISTA DE COMPOSIÇÃO GERAL DO MÓDULO PLATAFORMA



Fonte: (Autoria Própria)

Dentro do módulo Plataforma, o primeiro membro listado é o submódulo Console. Os seus submódulos encapsulados são os seguintes: GFC, LancadorInterface e LancadorComandos.

O elemento GFC é responsável por armazenar todas as classes que são necessárias para representar toda a estrutura de um grafo de fluxo de controle para permitir que um plugin possa exibi-lo graficamente.

O submódulo LancadorInterface tem como objetivo encapsular todas as classes que permitem que a execução dos testes possa ser estimulada via console. Ou seja, deve oferecer a possibilidade ao usuário de realizar testes vias ambiente de desenvolvimento. Portanto, essas classes devem ser capazes de lançar esses testes.

O submódulo LancadorComandos é composto por todas as classes que permitem que a ferramenta seja executada via linha de comandos. Ou seja, quando o usuário desejar utilizar o terminal para executar a ferramenta, essas classes devem possuir informações que possibilitem esse ato.

O segundo submódulo presente no módulo Plataforma é o elemento Motor. Esse submódulo é composto somente pelo o item Seletor. O submódulo Seletor tem como proposito armazenar todas as classes necessárias para que testes sejam gerados a partir de unidades anotadas. Essas classes devem realizar uma seleção a fim de decidirem qual teste será posto em execução primeiro pela ordem das unidades anotadas.

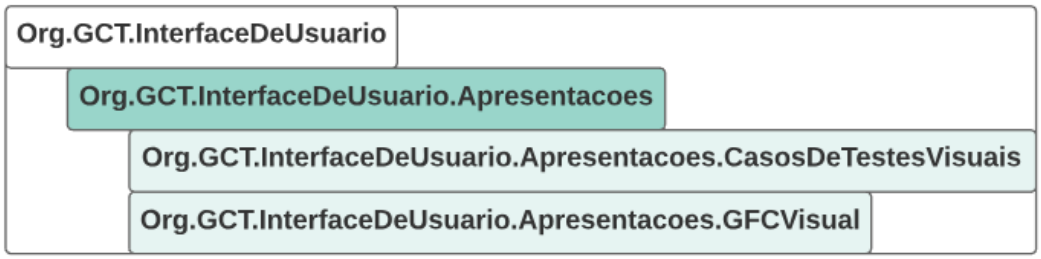
O terceiro submódulo listado entre os que compõem o módulo mais externo Plataforma é o submódulo Comuns. Esse elemento é composto somente pelo submódulo Utilitarios. O Utilitarios é responsável por encapsular todas as classes que gerenciam informações comuns a todos os tipos de testes que são gerados pela ferramenta utilizando diversos paradigma. Portanto, todas as informações comuns aos diversos tipos de teste são gerenciadas pelas classes armazenadas no submódulo Utilitarios.

O quarto membro presente no módulo Plataforma é o Anotacoes. Esse submódulo é composto pelos seguintes submódulos: Imperativo, OrientadoAObjetos e OutroParadigma. O submódulo Imperativo tem como objetivo armazenar todas as classes que são responsáveis por gerenciar as anotações de teste voltadas ao paradigma Imperativo. O submódulo OrientadoAObjetos tem como objetivo encapsular todas as classes que são responsáveis pela gerência das anotações de teste voltadas ao paradigma Orientado a objetos.

O submódulo OutroParadigma tem como objetivo armazenar todas as classes que são responsáveis por gerenciar as anotações de teste referentes a um paradigma de teste inserido pelo usuário. Apesar do foco neste trabalho estar em programas inscritos com linguagem

baseada no POO, como, por exemplo, Java. Fica a cargo do programador/testador a escolha da inserção de um critério associado a outro paradigma que se adeque ao contexto do programa testado. O intuito com essa funcionalidade é aumentar a flexibilidade e liberdade ao programador/testador durante a atividade de teste. Com isso, essas classes permitem que o usuário seja capaz de inserir novas anotações para referenciar os seus critérios de teste definidos para o novo paradigma. Portanto, definido a hierarquia de submódulos tanto do módulo Gerador, quanto do módulo Plataforma, o próximo passo e descrever toda a hierarquia referente ao submódulo Interface de Usuario, tal que, na Figura 15 é exibida toda a lista de submódulos presentes.

FIGURA 15 - LISTA DE COMPOSIÇÃO GERAL DO MÓDULO INTERFACE DE USUÁRIO



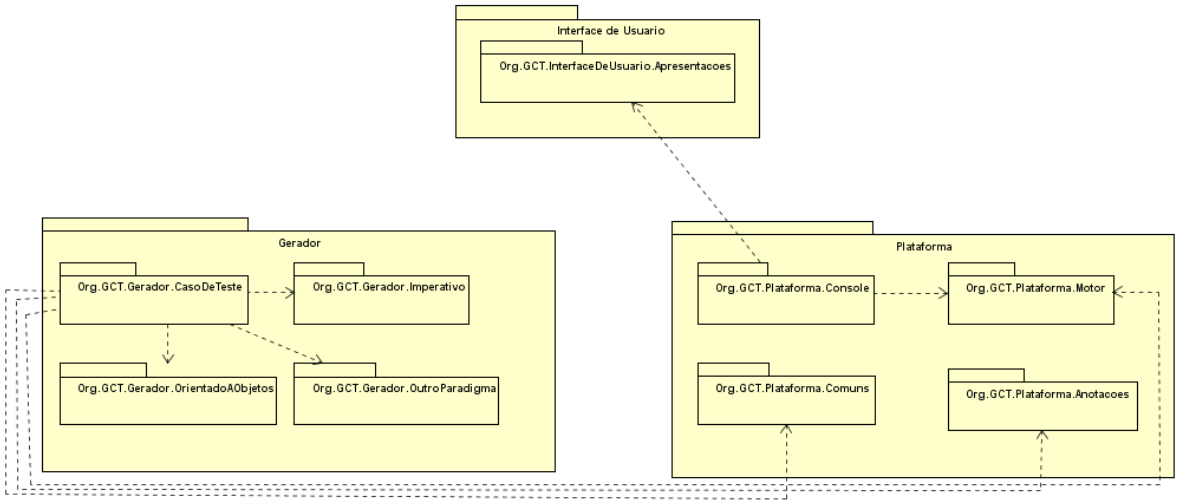
Fonte: (Autoria Própria)

O módulo Interface de Usuario é composto somente pelo item Apresentacoes. O submódulo Apresentacoes é composto apenas pelos elementos CasosDeTesteVisuais e GFCVisual. O submódulo CasosDeTesteVisuais tem responsabilidade de armazenar as classes necessárias para seja capaz de exibir de forma gráfica ao usuário os casos de teste gerados a partir de aplicação dos critérios. O submódulo GFCVisual encapsula classes que gerenciam toda a parte de exibição visual do grafo de fluxo de controle das unidades testadas. A ideia de manter um módulo mais externo voltado a interface de usuário é garantir a conformidade com os requisitos retratados nas Figuras 6 e 7.

4.3.2 Visão de uso entre submódulos

Após a descrição dos elementos que compõem os módulos mais externos do sistema, pode-se gerar um diagrama que englobe uma forma mais simples de analisar a dependência entres os mesmos. Portanto, esse diagrama é mostrado na Figura 16.

