FERNANDO DA CRUZ PINHEIRO

FERRAMENTA PARA EXECUÇÃO DE CASOS DE TESTE EM MODELO UML APLICADA EM SISTEMAS EMBARCADOS

JOINVILLE - SC

2012

UNIVERSIDADE DO ESTADO DE SANTA CATARINA – UDESC
CENTRO DE CIÊNCIAS TECNOLÓGICAS – CCT
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO – DCC

FERNANDO DA CRUZ PINHEIRO

FERRAMENTA PARA EXECUÇÃO DE CASOS DE TESTE EM MODELO UML APLICADA EM SISTEMAS EMBARCADOS

Trabalho de conclusão de curso (TCC) apresentado ao curso de graduação em Ciência da Computação, da Universidade do Estado de Santa Catarina (UDESC), como requisito parcial da disciplina de Trabalho de Conclusão de Curso.

Orientador: Marco Aurélio Wehrmeister

JOINVILLE – SC

2012

FERNANDO DA CRUZ PINHEIRO

FERRAMENTA PARA EXECUÇÃO DE CASOS DE TESTE EM MODELO UML APLICADA EM SISTEMAS EMBARCADOS

Trabalho aprovado como requisito parcial para obtenção do grau de bacharel, no curso de graduação em Ciência da Computação da Universidade do Estado de Santa Catarina UDESC.

Banca Examinadora:

Orientador: _____
Marco Aurélio Wehrmeister, Doutor.
Universidade do Estado de Santa Catarina UDESC

Membro: _____
Carla Diacui Medeiros Berkenbrock, Doutora.
Universidade do Estado de Santa Catarina UDESC

Membro: _____
Rui Jorge Tramontin Junior, Doutor.
Universidade do Estado de Santa Catarina UDESC

JOINVILLE - SC

2012

SUMÁRIO

RESUMO

Devido à necessidade de *softwares* com boa qualidade e baixo custo, técnicas de teste de *software* foram desenvolvidas com a finalidade de corrigir os erros pertinentes ao sistema, facilitando o seu desenvolvimento. Com o objetivo de encontrar falhas no projeto o mais cedo possível, este trabalho apresenta uma proposta de ferramenta de automação de testes funcionais executado em um diagrama de sequência direcionado para sistemas embarcados. O sistema em teste é inicialmente representado em um modelo UML onde é transformado em um modelo intermediário chamado DERCS. Este meta modelo contêm as informações como os requisitos funcionais e não funcionais do sistema testado. Estas informações são armazenadas em uma estrutura de dados, assim, possibilitando realizar testes comportamentais do sistema. Os casos de testes e a especificação dos resultados são definidos com o auxílio de arquivos XML. Essa ferramenta apresenta resultados de forma clara e objetiva, apontando erros da especificação da aplicação, caso eles existam.

PALAVRAS-CHAVE: teste de software, automação de testes funcionais, modelo UML, sistemas embarcados.

ABSTRACT

Due to the need for high productivity, good quality and low cost, new testing techniques have been developed and used during the whole software development. In order to find errors as soon as possible in the project , this work proposes a tool for test automation of functional requirements performed on UML models for embedded systems. The tested system is represented by an UML model which is transformed into a model called DERCS. This model provides informations on the functional and non-functional of the tested system. All information is stored into a data structure giving the possibility to perform behavioral tests on the system. The test cases and also the testing result are defined as XML files. This tool presents results clearly and objectively pointing to errors in specification of the application, if there are any.

KEY-WORDS: *software testing, automated functional testing, UML model, embedded systems.*

LISTA DE ILUSTRAÇÕES

LISTA DE QUADROS

1. INTRODUÇÃO E JUSTIFICATIVA

Com aumento da velocidade de desenvolvimento dos produtos de *software*, manter um padrão de qualidade do produto final acaba sendo um desafio para os projetistas. Para que o processo de desenvolvimento pudesse acompanhar e manter uma alta qualidade com baixo custo, foram estabelecidos várias técnicas, critérios, métodos e ferramentas de engenharia de *software*. Apesar dessas verificações serem aplicadas no desenvolvimento de um determinado projeto, falhas no produto final acabam ocorrendo, podendo causar problemas financeiros, de tempo e degradar a imagem da empresa que produz este *software* (SYLLABUS, 2007). Define-se falha de *software* como um comportamento diferente do esperado pelo usuário (CLÁUDIO, 2007). O custo de correção de um erro no *software* pode se tornar de 60 a 100 vezes maior dependendo do momento em que for constatado, isto é, quanto mais tarde o erro for revelado, mais elevado o custo e o esforço para consertá-lo (PRESSMAN, 2002). Define-se erro de *software* como um resultado inesperado na execução do programa (CLÁUDIO, 2007).

Com vistas a auxiliar no desenvolvimento do sistema, várias técnicas para testar e verificar se o modelo está livre de falhas foram desenvolvidas. Para encontrar erros, o processo de teste de *software* é uma das principais atividades a ser aplicada no projeto do *software*. Os testes de *software* são feitos de maneira a atender todas as funcionalidades, restrições e condições que o programa possui. Este processo deve ser aplicado desde o início do projeto, para que o custo de correção dos erros seja reduzido, evitando que se prolonguem por todas as fases do desenvolvimento (BARBOSA, 2003).

Das técnicas de testes existentes, algumas são baseadas em testes de funcionalidades (caixa preta), onde é avaliado o comportamento “externo” do *software*, verificando se as funcionalidades estão corretas de acordo com a especificação. Outras usam testes estruturais (caixa branca) que, com base no código fonte, avaliam os componentes internos de um programa (AMEIDA, 2010).

Neste trabalho, é abordado o teste estrutural, onde, com base em um modelo são especificados os casos de testes, buscando evitar erros no momento da geração do código. Exemplos de erros são: a especificação pode estar errada ou incompleta, um equívoco na definição de seus atributos, erros de entrada e saída, entre outros (TOZELLI, 2008).

Visando aplicar a ferramenta para execução de casos de testes, propõe-se usar o modelo UML para representar o sistema. O modelo UML é transformado em outro meta modelo independente de plataforma, chamado DERCS (*DIstributed Embedded Real-time Compact Specification*) (Wehrmeister, 2009). Este meta modelo especifica comportamentos e faz tratamento de alguns requisitos não-funcionais e estruturais, visando o uso em sistemas embarcados. A partir desse meta modelo os casos de teste são executados com a finalidade de detectar erros comportamentais do *software*.

Este trabalho consiste em desenvolver uma ferramenta em linguagem *Java* para automatizar a execução de casos de teste sobre o modelo DERCS que representa as funcionalidades de um *software* embarcado.

A ferramenta desenvolvida neste trabalho concentra-se em aplicar testes para sistemas embarcados, porém pode-se aplica-la para outros sistemas desde que as mesmas restrições de projeto sejam aplicadas.

Os casos de teste são definidos conforme os diagramas de sequência que representam os comportamentos do sistema embarcado. Com as informações dos casos de teste, o modelo DERCS é simulado com o auxílio de um *framework* chamado FUMBeS (Wehrmeister, 2011). Por fim, será gerado um relatório contendo informações dos testes realizados.

A representação dos casos de testes do sistema é especificada através de um arquivo no formato XML. Esse modelo é constituído por vários casos de testes, onde cada um é composto por um cenário inicial, os métodos testados (cada um com seu parâmetro inicial) e a afirmação do resultado.

Entende-se por cenário inicial os objetos a serem tratados e seus respectivos atributos definidos, podendo-se também inicializar esses atributos no decorrer da aplicação do teste com o uso de *scripts*. Os métodos são compostos pelos seus valores

de entrada e o cenário esperado. Entende-se por cenário esperado, o resultado do estado dos objetos que o testador espera ter após a execução dos testes. Assim, junto com o cenário resultante, o testador tem a possibilidade de efetuar a comparação (no relatório de teste) entre os cenários para definir se há defeitos no sistema. Os parâmetros de entrada dos métodos também serão criados dinamicamente no programa. Com o retorno do valor de cada método, pode-se fazer a comparação com o valor esperado, possibilitando saber se houve alguma mudança em algum elemento do método.

Após a execução dos testes realizados, o resultado de cada um dos casos de testes é descrito em um arquivo em formato XML. Esse relatório contêm os casos de testes que são compostos, de maneira geral, por um cenário inicial contendo todos os seus objetos e atributos que foram definidos, o cenário esperado, que contém o estado que os objetos terão depois do teste para poder fazer a comparação e, por último, estabelece o cenário do resultado, que representa o cenário após a execução dos testes de casos.

1.1 Objetivos

O objetivo deste trabalho consiste em implementar uma ferramenta para automatizar a execução de casos de teste sobre um modelo UML direcionado para sistemas embarcados.

1.1.1 Objetivos Específicos

Com base no objetivo apresentado acima, podem ser identificados os seguintes objetivos específicos:

- Especificar os casos de teste conforme o conceito de cenários de teste;
- Implementar uma ferramenta para aplicação dos casos de teste, onde foi definida em três etapas:
    - Ler os casos de teste em um arquivo de entrada no formato XML;

- Executar os casos de teste; e
-Gerar resultados em um arquivo de saída no formato XML.

1.2 Metodologia

Para atingir os objetivos específicos, primeiramente foram estudados os conceitos sobre tipos e técnicas de testes de *software*, com o objetivo de adquirir um embasamento adequado para a condução do restante do trabalho.

Após essa etapa, foi proposto um modelo dos casos de teste, que está representado em um diagrama de classe, onde foram definidos o cenário inicial, o resultado obtido, bem como o cenário inicial e o esperado.

O próximo passo foi criar o modelo do resultado obtido da execução dos casos de testes, onde são definidos o cenário inicial, o resultado obtido, o cenário esperado, bem como os objetos com seus respectivos atributos após a aplicação do teste.

Na sequência, foi realizada a implementação em *Java* dos modelos que definem os testes de casos, a execução dos testes e o resultado obtido pelos testes efetuados, podendo-se avaliar deste modo se há algum erro de implementação no sistema testado.

Por fim, para verificar se o resultado obtido atende aos objetivos previamente definidos, realizou-se a avaliação da ferramenta criada através da execução de estudos de caso. Servindo como estudo de caso, foi testado um sistema representado em um modelo UML, retirado de Wehrmeister (2008b).

1.3 Estrutura do trabalho

Este trabalho está dividido da seguinte forma:

O segundo capítulo apresenta um levantamento bibliográfico. Nele são abordados os conceitos que estão associados ao estudo do teste de *software,* tais como: os tipos de teste de *software*, suas técnicas, os níveis de teste de *software*, modelo UML e a definição de casos de teste.

O terceiro capítulo é dedicado ao estudo de trabalhos correlatos com o tema proposto, onde são levantadas vantagens e desvantagens do seu uso, uma análise do resultado e uma comparação sobre o que está sendo reusado em relação ao trabalho em desenvolvimento.

No quarto capítulo tem como objetivo focar o funcionamento do projeto, onde são descritos as etapas para a realização do projeto e é apresentado um diagrama de classe para demonstrar a estrutura da aplicação a ser implementada.

O quinto capítulo apresenta detalhes da ferramenta desenvolvida, destacando as tecnologias empregadas, Casos de Uso, Diagrama de Classes e características da implementação.

O sexto capítulo exibe os resultados gerados pela ferramenta desenvolvida, bem como apresenta a avaliação e as conquistas atingidas com o desenvolvimento do presente trabalho.

Por fim, tem-se a conclusão do que foi alcançado com o desenvolvimento do trabalho, apresentando as vantagens e desvantagens encontradas, bem como trabalhos futuros que podem dar continuidade ao tema abordado.

2. FUNDAMENTAÇÃO TEÓRICA

Este capítulo descreve a fundamentação teórica a respeito do trabalho desenvolvido, bem como técnicas de teste, além dos demais conceitos básicos para o entendimento completo do problema de pesquisa e de sua estratégia de solução.

Diante disso, o ponto de partida do trabalho consiste no estudo acerca da definição de teste de *software*, tornando claro os conceitos, fases, finalidade e a apresentação de um modelo de processo de testes de *software*.

2.1 Testes de *Software*

O teste de *software* é um processo que ocorre durante todo o desenvolvimento de um projeto. Executado pelo testador de *software*, responsável por algumas ações, tais como levantamento de requisitos e a execução do teste (TOZELLI, 2008). É um procedimento que tem como objetivo de revelar defeitos em um *software*, ajudando a identificar a exatidão, a integridade, a segurança e a qualidade do sistema (DIAS *et al.*, 2007). As falhas podem ser originadas por vários motivos, tais como: a especificação pode estar errada ou incompleta, defeitos no código, o algoritmo pode estar implementado de forma errada ou incompleta, entre outros.

Conforme descrito em Sommerville (2007), o processo de teste de *software* têm duas metas distintas:

1. Demonstrar ao desenvolvedor e ao cliente que o software atende aos requisitos. Envolve realizar testes em todos os requisitos de um sistema a pedido de um cliente ou para um desenvolvedor, podendo usar o *software* para desenvolver outro projeto.

2. Descobrir falhas ou defeitos no software que apresentam comportamento incorreto, não desejável ou em não conformidade com sua especificação. Esta etapa consiste em fazer a detecção de alguns comportamentos errados de um sistema, sendo esse o objetivo final deste trabalho em desenvolvimento.

Sommerville (2007), conforme apresentado na Figura 1, um modelo geral do processo de testes de *software* revelando os passos e os elementos essenciais para o teste.

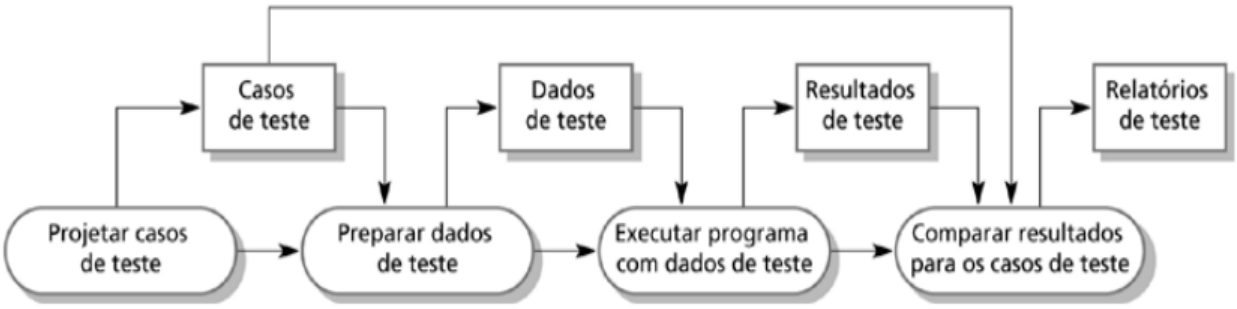

Figura 1 – Modelo de processo de testes de *software*.
Fonte: Somerville, (2007).

Para o início do teste de *software,* é preciso fazer uma especificação de entrada que define os casos de teste. A seguir, é feita a preparação de dados de entrada para ocorrer a execução do teste com base nestes dados. Após, é retornado um valor para poder fazer uma comparação do resultado conforme especificado na saída esperada. Por fim, é gerado um relatório para demonstrar o resultado do teste. A saída esperada (ou resultado esperado) só pode ser feita por pessoas que compreendem o que o sistema deve fazer.

2.2 Casos de teste

O caso de teste é basicamente um caso de uso que estará em teste. Um caso de uso é um comportamento de um sistema que foi requisitado pelo usuário no início do projeto. Os casos de teste mais escolhidos para o teste são aqueles que representarão maior relevância em relação ao custo, o risco e a necessidade de verificá-lo para o sistema (RUP - *Rational Unified Process, 2001*).

Na definição de um caso de teste devem-se especificar várias informações relevantes para fazer a execução do teste, tais como: as pré-condições, informações sobre as entradas, resultado esperado, etc. Neste trabalho são definidas algumas

informações precisas para que esse processo de automação seja executado corretamente. Entre as mais importantes, cita-se: o estado inicial, parâmetros de entrada, resultado esperado e os métodos a serem testados.

2.3 Automação de Teste de *Software*

A automação de teste de *software* é um processo pouco utilizado por desenvolvedores, pelo fato do tempo a ser gasto para desenvolver uma ferramenta com essa finalidade. Basicamente, a automatização de teste serve para fazer um *software* capaz de executar casos de teste automaticamente, com finalidade de ter mais confiabilidade e agilidade na execução dos casos de teste (SANTOS, 2011).

Entre várias atividades da automatização de teste de *software*, as mais importantes são: planejamento e controle, escolha das condições de teste, modelagem dos casos de teste, checagem dos resultados, avaliação de conclusão dos testes, geração de relatórios, como também a revisão dos documentos (ALMEIDA, 2010).

Essas atividades estão conciliadas na atividade de executar programa com dados de teste descrito anteriormente na seção 2.1 e demonstrado na Figura 1.

2.4 Técnicas de Teste de *Software*

Softwares podem ser testados de várias maneiras. Para isso, foram definidas algumas técnicas para determinar o que está sendo testado em um sistema. Dentre as mais importantes estão o teste de caixa-preta e o teste de caixa-branca.

2.4.1 Teste de Caixa-Preta

O teste de caixa-preta, também chamado de teste funcional, faz a avaliação externa do componente de um sistema, sem ter conhecimento da sua estrutura interna (implementação), isto é, tem-se apenas conhecimento da interface. Normalmente, este tipo de teste, testa as especificações de entrada e saída, as pré e pós condições dos

comportamentos dos componentes do sistema, entre outras especificações requisitadas (DIAS *et al*., 2007).

2.4.2 Teste de Caixa-Branca

O teste de caixa-branca, também chamado de teste estrutural, é uma técnica de teste que acontece com o conhecimento da implementação da aplicação. Ela tende a revelar defeitos que ocorrem no programa conforme a implementação da aplicação, isto é, determina o funcionamento correto do programa a partir da sua estrutura interna (ALMEIDA, 2010).

Neste trabalho é abordada a técnica de caixa-branca, onde o testador precisa ter conhecimento da composição da estrutura interna do *software* para executar os testes, assim podendo determinar se os elementos do sistema estão executando de forma correta (DIAS *et al.*, 2007).

2.5 Níveis de Teste de *Software*

Os testes de *softwares* podem ser executados em vários níveis, podendo-se assim analisar o que vai abranger o sistema a ser testado. Conforme o nível pode se determinar alguns aspectos como: seus objetivos genéricos, o que terá como base de teste, o que está sendo testado, etc. A Figura 2 ilustra níveis de teste de *software*. Os níveis de teste estão divididos em quatro níveis: teste de unidade, teste de integração, teste de sistema e teste de aceitação (SYLLABUS, 2007).

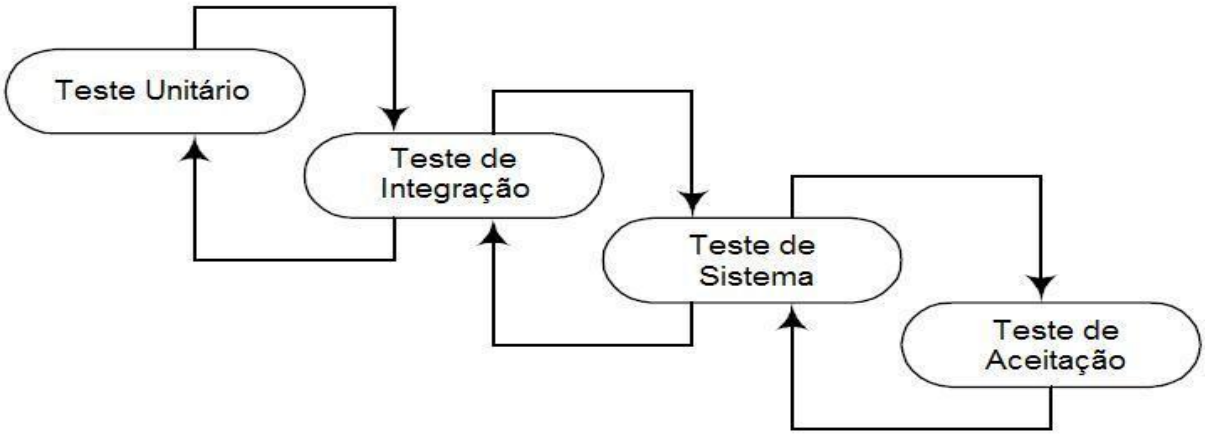

Figura 2 - Níveis de teste de *software.*
Fonte: Barbosa (2007).

