UNIVERSIDADE FEDERAL DO MARANHÃO
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA DA ELETRICIDADE

JÉSSICA BASSANI DE OLIVEIRA

UMA ABORDAGEM BASEADA EM ENGENHARIA DIRIGIDA POR MODELOS PARA SUPORTAR O TESTE DE SISTEMAS DE SOFTWARE NA PLATAFORMA DE COMPUTAÇÃO EM NUVEM

São Luís
2012

JÉSSICA BASSANI DE OLIVEIRA

UMA ABORDAGEM BASEADA EM ENGENHARIA DIRIGIDA POR MODELOS PARA SUPORTAR O TESTE DE SISTEMAS DE SOFTWARE NA PLATAFORMA DE COMPUTAÇÃO EM NUVEM

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia da Eletricidade da Universidade Federal do Maranhão, para a obtenção do título de mestre em Engenharia da Eletricidade - Área de Concentração: Ciência da Computação.

Orientador: Ph. D. Zair Abdelouahab

Co-Orientador: Dr.Denivaldo Lopes

São Luís

2012

Oliveira, Jéssica Bassani

UMA ABORDAGEM BASEADA EM ENGENHARIA DIRIGIDA POR MODELOS PARA SUPORTAR O TESTE DE SISTEMAS DE SOFTWARE NA PLATAFORMA DE COMPUTAÇÃO EM NUVEM / Jéssica Bassani Oliveira. – São Luís, 2012.

192 f.

Impresso por computador (fotocópia).

Orientador: Ph. D. Zair Abdelouahab.

Co-Orientador: Dr.Denivaldo Lopes.

Dissertação (Mestrado) – Universidade Federal do Maranhão, Programa de Pós-Graduação em Engenharia da Eletricidade . São Luís, 2012.

1. Software - Engenharia. 2. Engenharia Dirigida por Modelos. 3. Computação em Nuvem I. Título.

CDU 004.41

UMA ABORDAGEM BASEADA EM ENGENHARIA DIRIGIDA POR MODELOS PARA SUPORTAR O TESTE DE SISTEMAS DE SOFTWARE NA PLATAFORMA DE COMPUTAÇÃO EM NUVEM

Jéssica Bassani de Oliveira

Dissertação aprovada em 21 de dezembro de 2012.

Prof. Zair Abdelouahab, Ph.D
(Orientador)

Prof. Denivaldo Cícero Pavão Lopes, Dr.
(Co-Orientador)

Profa. Daniela Barreiro Claro, Dra.
(Membro da Banca Examinadora)

Prof. Cláudio Gottschalg Duque, Dr.
(Membro da Banca Examinadora)

Prof. Slimane Hammoudi, Dr.
(Membro da Banca Examinadora)

Dedico este trabalho ao meu tio Antônio Luiz Bassani.

RESUMO

Apresenta-se uma abordagem para suportar a criação de casos de teste para sistemas de *software* em ambientes de Computação em Nuvem. Esta abordagem é baseada na Engenharia Dirigida por Modelos (MDE). Um *framework*, uma metodologia e metamodelos são propostos para suportar a geração dos casos de teste. Metamodelos específicos para plataformas de Computação em Nuvem são propostos. Dois exemplos ilustrativos ajudam a compreender a abordagem e os passos para aplicá-la. A prototipagem do *framework* proposto reutiliza as ferramentas MT4MDE e SAMT4MDE para gerar definições de transformação semiautomaticamente para contribuir com a criação semiautomática de modelos de teste e códigos de teste para plataformas de Computação em Nuvem. Uma avaliação é feita, comparando com o *framework* desenvolvido abordagens existentes com nossa abordagem proposta, destacando os principais diferenciais, incluindo vantagens e desvantagens.

Palavras-chave: Engenharia Dirigida por Modelos, Modelos, Casos de Teste, Computação em Nuvem.

ABSTRACT

We present an approach to support the creation of test cases for software systems in cloud computing environments. This approach is based on Model Driven Engineering (MDE). A framework, a methodology and metamodels are proposed to support the generation of test cases. Metamodels for cloud computing environments and transformation definitions are proposed. Two illustrative examples help to understand our approach and the steps to apply it. A prototype implementing the proposed framework is presented and it works together with MT4MDE and SAMT4MDE to generate semi-automatically transformation definitions and contribute to the semi-automatic creation of testing model and test code for cloud computing platforms. We present an evaluation of our approach and compare it with other approaches, detaching main differences, including advantages and disavantages.