FIGURA 16 - VISÃO DE USO ENTRE SUBMÓDULOS



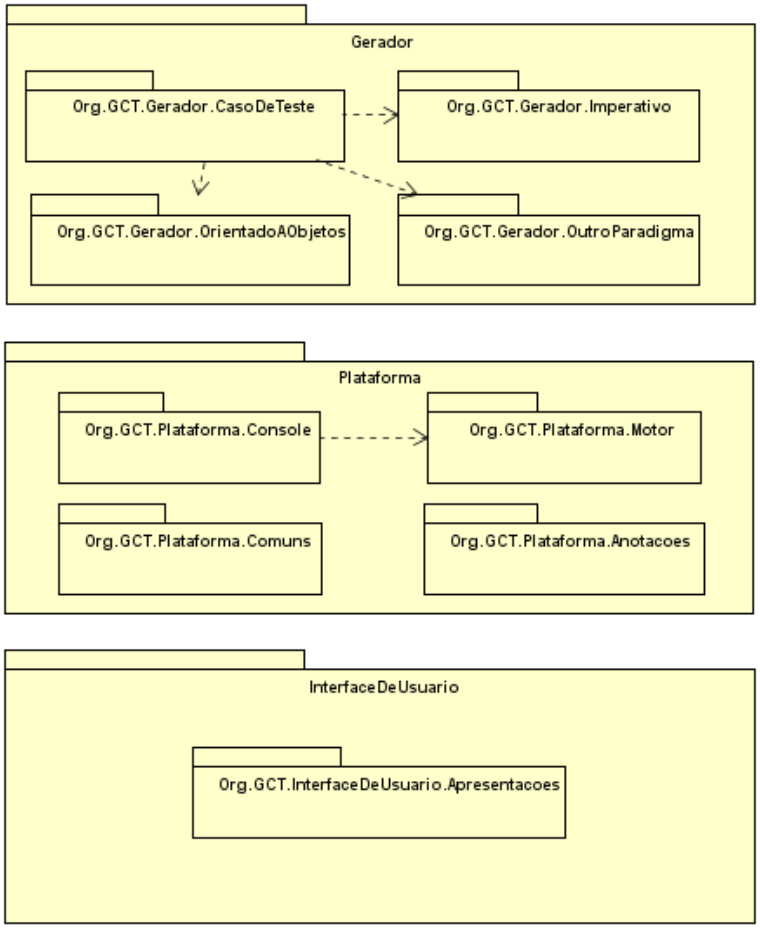
Fonte: (Autoria Própria)

Analisando a Figura 16, pode-se perceber que o submódulo Gerador.CasoDeTeste possui dependência com todos os membros que compõem o módulo Gerador, além de possuir dependência com elementos do modulo Plataforma. Isto é, por esse motivo foi abordada a dependência entre o submódulo Gerador e Plataforma na Figura 9. Percebe-se também que o submódulo Plataforma.Console além de possuir dependência com um dos submódulos que compõem o módulo Plataforma, possui relação de dependência com o submódulo do módulo InterfaceDeUsuario. Esse é o motivo que é esboçado a dependência entre Plataforma e InterfaceDeUsuario na Figura 9.

4.3.3 Visão em camadas

Com foco no atributo de modificabilidade, foi definido uma visão em camadas com base na visão de uso. O objetivo com essa visão é abordar que os módulos foram definidos de forma a possibilitar a extensão de suas funcionalidades de forma simples, além de possibilitar o conhecimento de níveis de dependência entre cada camada (BASS; CLEMENTS; KAZMAN, 2003). Na Figura 17 é retratada mais uma das estruturas arquiteturais utilizadas neste trabalho.

FIGURA 17 - VISÃO EM CAMADAS



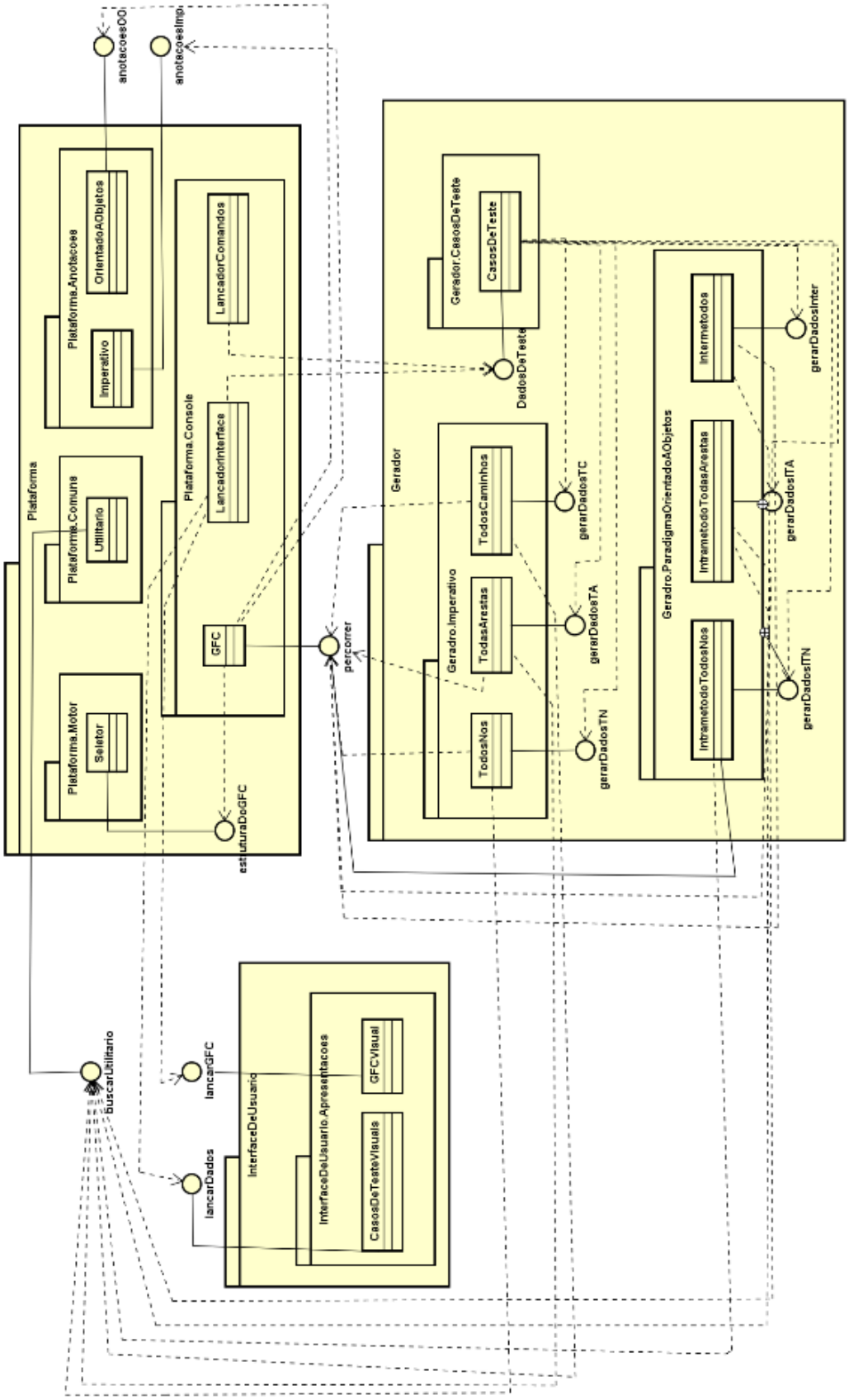
Fonte: (Autoria Própria)

Com base nessa figura, pode-se analisar que a camada Gerador possui dependências com a Plataforma. Essa dependência ocorre porque para que os casos de teste sejam gerados, alguns elementos presentes na camada Plataforma necessitam ser utilizados pela camada Gerador. Outra análise de dependência percebida ocorre entre a camada Plataforma e a camada Interface de Usuario, essa dependência é necessária para que a camada Plataforma consiga gerar dados a serem exibidos na interface de usuário.