2.5.1 Teste de Unidade

No nível do teste de unidade, cada unidade de código do sistema será testada. O termo unidade de código é o menor pedaço de um *software*, isto é, pode corresponder como um método ou uma classe correspondendo a um *software,* usando uma linguagem orientada a objeto (BARBOSA, 2007). Este tipo de teste foi abordado neste trabalho, pois os testes a serem executados serão aplicados em métodos, que correspondem a unidades do sistema.

2.5.2 Teste de Integração

O teste de integração ocorre depois que os componentes do teste de unidade se concretizam até formarem um módulo ou um subsistema. Mesmo que as unidades fossem testadas criteriosamente, não se pode garantir que na sua interação elas irão se comportar corretamente (BURNSTEIN, 2003). Após a formação desse módulo ocorre o teste para avaliar a interação entre essas unidades, que é feito por desenvolvedores que focam na colaboração de um conjunto de unidades (subsistema) (RAINSBERGER, 2005). Nesse teste se utiliza a técnica da caixa-branca.

2.5.3 Teste de Sistema

Após a realização do teste de unidade e o teste de integração, é feito o teste de sistema, onde uma equipe de desenvolvedores desconhece a estrutura interna do sistema e o teste é feito somente através da interface da aplicação (caixa-preta). Assim, são testados os comportamentos do sistema para avaliar se todas as especificações feitas pelo usuário estão ocorrendo de forma correta (BARBOSA, 2007).

2.5.4 Teste de Aceitação

Neste nível o teste é feito quando todo o sistema estiver completo, sendo feito pelo próprio usuário que está em uso, assim determinando se está faltando alguma coisa ou se é necessário algum complemento não especificado inicialmente, com objetivo de satisfazer as necessidades do cliente (GLENFORD, 2004).

2.6 Tipos de Teste de *Software*

Conforme classificado em RUP (2001), os testes de *software* podem ser classificado por vários tipos, onde cada tipo tem um objetivo e uma técnica de suporte específicos. Assim, cada tipo terá um foco, características ou atributos do sistema para testar. Conforme ilustrado na Figura 3, o RUP classifica teste sobre cinco fatores para a qualidade do modelo, e para cada um desses fatores existem os tipos de teste associados (BARBOSA, 2007).

Dimensões de Qualidade	Tipos de teste
Funcionalidade	Teste Funcional Teste de Segurança Teste de Volume
Usabilidade	Teste de Usabilidade
Confiabilidade	Teste de Integridade Teste de Estrutura Teste de Stress
Desempenho	Teste de Avaliação de Desempenho Teste de Contenção Teste de Carga Teste de Perfil de Desempenho
Suportabilidade	Teste de Configuração Teste de Instalação

Figura 3 – Tipos de teste de *softwares*.

Fonte: Barbosa (2007).

2.6.1 Funcionalidade

Os tipos de teste relacionados com a dimensão de qualidade do tipo *funcionalidade* estão associados aos testes onde sua principal função é ajudar a garantir que não haja erro do sistema em relação à especificação. O teste mais importante dessa dimensão é o teste de funcionalidade, que testa o comportamento do programa. Este teste foi efetuado neste trabalho onde, através das chamadas de funções, testes de caixa-branca foram realizados a fim de comparar se o resultado dessa função é o resultado esperado.

Entre outros está o teste de segurança, que consiste em garantir a segurança de dados do sistema e o teste de volume que testa a capacidade do sistema lidar com grandes volumes de dados (BARBOSA, 2007).

2.6.2 Usabilidade

O teste de usabilidade preocupa-se em definir o quanto o sistema é fácil de usar. Entre suas características está enfatizar a estética, assistentes e agentes, consistência na interface do usuário, fatores humanos e materiais de treinamento (RUP, 2001).

2.6.3 Confiabilidade

No teste de confiabilidade, ocorrem os testes internos do sistema, validando os resultados de entradas, saídas e operações conforme os requisitos pré definidos para a aplicação.

Há outros testes de confiabilidade no seu entorno, tais como: o teste de integridade que tem a finalidade de testar a resistência do *software* a falhas e compatibilidades, e o teste de *stress*, que tem como objetivo testar o *software* em condições anormais como, por exemplo, recursos compartilhados limitados (BARBOSA, 2007).

2.6.4 Desempenho

Os tipos de teste relacionados com a dimensão de qualidade de desempenho têm o objetivo de avaliar o desempenho de algumas características que o sistema poderia ter, podendo analisar o desempenho do fluxo de execução, os tempos de resposta de uma requisição, entre outros (RUP, 2001). O teste de avaliação de desempenho testa o *software* mediante uma carga de trabalho já conhecida.

O teste de contenção tem o objetivo de avaliar se o *software* pode trabalhar com vários usuários usando um mesmo recurso. O teste de carga avalia o limite operacional de um sistema e o perfil de desempenho faz análise sobre o uso do sistema com finalidade de identificar possíveis gargalos ou processos ineficientes (BARBOSA, 2007).

2.6.5 Suportabilidade

Este teste analisa a suportabilidade do sistema em relação a suas configurações e instalações. Entre esses testes está o teste de configuração, que tem o propósito de garantir que o *software* funcione em diversas configurações possíveis associados tanto ao *hardware* quanto ao *software*. O teste de instalação tenta garantir com que o *software* seja instalado em condições anormais, como o espaço insuficiente no disco ou uma interrupção de energia (BARBOSA, 2007).

3.TRABALHOS CORRELATOS

A automatização de casos de testes é um tema com uma quantidade crescente de pesquisa. Neste entorno foram verificados alguns trabalhos e projetos que foram desenvolvidos para criar estratégias para fazer uma ferramenta de casos de teste eficiente.

3.1 Derivações de casos de testes funcionais: uma abordagem baseada em modelos UML

Conforme especificado anteriormente, antes de aplicar os testes devemos definir os casos de teste do sistema, para depois então executar os testes. Com o mesmo propósito de realizar testes funcionais nas fases iniciais do processo de desenvolvimento, o trabalho desenvolvido por Orozco *et*. al., (2009) apresenta uma abordagem de derivação automática dos casos de teste de *software* baseada em modelos UML, objetivando a derivação automatizada completa ou parcial dos casos de testes.

Foi utilizado um diagrama de atividade com alguns estereótipos compostos por marcações caracterizando as ações e alguns elementos para permitir a derivação dos casos de teste (Orozco *et. al*, 2009). Isso difere do que é feito neste trabalho, pois os elementos em questão serão especificados em um diagrama de sequência. A Figura 4 exibe uma visão das atividades seguidas para ocorrer a derivação dos casos de teste.

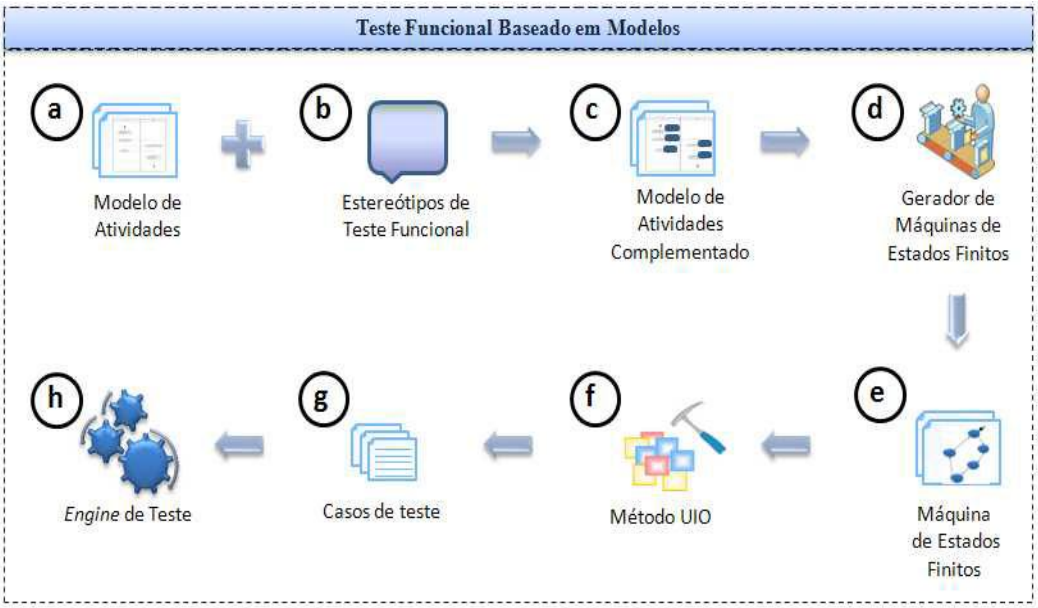

Figura 4 – Uma visão da abordagem proposta.
Fonte: Orozco et.*al*, (2009).

Com o modelo de atividade definido (a), é inserido um estereótipo em cada atividade (b), que terá como função especificar as ações realizadas no sistema e o resultado esperado dessa ação.

Após essas informações serem complementadas no diagrama de atividade (c), uma máquina de estado finito é gerada através desse diagrama, (d) que deverá representar todos os caminhos possíveis do sistema. Nesta máquina de estados finitos (e), será utilizada uma ferramenta (f) chamada método UIO (*Unique Input/Output*), onde a partir dela vai ser gerado um conjunto de sequências de entrada e saída para esta máquina. Cada uma dessas sequências será um caso de teste(g), para depois então usar uma ferramenta de execução de teste (h) a partir desses casos de teste gerado.

A Figura 5 mostra a máquina de estados finitos e as transições descrevendo cada caminho possível de duas funcionalidades de uma calculadora, somar e dividir.

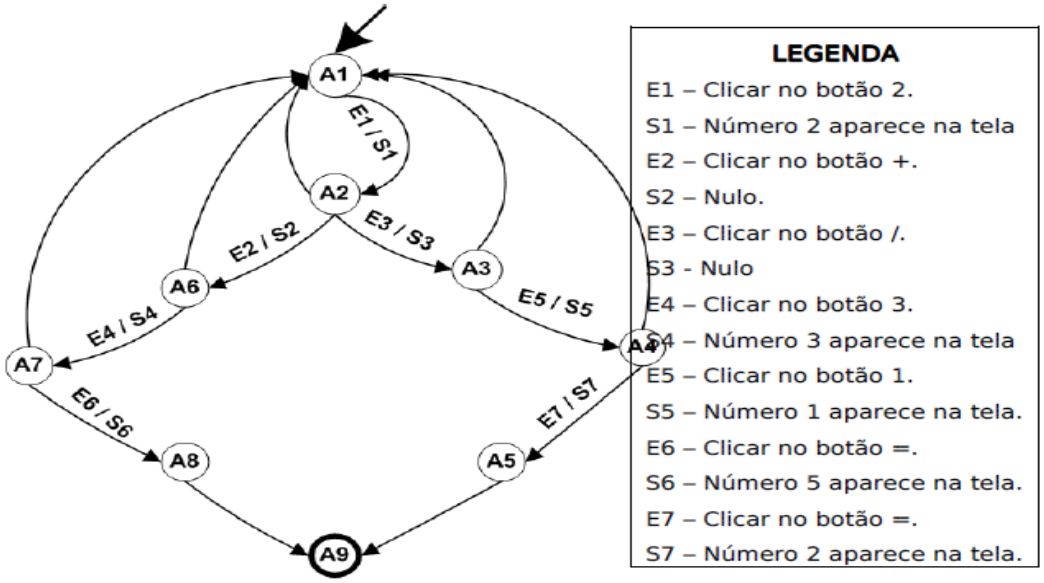

Figura 5 – Máquina de estado finitos de um sistema.

Fonte: Orozco *et*. al, (2009).

Após a aplicação da ferramenta método UIO, o resultado obtido apresentado será {(E1/S1), (E1/S1, E2/S2), (E1/S1, E2/S2, E4/S4), (E1/S1, E2/S2, E4/S4, E6/S6), (E1/S1, E3/S3), (E1/S1, E3/S3, E5/S5), (E1/S1, E3/S3, E5/S5, E7/S7)}, onde cada conjunto de elemento dentro dos parênteses representa um caso de teste. Assim, neste estudo foi determinado o valor inserido para realizar as operações de soma ou divisão, assim podendo fazer os testes de funcionalidade de cada caso de teste do sistema.

A vantagem desta abordagem é não precisar definir manualmente os casos de teste de um determinado sistema, assim reduzindo os esforços para a geração de testes, aumentando sua eficiência e evitando possíveis erros ao definir os casos de testes. Uma desvantagem encontrada no projeto é que, embora para um sistema simples seja eficiente, em sistemas complexos pode haver casos de teste gerados repetitivamente. Assim, é necessária uma composição dos diagramas de atividades para evitar com que se gerem casos de testes redundantes.

3.2 Testes de desempenho a partir de modelos UML para componentes de *software*

Além da automação da execução de casos de teste, existem várias técnicas para otimizar e avaliar a eficiência de um sistema. Dentre elas está o teste de desempenho, onde seu objetivo principal é quantificar e avaliar a eficácia do sistema.

O trabalho escrito por Dias *et. al*, (2008), descreve uma ferramenta para testar o desempenho de componente de *software*. Este trabalho foca no teste do DAO (*Data Access Object*, objeto de acesso de dados), componente que faz com que o sistema possa acessar e armazenar os seus dados. Este componente é um dos pontos críticos para o desempenho geral do sistema, pois qualquer erro pode corromper o banco de dados, sistemas de arquivos, entre outros sistemas de armazenamento.

O funcionamento do sistema está ilustrado na Figura 6, que mostra que os componentes do sistema são representados através de um modelo UML 2.0 especificado usando a ferramenta *Umbrello UML Modeller* (Modeller 2007). Neste trabalho, é preciso especificar e visualizar os requisitos (métricas) necessários para poder fazer o teste de desempenho do componente. Para isso é usada uma linguagem, chamada *UML SPT Profile (UML Profile for Schedulability,Performance and Time).* Através dela, os requisitos de desempenho são especificados, e esses dados estarão representados (descritos manualmente) em um arquivo XMI (XML *Metadata Interchange*).

A ferramenta tem como primeiro processo a leitura dos requisitos para o teste em um arquivo XMI. Para a realização do *parser* dos dados, é usada uma biblioteca chamada *SAX* (XML,2007). O armazenamento desses requisitos é alocado em uma estrutura de dados.

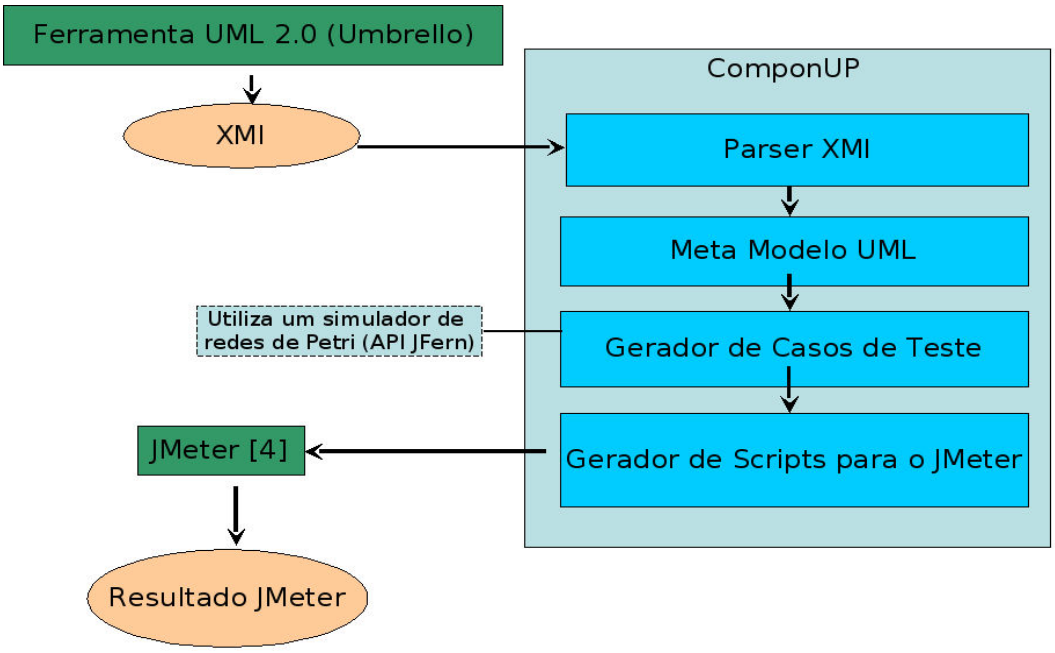

Figura 6 - Funcionamento do sistema.

Fonte: Dias *et. al*, (2008).

Logo após é feita uma análise sobre as informações coletadas, verificando se não está faltando nenhum requisito para o procedimento do teste ou se não há alguma inconsistência nos dados ou erros no arquivo XMI. Após, é criado um meta modelo que tem os pacotes e as classes necessárias executar o teste.

Em seguida, foi utilizado um algoritmo para criar uma Rede de Petri que representa os casos de uso desse sistema. Essa Rede de Petri é simulada com o auxílio de uma ferramenta chamada JFern (JFern 2006). Cada simulação dessa rede resulta em um conjunto de casos de teste (comportamento). Cada comportamento foi traduzido em um *script* para uma ferramenta chamado *Apache JMeter.* O resultado (tempos medidos) será mostrado em um *script* gerado em um arquivo de saída no formato XML.

Vários pontos positivos se destacam, tais como: a boa visualização do resultado do desempenho de cada requisição; casos de teste que são gerados; e a precisão em que as métricas foram definidas para análise de desempenho. Um ponto negativo do trabalho é em relação ao tempo para fazer o teste de desempenho, já que é preciso criar uma Rede de Petri manualmente (seguindo um algoritmo), onde para alguns

sistemas mais complexo leva muito tempo gerando incertezas sobre a precisão dessa rede criada.

Podemos destacar alguns pontos em comum deste trabalho com o projeto desenvolvido, entre eles: é a utilização de um diagrama de sequência; e o uso de uma ferramenta para simular o comportamento para retornar um resultado para comparação.

3.3 Um Método Automático de Teste Funcional para a Verificação de Componentes

O trabalho desenvolvido por Barbosa (2005) consiste no desenvolvimento de uma ferramenta que realiza a geração, execução e análise de resultados de testes funcionais para componentes de *softwares*. Para a realização do teste funcional é desenvolvido a ferramenta SPACES (*Specification Based Component Tester*). A arquitetura da ferramenta SPACES está representada na figura 7.

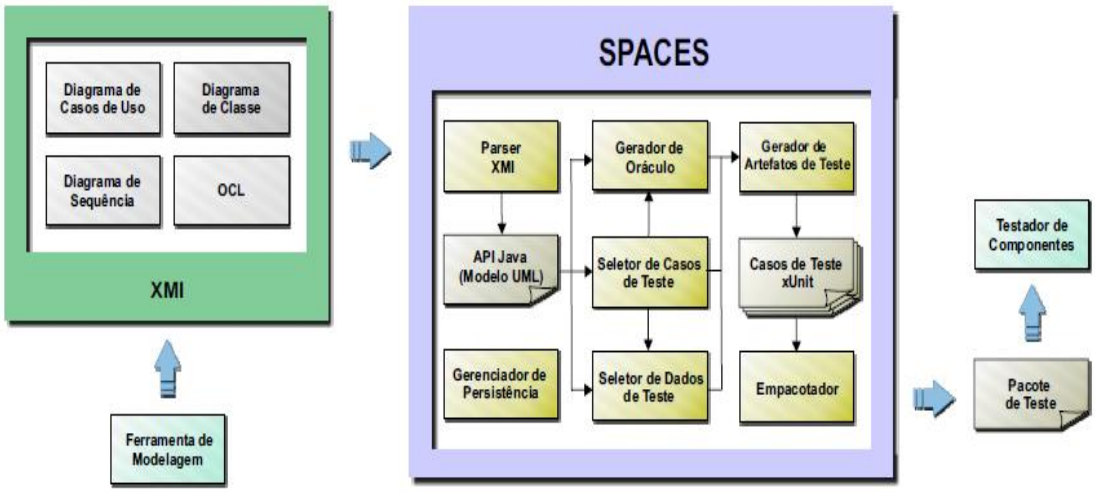

Figura 7: Arquitetura da Ferramenta SPACES.
Fonte: Barbosa (2005).

A ferramenta SPACES utiliza um arquivo de entrada do tipo XMI para importar as informações da especificação dos componentes. A estrutura destes é definida através do modelo UML e de restrições OCL (linguagem para definir restrições de componentes

de *software*), estas últimas usadas para definir as pré e pós-condições de cada funcionalidade do sistema, realizando a comparação de resultados.