Keywords: Model Driven-Engineering, Models, Test Case, Cloud Computing.

Agradecimentos

A Deus, por ter conduzido meus passos até agora.
Aos meus pais, irmãos e demais parentes: Nádia, Aguilar, Gianne, Giancarlo, Antônio, Genice e Oscar pela força;
Ao meu orientador, professor Ph.D. Zair Abdelouahab, pela oportunidade concedida, pela orientação, dedicação e conhecimento repassado. Os meus sinceros agradecimentos.
Ao meu co-orientador, professor Dr. Denivaldo Lopes, pela oportunidade concedida, pela orientação, pela paciência, compreensão, apoio e incentivo ao longo deste período. Meus sinceros agradecimentos.
Aos meus colegas e ex-colegas de mestrado, em especial a Amélia, Lianna, Vladimir, Marcus Vinícius, Eduardo Devidson, César, Jone, Fabio, Luis Eduardo, Ivo, Berto e Wagner pelo apoio em diversos momentos durante a realização deste mestrado.
Aos meus amigos Marcelo Nicomedes e Luis Oliveira pelo apoio, pela compreensão e pelo incentivo na realização deste mestrado.
Às pessoas especiais que conheci na cidade de São Luís durante este período, em especial a Wellington Luiz Rodrigues, Gustavo Dominices, Michael Costa e Eduardo Batista por todo apoio e companheirismo demonstrado, no qual foi de suma importância para mim.
À Aitan Viegas por ter ajudado no início dos trabalhos do mestrado e por todos momentos depois compartilhados.
Aos integrantes e ex-integrantes do LESERC, em especial Pablo, Steve, Jean e Wesley que contribuíram direta ou indiretamente para realização desse trabalho.
Aos amigos e colegas da UEMS, em especial a DINF, pela compreensão e colaboração.
Ao programa de Pós-Graduação em Engenharia de Eletricidade da Universidade Federal do Maranhão, pela oportunidade da realização do curso.
A CAPES, CNPq e FAPEMA pelo apoio concedido ao longo do mestrado.
A todos aqueles que direta ou indiretamente contribuíram para a realização deste trabalho.

"O impossível é questão de tempo".

Alberto Saltiel

Lista de Figuras

Lista de Tabelas

Lista de Siglas

API Application Programming Interface.

ATL Atlas Transformation Language.

CIM Computation Independent Model.

DOM Document Object Model.

DSL Domain Specific Languages.

EC2 Amazon Elastic Compute Cloud.

EMF Eclipse Modeling Framework.

GAE Google App Engine.

HTML HyperText Markup Language.

HTTP Hypertext Transfer Protocol.

IaaS Infrastructure-as-a-Service.

IBM International Business Machines.

MDA Model-Driven Architecture.

MDE Model-Driven Engineering.

MDT Model-Driven Testing.

MOF QVT Meta Object Facility - Query/View/Transformation.

MT4MDE Mapping Tool for MDE.

OMG Object Management Group.

OO Object Oriented.

OWL Web Ontology Language.

PaaS Platform-as-a-Service.

PIM Platform Independent Model.

PSM Platform Specific Model.

QEMU Quick EMUlator.

QoS Quality of Service.

SaaS Software-as-a-Service.

SAMT4MDE Semi-Automatic Matching Tool for MDE.

SAX Simple API for XML.

SLA Service-level Agreement.

SO Sistema Operacional.

SOA Service Oriented Architecture.

SQL Structured Query Language.

StAX Streaming API for XML.

SWRL Semantic Web Rules Language.

TaaS Testing-as-a-service.

TI Tecnologia da Informação.

UML Unified Modeling Language.

WSDL Web Service Definition Language.

XMI XML Metadata Interchange.

XML eXtensible Markup Language.

Sumário

1 Introdução

Este primeiro capítulo descreve a contextualização em que a dissertação foi desenvolvida, a problemática, a solução proposta e seus objetivos. Após, a metodologia empregada para a execução deste trabalho de pesquisa é descrita e, fechando o capítulo, a apresentação dos demais capítulos desta dissertação.

1.1 Contexto

O crescente avanço da tecnologia, a utilização da *Web* como meio de publicação de novas aplicações e disponibilização de funcionalidades, o armazenamento de dados, antes somente armazenados em servidores locais, têm consolidado o conceito da Computação em Nuvem (Cloud Computing) [79].