4.3.4 Visão de estrutura em classes

Dado a definição das classes concretas que compõem o sistema, pode-se utilizar o diagrama de classes para descrever como ocorre a relação de dependências entre esses elementos (PRESSMAM, 2011). O diagrama de classes derivado da arquitetura proposta é apresentado na Figura 18.

FIGURA 18 - DIAGRAMA DE CLASSES E SUAS INTERFACES



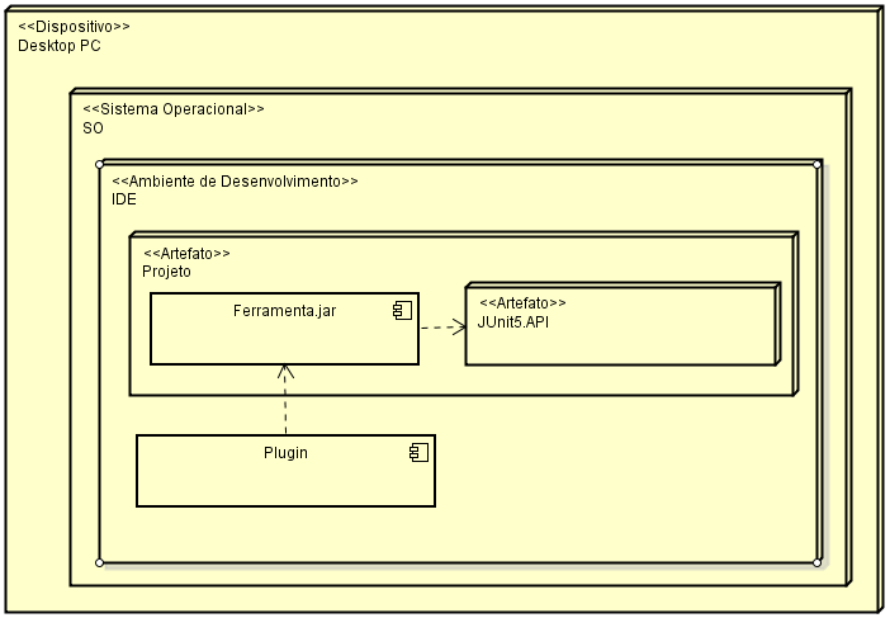
Fonte: (Autoria Própria)

Pode-se perceber no diagrama da Figura 18 que as interfaces de cada submódulo foram definidas, possibilitando uma melhor visão de quais métodos públicos irão compor cada classe. Portanto, tem-se uma visão de mais baixo nível dos elementos do sistema.

4.3.5 Visão de estrutura de alocação

Fora do contexto de modularização, mas para possibilitar uma ideia de elementos necessários na implantação do software proposto com a arquitetura, foi desenvolvida uma nova visão. A visão (baseada na estrutura arquitetural de alocação) tem como objetivo principal exibir toda combinação da arquitetura, tratando-se de elementos de software e elementos diversos, como, por exemplo, elementos de hardware e redes. Nessa visão, pode-se utilizar o diagrama de implantação para representar relações entre elementos de infraestrutura e artefatos de software que permitem a implantação do software. A Figura 19 aborda o diagrama de implantação da arquitetura definida neste trabalho.

FIGURA 19 - DIAGRAMADE IMPLANTAÇÃO



Fonte: (Autoria Própria)

No diagrama apresentado na Figura 19, pode-se perceber a relação entre 7 elementos, sendo eles, elementos de infraestrutura (Elementos que não são aplicações de software), artefatos (Aplicação de software) e componentes (Elementos de um software). Partindo do elemento mais externo, ou seja, o Desktop PC, percebe-se que todos os artefatos e componentes estão encapsulados nesse elemento. Isso é necessário porque a arquitetura proposta neste trabalho possui foco em plataforma desktop. Em seguida, o elemento SO representa os possíveis

sistemas operacionais que possibilitam a execução de uma ferramenta desenvolvida com base nessa arquitetura. Não foram definidas restrições relacionadas a sistemas operacionais.

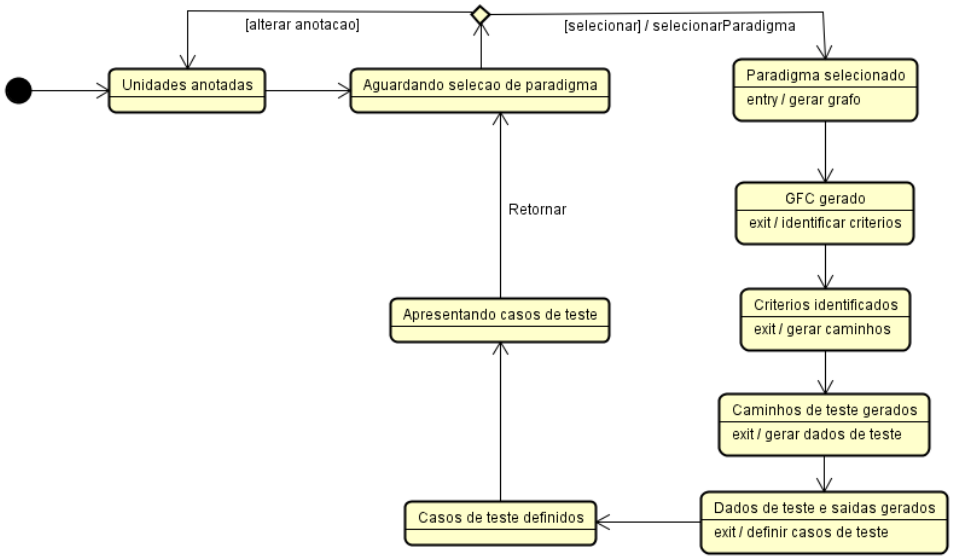
O sistema operacional é composto pelo artefato Ambiente de Desenvolvimento, que faz junção a características e ferramentas que possibilitam a construção de softwares. Portanto, encapsulados no Ambiente de Desenvolvimento tem-se o artefato Projeto e os componentes Ferramenta.jar e Plugin, que proveem todas as funcionalidades ao usuário. O artefato Projeto é composto pelo artefato JUnit5.API e o componente Ferramenta.jar, que é um componente final gerado a partir de todos os elementos necessários para que a ferramenta possa ser executada, inclusive, nele estão presentes todas as dependências entre as classes apresentadas na Figura 18. Já o artefato JUnit5.API disponibiliza todas as bibliotecas necessárias para que testes de interação entre métodos públicos possam ser realizados pela Ferramenta.jar (isso é retratado na dependência entre o Ferramenta.jar e JUnit5.API). O componente Plugin é apresentado como uma maneira de exibir informações aos usuários via interface gráfica. Para que esse componente tenha acesso a dados, ele mantém uma relação de dependência com o componente Ferramenta.jar.