A ferramenta SPACES foi desenvolvida para automatizar as atividades do método AFCT(*Automatic Functional Component Testing*). Com o uso do método AFCT, é realizada a derivação dos casos de teste, a geração e a execução de código de teste e a interpretação do resultado obtido. As atividades do método AFCT são representadas em 5 etapas: **planejamento de teste**, onde são tomadas decisões relativas ao processo de teste; **especificação de teste**, onde são derivados os casos de teste; **construção**, que é responsável pela geração do código de teste; **empacotamento**, onde o código de teste é compilado e empacotado para distribuição; **execução e análise de resultados**, que realiza a execução e análise de resultados dos códigos de teste gerados. A ferramenta SPACES ainda faz o uso da ferramenta *Junit* para realizar a execução dos casos de testes e fazer a análise dos resultados.

Essa ferramenta tem como ponto positivo fazer a re-execução dos casos de teste possibilitando que o testador possa avaliar várias vezes um mesmo cenário a partir de novos dados fornecidos e do uso de uma interface gráfica para a representação e a realização de todos os processos e funcionalidades da ferramenta SPACES. Um ponto negativo encontrado é o uso de vários diagramas UML para coletar as informações para a realização dos testes.

Pontos semelhantes entre o trabalho de Barbosa (2005) e o desenvolvido são a introdução de especificações comportamentais do sistema em teste para a ferramenta e o propósito de automatizar a execução dos casos de teste. Uma diferença encontrada foi a forma de especificar os casos de teste. Enquanto neste trabalho definem de forma manual os casos de teste, o trabalho de Barbosa (2005) deriva os casos de teste com o auxílio do método AFCT.

3.4 Estratégia de Automação em Testes: Requisitos, Arquitetura e Acompanhamento de sua Implantação

O trabalho de Costa (2004) apresenta uma abordagem que define um planejamento para o aperfeiçoamento de um projeto de automação de testes de softwares.

Nessa abordagem, Costa (2004) busca organizar diversos requisitos de automação apresentados em uma arquitetura para que os testes possam ter reusabilidade e a rastreabilidade dos artefatos de um teste, tais como: requisitos, casos de testes, execução e falhas. Dentro desta arquitetura são (requisitos relacionados às necessidades de automação). Dentre esses requisitos de automação estão: planejamento dos testes, construção dos artefatos, execução, análise de falhas, medições e gerência de configuração dos artefatos de teste.

A estratégia em estudo foi aplicada em uma empresa que faz o uso da automatização de teste (MTS - Metodologia de Teste de Sistema), servindo assim como um campo de estudo para o trabalho. Através das ferramentas existentes desta empresa ocorreu o acompanhamento da implantação da estratégia em questão e a análise dos requisitos de automação definidos. O resultado demonstra que algumas ferramentas não atendem aos requisitos necessários para a automação.

As vantagens encontradas neste trabalho são a possibilidade de analisar se as ferramentas para a execução de teste de *software* atendem aos requisitos necessários para a realização de um teste eficaz e, também, poder buscar soluções para os requisitos que não são capazes de serem atendidos. Uma desvantagem encontrada neste trabalho é que é preciso de tempo para implantar uma metodologia de teste para verificar se todos os requisitos de teste estão sendo atendidos, causando assim um grande custo para a empresa.

3.5 Combinatória de Teste de *Software*

Este trabalho aplica combinações matemáticas em testes de *software* de modo a definir os testes de um sistema sejam mais eficientes do que os métodos de seleção manual de casos de teste.

Kuhn *et al*.(2009) introduz um método chamado combinação em pares, onde demonstra que, com a interação entre os componentes de um projeto juntamente com uma combinação matemática, os testes podem ser com qualidade e eficiência. A combinação matemática estudada foi representada por matrizes binárias, verificando se todos os parâmetros do sistema estão sendo testados entre si (em pares).

Um exemplo básico realizado para aplicar este método é um aplicativo que pode ser desenvolvido em sistemas operacionais Linux ou Windows, os processadores podendo ser Intel ou AMD e usar protocolos Ipv4 ou Ipv6. Em princípio há 8 possibilidades de teste a serem estudadas (2x2x2). Aplicando o método de combinações em pares, mostrado no Quadro 1, os componentes de testes interagem entre si, podendo assim ser representados em 4 testes. Neste exemplo, a redução de casos de teste é pouca, mas aplicando em sistemas maiores a redução tenderá a ser mais significante (Kuhn, 2009).

Table 1. Pairwise test configurations.

Test case	OS	CPU	Protocol
1	Windows	Intel	IPv4
2	Windows	AMD	IPv6
3	Linux	Intel	IPv6
4	Linux	AMD	IPv4

Quadro 1: Teste com combinação em pares.

Fonte: Kuhn, 2009.

Assim, Kuhn defende a tese de que a maioria das falhas acontece através das interações entre os parâmetros dos componentes do sistema.

Em relação ao trabalho proposto, os testes foram executados através de um *software* onde serão definidos manualmente os casos de teste. Já o trabalho de Kuhn (2009), consiste em realizar testes de combinações entre componentes de um sistema, para testar todos os parâmetros possíveis, ajudando assim a capturar as possíveis falhas. A vantagem identificada é ajudar a encontrar erros no ciclo inicial de teste, possibilitando uma maior taxa de sucesso na detecção de falhas.

3.6 Teste de *Software* Automatizado: Uma Solução para Maximizar a Cobertura de Plano de Teste e Aumentar a Confiabilidade do *Software* e Qualidade em Uso

O trabalho realizado por Catelani *et*. *al*.(2011), tem como propósito automatizar testes de *software* para avaliar o quanto o software é confiável submetendo o sistema em teste a condições reais de operações de forma acelerada. Para isso, é aplicado testes de *stress*, onde os módulos ou o sistema é posto sob condições de uso extremo. Além disso, é feita uma maximização de plano de teste, sendo aplicado o teste de regressão, onde os componentes são adicionados em etapas ao sistema testado, verificando o seu comportamento.

O teste em questão é aplicado em componentes do sistema e a avaliação da sobrecarga da memória é feita tanto sobre os componentes ou sobre o sistema por inteiro. A principal característica do teste é a avaliação simultânea da memória ocupada do sistema durante a execução dos testes automatizados, podendo assim avaliar a taxa de ocupação da memória. Esta abordagem de teste poder ser vista na Figura 8.

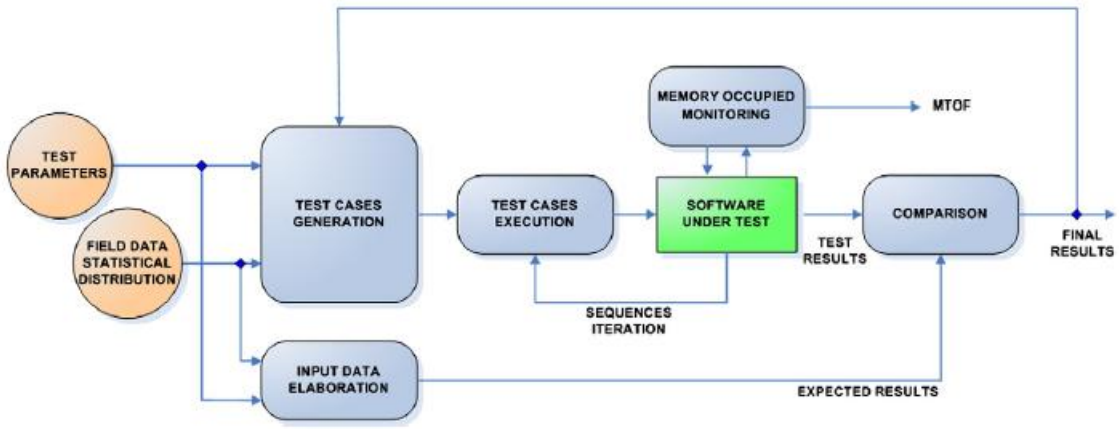

Figura 8: Diagrama de blocos do método de teste.

Fonte: Catelani *et*. *al*.(2011)

O processo responsável pela avaliação do estouro de memória é representado na atividade “*memory occupied monitoring*”, que gera uma variável MTOF(tempo médio de sobrecarga). Caso seja detectado algum vazamento de memória, o módulo é

repassado para a fase de implementação para verificar e corrigir o problema e em seguida é posto em teste novamente.

As vantagens desta metodologia: (i) são poder realizar os testes várias vezes conforme o testador de *software* achar necessário; e (ii) a forma como são definidos os casos de teste, onde são usados valores aleatórios compatíveis ao sistema, podendo realizar testes de forma dinâmicos. Uma desvantagem encontrada no trabalho é não poder avaliar onde está exatamente a causa do problema, aumentando a quantidade de trabalho para o programador que for analisar a falha dos módulos.

O trabalho de Catelani *et.* al. e o trabalho proposto têm em comum a avaliação dos testes onde é realizada a comparação do resultado final e o esperado através de um relatório de teste gerado pela ferramenta. Uma diferença seria a definição dos valores dos atributos dos casos de teste, que é feito de forma aleatória.

3.7 Discussão sobre os trabalhos correlatos

No trabalho desenvolvido por Orozco *et. al*,(2009), pode-se perceber que o objetivo foi o de gerar todos os casos de teste para um comportamento para fazer os testes funcionais. É de interesse demonstrar que os casos de teste podem ser gerados automaticamente, o que é útil para testadores de *software* que não têm conhecimento aprofundado do sistema. Já o trabalho desenvolvido por Dias *et*. *al*, (2008), demonstra que é perceptível que a utilização de modelos UML possui importância para a realização de teste de *software* em fase inicial de desenvolvimento de um sistema, já que através destes são definidos os comportamento para os casos de teste. No trabalho realizado por Barbosa (2005), percebe-se o uso de outras ferramentas para desenvolver a ferramenta SPACES, notando desta maneira um dinamismo na evolução do seu projeto.

O trabalho descrito por Costa (2004) é de interesse para as empresas que necessitam implantar a automatização de teste de *software*, assim o planejamento e a metodologia estudados podem ter bons resultados no projeto final. Hinchey (2009) também visa tornar o teste de *software* mais eficaz, fazendo uma análise de uma combinação matemática em cima dos componentes de um sistema para definir com maior qualidade os casos de teste.

No Quadro 2, algumas características relevantes dos projetos descritos nos trabalhos correlatos serão relatadas e comparadas à ferramenta desenvolvida.

	Trabalho proposto	**Derivação de Casos de teste (Orozco et. al)**	**Automação de Teste Funcional (Barbosa)**	**Teste de desempenho (Dias et. al)**	**Estratégia de Automação em Testes (Costa)**	**Combinatória de Teste de Software (Kuhn)**	**Solução para Maximizar a Cobertura de Plano de Teste (Catelani et. al.)**
Objetivo do Trabalho	Automatizar a execução de caso de teste	Derivação automatizada de casos de teste	Automatizar a execução de testes funcionais.	Automatizar teste de desempenho	Planejamento para a automação de testes de softwares	Estabelecer um método de teste para aumentar a eficiência.	Testar o software em condições acelerado de uso.
Tipo de testes	Teste Funcional	———	Teste Funcional	Teste de Desempenho	———	Teste Funcional	Teste de Confiabilidade
Diagrama usado para obter informações	Diagrama de sequência	Diagrama de atividade	Diagramas UML e restrições OCL	Diagrama de sequência e de Caso de Uso (Definir Métricas)	———	———	———
Definição dos casos de teste	Manual via arquivo de entrada XML	Automatizado	Método AFCT.	Automatizado (feramenta Jfern)	———	Manual	Automatizado com valores aleatórios
Execução dos casos de teste	Automatizado (auxílio da ferramenta FUMBeS)	———	Automatizado (feramenta JUnit).	Automatizado (auxílio da ferramenta Apache JMeter)	Automatizado	———	Teste Automatizado
Apresentação do resultado	Arquivo de saída no formato XML	Via Scripts	Interface gráfica	Arquivo de saída no formato XML	———	———	Via Scripts

Quadro 2 – Características entre trabalhos correlatos e o proposto.

4. ESPECIFICAÇÃO DO PROJETO

Este capítulo detalha a especificação do projeto, a qual descreve os passos que ocorrem no processo para a implementação da ferramenta de execução automática de casos de teste.

A especificação do projeto descreve o funcionamento do sistema. Além disso, é explicada uma estrutura de dados que representa o modelo de casos de teste que está representada em um diagrama de classe.

4.1 Funcionamento do sistema

Com a finalidade de representar o processo de automatização dos casos de teste proposto neste trabalho, é demonstrado na Figura 9 o funcionamento do sistema mostrando o fluxo de entrada e saída do sistema, além das ações executadas através da ferramenta implementada. A ferramenta desenvolvida é capaz de executar os testes em fase de projeto de *software*, ou seja, os testes são aplicados no modelo do sistema, não precisando da geração de código.

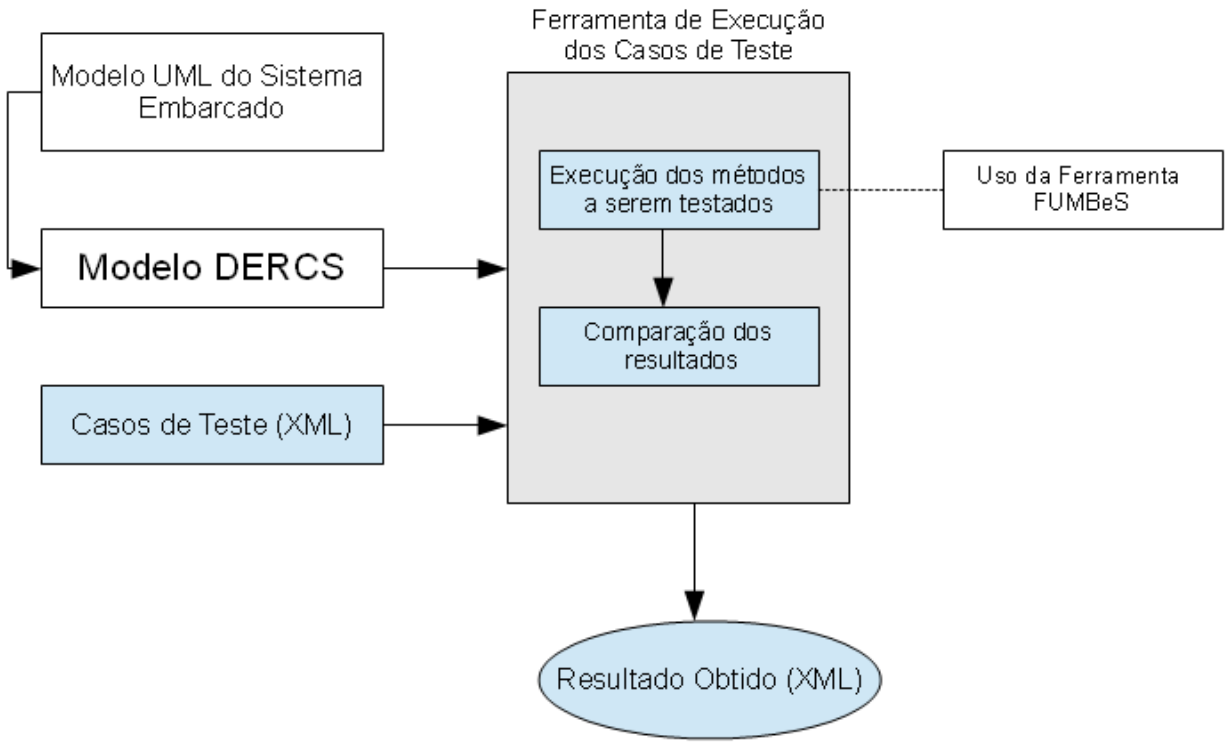

Figura 9 - Funcionamento do sistema

Na Figura 9, os elementos de cor azul claro representam os passos realizados neste trabalho, os de cor branca são as etapas do projeto que foram desenvolvidas no contexto de outros trabalhos. O sistema embarcado é inicialmente representado através da linguagem de modelagem UML. A representação do sistema através de modelos UML pode possuir informações repetidas, como por exemplo, definir o comportamento de um método mais de uma vez usando diagramas diferentes, tornando a manipulação dos elementos da especificação (i.e. os elementos do meta-modelo) mais complexa. Desse modo, este modelo UML é transformado em um modelo intermediário chamado modelo DERCS (*DIstributed Embedded Real-time Compact Specification* – Wehrmeister, 2009). Esse meta-modelo consegue representar os elementos do modelo UML de sem ambiguidade. Tais elementos representam as informações estruturais e/ou comportamentais do sistema. Assim, as informações do sistema, são estruturadas conforme o meta modelo DERCS e são armazenadas em memória não volátil através de um arquivo do tipo binário. A geração de arquivos binários é feita através da serialização de dados padrão fornecida pela API (*Aplication Programming Interface*) da

linguagem de programação JAVA A conversão entre modelos é feita de forma automática com auxílio de uma ferramenta desenvolvida por Wehrmeister (2009).

A solução proposta tem como início a importação do modelos do sistema embarcado, que está estruturado conforme o meta modelo DERCS e armazenado em um arquivo binário. Ao ler o arquivo, as informações estruturais e comportamentais são alocadas em memória volátil (e.q. RAM) através da estrutura definida na API DERCS, desenvolvida por Wehrmeister, (2009). Esta importação também é realizada através da API de serialização de dados do JAVA. Dessa forma, a ferramenta desenvolvida neste trabalho obtém todas as informações do sistema necessárias para realizar simulações comportamentais dos métodos.

Para definir os casos de teste, foram usados diagramas de sequência para coletar as informações sobre o sistema que será testado. Os casos de teste são construídos manualmente em um arquivo de entrada no formato XML. Neles são descritas as informações necessárias para realizar os testes, tais como: os valores dos atributos dos objetos e os métodos a serem testados. O uso desse tipo de arquivo de entrada permite a importação e exportação de informações produzidas pela ferramenta de testes, de modo a facilitar a integração dela com outras ferramentas CASE relacionadas com as atividades de verificação e validação do sistema. As informações do teste fornecidas no arquivo XML descrevem objetos, métodos, atributos, de forma autocontida e independente de plataforma, facilitando o seu uso em outras partes do projeto como, por exemplo, geração de código para os casos de teste na plataforma alvo.

A ferramenta para execução dos casos de teste implementada neste trabalho importa o modelo DERCS e o arquivo XML que representa os casos de teste. Os dados dos casos de teste são representados (em memória) através de uma estrutura de dados orientada a objetos usando a linguagem de programação Java.

Após a ferramenta importar as informações sobre o modelo, o próximo processo é executar os casos de teste, com o auxílio de uma ferramenta chamado FUMBeS (*Framework for UML Model Behavior Simulation*). Esta ferramenta desenvolvida por Wehrmeister *et*. *al* (2011) faz a simulação de um comportamento especificado no modelo UML/DERCS.

Na API DERCS, um comportamento do sistema embarcado é tratado como um conjunto de ações que são executadas em resposta ao recebimento de uma mensagem. Desse modo, quando um objeto recebe uma mensagem , ele inicia a execução de uma sequência de ações. Exemplos de ações executadas por um objeto são: atribuir um valor à uma variável, retornar um valor ou um objeto, envio de mensagens (acionando outro objeto), modificar o estado do objeto, entre outros (Wehrmeister, 2009). Com base nestas informações que estão acessadas através da API DERCS, a ferramenta FUMBeS realiza a simulação da execução dessas ações individualmente, possibilitando então a simulação comportamento do sistema. A ferramenta FUMBeS pode retornar um valor resultante da execução dos métodos, que então é comparado automaticamente com um valor esperado especificado no caso de teste.

Por fim, é gerado um relatório em um arquivo de saída em formato XML onde estão todas as informações do resultado obtido após a execução de todos os casos de teste. Assim, fornecem-se informações para verificar se o comportamento do sistema especificado no modelo UML está correto ou não.

Portanto, este trabalho consistiu em desenvolver as seguintes etapas:

- Definir os casos de testes;
- Importar as informações do sistema embarcados no meta modelo DERCS;
- Leitura dos casos de testes;
- Execução dos testes (com o auxílio da ferramenta FUMBeS);
- Comparar os resultados dos testes com o esperado; e
- Gerar o relatório dos testes.

4.2 Modelo para Representação dos Casos de Teste

A estrutura dos casos de teste é apresentada através de um diagrama de classes na Figura 10, onde são mostradas todas as classes e os atributos necessários para a execução dos casos de teste. Essa estrutura foi baseada na necessidade de alocar em memória os elementos dos casos de teste e gerar um relatório de informações do resultado obtido após a execução.