A Computação em Nuvem faz referência a infraestrutura e serviços que são disponibilizados na rede através de virtualização e acessados via Internet. As aplicações são executadas em servidores hospedados como serviços [79] [92]. Uma das vantagens da Computação em Nuvem é cortar custos operacionais e, o mais importante, permitir que departamentos de Tecnologia da Informação (TI) se concentrem em projetos estratégicos [92].

Aplicações podem ser desenvolvidas para o ambiente de Computação em Nuvem. Estas aplicações podem ser desenvolvidas localmente e hospedadas como serviços ou construídas dentro dos ambientes de desenvolvimento oferecido pelas nuvens.

Nesse contexto, o suporte de teste em ambientes de Computação em Nuvem vem como uma nova forma para testar aplicações *Web*. A nuvem de teste procura simular ambientes do mundo real por meio de testes de carga e sites *Web* de teste de estresse [33].

Sendo assim, a fase de Teste de *Software* se caracteriza como uma atividade crucial no processo de Verificação & Validação de *software*, que envolve custos altos e é fortemente dependente de automação. A falta de confiabilidade nos testes pode in-

validar todo um processo de Verificação & Validação de *software*. Os testes precisam ser continuamente mantidos e executados durante todo o ciclo de vida de desenvolvimento de um sistema [43].

Visando uma melhoria na qualidade do *software* desenvolvido, a *Model-Driven Engineering (MDE)* tem focado em utilizar modelos para prover benefícios, tais como: melhorar a qualidade final dos produtos de *software* e reduzir os custos e tempo no desenvolvimento.

A MDE pode auxiliar a criação de aplicações interoperáveis. A MDE fornece meios para adaptar novas tecnologias com sistemas legados e permitir a integração entre diferentes tecnologias [72].

Dentro da MDE, existe o *Model-Driven Testing (MDT)* com objetivo de automatizar o Teste de *Software*, de forma significativa, reduzindo esforços de teste durante o curso de desenvolvimento de sistemas [81]. Transformações entre modelos podem ser usadas para criar modelos específicos para a plataforma em nuvem, e criar o código-fonte de teste. Contribuindo assim para maior confiabilidade, produtividade e qualidade no desenvolvimento.

1.2 Problemática

Dentro de um estudo realizado na área de teste de *software* para plataformas de Computação em Nuvem, identificou-se dois principais problemas. O primeiro é a falta de interoperabilidade entre as plataformas de nuvens, considerada uma das principais dificuldades encontradas na Computação em Nuvem, pois existe uma ausência de padrões entre as diversas plataformas [33] [67]. O segundo problema foi identificado através de pesquisas baseadas em [90] [93] [6], onde comprovou-se uma ausência de padronização do processo de Teste de *Software* para Computação em Nuvem.

Assim, pode-se observar que um processo de automatização na geração de casos de teste para ambientes de Computação em Nuvem se faz necessário para reduzir o número de intervenções humanas na construção de casos de teste, garantindo qualidade e eficiência. Transformações de modelos podem apoiar tal processo contribuindo para automatizar a passagem de informação entre diferentes níveis de modelos, garantindo portabilidade e interoperabilidade.

Outra necessidade observada foi a ausência de um modelo de desenvolvimento para construção de casos de teste que seja aplicável a nuvens de plataformas diferentes. A MDE trabalha com a ideia de criar diferentes modelos em diferentes níveis de abstração e depois uni-los para formar uma implementação [45].

A MDE propõe o aumento da produtividade e da reutilização através da separação de conceitos e abstração [40]. MDT, que trabalha com o mesmo princípio da MDE, traz alguns benefícios como a redução do tempo e a reutilização de funções similares melhorando, a qualidade dos testes. Além de reduzir drasticamente os custos de manutenção, devido as mudanças da implementação serem realizadas nos modelos [27].

Portanto, observa-se que o MDT se constitui numa abordagem promissora no que se refere a Teste de *Software*. Na medida em que os modelos utilizados na concepção da lógica do sistema forem modificados, os modelos de testes serão automaticamente atualizados, proporcionando interoperabilidade em diferentes plataformas de Computação em Nuvem.

Supõe-se que, para a geração automática de casos de teste, faz-se necessário o desenvolvimento de uma abordagem baseada em MDT para solucionar o problema da falta de interoperabilidade entre as diferentes plataformas dentro da Computação em Nuvem.

1.3 Solução Proposta

Este trabalho propõe uma solução para automatizar a geração de casos de teste utilizando a MDE para aplicações construídas para ambientes de Computação em Nuvem.

Um *framework* é proposto com a finalidade de suportar interoperabilidade e ser aplicável em diversas plataformas de Computação em Nuvem. Esta interoperabilidade é conseguida através de modelos. Este *framework* trabalha com a geração de testes em nível de aplicação em forma de testes funcionais e testes de tempo de resposta e estresse.