4.4 Exemplos de comportamentos de módulos do sistema

Para analisar a dinâmica de execução do sistema, pode-se utilizar modelos de diagramas definidos pela UML, que auxiliam na compreensão dos possíveis comportamento dos objetos que compõem o sistema, como por exemplo, o diagrama máquina de estados. Esse diagrama é capaz de descrever todo o comportamento do sistema com base em eventos internos e externos que podem alterar o estado do sistema por meio de transições (SOMMERVILLE, 2011). Dentro de cada estado podem ser definidas ações e atividades que podem ocorrer em diferentes estágios. As atividades podem ser *entry* (ação realizada assim que o estado é iniciado), *do* (atividade realizada dentro de um estado) e *exit* (ações executadas quando o estado é finalizado).

Para analisar o comportamento de alguns módulos propostos na arquitetura, foram definidos três exemplos, dado que eles permitem a análise das ideias principais da ferramenta. Para cada exemplo foi desenvolvido um diagrama de máquina de estados e a sua descrição. O primeiro exemplo define o comportamento dos módulos quando um usuário da ferramenta decide gerar casos de teste aplicando um ou vários paradigmas de teste por meio de anotações que referenciam critérios. A Figura 20 retrata o diagrama responsável por ilustrar esse exemplo.

FIGURA 20 - DIAGRAMA DE ESTADO PARA GERAR CASOS DE TESTE



Fonte: (Autoria Própria)

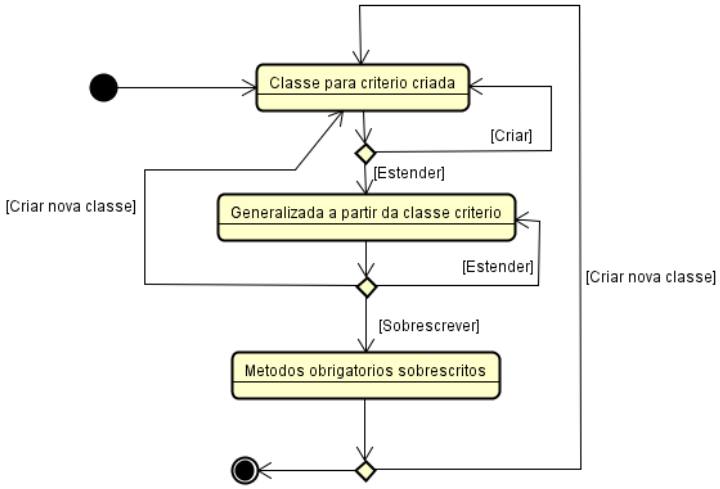
Após a transição do estado inicial ser disparada, o primeiro estado ordinário do sistema é denominado Unidades anotadas. Para que esse estado seja posto em execução, é necessário que um usuário tenha anotado devidamente as unidades a serem testadas por meio de *tags*, dado que essas *tags* referenciam cada critério de teste a ser aplicado. No próximo estado, o sistema aguarda uma decisão do usuário, entre inserir novas anotações ou definir o paradigma que deseja utilizar na geração de casos de teste. Se o usuário decidir inserir novas anotações, após o término dessa atividade, o sistema retorna ao estado Unidades anotadas. Entretanto, se o usuário selecionar o paradigma de teste, o sistema entrará em um outro estado identificado por Paradigma selecionado. Esse estado possui uma ação de entrada que é responsável por gerar o GFC de cada unidade anotada. Quando a transição é disparada, o novo estado iniciado é GFC gerado, esse estado deve garantir que todos os grafos de fluxo de controle foram gerados e possui como ação de saída percorrer todas as unidades anotadas e identificar os critérios de teste presentes em cada unidade. Em seguida, o próximo estado é iniciado, esse estado é denominado de Criterios identificados. Nesse estado, todos os critérios de teste que a serem utilizados para gerar caminhos de teste devem ter sido identificados.

Além disso, o estado possui uma ação de saída, definindo que no final da sua execução devem ser gerados todos os caminhos de teste com base na aplicação de critérios que percorrem a estrutura do GFC de cada unidade. Com isso, a transição ativa o estado Caminhos de teste gerados. Quando esse estado é posto em execução, todos os caminhos de teste devem estar definidos para que os dados de teste sejam selecionados, dado que esses dados devem ser

gerados em uma ação realizada no final da execução do estado Caminhos de teste gerados. Após a identificação dos dados de teste, o estado conhecido por Dados de teste e saídas gerados é executado. Isso representa que foram mapeadas as saídas para cada dados de entrada, e no fim da sua execução, o caso de teste é gerado por meio da ação “definir casos de teste”. Após todo o processo descrito, o estado Casos de teste definidos é iniciado, esse representa que o conjunto de testes para todos as unidades anotadas foi identificado e serão exibidos ao usuário no próximo estado, conhecido por Apresentando casos de teste. Ou seja, nesse estado os casos de teste são exibidos ao usuário até que o mesmo decida retornar ao estado de Aguardando seleção de paradigma.

O segundo exemplo aborda todo o comportamento necessário para que um programador consiga inserir um novo critério de teste em um paradigma de teste já existente na ferramenta. A Figura 21 exibe o grafo que retrata todo o fluxo.

FIGURA 21 - DIAGRAMA DE ESTADO PARA INSERIR CRITÉRIO CUSTOMIZADO



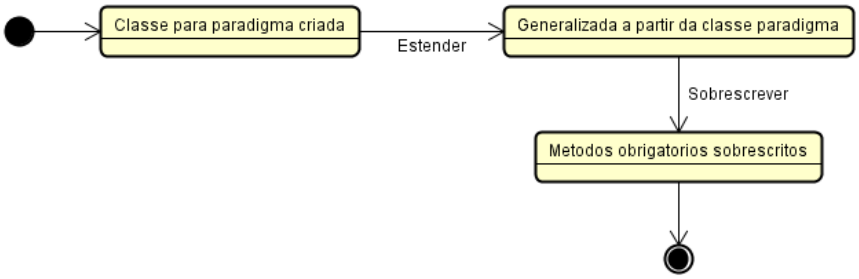
Fonte: (Autoria Própria)

Nesse diagrama, o contexto de comportamento do sistema foi derivado para verificar também possíveis comportamentos do programador que ocasionarão mudanças no estado do sistema. Após a transição do estado inicial ser disparada, o próximo estado é denominado Classe para criterio criada. Para que esse estado seja posto em execução, é necessário que um programador tenha criado uma classe com origem de uma classe abstrata responsável pela implementação de critérios. Com isso, o programador pode decidir criar mais classes ou estender a(s) classe(s) já criada(s).

Quando o desenvolvedor decide criar novas classes, o fluxo retorna ao estado Classe para criterio criada. Caso ele decida estender uma classe, o fluxo segue para o estado Generalizada a partir da classe criterio. Portanto, o programador pode tomar mais uma decisão entre estender mais classes dentre as que foram criadas, sobrescrever os métodos das classes estendidas, ou então criar mais classes. Caso ele decida estender outra classe, o fluxo retorna ao estado Generalizada a partir da classe criterio, ou seja, mais uma classe é estendida. Caso ele decida sobrescrever os métodos de uma classe estendida, o fluxo segue para o estado Metodos obrigatorios sobrescritos. Isto é, os métodos das classes abstratas foram reimplementados e o programador pode decidir finalizar o fluxo ou então criar novas classes para que o fluxo retorne ao estado Classe para criterio criada. A outra opção partindo do estado Generalizada a partir da classe criterio permite ao programador criar novas classes e o fluxo retorna para o estado Classe para criterio criada. Portanto, essa ideia de estender classe abstrata garante que o programador realize no mínimo a implementação dos métodos necessários para que o critério possa ser integrado a um paradigma de teste.