Existe uma lista de casos de teste a ser testada na classe *TestSuite*. Para cada caso de teste, é preciso ser definido o cenário inicial (associação entre as classes *TestCase* e *ScenarioInput*). Este cenário inicializa valores aos atributos dos objetos. A cada método executado (através da ferramenta FUMBeS) os valores dos atributos alocados na memória podem ser alterados. As comparações após a execução dos métodos ocorrem entre: o cenário resultante e o esperado; e o valor retornado do método com o seu valor esperado.

Tanto um caso de teste como um método a ser testado pode ter um cenário inicial, que está representado através da classe *ScenarioInput*, podendo representar as informações dos seus objetos e atributos através das classes *Object* e *Atributte*, respectivamente.

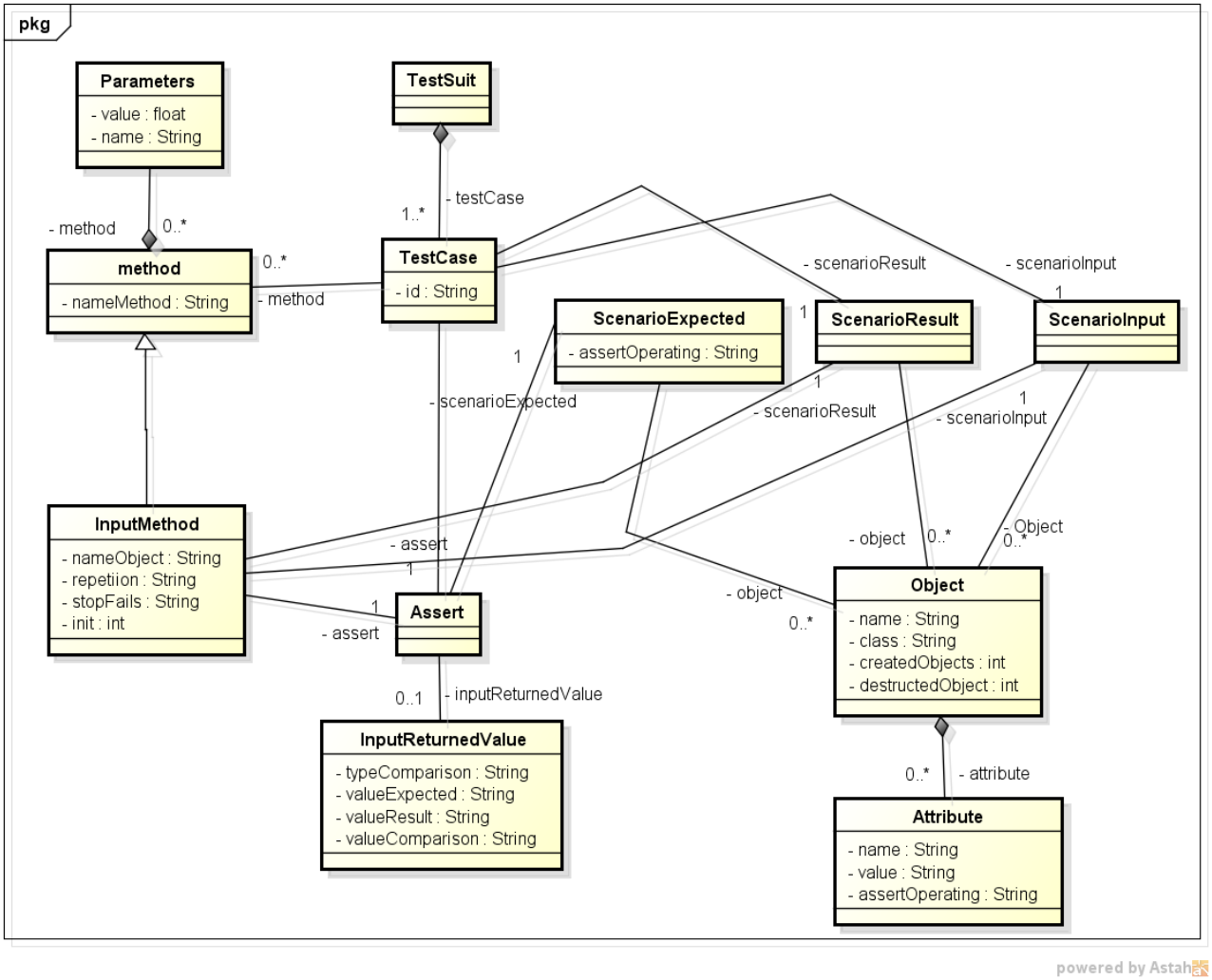

Figura 10 – Diagrama de classe do Modelo dos Casos de teste.

Os métodos a serem testados pertencem à classe *InputMethod*. Para cada método são definidos: os parâmetros de entrada (caso existam) na classe *Parameters*; o valor de retorno esperado da sua execução na classe *InputReturnedValue;* e os cenários esperado e o resultante (associações entre as classes *InputMethod*, *Assert*, *ScenarioExpected* e *ScenarioResult*). Assim, no teste, são feitas duas comparações: entre o valor obtido na execução do método e o valor esperado (o atributo *assertOperating* da classe *Attribute* representa o resultado da operação); e entre os cenários esperado e o resultante (em caso de inconsistência, a configuração é feita pelo atributo *assertOperating* da classe *ScenarioExpected*).

Também é possível representar um cenário de entrada, esperado e resultante (associação entre a classe *TestCase* com as classes *ScenarioExpected*, *ScenarioResult* e *ScenarioInput*). Caso o testador queira saber quantos objetos são destruídos ou criados (para, por exemplo, testar a quantidade gasta de memória), pode-se obter esta informação através do atributo *createdObjects* e *destructedObjects* da classe *Object*.

5. PROJETO E IMPLEMENTAÇÃO DA FERRAMENTA PARA AUTOMATIZAÇÃO DOS CASOS DE TESTES

O presente capítulo descreve as etapas necessárias para o desenvolvimento da ferramenta, especificando as tecnologias utilizadas, bem como os detalhes do projeto e da implementação.

5.1 Diagrama de Caso de Uso

Esta seção apresenta a análise das funcionalidades da ferramenta construída. Esta ferramenta recebeu o nome de AutoTeste, tendo como objetivo de automatizar a execução dos casos de teste.

As funcionalidades da ferramenta desenvolvida são apresentadas pela Figura 11, através do Diagrama de Caso de Uso:

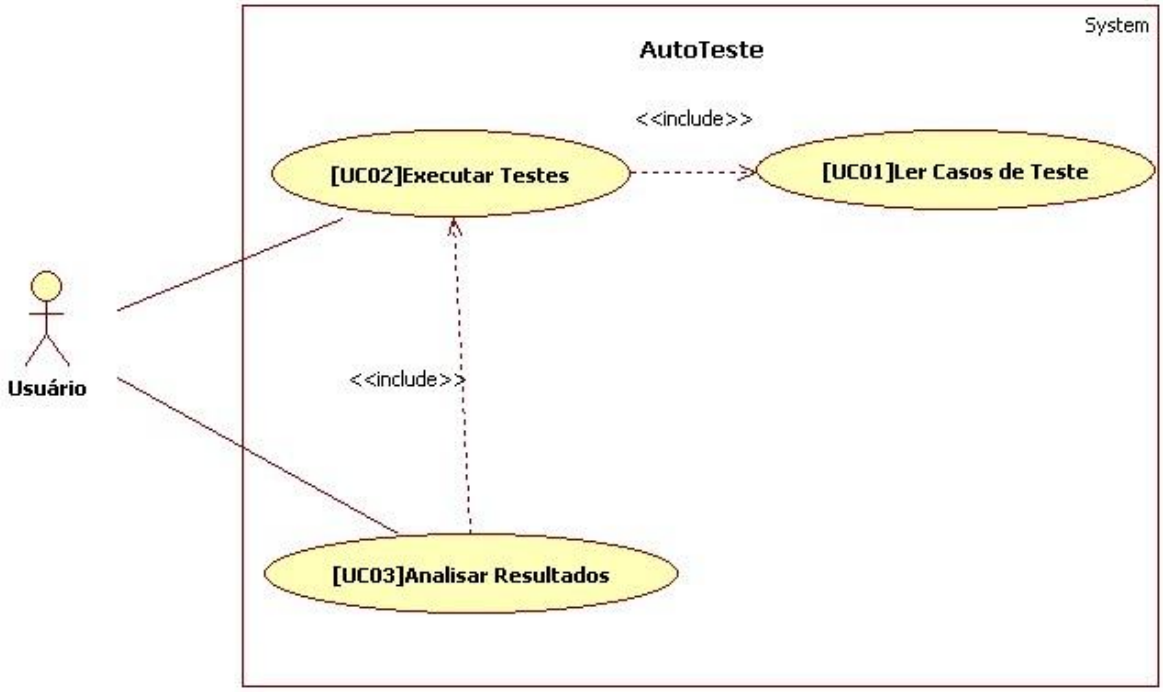

Figura 11 – Diagrama de Caso de Uso da Ferramenta AutoTeste.

Os detalhes dos casos de uso são descritos a seguir:

[UC01]: Ler Casos de Teste

Sumário: A ferramenta realiza a leitura do arquivo de entrada que representa os casos de teste e coloca seus respectivos dados em memória através de uma estrutura de dados (Figura 10). Após a execução dos casos de teste, o resultado dos testes também é armazenado em memória através desta estrutura de dados.

Ator principal: Usuário.

Ator secundário: Não existe.

Entrada: Arquivo no formato XML.

Pré-condições: Respeitar a estrutura de dados definido no diagrama de classe da ferramenta.

Fluxo principal:

1. Lê o documento de entrada.
2. Armazena os dados no modelo de casos de teste.
3. Retorna o modelo de caso de teste para uma instância da classe *ExecutarTeste*.

[UC02]: Executar Testes

Sumário: É necessário submeter para a ferramenta o arquivo de entrada onde estão definidos os casos de teste. Logo após, a ferramenta faz as execuções dos testes do sistema de forma automática gerando relatório dos testes e um arquivo de saída.

Ator principal: Usuário.

Ator secundário: Não existe.

Entrada: Arquivo XML do modelo Caso de Teste, arquivo binário do modelo DERCS.

Pré-condições: Arquivo de formato válido, ou seja, XML e respeitando a estrutura de dados.

Fluxo Principal:

1. Leitura do modelo DERCS.
2. A ferramenta recebe o arquivo de entrada que representa os casos de teste.
3. Executa os casos de teste.
4. Armazena o resultado dos testes no modelo de casos de teste.

[UC03]: Analisar Resultados

Sumário: Após a execução dos testes, a ferramenta possibilita o usuário analisar os testes realizados através de um documento gerado em formato XML. A partir dos dados gerados deste arquivo, é possível o usuário visualizar e concluir sobre a validade do teste aplicado.

Ator principal: Usuário

Ator secundário: Não existe

Entrada: Dados do modelo dos casos de teste.

Pré-condições: Dados válidos.

Fluxo principal:

1. Gerar um documento de saída apresentando informações dos testes.
2. Compara os cenários esperados e resultantes dos casos de testes e dos métodos testados.
3. Compara o valor esperado e o retornado da execução de cada método.

5.2 Visão Geral do Processo

As trocas de mensagens entre os objetos no decorrer do tempo para a realização dos testes da ferramenta AutoTeste podem ser representadas através do diagrama de sequência ilustrada na Figura 12.

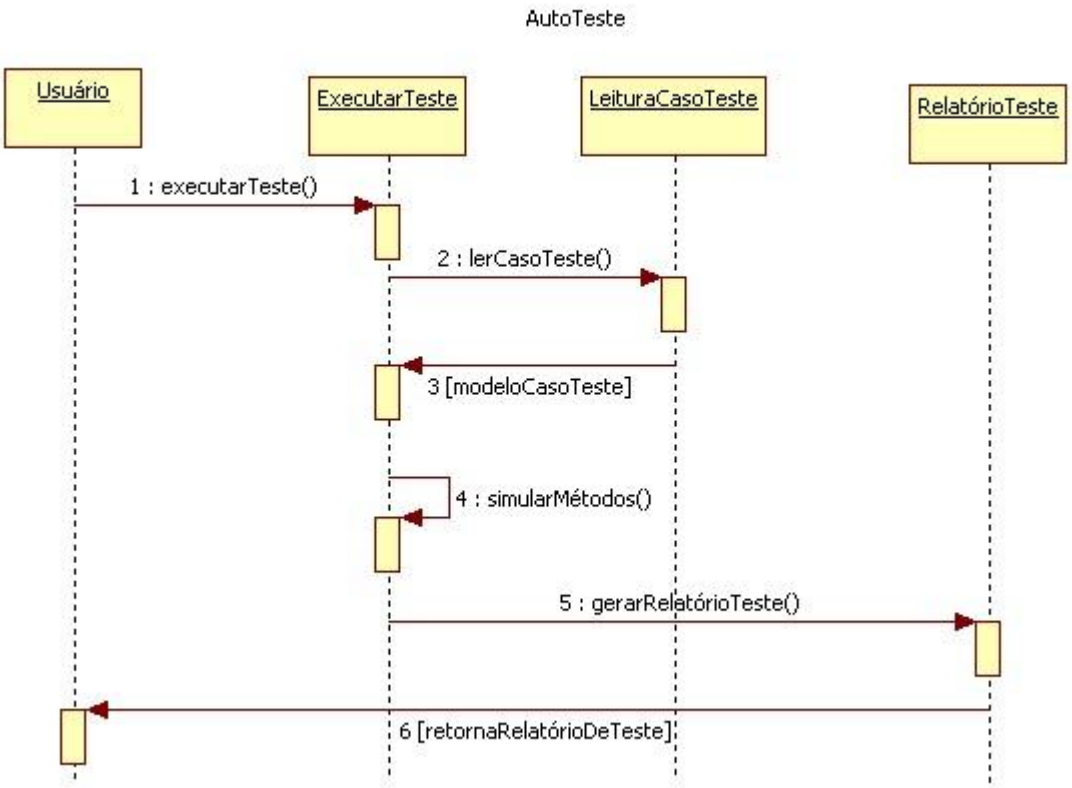

Figura 12: Diagrama de Sequência da Ferramenta AutoTeste.

Inicialmente, a ferramenta é executada pelo testador de *software*. Após isso, o objeto *ExecutarTeste* realiza a leitura dos casos de teste com o auxílio do objeto *LeituraCasoTeste,* responsável por realizar o *parser* do documento de entrada que define os casos de teste e alocar estas informações no modelo de caso de teste.

Após esta leitura, é retornado para o objeto *Executar* o modelo de caso de teste para então executar os testes, representado na iteração da mensagem 4. Por fim, é feito o relatório de teste através do objeto *RelatórioTeste,* onde é relatada as informações dos testes e retornado para o usuário um documento de saída para a sua visualização.

5.3 Diagrama de Classe

Esta seção descreve o projeto estrutural da ferramenta AutoTeste, baseado nos requisitos apresentados no Diagrama de Casos de Uso da seção 5.1.

Posto isso, o Diagrama de Classes da ferramenta é demonstrado na Figura 13:

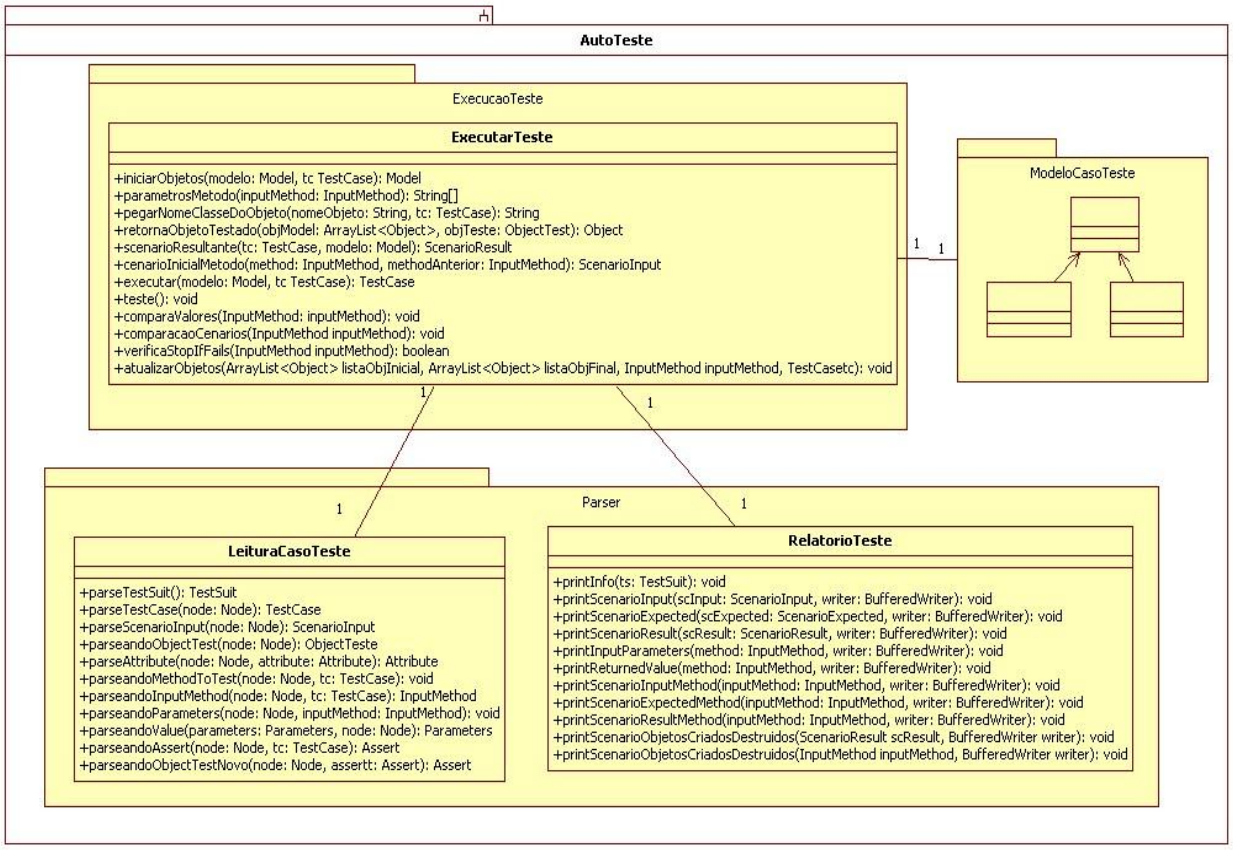

Figura 13 – Diagrama de Classe da Ferramenta AutoTeste.

Conforme o modelo, a ferramenta é composta pelos seguintes pacotes:

a) Parser: Este pacote tem as classes responsáveis realizar a leitura dos casos de testes e gerar o relatório. A classe *LeituraCasoTeste*, realiza o *parsing* (processo de analisar uma sequência de entrada) do arquivo de entrada, que contém os casos de testes. Os dados são postos em memória através do modelo dos casos de teste. Após a execução dos testes, a classe *RelatórioTeste* realiza a

geração dos resultados que estão no modelo dos casos de teste, gerando assim um novo arquivo de saída no formato XML.

b) **ModeloCasoTeste:** O pacote *ModeloCasoTeste* contém as classes e os atributos responsáveis por representar os casos de testes e o resultado da execução. Tanto as funcionalidades como as suas classes já foram apresentada na seção 4.2.

c) **ExecucaoTeste:** Este pacote contém a classe *ExecutarTeste*, que é o “motor” de execução dos testes. Esta classe é responsável por: executar os casos de testes, gerenciar a configuração dos cenários resultantes (relatório dos testes) e manipular as informações do sistema para efetuar os testes. Para a realização de tais procedimentos, foi preciso a importação de duas APIs:

   - DERCS: com o uso desta API foi possível obter informações correspondentes ao sistema em teste. Esta API representa o modelo do sistema em teste e, dentre as suas principais finalidades a é obtenção e definição dos valores dos atributos dos objetos do sistema em teste. Basicamente, todas as informações relativas ao sistema em teste (métodos, parâmetros, classes, atributos, etc) para realizar o teste foram obtidas através do DERCS.

   - FUMBeS: uma das etapas desta automatização de teste é realizar a simulação do comportamento dos métodos em testes. Este processo foi realizado com a ferramenta FUMBeS, onde o comportamento de cada método em teste é simulado através da API DERCS. Por fim, o FUMBeS retorna o valor do resultado da execução (se tiver retorno) alterando o estado dos objetos (podendo ou não ser alterado) do modelo.

5.4 Construção da Ferramenta AutoTeste

Este seção descreve os procedimentos para a implementação do projeto. Tal construção se divide em três etapas: fazer o análise do diagrama de sequência, a definição dos casos de teste e a implementação da ferramenta.