A geração de casos de teste é realizada a partir das informações obtidas pelo modelo de negócio da aplicação. O *framework* trabalha com três opções de aplicação. A primeira opção é reutilizar um serviço, ou seja, testar uma aplicação considerando somente a utilização de serviços como um consumidor. A segunda opção é testar uma aplicação considerando que os serviços sejam construídos a partir do zero. A terceira opção é híbrida, pois permite utilizar um serviço existente (através de sua descrição em Web Service Definition Language (WSDL)) ou um novo serviço em desenvolvimento.

Um algoritmo é proposto para a criação automática dos modelos de teste independentes de plataforma. Esta geração automática se apoia na escolha dos casos de teste que serão gerados e quais os critérios de testes utilizados, a partir dos dados de entrada, obtidos pelo modelo de negócio, fornecido pelo usuário.

1.4 Objetivos

O trabalho desenvolvido foi baseado em dois principais objetivos: Primeiro, fornecer uma abordagem baseada na MDE para suportar o teste em ambientes de Computação em Nuvem. E segundo, propor um *framework* para geração de casos de teste a serem aplicados em *software* desenvolvido em plataformas de Computação em Nuvem.

1.4.1 Objetivos Específicos

Os objetivos específicos são os seguintes:

- Uma proposta de construção de uma Domain Specific Languages (DSL) para o processo de teste em ambientes de Computação em Nuvem;
- O desenvolvimento de uma metodologia para obtenção e construção de casos de teste para *software* baseado nestas plataformas;
- O estudo e utilização da MDE para a construção de um protótipo para geração de casos de teste aplicado à Computação em Nuvem;
- E a proposta de geração automática dos casos de teste e de códigos de teste para a aplicação no *software* desenvolvido.

1.5 Metodologia

A metodologia proposta, para o desenvolvimento desta dissertação, pode ser listada a seguir:

1. Pesquisa bibliográfica, com intuito de coletar informações de livros, artigos, teses, dissertações, periódicos, anais de congressos, tutoriais e *websites* para uma total ambientação da literatura sobre o estado da arte de:
   - MDE, incluindo linguagens de transformação, definição de transformação, especificação de correspondências [72] [22] [34] [45] [52] [49];
   - Teste de *Software* [47] [78];
   - *Model Driven Testing (MDT)* [27] [35];
   - *Service Oriented Architecture (SOA)* [32];
   - Computação em Nuvem [92], [25], [66], [79] .
2. Proposta de uma abordagem para criação de modelos de casos de teste e códigos de teste segundo MDE;
3. Proposta de uma abordagem para aplicação dos casos de teste e códigos gerados para um sistema numa plataforma de Computação em Nuvem segundo MDE;
4. Utilização de linguagem de transformação *Atlas Transformation Language (ATL)* [22] e *Meta Object Facility - Query/View/Transformation (MOF QVT)* [55] [54] para escrever as definições de transformação;
5. Proposta de criação de uma DSL para o processo de Teste de *Software* em ambientes de Computação em Nuvem;

1.6 Apresentação da Dissertação

Este trabalho de dissertação está estruturado em 7 (sete) capítulos como seguem:

O primeiro Capítulo introduz o contexto tecnológico no qual este trabalho está inserido, o cenário em que foi desenvolvido, a problemática que proporcionou a motivação para a proposta da ferramenta, tecnologias aplicadas e abordagens presentes na solução. Também neste capítulo, o objetivo geral, os objetivos específicos e a metodologia de pesquisa são apresentados.

O segundo Capítulo apresenta a fundamentação teórica para situar e levar o leitor ao entendimento do desenvolvimento do presente trabalho. Os conceitos relacionados e aplicados à solução proposta são apresentados, tais como: Teste de *Software*, incluindo os critérios de testes para a escolha dos testes a serem executados; e o ambiente da Computação em Nuvem (cenário da aplicação dos testes). Também a arquitetura SOA e Serviços *Web* são relatados. Encerrando o capítulo, abordaremos a MDE e suas aplicações como o MDT e *Framework* de Modelagem do Eclipse com *Eclipse Modeling Framework (EMF)*.

No terceiro Capítulo, o estado da arte da pesquisa e trabalhos relacionados ao processo de Teste de *Software* para ambientes de Computação em Nuvem são apresentados. Também, uma análise dos trabalhos relacionados é apresentada.