O terceiro exemplo de representação de diagramas de estado também está atrelado a comportamentos executados de acordo com as decisões do programador quando o mesmo decide inserir um novo paradigma de teste. A Figura 22 retrata esse diagrama.

FIGURA 22 - DIAGRAMA DE ESTADO PARA INSERIR NOVO PARADIGMA DE TESTE



Fonte: (Autoria Própria)

Nesse exemplo, a transição do estado inicial é disparada quando o programador decide criar uma nova classe derivada de uma classe abstrata que permite que um novo paradigma de teste passe a ser integrado a ferramenta. Com isso, o fluxo segue para o estado Classe para paradigma criada. Pode-se observar que esse diagrama é mais simples que o diagrama da Figura 19. Essa ideia de simplicidade foi utilizada para que seja representado um fluxo de troca de estados com maior clareza, porém, nada impede que após a criação de uma classe, o programador decida gerar novas classes antes de estender as classes definidas no primeiro

momento. Entretanto, a ideia desse exemplo é explicar uma forma mais sequencial entre os passos. Portanto, após uma classe ser criada, o programador pode estendê-la, ou seja, o fluxo inicia o estado Generalizada a partir da classe paradigma, onde esse estado representa que a classe foi instanciada e o programador pode decidir sobrescrever os métodos obrigatórios. Quando os métodos são sobrescritos, o fluxo segue para o próximo estado denominado Metodos obrigatórios sobrescritos e logo após é encerrado.

5. AVALIAÇÃO DA ARQUITETURA APLICANDO A TÉCNICA ATAM

O método ATAM tem como propósito analisar o nível de satisfação que uma arquitetura atende aos seus nortes particulares de qualidade. Essa análise é feita por equipes com papéis definidos. Porém, no contexto deste trabalho não foi abordado a criação de uma equipe específica, pois existem limitações na quantidade de pessoas envolvidas nesse projeto. Os papéis de líder de equipe, líder de avaliação, escriba de cenário, escriba do ocorrido, controlador de tempo, observador de processo, catalisador do processo e questionador foram desenvolvidos todos pelos participantes deste trabalho. Dado que a arquitetura proposta no trabalho está desenvolvida com uma técnica que se baseia nos atributos de qualidade almejados, a técnica ATAM ajudou significativamente nessa análise de conformidade dos requisitos não funcionais com o atributo de qualidade buscado.

A técnica ATAM, como descrita na Subseção 2.4 do documento, é aplicada por meio de passos sequências, que focam em apresentação, análise e resultados. O passo 1 definido pela técnica consiste na apresentação da técnica a todos os representantes do projeto. Nesse caso, os representantes desse projeto é apenas o aluno desenvolvedor do documento e ele tem acesso a descrição geral da técnica.

O passo 2 define que o gerente de projeto ou clientes do sistema contextualizam o sistema em perspectiva de negócio. No caso da arquitetura provida no trabalho, não é possível existir uma perspectiva única de negócio, visto que o público alvo são todos os programadores e testadores de software. Portanto, o público alvo pode possuir características diferentes.

O passo 3 solicita que a arquitetura seja apresentada em um nível apropriado de detalhes. Essa apresentação já foi apresentada no Capítulo 4 deste trabalho com um nível considerável de detalhes e pode ser consultada a qualquer momento.

O passo 4 consiste na definição de padrões arquiteturais desenvolvidos de acordo com os atributos de qualidade objetivados pela arquitetura. Dado a aplicação da técnica ADD no desenvolvimento da arquitetura proposta neste trabalho, essa técnica define em um dos seus passos a criação de um padrão arquitetural. Ou seja, o padrão arquitetural desta arquitetura proposta já foi definido durante a sua criação e pode ser visto na Figura 9.

O passo 5 define que as partes interessadas no projeto devem definir prioridades, com base em pesos para os atributos de qualidade almejados e devem quantificar os requisitos não funcionais para cada atributo de qualidade com base em cenários de uso. No caso deste trabalho, as partes interessadas identificaram somente um atributo de qualidade, que nesse caso é o de Modificabilidade. portanto, não foi necessário definir pesos, visto que esse atributo é o único existente e possui prioridade máxima. Quanto a quantidade de requisitos responsáveis por proverem esse atributo foram identificados 5, com cenários descritos nas Figuras 3,4,5,6 e 7.

O passo 6 consiste em identificar os cenários mais bem classificados no passo anterior e analisar como a arquitetura é capaz de suportar cada cenário, esse passo é contemplado na Subseção 5.1. Os passos 7 e 8 não possuem relevância no cenário da arquitetura avaliada, pois basicamente, o passo 7 consiste em os representantes do cliente priorizarem os cenários que descrevem os requisitos não funcionais mais relevantes. Logo após, acontece uma comparação entre os cenários priorizados nesse momento e os cenários priorizados com base no peso dos atributos de qualidade do passo 5. O passo 8 consiste na explicação do arquiteto em como a arquitetura suportará esses cenários, porém, no caso da arquitetura proposta não faz sentido refazer essa análise, pois só existe um atributo de qualidade e os cenários serão os mesmos. O passo 9 consiste em apresentar todas as informações obtidas com a avaliação. Portanto, durante a avaliação não foram identificados conflitos na arquitetura projetada. Todos os cenários de qualidade foram definidos, possibilitando a análise de como a arquitetura suporta cada cenário. Essa conclusão é feita com base nas decisões arquiteturais e nos requisitos de modificabilidade definidos.

5.1 Avaliação dos requisitos

Nessa subseção foi definido como a arquitetura suporta cada cenário das Figuras 3,4,5,6 e 7 derivados com base em requisitos e no atributo de modificabilidade.

5.1.1 Imperativo customizado

Como a arquitetura do sistema foi desenvolvida com base na estrutura de modularização, um modulo foi definido com o objetivo de contemplar esse requisito. O módulo denominado Gerador.ParadigmaImperativo.Customizado é composto por uma classe abstrata que encapsula todos os métodos obrigatórios a serem implementados para que o critério seja inserido. Portanto, dado a existência de um módulo específico para a inserção de um novo critério de teste, o desenvolvedor é capaz de integrar o novo critério de teste a ferramenta de forma rápida e concisa sem demandar que o mesmo se preocupe com outras partes do software.

Portanto, durante a avaliação não foram identificados conflitos na arquitetura projetada. Todos os cenários de qualidade foram definidos, com base nessa definição foi apontado como a arquitetura suporta cada cenário.

5.1.2 POO customizado

Seguindo a ideia descrita na Subseção 5.1.2, também foi projetado um módulo denominado Gerador.ParadigmaOrientadoAObjetos.Customizado que encapsula uma classe abstrata com métodos obrigatório a serem implementados para que o critério de teste customizado seja considerado. Com isso, o módulo específico permite ao desenvolvedor uma maior agilidade, além de garantir que não seja necessário entender todos os módulos do sistema para que esse ato seja efetuado.

5.1.3 Novo paradigma

Para contemplar esse requisito, foi definido um submódulo denominado de Gerador.OutroParadigma.Criterio que encapsula uma classes abstratas com métodos obrigatórios a serem implementados para que o usuário consiga integrar esse novo paradigma a ferramenta. Portanto, dado a definição desse módulo, o programador será capaz de implementar esse paradigma de forma rápida, visto que são definidos os métodos necessários, é apenas necessário a preocupação com a implementação desses métodos para que o novo paradigma funcione corretamente.