5.4.1 Análise do Diagrama de Sequência

Os casos de teste devem ser baseados nos diagramas de sequência que representam o comportamento do sistema em teste. O testador de *software* deve ter o conhecimento sobre diagrama de sequência para fazer a análise visual, isto é, saber a atividade de cada método.

Através dessa análise, o testador vai determinar os pontos crucias para o teste, conforme a sua essencialidade. A amplitude dos testes pode ser em níveis unitários (testes de unidade) ou de interações (teste de integração).

5.4.2 Definindo os casos de teste

Para realizar a execução de testes na ferramenta AutoTeste, antes é preciso especificar os casos de teste. O documento de entrada que representa os casos de testes deve estar no formato XML.

O conteúdo dos testes deve ser baseado no sistema que está em teste e suas informações devem ser retiradas do diagrama de sequência. Após a análise do diagrama de sequência, basta descrever os casos de teste respeitando a estrutura de dados. A estrutura para definir os casos de testes dentro do documento está representada na Figura 14:

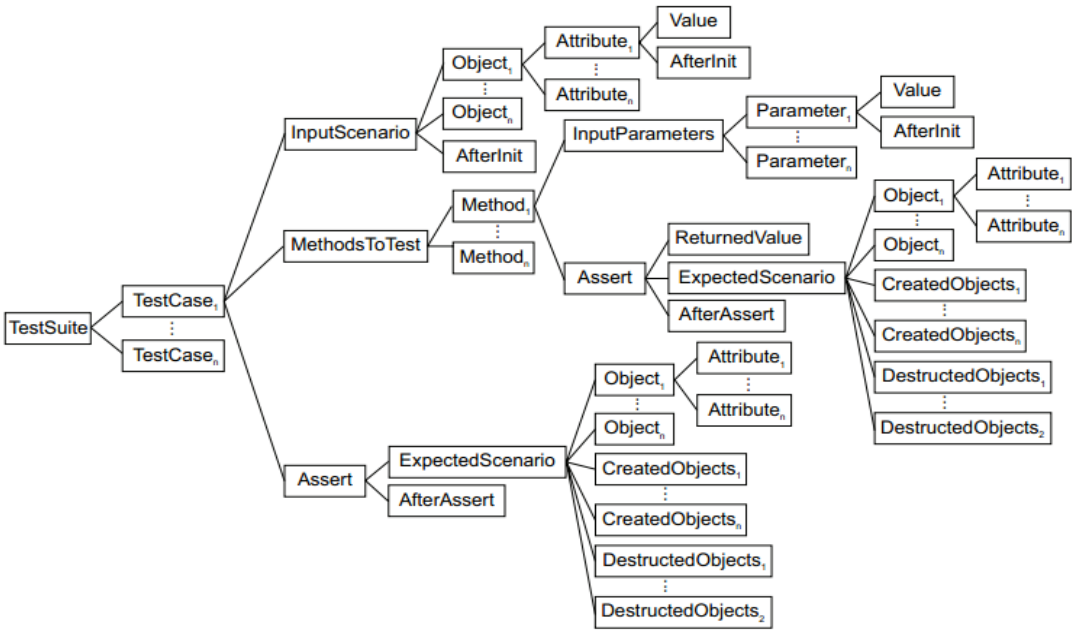

Figura 14: Estrutura do arquivo XML que descreve os casos de teste.

Fonte: Wehrmeister, (2009).

A figura 14 representa todos os elementos necessários para a realização dos casos de testes. Um exemplo da definição de um caso de teste em formato XML está mostrado no Quadro 3 do capítulo 6.

O primeiro passo desse caso de teste é inicializar os objetos em teste (conforme definido no diagrama de sequência) em um cenário de entrada. Através desse cenário, são dadas as informações dos objetos como o seu nome, a classe à qual pertence e os nomes dos atributos com seus respectivos valores.

Após isso, as informações sobre os métodos em testes são especificadas com os parâmetros de entradas e o valor retornado de cada método. Além disso, o usuário pode definir um cenário esperado após a execução do método, para poder comparar os estados dos objetos do cenário resultante.

Para cada método, algumas informações são definidas como, por exemplo, o seu nome e o objeto que pertence. Além disso, outras informações adicionais são definidas, tais como:

a) **Executar o método várias vezes:** O testador pode querer analisar o comportamento do método sendo executado mais de uma vez. Assim, a cada execução do método, o estado do objeto pode ser alterado. Esta informação é fornecida pelo elemento *repetition*.
b) **Parar o caso de teste se houver falha:** O usuário tem a possibilidade de parar de executar o caso de teste corrente se houver alguma inconsistência no método testado. Assim, a ferramenta continuará a executar o próximo caso de teste, se houver. Informação definido pelo elemento *stopIfFails* (define-se *stopIfFails* = 1 caso o teste pare, caso contrário, *stopIfFails* = 0).
c) **Reiniciar o modelo[1] em teste:** A cada método testado, o modelo que representa o sistema (definido no arquivo de entrada contendo o modelo DERCS) é alterado conforme o seu comportamento. Através do elemento *init*, é possível o testador poder reiniciar o modelo antes de a ferramenta executar um método.

Após as informações dos métodos serem especificadas, o testador pode definir o cenário esperado após todos os métodos serem executados. Este cenário esperado será comparado com o cenário resultante das execuções dos métodos.

Estas informações são a base necessária para realizar a execução dos casos de teste. No Capítulo 6 é feito um estudo desse caso de teste e o seu resultado gerado no relatório de teste produzido pela ferramenta AutoTeste. Na próxima seção, são detalhados, em etapas, os algoritmos da implementação desta ferramenta.

5.4.3 Detalhes da Implementação

Para o desenvolvimento da ferramenta AutoTeste, foi utilizada a linguagem de programação JAVA, pelo fato das ferramentas auxiliares DERCS e FUMBES serem implementadas nesta linguagem. Em relação ao ambiente de programação selecionado para o desenvolvimento, optou-se pelo *Eclipse,* por dar suporte às ferramentas necessárias para a implementação do projeto.

[1] Entende-se por reiniciar o modelo do sistema em atribuir os valores do estado dos objetos conforme definido no cenário de entrada. Para reiniciar o modelo, fixa *init*=1, caso contrário, *init*=0.

A implementação da ferramenta pode ser dividida em 3 partes: leitura dos casos de teste, a execução automática dos casos de teste e a produção relatório de teste.

5.4.3.1 Leitura dos Casos de Teste

Esta etapa coleta as informações dos casos de teste que estão presentes em um documento e coloca essas informações no modelo de casos de teste. Para realizar a manipulação e transformação dos dados do documento XML para a memória, foi usado o DOM (*Document Object Model*), que é uma API pertencente à linguagem *Java*.

A API DOM foi escolhida pelo fato desta montar, de forma automática, uma árvore com os conteúdos do arquivo XML, e também possuir funções que facilitam a manipulação e a navegação dos dados do arquivo XML.

A figura 15 mostra um fluxograma que representa o algoritmo implementado para realizar a leitura dos casos de teste. A partir de um nó representando a raiz de um documento XML, são percorridas todas as informações relativas aos casos de teste definidos pelo usuário.

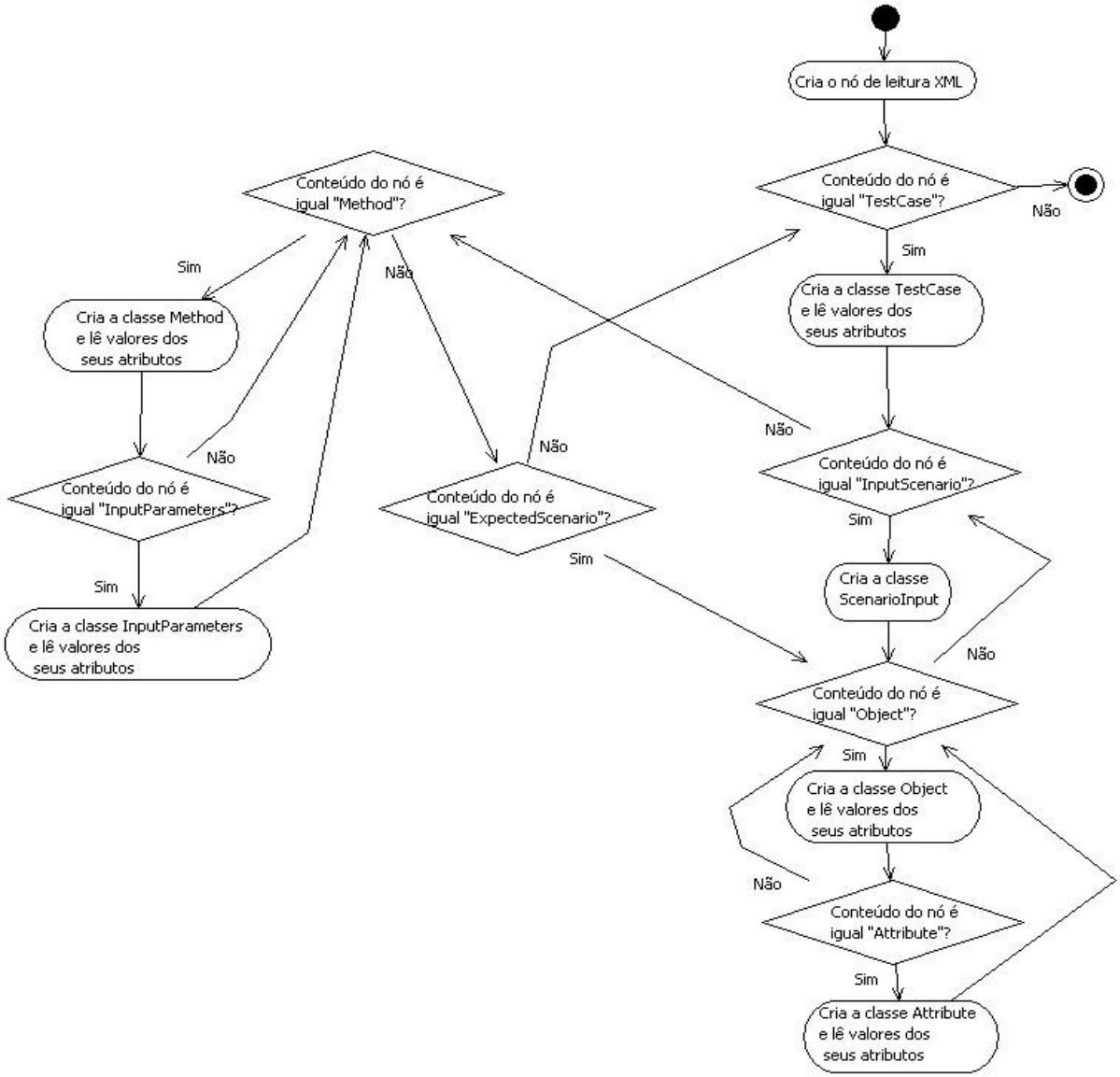

Figura 15: Fluxograma da leitura dos Casos de Teste.

A leitura do arquivo só termina quando todos os casos de teste e seus conteúdos são alocados em memória através do modelo de caso de teste. Todas as implementações dos fluxogramas pertinentes a este capítulo estão disponibilizadas na seção 1 do anexo.

5.4.3.2 Execução dos Casos de Teste

Com os dados definidos no modelo estrutural de casos de teste, a próxima etapa da ferramenta é executá-lo de forma automatizada. Este processo pode se dividir em 4 passos:

a) **Executar os casos de teste:** A Figura 16 mostra como fica o fluxograma para executar os casos de teste. Primeiramente é feito o *parsing* dos casos de teste. Após, é percorrido todos os casos de teste. Para cada caso de teste, o modelo DERCS que representa o sistema em teste é carregado. Após a execução de todos os casos de teste, é gerado o relatório de teste dos casos de testes.

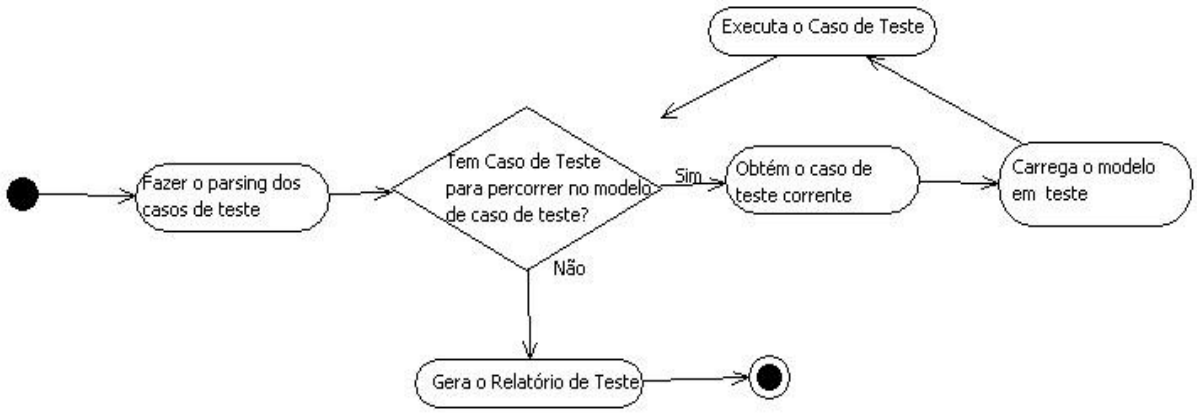

Figura 16: Fluxograma para a execução dos casos de teste

b) Configuração do cenário do caso de teste: Esta etapa consiste em inicializar os objetos do sistema em teste conforme especificado no cenário de entrada do caso de teste. O fluxograma da Figura 17 demonstra como ficaria o algoritmo para a implementação desta etapa. O estado dos objetos do modelo em teste é inicializado conforme os objetos e seus valores definidos no modelo de caso de teste.

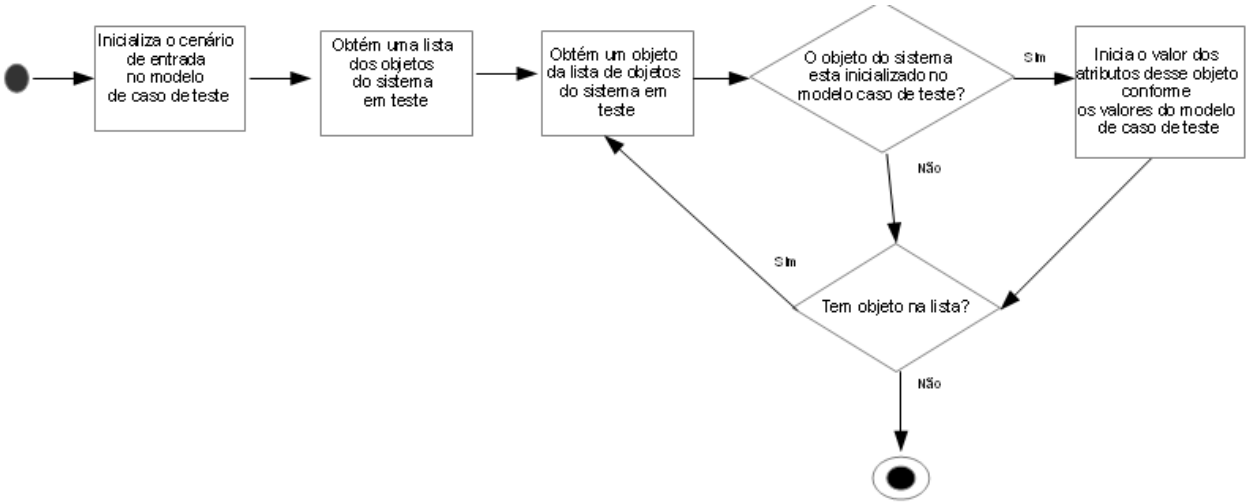

Figura 17: Inicializando os valores dos objetos do sistema.

Com o auxílio da ferramenta DERCS, foi possível obter o acesso (atribuir ou consultar valores) aos atributos dos objetos do sistema. O DERCS oferece a Classe *Model* que tem acesso a todos os objetos e seus respectivos atributos com seus valores. Através da classe *RuntimeInformation* da ferramenta DERCS, é possível representar o estado dos objetos em tempo de execução do sistema em teste, assim os objetos são inicializados. Inicialmente o estado de todos os objetos é inicializado como *null*.

c) **Simulação dos métodos:** Neste trabalho, executar um teste de *software* é realizar uma simulação do comportamento dos objetos estabelecido no modelo DERCS. Esta simulação é realizada através do método *startSimulation*() da classe *Simulation* que pertence à ferramenta FUMBeS. Esta etapa está representada na Figura 18.

```
//simula o comportamento do método
dercsSimulation.startSimulation(testMethod, testObj, valParametros, modelo, inputMethod.getInit());
//colocar o valor retornado do metodo no modelo
inputMethod.getAssertt().getInputReturnedValue().setValorResultado(dercsSimulation.getValorRetornoMetodo());
```

Figura 18: Simulação dos métodos em teste.

Este código foi retirado do método executar da classe Executa. O método *startSimulation()* têm como parâmetros de entrada: o método a ser testado, o objeto ao qual o método em teste pertence, os parâmetros do método em teste, o modelo em teste e a opção de reinicializar o modelo. Após a simulação de cada método, os objetos pertencentes a este modelo podem mudar de valor conforme o resultado da simulação. Através do método *getValorRetornoMetodo()* da ferramenta FUMBeS é retornado o resultado da execução do método que está sendo atualmente testado.

d) Configuração dos cenários para o relatório de teste: esta etapa é importante para a validação do teste ao fazer a comparação do cenário esperado junto ao cenário resultante (cenário após a execução dos métodos). As informações dos cenários dos testes estão definidas no modelo de caso de teste e é a partir delas que é gerado o relatório da execução dos casos de teste.

5.4.3.3 Demonstração do Relatório de Teste

Após os testes terem sidos executados, é preciso gerar o relatório do seu resultado. Esse resultado é exposto em um documento de saída no formato XML. Este documento deve respeitar uma estrutura de representação de dados, ilustrada na Figura 19:

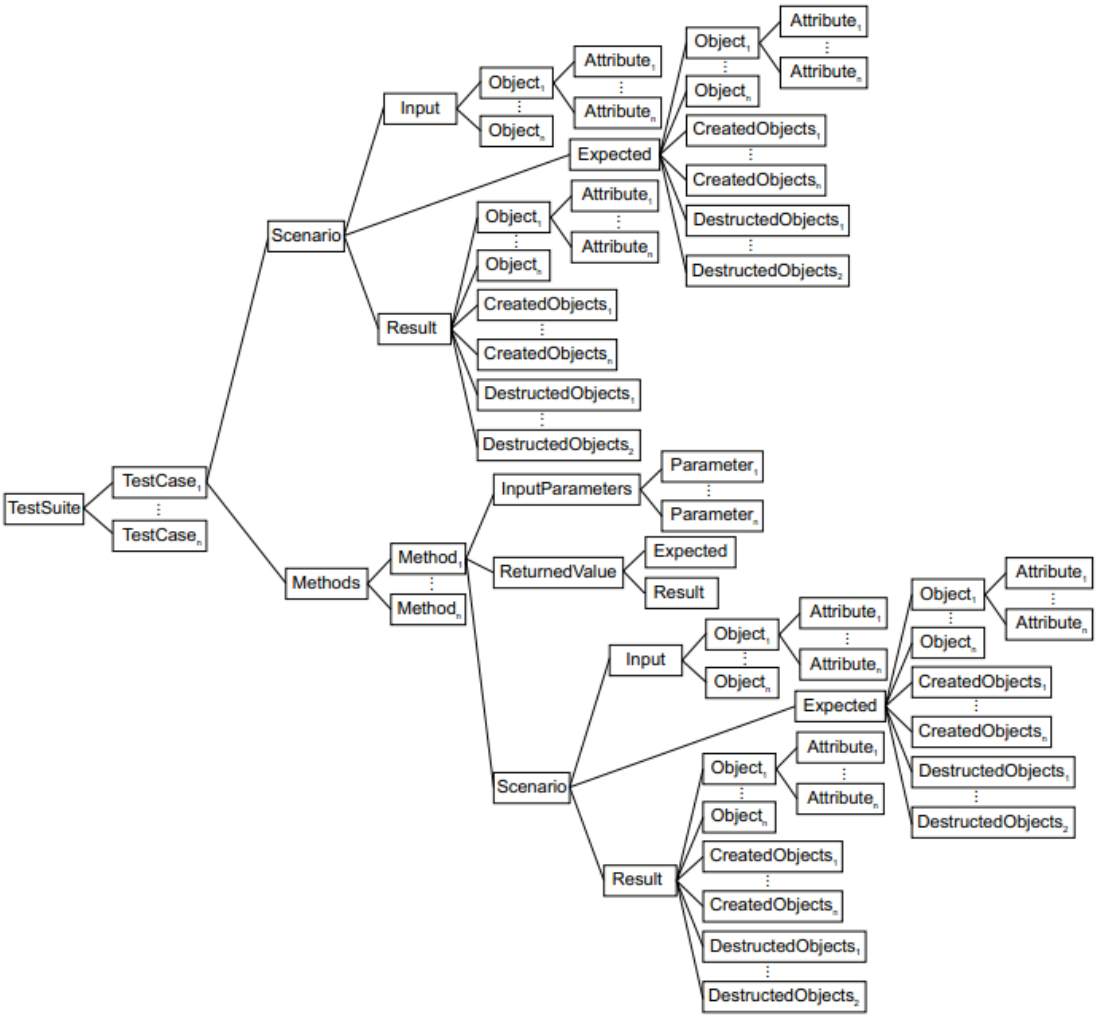

Figura 19: Estrutura do arquivo XML que representa o relatório dos testes.
Fonte: Wehrmeister, (2009).