No quarto Capítulo, a proposta do *framework* para suporte de Teste de *Software* para ambientes de Computação em Nuvem é apresentada. O suporte de testes é exibido, mostrando: a abordagem utilizada; a metodologia de desenvolvimento aplicada; os metamodelos propostos; o *framework* baseado em MDE; a utilização de *matching* de modelos; a geração automática de modelos de teste.

No quinto Capítulo, a implementação do protótipo é apresentada através das ferramentas para suportar a geração de casos de teste para ambientes de Computação em Nuvem. A ferramenta para a criação automática de modelos de testes, a ferramenta para auxiliar no processo de especificações de correspondências para MDE (*Mapping Tool for MDE (MT4MDE)*) e para geração semiautomática de correspondências para MDE (*Semi-Automatic Matching Tool for MDE (SAMT4MDE)*) são apresentadas neste Capítulo.

No sexto Capítulo, exemplos ilustrativos são propostos, incluindo suas modelagens. Em seguida, a utilização do protótipo, a avaliação dos resultados e os testes são descritos.

Fechando a dissertação, no sétimo Capítulo, as conclusões, as contribuições, as limitações, os resultados alcançados e os trabalhos futuros do trabalho desenvolvido são apresentados.

2 Fundamentação Teórica

Este capítulo apresenta a fundamentação teórica, com os principais conceitos e tecnologias utilizadas, com o objetivo de promover ao leitor uma ambientação para o posterior conhecimento do *framework* desenvolvido para a geração de casos de teste para ambientes de Computação em Nuvem utilizando a abordagem da Engenharia Dirigida por Modelos.

Inicialmente, apresentam-se os conceitos de Teste de *Software* com seus critérios e tipos de testes. O assunto seguinte é a Computação em Nuvem com suas principais características e classificações quanto ao tipo de serviço oferecido. Ainda neste capítulo, a MDE é apresentada, incluindo o *Eclipse Modeling Framework* EMF e *Model-Driven Architecture (MDA)*.

2.1 Teste de *Software*

Com o avanço das tecnologias, a qualidade do *software* se tornou de extrema importância para o processo de desenvolvimento. Para garantir esta qualidade, a área de teste de *software* vem crescendo nos últimos anos. O teste de *software* é considerado uma das áreas mais onerosas do desenvolvimento de *software*. Segundo [47], "o teste é uma mola mestra no que tange impulsionar algum processo de qualidade de *software*".

Pode ser considerado que o processo de teste possui duas metas distintas. A primeira meta é que o *software* atenda os requisitos. A segunda meta é descobrir falhas ou defeitos no *software* que apresentam comportamentos incorretos, ou que não estejam em conformidade com que foi especificado.

2.1.1 Termos Utilizados

Antes de descrevermos sobre os testes, algumas definições e termos serão apresentados para a melhor compreensão deste capítulo. Os termos foram retirados do

Glossário de Termos Utilizados em Teste de *Software* produzido pelo "*Glossary Working Party*"do *International Software Testing Qualification Board* [91]. Tais definições são:

- **Abordagens de teste *(Test Approach)*:** Implementação da estratégia de teste para um projeto específico. Normalmente, inclui as decisões tomadas e baseadas no objetivo do projeto (teste) e na avaliação do risco feita, nos pontos de inicio relacionados ao processo de teste, nas técnicas de modelagem de teste a serem aplicadas, nos critérios de saída e nos tipos de testes a serem desempenhados;

- **Ambiente de teste *(Test environment)*:** Ambiente que contém *hardware*, instrumentação, simuladores, ferramentas de *software* e outros elementos de suporte necessários à realização de um teste;

- **Automatização de teste *(Test automation)*:** Utilização de *software* para desempenhar ou dar suporte às atividades de teste, por exemplo, gerenciamento de teste, modelagem de teste, execução de teste e verificação de resultados;

- **Caso de teste *(Test case)*:** Conjunto de valores de entrada, pré-condições de execução, resultados esperados e pós-condições de execução desenvolvidas para um determinado objetivo ou condição de teste, tais como para exercitar o caminho de um determinado programa ou verificar o atendimento a um requisito especifico;

- **Condição de teste *(Test condition):*** Item ou evento de um componente ou sistema que pode ser verificado por um ou mais casos de teste, por exemplo, função, transação, característica, atributo de qualidade ou elemento estrutural;

- **Critérios de entrada *(Entry criteria)*:** Conjunto de condições genéricas e específicas que permitem que um processo avance com uma determinada tarefa (por exemplo, fase de teste). A finalidade dos critérios de entrada é evitar que uma tarefa implique em mais esforços (desperdício) em comparação com o esforço necessário;