5.1.4 Integrar Plugin

Esse requisito é comtemplado com a definição do módulo Plataforma.console, pois esse submódulo encapsula classes que armazenam as informações necessárias para que o plugin possa requisitar a execução da ferramenta para gerar dados gráficos ao usuário. Ou seja, o desenvolvedor somente irá se preocupar em definir o plugin com o objetivo de que ele realize requisições ao módulo Plataforma.Console.

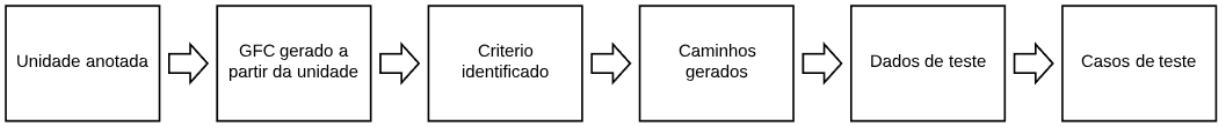
5.1.5 Interface de usuário

A modificação de interface de usuário é permitida através da definição do módulo InterfaceDeUsuario.Apresentacoes que encapsula informações de dados de testes e grafos. Essas informações podem ser alteradas de maneira simples, visto que o programador terá acesso a elas por meio desses módulos.

5.2 Simulação de uso de uma ferramenta derivada a partir da arquitetura proposta

Nesta subseção, é apresentada uma simulação de como são gerados casos de testes por uma ferramenta oriunda da arquitetura proposta neste trabalho, na qual essa ferramenta é capaz de gerar casos de teste com critérios referentes ao POO e ao paradigma imperativo utilizando fluxo de controle. Ou seja, é uma simulação de cenários de uso da ferramenta, além de ser exibido exemplos de inserção de novo critério de teste para o POO e para o paradigma imperativo. Na Figura 23 é mostrado um fluxo sequencial que facilita a compreensão do funcionamento dessa ferramenta para a geração de casos de teste de forma automatizada. Essa simulação de geração de casos de teste é utilizada tanto quando os testes são realizados via linha de comandos ou quando os testes são gerados por meio de ambientes de desenvolvimento.

FIGURA 23 - FLUXO QUE REPRESENTA A GERAÇÃO DE CASOS DE TESTE



Fonte: (Autoria Própria)

O primeiro passo consiste na anotação da unidade de entrada com uma *tag* que referencia algum critério de teste com base no paradigma selecionado. Feito isso, o próximo passo especifica a geração de um GFC com base nessa unidade. Dado a geração do GFC, a próxima etapa é identificar qual critério de teste a unidade está anotada. Após identificar esse critério, o GFC é percorrido, e com isso, gera-se os caminhos de teste. Portanto, com base nos caminhos de teste, são gerados os dados de testes que ocasionam em casos de teste.

5.2.1 Geração de casos de testes com POO

Para a geração de casos de testes no POO, é analisado o fluxo de controle de uma unidade. Especificamente neste trabalho, considera-se como uma unidade de teste, um método de uma classe. Gerado o GFC e com base no critério anotado, os caminhos de testes são definidos e os dados de teste são coletados para formar o conjunto de casos de teste.

FIGURA 24 – CRITERIO INTRAMÉTODO

```
public class MyClass {
    @IntrametodoTNSE
    public void verificaNumero(int n){
        if(n%2 != 0){
            System.out.println("Numero impar");
        }else{
            System.out.println("Numero par");
        }
    }
}
```

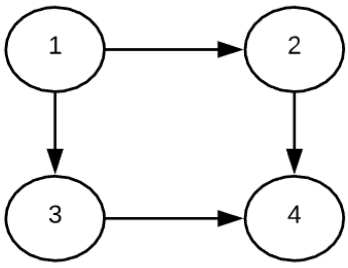
Fonte: (Autoria Própria)

Na Figura 24 é mostrada uma classe desenvolvida em Java, na qual existe um método anotado com a *tag* @IntrametodoTNSE. Essa *tag* faz referência ao critério intramétodo Todos-Nós presente no POO, onde quando essa *tag* é utilizada não é considerado o tratamento de exceções. Para que fosse considerado o tratamento de exceções a *tag* deveria ser inscrita @IntrametodoTNCE.

Portanto, dado que na classe existe uma unidade anotada com um critério de teste, a ferramenta deve gerar o GFC correspondente a essa unidade.

Para gerar o GFC, a ferramenta analisa quais as instruções sequenciais e quais as instruções de desvios para que todo esse fluxo seja representado no GFC. Na Figura 25 é mostrado o GFC derivado com base nessa unidade.

FIGURA 25 - GFC GERADO COM BASE EM UMA UNIDADE DE POO



Fonte: (Autoria Própria)

Percebe-se que o GFC é composto somente por quatro nós, onde o nó 1 representa o nó inicial, no qual é uma estrutura condicional que pode seguir o fluxo para o nó 2 ou para o nó 3, caso o fluxo siga para o nó 2 ele executa uma instrução informando que o número é ímpar e segue para o nó final 4. Caso contrário, ele executa a instrução informando que o número é par

e o fluxo segue para o nó final 4. A forma de geração do GFC foi apresentado na Subseção 2.1 deste trabalho.

Quando o GFC é gerado, a ferramenta identifica quais os critérios que o método está anotado para que o GFC seja percorrido com base nesse critério e os caminhos de teste sejam gerados. Com isso, dado que o método está anotado com a *tag* @IntrametodoTNSE, isso significa que os caminhos de teste devem de ser definidos de modo que todos os nós presentes no GFC sejam percorridos ao menos uma vez. Portanto, para esse GFC deriva-se dois caminhos de teste, nos quais C1 faz referência ao caminho 1 e C2 faz referência ao caminho 2. C1 é composto pelo nó 1, nó 2 e nó 4. Já o caminho C2 é composto pelo nó 1, nó 3 e nó 4.

Dado a identificação desses caminhos de teste, a ferramenta deve ser capaz de analisar quais dados de teste conseguem forçar a passagem por esses caminhos. Para que esses caminhos sejam contemplados, é necessário um dado de teste que seja um número ímpar e um dado de teste que seja um número par. Ou seja, para percorrer o caminho C1, pode-se ter o seguinte caso de teste: entrada = 1 e saída = "Numero impar". Para percorrer o caminho de teste C2, pode ter o seguinte caso de teste: entrada = 2 e saída = "Numero par". Portanto, percebe-se que para essa unidade, a ferramenta identifica de forma automatizada dois casos de teste que são capazes de analisar todo o fluxo que pode ser seguido por essa unidade quando dados de entrada são obtidos. Na Tabela 2 é exibido um melhor detalhamento desses dados de teste.

TABELA 2 - CASOS DE TESTE DO POO

Caso de teste	**Caminho de teste**	**Dados de entrada**	**Saída esperada**
Caso 01	nó 1 → nó 2 → nó 4	n = 1	"Numero impar"
Caso 02	nó 1 → nó 3 → nó 4	n = 2	"Numero par"

Fonte: (Autoria Própria)

5.2.2 Geração de casos de testes usando critério do paradigma imperativo

Para a geração de casos de teste com base nos critérios paradigma imperativo, também é analisado o fluxo de controle de uma unidade. Gerado o GFC e com base no critério anotado, os caminhos de testes são definidos e os dados de teste são coletados para formar o conjunto de

casos de teste. Na Figura 26 é mostrado outra classe desenvolvida em Java, na qual existe um método anotado com a tag @TodasArestas, critério esse, pertencente ao paradigma imperativo.