A partir dos elementos dessa estrutura é possível fazer a validação da execução dos casos de testes automatizados. Esta estrutura se assemelha com o arquivo de configuração dos casos de teste. A diferença deste arquivo de relatório está na informação sobre os casos de testes executados.

Foi necessário importar o pacote *CasoTeste*, que é onde estão as classes que modelam a estrutura de dados dos casos de teste. O fluxograma apresentado na Figura 20 representa o algoritmo para realizar a impressão do relatório de teste.

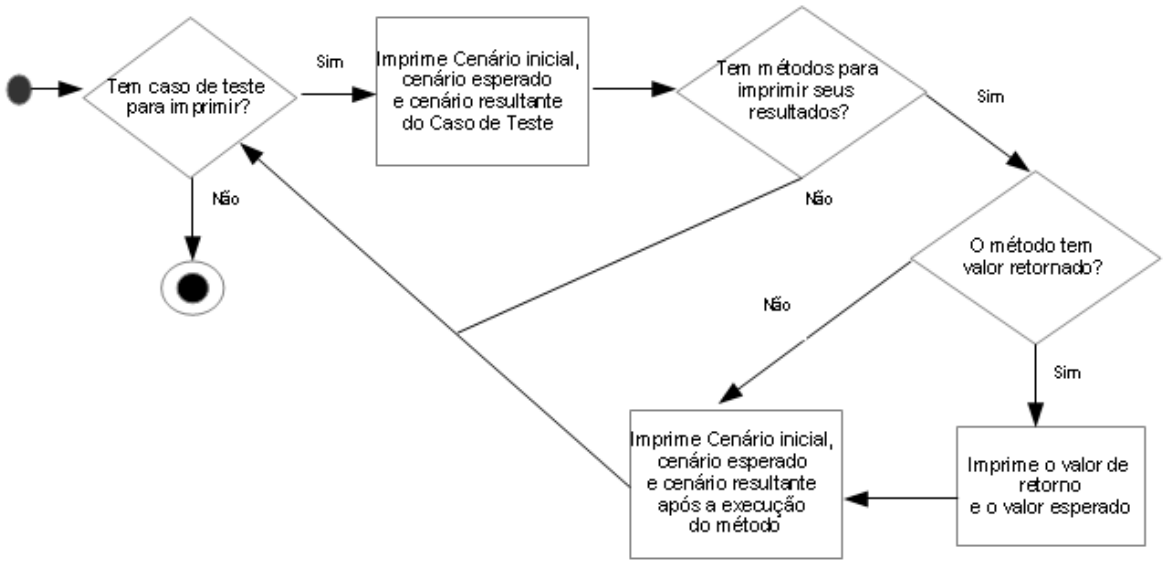

Figura 20: Gerando relatório dos testes.

Os resultados dos casos de teste presentes no modelo são percorridos. Assim, todos os cenários que envolvem o caso de teste e os valores de retorno dos métodos são relatados em um documento de saída.

6 AVALIAÇÃO DOS RESULTADOS

Este capítulo é dedicado à avaliação dos resultados atingidos pela ferramenta desenvolvida durante este trabalho.

Um sistema foi testado aplicando a ferramenta AutoTeste, servindo como um estudo de caso. A partir do relatório de teste gerado, foi seguida esta metodologia de avaliação para verificar se houve alguma falha no sistema.

6.1 Metodologia de Avaliação

Esta seção descreve os passos de avaliação dos resultados alcançados pela ferramenta AutoTeste, a fim de fazer a verificação da ferramenta e analisar os resultado dos testes.

Sendo assim, a metodologia de avaliação é descrita na Figura 21:

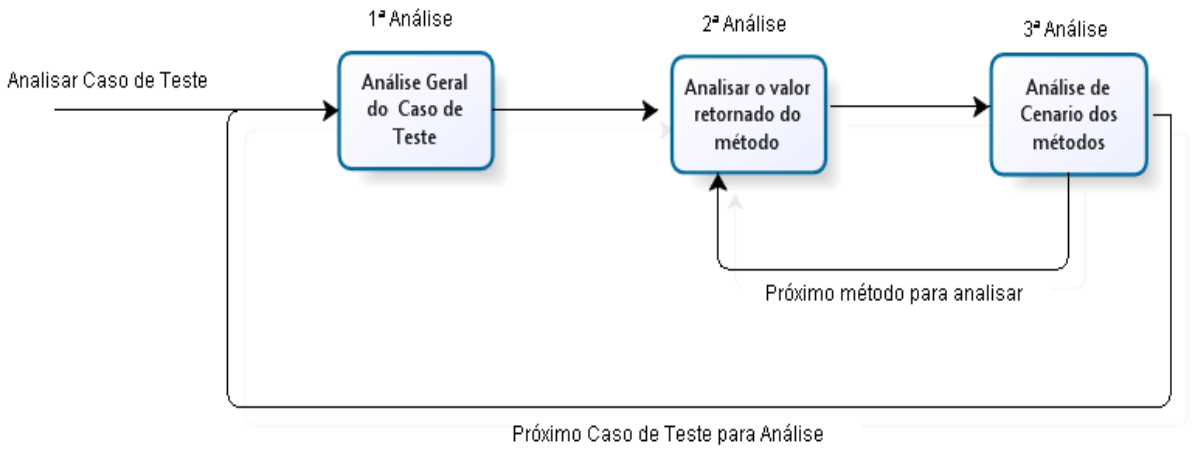

Figura 21: Metodologia de Avaliação dos Resultados.

Esta metodologia de avaliação dos resultados foi baseada na ordem da geração de dados do relatório dos testes. Primeiramente o relatório gera o resultado final do caso de teste depois o resultado de cada método testado. Conforme apresenta a Figura 21, a avaliação de cada caso de teste é composta por três análises:

1ª Análise: O testador analisa o resultado final do caso de teste. Este resultado está representado através de três cenários: cenário de entrada, cenário esperado e cenário resultante. Os dois primeiros cenários são definidos pelo testador no caso de teste. O cenário resultante apresentam os valores dos estados dos objetos após todos os métodos serem executados. Basicamente, é feito uma análise dos valores dos atributos dos objetos do cenário resultante com os valores do cenário esperado. Se tiver algum valor divergente, é por que ocorreu algum de comportamento não esperado na execução dos casos de teste.

2ª Análise: A segunda análise que o testador pode fazer é comparar o valor de retorno de cada método com o valor esperado, caso a execução do método tenha algum retorno.

3ª Análise: Por fim, é feita a comparação do cenário esperado e o cenário resultante de cada método testado. Novamente é realizado um exame para verificar se houve alguma divergência de valores entre os atributos apresentados nesses cenários.

6.2 Casos Avaliados

Os casos avaliados tratam de um sistema de controle de movimento de um Veículo Aéreo Não-Tripulado (VANT) (Wehrmeister, 2008b).

No trabalho realizado por Wehrmeister(2008b), foi especificado este sistema em um modelo UML, que foi posteriormente transformado no meta modelo DERCS. Em seguida é feita a importação das informações desse meta modelo para a estrutura de dados definida pela API DERCS representando as informações comportamentais e estruturais do sistema VANT.

O trabalho de Wehrmeister(2008b) também fez a representação desse sistema em diagramas de sequência, ilustrada na Figura 22, servindo como fonte para coletar as informações dos casos de teste. Os casos de teste são definidos manualmente através de um arquivo XML. O caso de teste posto em estudo neste trabalho está representado no Quadro 3.

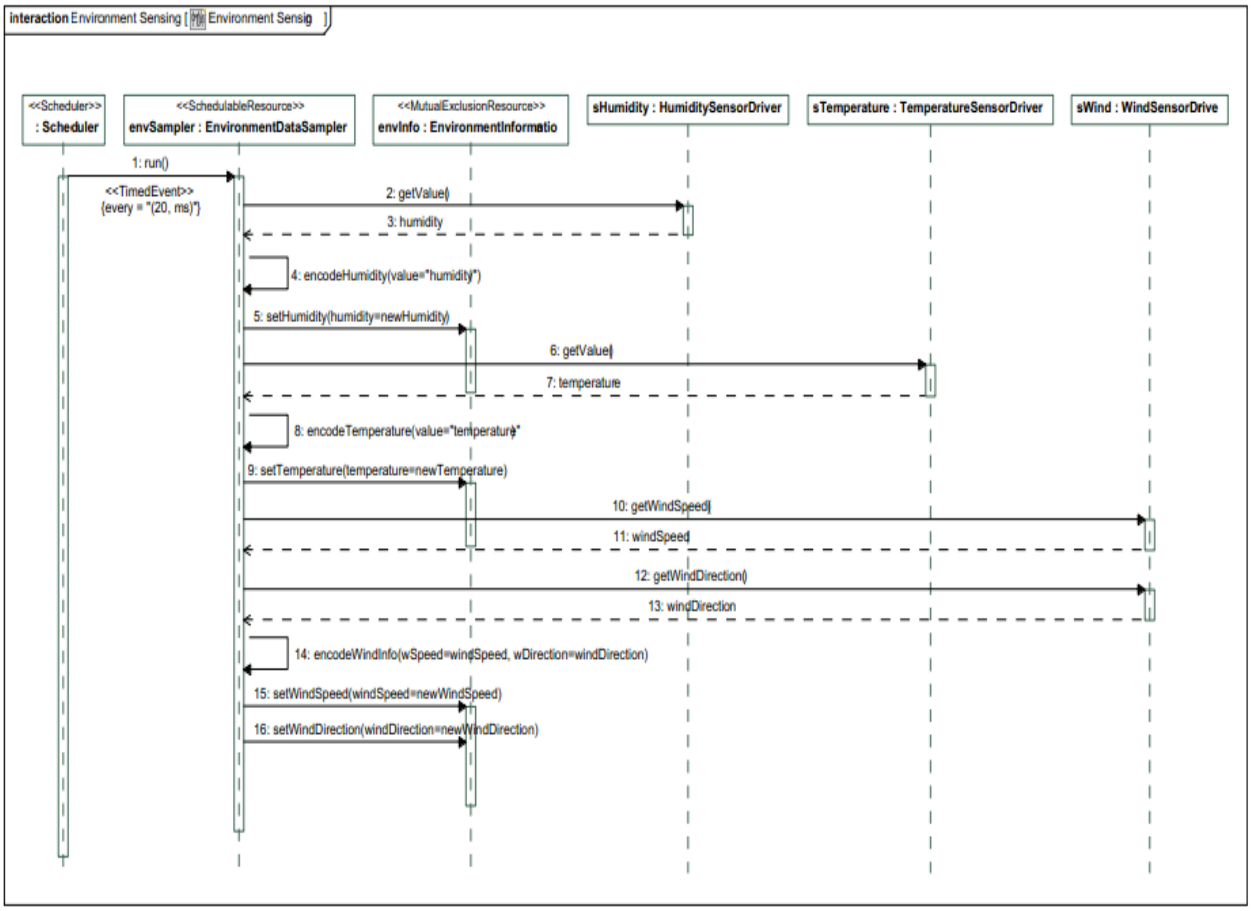

Figura 22: Diagrama de sequência.

Fonte: Wehrmeister, (2009).

Neste caso de teste (do Quadro 3), foram realizados três testes.

Teste 1: O primeiro teste é feito em cima do método *getValue*() que pertence à classe *HumiditySensorDriver*. Este teste é definido como teste unitário, pois tem como objetivo analisar uma unidade do sistema VANT. Como especificado no diagrama de sequência, este método tem como ação fazer o retorno do valor do atributo *humidity*. Desse modo, o objeto do qual pertence este atributo deve ser inicializado. Assim, valor definido no caso de teste para o atributo *humidity* foi definida no cenário de entrada sendo igual a 10 (linhas 18-22 do Quadro 3), através do objeto *sHumidity*. Com base nisso, foi definido um valor esperado e o sinal de comparação (linhas 50-52), onde vai ser comparado com o valor resultante da execução (valor retornado pelo FUMBeS) . Não foi definido

```
01 <TestSuit>
02  <TestCase id="TC_1">
03   <InputScenarion>
04    <Object name="envSampler"
class="EnvironmentDataSampler">
05     <Attribute name="newHumidity">
06      <Value>31</Value>
07     </Attribute>
08     <Attribute name="newTemperature">
09      <Value>32</Value>
10     </Attribute>
11     <Attribute name="newWindSpeed">
12      <Value>33</Value>
13     </Attribute>
14     <Attribute name="newWindDirection">
15      <Value>34</Value>
16     </Attribute>
17    </Object>
18    <Object name="sHumidity"
class="HumiditySensorDriver">
19     <Attribute name="Value">
20      <Value>10</Value>
21     </Attribute>
22    </Object>
23    <Object name="sTemperature"
class="TemperatureSensorDriver">
24      <Attribute name="Value">
25       <Value>20</Value>
26      </Attribute>
27     </Object>
28    <Object name="sWind"
class="WindSensorDriver">
29      <Attribute name="Value">
30       <Value>30</Value>
31      </Attribute>
32    </Object>
33    <Object name="envInfo"
class="EnvironmentInformation">
34     <Attribute name="Humidity">
35      <Value>500</Value>
36     </Attribute>
37     <Attribute name="WindSpeed">
38      <Value>600</Value>
39     </Attribute>
40     <Attribute name="WindDirection">
41      <Value>700</Value>
42     </Attribute>
43     <Attribute name="Temperature">
44      <Value>800</Value>
45     </Attribute>
46    </Object>
47   </InputScenarion>
48   <MethodsToTest>
49    <Method name="getValue"
obj="sHumidity" repetition="0"
stopIfFails="0" init="0">
50     <Assert>
51      <ReturnedValue
assertOperation="=">10</ReturnedValue>
52     </Assert>
53    </Method>
54    <Method name="getWindDirection"
obj="swind" repetition="0"
stopIfFails="0" init="0">
55     <Assert>
56      <ReturnedValue
assertOperation="=">0</ReturnedValue>
57     </Assert>
58    </Method>
59    <Method name="run" obj="envSampler"
repetition="0" stopIfFails="0" init="0">
60     <Assert>
61      <ExpectedScenarion>
62       <Object name="envInfo"
class="EnvironmentInformation">
63        <Attribute name="Humidity">
64         <Value>31</Value>
65        </Attribute>
66        <Attribute name="Temperature">
67         <Value>32</Value>
68        </Attribute>
69        <Attribute name="WindSpeed">
70         <Value>33</Value>
71        </Attribute>
72       <Attribute name="WindDirection">
73        <Value>34</Value>
74       </Attribute>
75   </Object>
76     </ExpectedScenarion>
77    </Assert>
78   </Method>
79  </MethodsToTest>
80  <Assert>
81   <ExpectedScenarion>
82    <Object name="envInfo"
class="EnvironmentInformation">
83     <Attribute name="Humidity">
84      <Value>31</Value>
85     </Attribute>
86     <Attribute name="Temperature">
87      <Value>32</Value>
88     </Attribute>
89     <Attribute name="WindSpeed">
90      <Value>33</Value>
91     </Attribute>
92     <Attribute name="WindDirection">
93      <Value>34</Value>
94     </Attribute>
95    </Object>
96   </ExpectedScenarion>
97  </Assert>
98 </TestCase>
99</TestSuit>
```

Quadro 3: Definição dos Casos de Teste.

um cenário esperado, pois a execução deste método não altera nenhum estado do objeto.

Teste 2: O próximo método testado é o *getWindDirection*() que pertence à classe *WindSensorDrive*. Este também é um teste unitário. Neste caso, o método tem um valor de retorno fixo, que é o resultado de uma combinação de iterações e condição. Para estipular o seu valor de retorno, foi preciso analisar o diagrama de sequência da Figura 23. Com base neste diagrama, o valor de retorno do método *getWindDirection*() deve ser igual a 0, que é o mesmo valor esperado definido no caso de teste (linha 53). Como este método não altera nenhum valor dos objetos, não é necessário definir um cenário esperado.

Teste 3: O último teste foi verificar o comportamento do método *run(),* da classe *EnvironmentDataSampler*. Este teste é definido como teste de integração, tendo o objetivo de testar o resultado do comportamento das interações entre os métodos representados no diagrama de sequência. O resultado desse teste, conforme o diagrama de sequência, foi transferir os valores dos atributos da classe *EnvironmentDataSampler* (linhas 4-17) para os atributos da classe *EnvironmentInformation* (linhas 33-46). Com base nisto, foi definido um cenário esperado após a execução deste método (linhas 60-77). Assim o testador do *software* poderá verificar os valores dos objetos antes e depois da execução do método através dos cenários esperado e o resultante, respectivamente. Como este método não retorna valor, não foi definido um valor esperado.

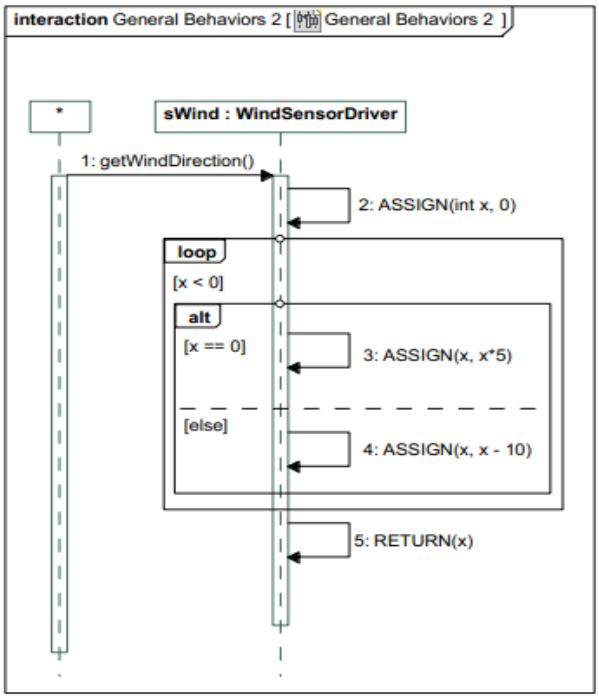

Figura 23: Comportamento do método *getWindDirection()*.
Fonte: Wehrmeister, (2009).

Nota-se que todas as classes que envolveram nos testes foram inicializadas com seus respectivos objetos (linhas 3-47 do Quadro 3), definindo dessa forma, o cenário de entrada do caso de teste.

Por fim, foi definido um cenário esperado (linhas 80-97) depois da execução de todos os métodos, relatando os valores esperados de cada atributo da classe *EnvironmentInformation.* Assim, junto ao cenário resultante dos testes, o testador pode fazer a comparação dos estados dos objetos.

6.3 Analisando o Relatório de Teste

Esta seção descreve avaliação dos resultados dos casos de teste através do relatório de teste gerado pela ferramenta AutoTeste. Baseando na metodologia de avaliação descrita na seção 6.1, esta avaliação analisa o resultado do caso de teste e o resultado de cada método executado, sendo que o último faz a análise da comparação entre o valor retornado e o esperado e/ou a comparação entre o cenário esperado e o

cenário resultante.