- **Critérios de saída *(Exit criteria)*:** Conjunto de condições genéricas e específicas, acordadas pelos *stakeholders*, que permite que um processo seja oficialmente considerado completado. A finalidade dos critérios de saída é evitar que uma tarefa seja considerada completa quando ainda existirem partes importantes dela que ainda não tenham sido terminadas. Os critérios de saída são utilizados para relatar e para planejar o momento de interromper os testes;

- **Dados de teste *(Test data)*:** Dados existentes (por exemplo, em um banco de dados) antes do início da execução de um teste e que afetam ou são afetados pelo componente ou sistema sendo testado;

- **Defeito *(Defect)*:** Falha em um componente ou sistema que pode fazer com que o componente ou sistema falhe ao desempenhar sua função (por exemplo, uma sentença incorreta ou uma definição de dados incorreta). Um defeito, se descoberto durante a execução, pode levar a falha do componente ou do sistema;

- **Falha *(Failure)*:** Desvio do componente ou sistema da entrega, resultado ou serviço esperado;

- **Funcionalidade *(Functionality)*:** Capacidade do produto de *software* de oferecer funções que atendam às necessidades declaradas ou implícitas quando utilizado sob condições específicas;

- **Objetivo de teste *(Test objective)*:** Razão ou finalidade por trás da modelagem e da execução de um teste;

- **Oráculo de teste (*Test oracle*):** Fonte utilizada para determinar os resultados esperados e compará-los com os resultados reais produzidos pelo *software* em teste. Um oráculo pode ser um sistema existente (para um *benchmark*), um manual de usuário ou o conhecimento especializado de um indivíduo, porém, não deve ser o código;

- **Plano de teste *(Test plan)*:** Documento descrevendo o escopo, abordagem, recursos e cronograma das atividades de teste que se destina. Ela identifica, entre outros, itens de teste, recursos a serem testados, tarefas de teste, executor de cada tarefa, grau de independência do testador, o ambiente de teste, as técnicas de projeto de teste e critérios de entrada e de saída a serem usados, as razões de sua escolha, e os eventuais riscos que exigem planos de contingência. É um registro do processo de planejamento de teste;

- **Relatório de defeito *(Defect report)*:** Documento que relata qualquer falha em um componente ou sistema que possa fazer com este componente ou sistema deixe de desempenhar sua função requisitada;

- **Requisito de teste *(Test requirement)*:** Ver condição de teste;

- **Técnica de modelagem de teste *(Test design technique)*:** Procedimento utilizado para derivar e/ou selecionar casos de teste;

- **Tipos de testes *(Test type)*:** Grupo de atividades que testa um componente ou sistema enfocando um objetivo de teste específico, ou seja, funcional, usabilidade, regressão, etc. Um tipo de teste pode acontecer em um ou mais níveis ou fases de teste.

2.1.2 Tipos de Defeitos

Depois de codificar os componentes do programa, geralmente examina-se o código, a procura de defeitos para eliminá-los imediatamente. Quando não existe nenhum defeito óbvio, testam-se os programas para verificar se podem isolar mais defeitos criando condições em que o código não reage como planejado [62]. Assim, é importante saber que tipo de defeitos se está procurando.

Em relação à "defeito"(do inglês, *fault*), este termo trata-se como sendo um passo, processo ou definição de dados incorretos e "engano " (do inglês, *mistake*) como uma ação que produz um defeito. A existência de um defeito pode ocasionar um "erro" (do inglês, *error*) durante a execução de um programa, reconhecido como um estado inconsciente ou inesperado. Este estado inesperado pode levar o sistema a uma "falha" (do inglês, *failure*). Uma falha pode fazer com que o resultado produzido pela execução seja diferente do resultado esperado [9].

Segundo [62], os defeitos podem ser classificados de acordo com as seguintes ocorrências:

- **Defeitos de algoritmo:** Ocorre quando a lógica ou o próprio algoritmo não produz a saída esperada de acordo com uma entrada especificada;

- **Defeitos de computação e/ou precisão:** Este tipo de defeito pode ser detectado quando a fórmula do algoritmo está errada ou não traz o resultado com a precisão esperada;

- **Defeitos na documentação:** Ocorre quando a documentação não está de acordo com o que o programa está realizando, ou seja, a documentação está divergente do programa implementado;

- **Defeitos de *stress* ou sobrecarga:** Este defeito ocorre quando se trabalha com os mecanismos de filas, pilhas, *buffers*, dimensão de tabelas, etc. O defeito acontece quando estes mecanismos recebem mais dados do que a capacidade especificada nos requisitos;

- **Defeitos relativos à capacidade ou limites:** Este tipo de defeito pode ser detectado quando o desempenho do sistema não atende ao especificado à medida que o sistema alcança um limite especificado;

- **Defeitos de sincronia ou coordenação:** São defeitos que ocorrem em sistemas de tempo real. Este defeito pode ser detectado quando existem diversos processos e eventos e os códigos que coordena estes eventos são inadequados.