FIGURA 26 - CRITÉRIO TODAS-ARESTAS

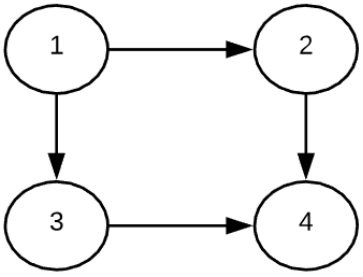
```
public class MyClass {

    @TodasArestas
    public void verificaValor(int n){
        if(n <= 0){
        System.out.println("Numero não positivo");
        }else{
        printfln("Numero positivo");
        }
    }
}                                                    //
```

Fonte: (Autoria Própria)

Portanto, dado que no programa existe uma unidade anotada com um critério de teste, a ferramenta deve gerar o GFC correspondente a essa unidade.

Para gerar o GFC, a ferramenta analisa quais as instruções sequenciais e quais as instruções de desvios para que todo esse fluxo seja representado no GFC. Na Figura 27 é mostrado o GFC derivado com base nessa unidade, neste caso específico, o GFC da Figura 27 ficou semelhante ao Figura 25, visto que os métodos possuem estruturas semelhantes.

FIGURA 27 - GFC GERADO COM BASE EM NOVA UNIDADE DO POO



Fonte: (Autoria Própria)

Percebe-se que o GFC é composto somente por quatro nós, onde o nó 1 representa o nó inicial, no qual é uma estrutura condicional que pode seguir o fluxo para o nó 2 ou para o nó 3. Caso o fluxo siga para o nó 2 ele executa uma instrução informando que o número é não positivo e segue para o nó final 4, ou caso o fluxo siga para o nó 3 ele executa a instrução informando que o número é positivo e o fluxo segue para o nó final 4.

Após a geração do GFC, a ferramenta identifica quais critérios a função está anotada. Nesse caso, o método da Figura 26 está anotada com a *tag* @TodasArestas e isso significa que o GFC deve ser percorrido de maneira que todas as arestas presentes nesse grafo sejam percorridas ao menos uma vez. Portanto, para esse GFC deriva-se dois caminhos, nos quais C1 referencia o caminho 1 e C2 referencia o caminho 2. O caminho C1 é composto pelo nó 1, nó 3 e nó 4. O caminho C2 é composto pelo nó 1, nó 2 e nó 4.

Dado a identificação desses caminhos, a ferramenta gera dados que implicam na execução dos mesmos. Portanto, para que esses caminhos sejam percorridos é necessário um número menor ou igual a zero e um número maior que zero como dados de entrada. Ou seja, para percorrer o caminho C1, pode-se ter o seguinte caso de teste: Entrada = -1 e saída = "Numero não positivo". Para percorrer o caminho de teste C2, pode ter o seguinte caso de teste: Entrada = 2 e saída = "Numero positivo". Portanto, percebe-se que para essa unidade, a ferramenta identifica de forma automatizada dois casos de teste que são capazes de analisar todo o fluxo que pode ser seguido por essa unidade quando dados de entrada são obtidos. Na Tabela 3 é exibido um melhor detalhamento desses dados de teste.

TABELA 3 - CASOS DE TESTE DO PARADIGMA IMPERATIVO

Caso de teste	**Caminho de teste**	**Dados de entrada**	**Saída esperada**
Caso 01	nó 1→nó 3→nó 4	n = 2	"Numero positivo"
Caso 02	nó 1→nó 2→nó 4	n = -1	"Numero não positivo"

Fonte: (Autoria Própria)

Portanto, nota-se que foi possível gerar casos de teste para unidades do POO utilizando critérios de teste vinculados a um paradigma diferente, neste caso, o imperativo. Mais uma vez demonstrando a flexibilidade da solução proposta.

5.2.3 Inserindo critério customizado no POO

Para inserir um critério de teste no POO, o desenvolvedor deve criar uma classe derivada de uma classe abstrata que possui métodos obrigatórios a serem implementados. Os métodos

referenciam como o GFC deve ser percorrido para que os caminhos de teste sejam gerados, além disso, nesses métodos também deve ser definida a anotação na qual irá representar esse critério. Esses métodos são semelhantes aos métodos utilizados nos critérios já inseridos na ferramenta, como, por exemplo, o critério intramétodo e o intermétodo.

Após a definição de uma nova *tag*, o novo critério passa a ser interpretado. O exemplo mostrado na Figura 28 ilustra a inserção de um novo critério no POO.

FIGURA 28 - INSERINDO CRITÉRIO CUSTOMIZADO NO POO

```
package Org.GCT.OrientadoAObjetos.Customizado;

@CustomizaCom(CoberturaDeFilhos. class)
public @interface CoberturaDeFilhos{
        //implementação de regras
}


public class CoberturaDeFilhos extends CriterioCustomPOO{
        @Override
        public void execucao(){
            //Implementação do critério
        }
}
```

Fonte: (Autoria Própria)

Na Figura 28 é mostrado que é criada uma classe chamada CoberturaDeFilhos com base em uma classe abstrata denominada CriterioCustomPOO que define métodos obrigatórios a serem implementados para que o critério seja considerado. Com isso, é vinculada a essa classe a tag @CoberturaDeFilhos que faz referência ao critério customizado. Essa vinculação é realizada por meio da tag @CustomizaCom que está presente no módulo Plataforma, seguindo a mesma ideia utilizada no framework JUnit5.

5.2.4 Inserindo critério customizado do paradigma imperativo

Para inserir um critério de teste vinculado ao paradigma imperativo de modo a ser aplicado em POO, o desenvolvedor também deve criar uma classe derivada de uma classe abstrata que possui métodos obrigatórios as serem implementados. Os métodos referenciam como o GFC deve ser percorrido para que os caminhos de teste sejam gerados. Além disso, nesses métodos também deve ser definida a anotação na qual irá representar esse critério. Esses métodos são semelhantes aos métodos utilizados nos critérios já inseridos na ferramenta, como por exemplo, o critério intramétodo e o intermétodo.

Após a definição de uma nova *tag*, o novo critério passa a ser interpretado. O exemplo mostrado na Figura 29 ilustra a inserção de um novo critério do paradigma imperativo.

FIGURA 29 - INSERINDO CRITERIO CUSTOMIZADO DO PARADIGMA IMPERATIVO

```
package Org.GCT.ParadigmaImperativo.Customizado;

@CustomizaCom(TodosUsos. class)
public @interface TodosUsos{
        //implementação de regras
}


public class TodosUsos extends CriterioCustomPI{
        @Override
        public void execucao(){
            //Implementação do critério
        }
}
```

Fonte: (Autoria Própria)

Na Figura 29 é mostrado que é criada uma classe chamada TodosUsos com base em uma classe abstrata denominada CriterioCustomPI que define métodos obrigatórios a serem implementados. Com isso, é vinculada a essa classe a tag @TodosUsos que faz referência ao critério customizado. Essa vinculação é realizada por meio da tag @CustomizaCom que está presente no módulo Plataforma.

Portanto, percebe-se que é simples a inserção de um novo critério de teste nos paradigmas de teste dispostos na ferramenta. Com isso, é possível abordar que a arquitetura proposta suporta os requisitos definidos na Subseções 4.2.1 e 4.2.2. Para atender aos outros requisitos descritos nas Subseções 4.2.3, 4.2.4 e 4.2.5, ideias semelhantes as das subseções 5.2.3 e 5.2.4 são utilizadas.