6.3.1 Analisando o Resultado do Caso de Teste

Conforme definido na metodologia de avaliação dos resultados, primeiramente é feito a análise do resultado dos cenários do caso de teste. A Figura 24 apresenta o cenário esperado (linhas 27-34 da Figura 24) e o cenário resultante (linha 35-67) do caso de teste realizado. O cenário resultante relata o resultado dos atributos de todos os objetos inicializados nos testes. Neste caso só nos interessa comparar os valores do objeto *envInfo*, conforme definido no cenário esperado. Podemos observar entre os cenários, que o valor do resultado dos atributos deste objeto foi conforme esperado.

```
27          <Expected>
28              <Object name="envInfo" class="EnvironmentInformation">
29                  <Attribute name="Humidity">31</Attribute>
30                  <Attribute name="Temperature">32</Attribute>
31                  <Attribute name="WindSpeed">33</Attribute>
32                  <Attribute name="WindDirection">34</Attribute>
33              </Object>
34          </Expected>
35          <Result>
36              <Object name="envSampler" class="EnvironmentDataSampler">
42              <Object name="sHumidity" class="HumiditySensorDriver">
45              <Object name="sTemperature" class="TemperatureSensorDriver">
48              <Object name="sWind" class="WindSensorDriver">
51              <Object name="envInfo" class="EnvironmentInformation">
52                  <Attribute name="Humidity">31.0</Attribute>
53                  <Attribute name="WindSpeed">33.0</Attribute>
54                  <Attribute name="WindDirection">34.0</Attribute>
55                  <Attribute name="Temperature">32.0</Attribute>
56              </Object>
57              <CreatedObjects class="EnvironmentDataSampler">0</CreatedObjects>
58              <DestructedObjects class="EnvironmentDataSampler">0</DestructedObjects>
59              <CreatedObjects class="HumiditySensorDriver">0</CreatedObjects>
60              <DestructedObjects class="HumiditySensorDriver">0</DestructedObjects>
61              <CreatedObjects class="TemperatureSensorDriver">0</CreatedObjects>
62              <DestructedObjects class="TemperatureSensorDriver">0</DestructedObjects>
63              <CreatedObjects class="WindSensorDriver">0</CreatedObjects>
64              <DestructedObjects class="WindSensorDriver">0</DestructedObjects>
65              <CreatedObjects class="EnvironmentInformation">0</CreatedObjects>
66              <DestructedObjects class="EnvironmentInformation">0</DestructedObjects>
67          </Result>
68      </Scenario>
```

Figura 24: Cenário Final do Relatório de teste.

O relatório de teste possibilita examinar quantos objetos foram criados e destruídos (linha 57-66) no teste realizado (envolvendo apenas as classes pertencentes aos objetos em teste). Neste caso não sofreu nenhuma alteração nas quantidades de objetos de cada classe.

Este capítulo são apresentadas figuras com partes do relatório de teste. O relatório por inteiro dos testes está em anexo.

6.3.2 Analisando o Resultado da Execução do Método *getValue()*

A segunda análise realizada é a comparação do valor de retorno dos métodos testados, caso o método tiver algum retorno. Conforme foi especificado no caso de teste, o primeiro método testado é o *getValue()* (relativo ao teste 1 da seção 6.2), que possui um valor de retorno igual a 10. A Figura 25 ilustra o relatório de teste que representa esta comparação.

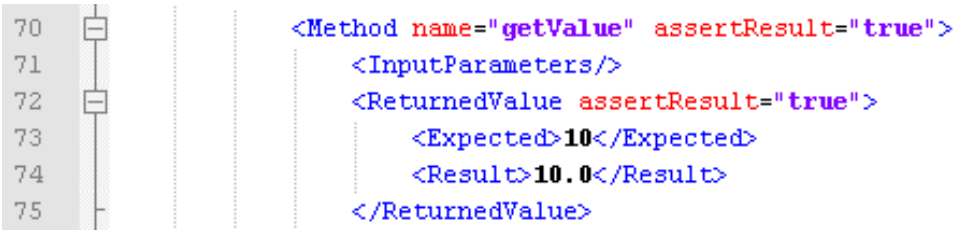

Figura 25: Valor retornado no relatório de teste do método *getValue()*.

O resultado da comparação entre o valor da execução do método *getValue()* e o valor esperado é representado pelo elemento *assertResult,* na linha 72 da Figura 25. Neste caso retorno verdadeiro, confirmando que o valor de retorno foi o esperado.

Na linha 70 é relatado o nome do método testado e o resultado do teste representado pelo elemento *assertResult*. Este elemento relata *true* quando a comparação do valor da execução do método com o valor esperado for verdadeira e a comparação dos cenários esperado e resultante for verdadeiro, caso contrário o valor é *false*.

De acordo com a metodologia de avaliação, a próxima análise deveria ser realizada entre o cenário esperado e o resultante da execução do método. Mas como

este teste não tem a intenção de realizar esta análise, não foi definido nenhum cenário esperado. Assim o valor do elemento *assertResult* da linha 70 é verdadeiro.

Podemos concluir então que o comportamento do método *getValue()* pertencente à classe *HumiditySensorDriver* está correto.

6.3.3 Analisando o Resultado da Execução do Método *getWindDirection()*

Seguindo a metodologia de avaliação, o mesmo procedimento é feito para o próximo método testado, que é o *getWindDirection()*. Como pode ser notado na Figura 26, ocorreu um erro neste método, pois o valor do elemento *assertResult* da linha 138 retornou *false*.

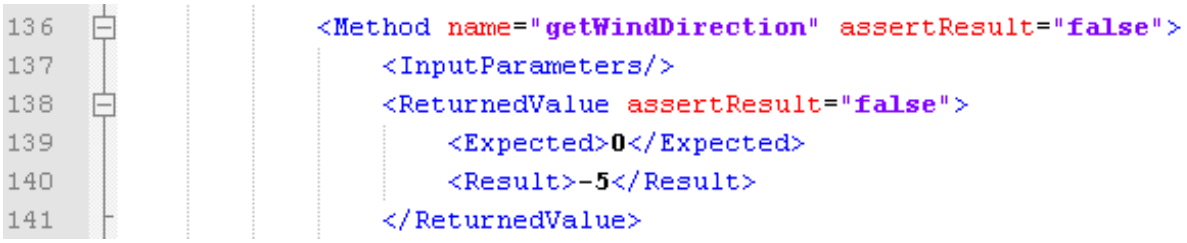

Figura 26: Valor retornado no relatório de teste do método *getWindDirection()*.

Como foi definido no teste 2, o valor esperado deveria ser igual a 0 (como definido no caso de teste no Quadro 3), mas o resultado da operação retornou -5, mostrando erro de comportamento desse método testado. Assim, o resultado da comparação do método em teste foi *false* (representado pelo elemento *assertResult* da linha 138). Como no teste anterior, não foi definido um cenário esperado.

Dessa maneira, a afirmação do resultado do teste (linha 136 da Figura 26) é falsa. Assim, permite-se concluir no sistema VANT representado no modelo UML definido pelo Wehrmeister (2008b), ocorreu um erro de comportamento em relação ao método *getWindDirection()* da classe *WindSensorDriver.*

6.3.4 Analisando o Resultado da Execução do Método run()

O próximo teste a ser avaliado é a execução do método run(). Para este, não foi definido nenhum valor de retorno, mas foi especificado um cenário esperado. Na Figura 27, o cenário resultante do método pode ser observado junto com o cenário esperado.

Figura 27: Resultado do teste do método *run()*.

Como neste caso só interessa comparar o estado do objeto *envInfo*, o valor dos outros objetos foram omitidos. Nota-se que nas linhas 206-228, relata o cenário de entrada definido no caso de teste.

Podemos observar então que o cenário esperado (linhas 229-236) e o cenário resultante do relatório de teste (linhas 237-258 da Figura 27) foram iguais, conforme o

valor dos seus atributos. Portanto, é concluso que o comportamento do método *run()* da classe *EnvironmentDataSampler* é correta. Na linha 205, o *assertResult* representa a comparação entre os cenários esperado e o resultante do método testado, neste caso sendo verdadeiro.

6.4 Considerações Finais

A aplicação de testes a partir de um modelo UML ajuda com que a geração de código do sistema em desenvolvimento seja livre de falha. Com a criação da ferramenta AutoTeste, foram executado casos de testes de forma automatizada, fazendo com que a fase de teste seja dinâmica.

Em relação à especificação dos casos de teste, com o conceito de cenários de testes, foi possível definir as informações fundamentais para aplicar testes e verificar a validade de cada método testado. Além disso, pode-se observar os estados dos objetos.

No tocante à execução dos casos de teste, este objetivo foi alcançado com sucesso. A ferramenta construída efetuou a leitura dos casos de teste para definir a modelagem da estrutura dos casos de teste (Figura 10). Através das ferramentas DERCS e FUMBeS, foram obtidas as informações correspondente ao sistema testado, simulando o comportamento de cada método testado. O Quadro 4 apresenta um resumo dos três testes executados pela ferramenta AutoTeste.

Com os testes aplicados neste capítulo, pode-se perceber que o sistema VANT contém erros de modelagem, por exemplo, o método *getWindDirection()* que não retorna um valor esperado com base nos requisitos do sistema testado. Já os métodos *getValue()* e *run()* tiveram o comportamento esperado.

	Método Testado	**Valor de Retorno Esperado**	**Cenário Esperado**	**Valor de Retorno do Método Testado**	**Resultado da Comparação entre o Valor Retornado e Valor Esperado**	**Resultado da Comparação entre Cenário Esperado e o Resultante**
Teste 1	*getValue*()	10	Não definido	10	*True*	_____
Teste 2	*getWindDirection*()	0	Não definido	-5	*False*	_____
Teste 3	*run*()	Não definido	Definido	_____	_____	*True*

Quadro 4: Testes realizados.

Com o relatório de teste gerado pela ferramenta desenvolvida neste trabalho, foi possível analisar, a partir de um modelo UML, o quanto um sistema embarcado é consistente com os resultados de seus comportamentos. Assim, cabendo ao testador do *software* verificar onde está o erro a partir do modelo UML e corrigir a falha.

CONCLUSÃO

A realização de testes de *software* em fase inicial de um sistema é tendência crescente entre projetistas e empresas que trabalham com desenvolvimento de *software*, com o objetivo de aumentar a confiabilidade e eficiência do sistema produzido. Um impacto desta metodologia é a redução de custo e de tempo do desenvolvimento do projeto final, pois desse modo as falhas dos sistemas são tratadas no início do projeto, podendo dessa maneira, atender a crescente demanda por software com boa qualidade.

Para abordar essas questões, neste trabalho foi desenvolvida uma ferramenta chamada AutoTeste capaz de executar casos de teste em *softwares* de forma automatizada. O foco da ferramenta consiste em realizar testes em sistemas embarcados na fase inicial do seu desenvolvimento.

Inicialmente o sistema testado foi definido em um modelo UML. A partir de uma conversão desse modelo para um meta modelo chamado DERCS, é possível representar o sistema embarcado com suas funcionalidades e características estruturais. Após a importação deste meta modelo foram feitas simulações comportamentais com o auxílio da ferramenta FUMBeS. Assim, os testes aplicados neste trabalho se baseiam no resultado do comportamento dos métodos testados.

Por meio de um modelo estrutural e de um *parser* realizado em um arquivo de entrada do tipo XML, são obtidas as informações dos casos de testes e realizadas a sua execução de forma automática.

Quanto aos resultados atingidos, foram testados o comportamento de alguns métodos de um sistema usado como caso de estudo. Com os comportamentos e valores resultantes das execuções dos métodos (processo realizado pela ferramenta FUMBeS) juntamente com as informações definidas nos casos de teste, foram realizadas comparações para analisar a validade dos testes. Para o correto funcionamento da ferramenta, não pode conter informações inválidas nos casos de teste em relação ao sistema em teste (i.e. nome do objeto errado).

As análises foram feitas através do relatório de teste gerado pela ferramenta AutoTeste. Com base nas informações colhidas dos métodos testados foi avaliada a consistência do sistema em relação ao definido no modelo UML.

Tendo em vista os objetivos deste trabalho, pode-se dizer que a ferramenta AutoTeste cumpre com êxito as tarefas de automatizar a execução dos casos de teste.

Para atingir esta tarefa, foi preciso seguir alguns procedimentos, dentre os principais estão: efetuar a leitura dos casos de teste, realizar a execução dos testes, gerar o relatório de resultado dos testes.

Algumas limitações foram encontradas no desenvolvimento do projeto. A ferramenta AutoTeste só permite informar, em um determinado teste, a criação ou a destruição dos objetos de classes especificadas no início do teste, não podendo avaliar estas informações sobre o restante dos objetos de outras classes no decorrer da execução do método. Além disso, não há possibilidade de se realizar testes com dois ou mais objetos com os mesmos nomes. Isso se dá pelo fato da versão atual da transformação UML-DERCS não possuir informações suficientes para diferenciar quais dos objetos está fazendo parte do teste somente com base no nome do objeto.

Acerca dos possíveis trabalhos futuros, pode-se sugerir a adição de novas funcionalidades na ferramenta desenvolvida neste trabalho para obter mais informações e com maior precisão sobre os testes realizados. Como por exemplo, fornecer informações dos objetos criados ou destruídos de um teste, dentre elas seus atributos. Outra sugestão é a possibilidade de exibir o relatório do cenário sobre cada iteração ao ser determinado a execução repetitiva de um método.

A realização de testes visando avaliar o comportamento de alguns requisitos não-funcionais de um sistema embarcado, melhoria a abrangência e variedade de testes aumentando a possibilidade de encontrar possíveis falhas do sistema. Além disso, para o usuário poder visualizar o relatório de teste de forma mais clara, arquivos HTML ou RTF poderiam ser gerados a partir do arquivo XML. Para tal fim um arquivo XSLT pode ser utilizado. Da mesma forma, visando o desenvolvimento ágil, outras ferramentas tais como analisadores de métricas, ferramentas de documentação, podem ser desenvolvidas de forma a utilizar a saída em XML para realizarem seus propósitos.

Referências

ALMEIDA, C. **Introdução ao Teste de Software**. 2010. Disponível em: http://www.linhadecodigo.com.br/artigo/2775/introducao-ao-teste-de-software.aspx. Acessado em: 27 out.2011.

BARBOSA, D. **Um Método Automático de Teste Funcional para a Verificação de Componentes,** 2005. 192 p. Dissertação de Mestrado. Universidade Federal de Campinas Grande - Ciência da Computação – Campus I. Disponível em: http://docs.computacao.ufcg.edu.br/posgraduacao/dissertacoes/2005/Dissertacao_DanielLimaBarbosa.pdf. Acessado em: Julho 2012.

BURNSTEIN, I. **Practical Software Testing:** A Process-oriented Approach**.** Publisher: Springer-Verlag. Nova York, 2002. 699 p. Department of Computer Science. Illinois Institute of Technology.

CATELANI *et. al.* **Teste de *Software* Automatizado:** Uma Solução para Maximizar a Cobertura de Plano de Teste e Aumentar a Confiabilidade do *Software* e Qualidade em Uso. [S.l] : Computer Standards & Interfaces. Volume 33, páginas 152-158, 2011.

CLÁUDIO, A. **Introdução ao Teste de Software.** Editora ClubeDelphi. Brasil. Engenharia de Software Magazine. 54-59 p. edição1, 2007.

COSTA, M. **Estratégia de Automação em Testes:** Requisitos, Arquitetura e Acompanhamento de sua Implantação**,** 2004. 93 p. Dissertação de Mestrado. Instituto de Computação – Universidade Estadual de Campinas.

DELAMARO M., MALDONADO J., JINO M. **Introdução ao Teste de Software**. Editora Campus, Rio de Janeiro, primeira Edição, 2007.

DIAS L., MENNA R. **Teste de Desempenho a partir de Modelos UML para Componentes de Software,** 2008. 60 p. Monografia. Departamento de Informática Pontifícia Universidade Católica do Rio Grande do Sul.

JFERN 2006, R. **Framework para Redes de Petri** . 2006. Disponível em: http://sourceforge.net/projects/jfern. Acessado em: Junho 2012.

JMETER 2007, A. **Apache Jmeter**. 2007. Disponível em: http://jakarta.apache.org/jmeter/. Acessado em: Junho 2012.

KUHN R., *et*. *al.* **Combinatorial Software Testing.** IEEE Computer Society, p 94-96, 2009.

MACORATTI, J. **UML - Conceitos Básicos II**. 2011. Disponível em: http://www.macoratti.net/vb_uml2.htm. Acessado em: 30 nov.2011.

MODELLER 2007, U. **UML Modeller Umbrello**. 2007. Disponível em: http://uml.sourceforge.net/index.php. Acessado em: Junho 2012.

GLENFORD J.. **The Art of Software Testing**. Publisher John Wiley & Sons, New York, Second Edition, 2004.

NEVES, V**. Uma Visão Geral de Teste de Sistemas Embarcados Com Enfoque no Teste Estrutural**. [S.l]. Instituto de Ciências Matemáticas e de Computação Universidade de São Paulo , São Carlos, 2010.

OROZCO A. *et al*. **Derivações de casos de testes funcionais: uma abordagem baseada em modelos UML.** 2009. Revista Eletrônica de Sistemas de Informação. Volume 8, n. 1, artigo 3.

PRESSMAN, S. **Engenharia de Software**. [S.l.] : McGraw Hill, quinta dição, 2002.

RAINSBERGER J., STIRLING S. **JUnit Recipes**: Practical Methods for Programmer Testing. CT: Manning Publications, Co. Greenwich, 2004.

RUP Rational Software Corporation**. Rational Unified Process®, RUP.** 2001. Disponível em http://www.wthreex.com/rup/. Acessado em outubro de 2011.

SANTOS, V. **Automação de Testes**. 2011. Disponível em: http://www.artigonal.com/programacao-artigos/automacao-de-testes-4665417.html. Acessado em: 27 nov.2011.

SOMMERVILLE, I**. Engenharia de Software**. [S.l.] : Editora Pearson Education, oitava edição. São Paulo, 2006.

SYLLABUS, **Certificação em Teste, item 1.2. Foundation Level Syllabus**. Versão 2007br. BSTQ – Brazilian Software Testing Qualifications Board.

TOZELLI, P. **Processo de Teste de Software**. 2008. Disponível em: http://www.artigonal.com/programacao-artigos/processo-de-teste-de-software-505672.html. Acessado em: out.2011.

WEHRMEISTER, M. **Uma Abordagem para Automação de Testes Funcionais em Modelos UML para Software Embarcado**. 2010. 32 p. Trabalho Individual. Relatório de Projeto de Pós-Doutorado. - Programa de Pós-Graduação em Engenharia de Automação e Sistemas. Universidade Federal de Santa Catarina, Florianópolis.

WEHRMEISTER M., PACKER J., CERON L. **Framework to Simulate the Behavior of Embedded Real-TimeSystems Specified in UML Models**, In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SISTEMAS COMPUTACIONAIS, 2011. **Anais …** Florianópolis: Editora da SBC, 2011, p.1-12.

Wehrmeister M, *et al*. **A Case Study to Evaluate Pros/Cons of Aspect- and Object-Oriented Paradigms to Model Distributed Embedded Real-Time Systems**, 2008. In: 5th International Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES), IEEE Computer Society Press, p. 44-54.

WEHRMEISTER, A. **An Aspect-Oriented Model-Driven Engineering Approach for Distributed Embedded Real-Time Systems**. 2009. 206 p. Tese de Doutorado. Universidade Federal do Rio Grande do Sul – Programa de Pós-Graduação em Computação, Porto Alegre.

XML 2007, S**. Simple API for XML**. 2007. Disponível em: http://www.saxproject.org/ Acessado em: Junho 2012.