2.1.3 Classificação Ortogonal de Defeitos

Segundo [62], a *International Business Machines (IBM)* desenvolveu uma a abordagem para o acompanhamento de defeitos chamada de classificação ortogonal de defeitos. Estes defeitos estão classificados em categorias que podem sugerir quais as partes do processo de desenvolvimento necessitam de atenção especial. A classificação destes defeitos pode ser vista na Tabela 2.1.

2.1.4 Tipos de Testes

Existem diversos testes que podem ser aplicados num plano de teste. Esses testes são escolhidos de acordo com o objetivo do *software*, documentação existente e também pela experiência do profissional da área. Nesta subseção, iremos abordar os principais tipos de testes e os critérios utilizados para a criação dos casos de teste.

Testes Funcionais

Caracterizado por mostrar se a aplicação funciona ou não em tudo aquilo que ela se propõe a atender em termos de funcionalidades [47]. Também conhecido como teste de caixa-preta ou teste baseado na especificação, é um teste voltado para verificação, isto é, voltado para encontrar discrepâncias entre o que um programa faz

Tabela 2.1: Classificação Ortogonal de Defeitos [62]

Tipo de Defeito	**Significado**
Função	Defeito que afeta a capacidade, as interfaces com o usuário final, do produto,com a arquitetura de hardware ou as estruturas globais de dados.
Interface	Defeito na interação com outros componentes ou drivers, por meio de chamadas, macros, blocos de controle ou lista de parâmetros.
Verificação	Defeito na lógica do programa, que falha ao validar dados e valores apropriadamente, antes de utilizá-los.
Atribuição	Defeito na estrutura de dados ou na inicialização do bloco de código.
Sincronia/sequência	Defeito que envolve a sincronia de recursos compartilhados em tempo real.
Versão/empacotamento	Defeito que ocorre devido a problemas nos repositórios, de mudanças de gerenciamento ou no controle de versões.
Documentação	Defeito que afeta informações nas publicações ou sobre a manutenção.
Algoritmo	Defeito que envolve a eficiência ou a correção do algoritmo ou da estrutura de dados, mas não do projeto.

e o que deveria fazer. Deve se fazer referência a requisitos expressos por usuários e especificados por engenheiros de *software* [61].

Para a construção dos casos de teste, diferentes abordagens podem ser usadas e combinadas. O objetivo no final é gerar uma quantidade de casos de teste mínima, que possa atingir toda a aplicação e que revele todos os defeitos, visto que a quantidade de combinações de casos de teste é alta.

A escolha da abordagem para derivar casos de teste funcional depende de diversos fatores: a natureza da aplicação, a forma da especificação, o conhecimento e a experiência dos projetistas de teste, a estrutura da organização, a disponibilidade de ferramentas, as restrições de orçamento e qualidade, e custos do projeto e implementação de código de suporte [61].

A seguir serão listados os principais critérios de testes utilizados para a construção dos casos de teste funcionais [47]:

- **Teste de equivalência de classe:** Tem como objetivo reduzir o número de casos de teste. Trabalha por dedução lógica. Esta redução é obtida através da divisão do domínio de entrada (por exemplo, uma regra de negócio) em subdomínios de elementos equivalentes;

- **Teste de valor limite:** Assim como o teste de equivalência de classes, tem como objetivo reduzir o número de casos de teste testando os valores de fronteiras das classes;

- **Teste por tabela de decisão:** Monta um conjunto de casos de teste que representam regras e conjunto de ações envolvidas a partir de uma tabela. É utilizado no caso onde existem várias regras e uma possa interferir na outra;

- **Pairwise Testing (Array ortogonal):** Tem por objetivo obter um conjunto de casos de teste significativo dentro de um conjunto inviável de combinações. Baseia-se na utilização de arrays ortogonais;

- **Teste de transição de estado:** Diagramas de transição de estados são utilizados para encontrar (ou deduzir) casos de teste. É aplicado quando se tem várias regras de negócios que estão associadas a estados de objetos e suas transições;

- **Teste de análise de domínio:** Levanta casos de teste considerando valores exatos e de fronteira. Técnica utilizada para obter um conjunto de testes válidos quando existem variáveis que possuem regras associadas e estas estão sujeitas a operadores relacionais.