6. CONCLUSÃO

Projetar a arquitetura de um software é uma tarefa altamente complexa, pois a arquitetura desenvolvida servirá como base para que o software seja construído. Portanto, projetar com pouca qualidade acarretará problemas no produto final e isso possibilitará insatisfação por parte de um possível usuário. A definição de requisitos em conjunto com o atributo de qualidade almejado na ferramenta foi essencial na aplicação da técnica ADD para desenvolver todos os detalhes desse projeto arquitetural.

A revisão bibliográfica e a pesquisa por trabalhos relacionados contribuíram para concluir que existem poucas ferramentas capazes de desenvolverem o teste de software com base em distintos paradigmas de programação, além de não oferecerem um alto suporte a extensão de suas funcionalidades. A aplicação da técnica ADD serviu para definir módulos funcionais e por meio da linguagem UML foi possível representar as dependências entre esses módulos. A aplicação da técnica de avaliação de arquitetura possibilitou uma investigação por cenários de conflitos na arquitetura.

A arquitetura proposta neste trabalho é adequada a desenvolvedores que necessitam construir ferramentas de teste de software que possibilitem modificações ou adequações de acordo com a necessidade dos testes. Portanto, pode-se dizer que o fato da arquitetura possibilitar a modificabilidade facilita a sua adequação em vários cenários de teste.

A ferramenta busca oferecer suporte a vários tipos de teste por meio de módulos que permitem a sua extensão e adequação de acordo com as necessidades dos desenvolvedores de software.

Para trabalhos futuros, pretende-se desenvolver uma ferramenta com base na arquitetura proposta no trabalho, visto que o nível de detalhes propõe uma base suficiente para a sua construção.

REFERÊNCIAS

ACKERMAN, R. et al. **Metacognition and system usability: Incorporating metacognitive research paradigm into usability testing. Computers In Human Behavior**, [s.l.], v. 54, p.101-113, jan. 2016. Elsevier BV.

ALIMUCAJ, A. **Control Flow Graph Generator: Documentation.** Hochschule Konstanz Technik, Wirstchaft Und Gestaltung. Junho 2009.

BASS, L.; CLEMENTS, P.; KAZMAN, R. **Software Architecture in Practice**. Addison-Wesley Professional, ed. 2,. 2003.

BECHTOLD, Stefan et al. **JUnit 5 User Guide.** : Junit.org, 2018. 95 p.

BERNARDO, P. C.; KON, F. **A Importância dos Testes Automatizados.** Artigo publicado na Engenharia de Software Magazine, 2008.

CARVALHO, Maira B. et al. **A case study on Service-Oriented Architecture for Serious Games**. Entertainment Computing**,** [s.l.], v. 6, p.1-10, jan. 2015. Elsevier BV.

CLARO, D. B., SOBRAL, J. B. M. **Programação em Java**. Copyleft Pearson Education. Florianópolis, SC. 2008.

DELAMARO, M. E., MALDONADO, J. C., MARIO, J. **Introdução ao Teste de Software.** 2a Edição. Elsevier, 2016.

FAISAL, Muhammad; NAZIR, Shoaib; BABAR, Muhammad. **Notational Modeling for Model Driven Cloud Computing using Unified Modeling Language. Icst Transactions On Scalable Information Systems**, [s.l.], v. 5, n. 16, p.154475-7, 13 abr. 2018. European Alliance for Innovation no.

FRAME, Scott; COFFEY, John W.. **A Comparison of Functional and Imperative Programming Techniques for Mathematical Software Development.** Journal Of

Systemics, Cybernetics And Informatics. Pensacola, Fl, p. 49-53. dez. 2014.

GALLARDO, D. **Desenvolvendo Plug-ins do Eclipse:** Como criar, depurar e instalar seu plug-in. IBM. Dez. 2002.

GALLAGHOR, L., OFFUTT, J., CINCOTTA, A.: I**ntegration testing of object-oriented componentes using finite state machines**. Research articles. Softw. Teste. Verif. Reliab. 16(4), 215-266 2006.

GARCIA, Boni. **Mastering Software Testing with JUnit 5:** A comprehensive, hands-on guide on unit testing framework for Java programming language. Mumbai: Packt Publishing Ltd., 2017

HARROLD, M. J. e ROTHERMEL, G. **Performing data flow testing on classes.** In: *II ACM SIGSOFT Symposium on Foundations of Software Engineering*, p. 154-163, Nova York, NY, EUA, dez. 1994. ACM Press.

LEAL, Igor Gonçalves. **Requisitos de Metodologias de Teste de Software para Processos Ágeis.** Artigo, Universidade Federal de Minas Gerais, Belo Horizonte, MG, 2009.

INTHURN, Cândida. **Qualidade & Teste de Software.** Florianópolis: Visual Books, 2001.

KAPOOR, Akshay; DHAVALE, Sunita. **Control Flow Graph Based Multiclass Malware Detection Using Bi-normal Separation.** Defence Science Journal, [s.l.], v. 66, n. 2, p.138-146, 23 mar. 2016. Defence Scientific Information and Documentation Centre. http://dx.doi.org/10.14429/dsj.66.9701

KOREL, B. **Automated Software Test Data Generation.** IEEE Transactions on software engineering , vol. 16, no. 8, august 1990.

MANRAL, Jai. **Automated Test Case Generation using Petri Nets**. Cornell University Library, Bangalore, India, p.1-11, 28 set. 2015.

PANICHELLA, Annibale; KIFETEW, Fitsum Meshesha; TONELLA, Paolo. **Automated**

Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets. Ieee Transactions On Software Engineering, [s.l.], v. 44, n. 2, p.122-158, 1 fev. 2018. Institute of Electrical and Electronics Engineers (IEEE).

PRESSMAM, R. S. **Engenharia de software**: Uma abordagem professional. Tradução de Ariovaldo Griese. 7. ed. [S.l.]: AMGH Editora Ltda., 2011. 934 p.

RAPPS, S. e WEYUKER, E. J. **Selecting Software Test Data Using Data Flow Information.** IEEE Transactions on software engineering , vol. se-1l, no. 4, april 1985.

SOBRINHO, A. et al. **Design and evaluation of a mobile application to assist the self-monitoring of the chronic kidney disease in developing countries. Bmc Medical Informatics And Decision Making**, [s.l.], v. 18, n. 1, p.1-14, 12 jan. 2018. Springer Nature.

SOMMERVILLE, Ian. **Engenharia de software**. 9. ed. São Paulo, SP: Pearson Prentice Hall, 2011. 529 p.

SON, Hyun Seung; PARK, Young B.; KIM, R. Young Chul. **MCCFG: an MOF-based multiple condition control flow graph for automatic test case generation.** Cluster Computing**,** [s.l.], p.1-10, 9 dez. 2016. Springer Nature. http://dx.doi.org/10.1007/s10586-016-0687-1.

VIRIUS, Miroslav; PECINOVSKÝ, Rudolf. **Object oriented computing. Wiley Interdisciplinary Reviews**: Computational Statistics, [s.l.], v. 6, n. 3, p.164-176, 20 mar. 2014.

VISWANATHAN, Sunitha Edacheril; SAMUEL, Philip. **Automatic code generation using unified modeling language activity and sequence models. Iet Software,** [s.l.], p.164-172, 11 jul. 2016. Institution of Engineering and Technology (IET).

YIN, Yufei. **An Automated Test Case Generation Approach based on Activity Diagrams of SysML.** International Journal Of Performability Engineering**,** [s.l.], 2017. Totem Publisher, Inc.. http://dx.doi.org/10.23940/ijpe.17.06.p13.922936.