ANEXOS

1. CÓDIGOS DA FERRAMENTA AUTOTESTE

```
public class Parseando {

    public TestSuit parseTestSuit() throws Exception{
        DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
        DocumentBuilder db = dbf.newDocumentBuilder();
        Document doc = db.parse("TestCase.xml");
        doc.normalize();

        Element raiz = doc.getDocumentElement();
        NodeList children = raiz.getChildNodes();
        TestSuit ts = new TestSuit();

        for (int i = 0; i < children.getLength(); i++) {
            Node n = children.item(i);
            if( n.getNodeType() == Node.ELEMENT_NODE && n.getNodeName() == "TestCase" ){
                TestCase tc = parseTestCase(n);
                ts.getTc().add(tc);
            }
        }
        return ts;
    }
```

Leitura dos Casos de testes.

```
public Model iniciarObjetos( Model modelo, TestCase tc){

 for( int j = 0; j < tc.getScInput().getObj().size(); j++ ){
   ArrayList<dercs.structure.runtime.Object> listObj = modelo.getObjects();
   for( int m = 0; m<listObj.size(); m++){
     if(listObj.get(m) != null){

       if(listObj.get(m).getName().equals(tc.getScInput().getObj().get(j).getNome())){
      //atribui os valores para cada objeto
        for( int k = 0; k < tc.getScInput().getObj().get(j).getAtt().size(); k++ ){

       if(listObj.get(m).getRuntimeInformation(tc.getScInput().getObj().get(j).
       getAtt().get(k).getNome()) == null ){

              listObj.get(m).createRuntimeInformationForAttributes();
         }
           //define o valor ao atributo
           listObj.get(m).setValue(
           tc.getScInput().getObj().get(j).getAtt().get(k).getNome(),
           tc.getScInput().getObj().get(j).getAtt().get(k).getValor());
         }
       }
      }
     }
   }
  return modelo;
}
```

Inicializando os Valores dos Objetos.

```
public void printInfo(TestSuit ts) {
    BufferedWriter writer = null;
    try {
        //passo1 : criar instancia do objeto
        writer = new BufferedWriter(new FileWriter(new File("saida.xml")));

        writer.write("<TestSuit>");
        //passo2: acessar os dados do modelo de casos de teste
        for(int i=0; i < ts.getTc().size(); i++){
            writer.write("<TestCase" + " id=" + "\""+ ts.getTc().get(i).getId()+ "\""+">");

            writer.write("<Scenario>");
            printScenarioInput(ts.getTc().get(i).getScInput(), writer);
            printScenarioExpected(ts.getTc().get(i).getAst().getScenarioExpected(), writer);
            printScenarioResult(ts.getTc().get(i).getScResult(), writer);
            writer.write("</Scenario>");
```

Gerando Relatório dos Testes.

```
public void teste(){

    try {
        //fazer o parser dos casos de teste
        Parseando parseando = new Parseando();
        TestSuit ts = parseando.parseTestSuit();

        //laco de repeticao para percorrer todos os casos de teste do modelo
        for( int i = 0; i < ts.getTc().size(); i++){
            Model uavDERCSModel = null;
            //obtem o caso de teste corrente
            TestCase tc = ts.getTc().get(i);
            try {
                //carrega o modelo e executa o caso de testes
                uavDERCSModel = Model.loadFrom("dercs_model.bin");
                //executa o caso de teste
                tc = executar(uavDERCSModel, tc);
            }
            catch (Exception e) {
                System.out.println("Error: " + e.getMessage() + "\n\n");
                e.printStackTrace();
            }
        }
        //gerar o relatorio de saida do resultado dos testes
        Desparseando desparser = new Desparseando();
        desparser.printInfo(ts);
    }
    catch (Exception e1) {
        e1.printStackTrace();
    }
}
```

Execução dos casos de teste.

2. CASOS DE TESTE

```
<TestSuit>
    <TestCase id="TC_1">
        <InputScenarion>
            <Object name="envSampler" class="EnvironmentDataSampler">
                <Attribute name="newHumidity">
                    <Value>31</Value>
                </Attribute>
                <Attribute name="newTemperature">
                    <Value>32</Value>
                </Attribute>
                <Attribute name="newWindSpeed">
                    <Value>33</Value>
                </Attribute>
                <Attribute name="newWindDirection">
                    <Value>34</Value>
                </Attribute>
            </Object>
            <Object name="sHumidity" class="HumiditySensorDriver">
                <Attribute name="Value">
                    <Value>10</Value>
                </Attribute>
            </Object>
            <Object name="sTemperature" class="TemperatureSensorDriver">
                <Attribute name="Value">
                    <Value>20</Value>
                </Attribute>
            </Object>
            <Object name="sWind" class="WindSensorDriver">
                <Attribute name="Value">
                    <Value>30</Value>
                </Attribute>
            </Object>
            <Object name="envInfo" class="EnvironmentInformation">
                <Attribute name="Humidity">
                    <Value>500</Value>
```

```
        </Attribute>
        <Attribute name="WindSpeed">
            <Value>600</Value>
        </Attribute>
        <Attribute name="WindDirection">
            <Value>700</Value>
        </Attribute>
        <Attribute name="Temperature">
            <Value>800</Value>
        </Attribute>
    </Object>
</InputScenarion>
<MethodsToTest>
    <Method name="getValue" obj="sHumidity" repetition="0" stopIfFails="0" init="0">
        <Assert>
            <ReturnedValue assertOperation="=">10</ReturnedValue>
        </Assert>
    </Method>

    <Method name="getWindDirection" obj="sWind" repetition="0" stopIfFails="0" init="0">
        <Assert>
            <ReturnedValue assertOperation="=">0</ReturnedValue>
        </Assert>
    </Method>
    <Method name="run" obj="envSampler" repetition="0" stopIfFails="0" init="0">
        <Assert>
            <ExpectedScenarion>
                <Object name="envInfo" class="EnvironmentInformation">
                    <Attribute name="Humidity">
                        <Value>31</Value>
                    </Attribute>
                    <Attribute name="Temperature">
                        <Value>32</Value>
                    </Attribute>
                    <Attribute name="WindSpeed">
                        <Value>33</Value>
                    </Attribute>
                    <Attribute name="WindDirection">
```

```
                        <Value>34</Value>
                    </Attribute>
                </Object>
            </ExpectedScenarion>
        </Assert>
    </Method>
</MethodsToTest>
<Assert>
    <ExpectedScenarion>
        <Object name="envInfo" class="EnvironmentInformation">
            <Attribute name="Humidity">
                <Value>31</Value>
            </Attribute>
            <Attribute name="Temperature">
                <Value>32</Value>
            </Attribute>
            <Attribute name="WindSpeed">
                <Value>33</Value>
            </Attribute>
            <Attribute name="WindDirection">
                <Value>34</Value>
            </Attribute>
        </Object>
    </ExpectedScenarion>
</Assert>
</TestCase>
<TestCase id="TC_2">
<InputScenarion>
    <Object name="envSampler" class="EnvironmentDataSampler">
        <Attribute name="newHumidity">
            <Value>1</Value>
        </Attribute>
        <Attribute name="newTemperature">
            <Value>2</Value>
        </Attribute>
        <Attribute name="newWindSpeed">
            <Value>3</Value>
        </Attribute>
```

```
            <Attribute name="newWindDirection">
                <Value>4</Value>
            </Attribute>
        </Object>
        <Object name="sHumidity" class="HumiditySensorDriver">
            <Attribute name="Value">
                <Value>100</Value>
            </Attribute>
        </Object>
        <Object name="sTemperature" class="TemperatureSensorDriver">
            <Attribute name="Value">
                <Value>200</Value>
            </Attribute>
        </Object>
        <Object name="sWind" class="WindSensorDriver">
            <Attribute name="Value">
                <Value>300</Value>
            </Attribute>
        </Object>
        <Object name="envInfo" class="EnvironmentInformation">
            <Attribute name="Humidity">
                <Value>45</Value>
            </Attribute>
            <Attribute name="WindSpeed">
                <Value>55</Value>
            </Attribute>
            <Attribute name="WindDirection">
                <Value>65</Value>
            </Attribute>
            <Attribute name="Temperature">
                <Value>75</Value>
            </Attribute>
        </Object>
    </InputScenarion>
    <MethodsToTest>
        <Method name="getWindSpeed" obj="sWind" repetition="0" stopIfFails="0" init="0">
            <Assert>
                <ReturnedValue assertOperation="=">15</ReturnedValue>
```

```
            </Assert>
        </Method>
        <Method name="setHumidity" obj="envInfo" repetition="0" stopIfFails="0" init="0">
            <InputParameters>
              <Parameter name="humidity">
            <Value>99</Value>
              </Parameter>
             </InputParameters>
             <Assert>
              <ExpectedScenarion>
                <Object name="envInfo" class="EnvironmentInformation">
                  <Attribute name="Humidity" assertOperation="=">99</Attribute>
                 </Object>
            </ExpectedScenarion>
             </Assert>
      </Method>
     </MethodsToTest>
  </TestCase>
  </TestCase>
</TestSuit>
```

3. RELATÓRIO DOS CASOS DE TESTE.

```
<TestSuit>
    <TestCase id="TC_1">
        <Scenario>
            <Input>
                <Object name="envSampler" class="EnvironmentDataSampler">
                    <Attribute name="newHumidity">31</Attribute>
                    <Attribute name="newTemperature">32</Attribute>
                    <Attribute name="newWindSpeed">33</Attribute>
                    <Attribute name="newWindDirection">34</Attribute>
                </Object>
                <Object name="sHumidity" class="HumiditySensorDriver">
                    <Attribute name="Value">10</Attribute>
                </Object>
                <Object name="sTemperature" class="TemperatureSensorDriver">
```

```
            <Attribute name="Value">20</Attribute>
        </Object>
        <Object name="sWind" class="WindSensorDriver">
            <Attribute name="Value">30</Attribute>
        </Object>
        <Object name="envInfo" class="EnvironmentInformation">
            <Attribute name="Humidity">500</Attribute>
            <Attribute name="WindSpeed">600</Attribute>
            <Attribute name="WindDirection">700</Attribute>
            <Attribute name="Temperature">800</Attribute>
        </Object>
    </Input>
    <Expected>
        <Object name="envInfo" class="EnvironmentInformation">
            <Attribute name="Humidity">31</Attribute>
            <Attribute name="Temperature">32</Attribute>
            <Attribute name="WindSpeed">33</Attribute>
            <Attribute name="WindDirection">34</Attribute>
        </Object>
    </Expected>
    <Result>
        <Object name="envSampler" class="EnvironmentDataSampler">
            <Attribute name="newHumidity">31</Attribute>
            <Attribute name="newTemperature">32</Attribute>
            <Attribute name="newWindSpeed">33</Attribute>
            <Attribute name="newWindDirection">34</Attribute>
        </Object>
        <Object name="sHumidity" class="HumiditySensorDriver">
            <Attribute name="Value">10</Attribute>
        </Object>
        <Object name="sTemperature" class="TemperatureSensorDriver">
            <Attribute name="Value">20</Attribute>
        </Object>
        <Object name="sWind" class="WindSensorDriver">
            <Attribute name="Value">30</Attribute>
        </Object>
        <Object name="envInfo" class="EnvironmentInformation">
            <Attribute name="Humidity">31.0</Attribute>
```

```
            <Attribute name="WindSpeed">33.0</Attribute>
            <Attribute name="WindDirection">34.0</Attribute>
            <Attribute name="Temperature">32.0</Attribute>
        </Object>
        <CreatedObjects class="EnvironmentDataSampler">0</CreatedObjects>
        <DestructedObjects class="EnvironmentDataSampler">0</DestructedObjects>
        <CreatedObjects class="HumiditySensorDriver">0</CreatedObjects>
        <DestructedObjects class="HumiditySensorDriver">0</DestructedObjects>
        <CreatedObjects class="TemperatureSensorDriver">0</CreatedObjects>
        <DestructedObjects class="TemperatureSensorDriver">0</DestructedObjects>
        <CreatedObjects class="WindSensorDriver">0</CreatedObjects>
        <DestructedObjects class="WindSensorDriver">0</DestructedObjects>
        <CreatedObjects class="EnvironmentInformation">0</CreatedObjects>
        <DestructedObjects class="EnvironmentInformation">0</DestructedObjects>
    </Result>
</Scenario>
<Methods>
    <Method name="getValue" assertResult="true">
        <InputParameters/>
        <ReturnedValue assertResult="true">
            <Expected>10</Expected>
            <Result>10.0</Result>
        </ReturnedValue>
        <Scenario assertResult="">
            <Input>
                <Object name="envSampler" class="EnvironmentDataSampler">
                    <Attribute name="newHumidity">31</Attribute>
                    <Attribute name="newTemperature">32</Attribute>
                    <Attribute name="newWindSpeed">33</Attribute>
                    <Attribute name="newWindDirection">34</Attribute>
                </Object>
                <Object name="sHumidity" class="HumiditySensorDriver">
                    <Attribute name="Value">10</Attribute>
                </Object>
                <Object name="sTemperature" class="TemperatureSensorDriver">
                    <Attribute name="Value">20</Attribute>
                </Object>
                <Object name="sWind" class="WindSensorDriver">
```

```
            <Attribute name="Value">30</Attribute>
        </Object>
        <Object name="envInfo" class="EnvironmentInformation">
            <Attribute name="Humidity">500</Attribute>
            <Attribute name="WindSpeed">600</Attribute>
            <Attribute name="WindDirection">700</Attribute>
            <Attribute name="Temperature">800</Attribute>
        </Object>
    </Input>
    <Expected/>
    <Result>
        <Object name="envSampler" class="EnvironmentDataSampler">
            <Attribute name="newHumidity">31</Attribute>
            <Attribute name="newTemperature">32</Attribute>
            <Attribute name="newWindSpeed">33</Attribute>
            <Attribute name="newWindDirection">34</Attribute>
        </Object>
        <Object name="sHumidity" class="HumiditySensorDriver">
            <Attribute name="Value">10</Attribute>
        </Object>
        <Object name="sTemperature" class="TemperatureSensorDriver">
            <Attribute name="Value">20</Attribute>
        </Object>
        <Object name="sWind" class="WindSensorDriver">
            <Attribute name="Value">30</Attribute>
        </Object>
        <Object name="envInfo" class="EnvironmentInformation">
            <Attribute name="Humidity">500</Attribute>
            <Attribute name="WindSpeed">600</Attribute>
            <Attribute name="WindDirection">700</Attribute>
            <Attribute name="Temperature">800</Attribute>
        </Object>
        <CreatedObjects class="EnvironmentDataSampler">0</CreatedObjects>
        <DestructedObjects class="EnvironmentDataSampler">0</DestructedObjects>
        <CreatedObjects class="HumiditySensorDriver">0</CreatedObjects>
        <DestructedObjects class="HumiditySensorDriver">0</DestructedObjects>
        <CreatedObjects class="TemperatureSensorDriver">0</CreatedObjects>
        <DestructedObjects class="TemperatureSensorDriver">0</DestructedObjects>
```

```
            <CreatedObjects class="WindSensorDriver">0</CreatedObjects>
            <DestructedObjects class="WindSensorDriver">0</DestructedObjects>
            <CreatedObjects class="EnvironmentInformation">0</CreatedObjects>
            <DestructedObjects class="EnvironmentInformation">0</DestructedObjects>
        </Result>
    </Scenario>
</Method>
<Method name="getWindDirection" assertResult="false">
    <InputParameters/>
    <ReturnedValue assertResult="false">
        <Expected>0</Expected>
        <Result>-5</Result>
    </ReturnedValue>
    <Scenario assertResult="">
        <Input>
            <Object name="envSampler" class="EnvironmentDataSampler">
                <Attribute name="newHumidity">31</Attribute>
                <Attribute name="newTemperature">32</Attribute>
                <Attribute name="newWindSpeed">33</Attribute>
                <Attribute name="newWindDirection">34</Attribute>
            </Object>
            <Object name="sHumidity" class="HumiditySensorDriver">
                <Attribute name="Value">10</Attribute>
            </Object>
            <Object name="sTemperature" class="TemperatureSensorDriver">
                <Attribute name="Value">20</Attribute>
            </Object>
            <Object name="sWind" class="WindSensorDriver">
                <Attribute name="Value">30</Attribute>
            </Object>
            <Object name="envInfo" class="EnvironmentInformation">
                <Attribute name="Humidity">500</Attribute>
                <Attribute name="WindSpeed">600</Attribute>
                <Attribute name="WindDirection">700</Attribute>
                <Attribute name="Temperature">800</Attribute>
            </Object>
        </Input>
        <Expected/>
```

```
        <Result>
            <Object name="envSampler" class="EnvironmentDataSampler">
                <Attribute name="newHumidity">31</Attribute>
                <Attribute name="newTemperature">32</Attribute>
                <Attribute name="newWindSpeed">33</Attribute>
                <Attribute name="newWindDirection">34</Attribute>
            </Object>
            <Object name="sHumidity" class="HumiditySensorDriver">
                <Attribute name="Value">10</Attribute>
            </Object>
            <Object name="sTemperature" class="TemperatureSensorDriver">
                <Attribute name="Value">20</Attribute>
            </Object>
            <Object name="sWind" class="WindSensorDriver">
                <Attribute name="Value">30</Attribute>
            </Object>
            <Object name="envInfo" class="EnvironmentInformation">
                <Attribute name="Humidity">500</Attribute>
                <Attribute name="WindSpeed">600</Attribute>
                <Attribute name="WindDirection">700</Attribute>
                <Attribute name="Temperature">800</Attribute>
            </Object>
            <CreatedObjects class="EnvironmentDataSampler">0</CreatedObjects>
            <DestructedObjects class="EnvironmentDataSampler">0</DestructedObjects>
            <CreatedObjects class="HumiditySensorDriver">0</CreatedObjects>
            <DestructedObjects class="HumiditySensorDriver">0</DestructedObjects>
            <CreatedObjects class="TemperatureSensorDriver">0</CreatedObjects>
            <DestructedObjects class="TemperatureSensorDriver">0</DestructedObjects>
            <CreatedObjects class="WindSensorDriver">0</CreatedObjects>
            <DestructedObjects class="WindSensorDriver">0</DestructedObjects>
            <CreatedObjects class="EnvironmentInformation">0</CreatedObjects>
            <DestructedObjects class="EnvironmentInformation">0</DestructedObjects>
        </Result>
    </Scenario>
</Method>
<Method name="run" assertResult="true">
    <InputParameters/>
    <ReturnedValue assertResult=""/>
```

```
<Scenario assertResult="true">
    <Input>
        <Object name="envSampler" class="EnvironmentDataSampler">
            <Attribute name="newHumidity">31</Attribute>
            <Attribute name="newTemperature">32</Attribute>
            <Attribute name="newWindSpeed">33</Attribute>
            <Attribute name="newWindDirection">34</Attribute>
        </Object>
        <Object name="sHumidity" class="HumiditySensorDriver">
            <Attribute name="Value">10</Attribute>
        </Object>
        <Object name="sTemperature" class="TemperatureSensorDriver">
            <Attribute name="Value">20</Attribute>
        </Object>
        <Object name="sWind" class="WindSensorDriver">
            <Attribute name="Value">30</Attribute>
        </Object>
        <Object name="envInfo" class="EnvironmentInformation">
            <Attribute name="Humidity">500</Attribute>
            <Attribute name="WindSpeed">600</Attribute>
            <Attribute name="WindDirection">700</Attribute>
            <Attribute name="Temperature">800</Attribute>
        </Object>
    </Input>
    <Expected>
        <Object name="envInfo" class="EnvironmentInformation">
            <Attribute name="Humidity">31</Attribute>
            <Attribute name="Temperature">32</Attribute>
            <Attribute name="WindSpeed">33</Attribute>
            <Attribute name="WindDirection">34</Attribute>
        </Object>
    </Expected>
    <Result>
        <Object name="envSampler" class="EnvironmentDataSampler">
            <Attribute name="newHumidity">31</Attribute>
            <Attribute name="newTemperature">32</Attribute>
            <Attribute name="newWindSpeed">33</Attribute>
            <Attribute name="newWindDirection">34</Attribute>
```

```
                </Object>
                <Object name="sHumidity" class="HumiditySensorDriver">
                    <Attribute name="Value">10</Attribute>
                </Object>
                <Object name="sTemperature" class="TemperatureSensorDriver">
                    <Attribute name="Value">20</Attribute>
                </Object>
                <Object name="sWind" class="WindSensorDriver">
                    <Attribute name="Value">30</Attribute>
                </Object>
                <Object name="envInfo" class="EnvironmentInformation">
                    <Attribute name="Humidity">31.0</Attribute>
                    <Attribute name="WindSpeed">33.0</Attribute>
                    <Attribute name="WindDirection">34.0</Attribute>
                    <Attribute name="Temperature">32.0</Attribute>
                </Object>
                <CreatedObjects class="EnvironmentDataSampler">0</CreatedObjects>
                <DestructedObjects class="EnvironmentDataSampler">0</DestructedObjects>
                <CreatedObjects class="HumiditySensorDriver">0</CreatedObjects>
                <DestructedObjects class="HumiditySensorDriver">0</DestructedObjects>
                <CreatedObjects class="TemperatureSensorDriver">0</CreatedObjects>
                <DestructedObjects class="TemperatureSensorDriver">0</DestructedObjects>
                <CreatedObjects class="WindSensorDriver">0</CreatedObjects>
                <DestructedObjects class="WindSensorDriver">0</DestructedObjects>
                <CreatedObjects class="EnvironmentInformation">0</CreatedObjects>
                <DestructedObjects class="EnvironmentInformation">0</DestructedObjects>
            </Result>
        </Scenario>
    </Method>
  </Methods>
 </TestCase>
</TestSuit>
```