Testes de Performance

Este teste tem por objetivo demonstrar se numa carga qualquer de informações o desempenho (tempo) do aplicativo atende às metas desejadas [47]. O teste de performance, também chamado de teste de desempenho trabalha com os requisitos não funcionais. Dentre os requisitos não funcionais, o teste de desempenho com confiabilidade, segurança, velocidade, precisão, etc.

Principais critérios de testes utilizados para a construção dos casos de teste de performance [62]:

- **Teste de estresse:** Utilizado principalmente para sistemas que operam abaixo de sua capacidade máxima. Avalia quando o sistema é levado aos seus limites por um pequeno período. Geralmente estas capacidades e limites estão descritos nos requisitos não funcionais;

- **Teste de volume:** Teste que aborda grandes quantidades de dados do sistema;

- **Teste de configuração:** Teste que analisa várias configurações de *software* e de *hardware* especificadas nos requisitos;

- **Teste de compatibilidade:** Teste utilizado para assegurar a compatibilidade entre sistemas ou aplicações. Um exemplo simples é o teste para assegurar compatibilidade de um sistema que possa ser executado em diversos *browsers*, em diferentes sistemas operacionais e diferente plataformas de *hardware*. Utilizado também quando o sistema tem interface para outro sistema;

- **Teste de regressão:** Teste que identifica defeitos que já foram corrigidos, porém, sofreram problemas de qualidade durante um processo de manutenção. Utiliza-se quando o sistema que está sendo testado irá substituir um sistema existente;

- **Teste de segurança:** Testes relacionados com disponibilidade, integridade e confidencialidade;

- **Teste de tempo:** Relacionado ao tempo que o sistema leva para atender uma funcionalidade;

- **Teste de ambiente:** Testa a capacidade de funcionamento do sistema no local de instalação;

- **Teste de qualidade:** Utilizado para testar confiabilidade, manutenibilidade e a disponibilidade do sistema. Inclui nesse teste o cálculo do tempo médio entre as falhas e o tempo médio dos reparos e também o tempo médio para encontrar e resolver um defeito;

- **Teste de recuperação:** Teste que trata da resposta à presença de falhas ou à perda de dados, energia, dispositivos ou serviços;

- **Teste de manutenção:** Teste que aborda a necessidade de ferramentas e procedimentos de diagnóstico para ajudar a encontrar a fonte dos problemas;

- **Teste de documentação:** Teste que avalia a conformidade da documentação a estes requisitos;

- **Teste de fatores humanos:** Teste onde se investigam os requisitos que se relacionam com a interface e com o usuário do sistema.

Testes de Usabilidade

Teste cujo foco é testar a usabilidade de um aplicativo, que é a facilidade de uso que as pessoas podem empregar em uma ferramenta ou um objeto a fim de realizar uma tarefa específica [47]. Segundo o autor Pezzè [61], o processo para verificar e validar a usabilidade inclui os seguintes passos principais:

- Inspecionar as especificações usando listas de verificação de usabilidade. A inspeção provê um retorno antecipado da usabilidade;

- Testar os primeiros protótipos com usuários finais para explorar seu modelo mental (teste exploratório), avaliar alternativas (teste de comparação) e validar a usabilidade do *software*;

- Testar as versões de maneira incremental tanto com especialistas em usabilidade como com usuários finais para monitorar o progresso e antecipar problemas de usabilidade;

- Executar testes de sistema e aceitação que incluam inspeções e testes usando especialistas, usuários finais e verificações normalmente feitas de maneira automática como teste de conectividade e de compatibilidade com navegadores.

Principais critérios de testes utilizados para a construção dos casos de teste de usabilidade [61]:

- **Teste de exploratório:** Utilizado para perguntar aos usuários finais a abordagem para interagir com o sistema, avaliar alternativas do projeto e validação em oposição a requisitos de usabilidade estabelecidos e padrões;

- **Teste de comparação:** Consiste em observar as reações do usuário em diferentes padrões de interação;

- **Teste de validação:** Avalia a usabilidade em geral, incluindo a identificação de dificuldades e obstáculos que os usuários encontram quando interagem com o sistema;

Testes de Unidade

"O teste de unidade tem como foco as menores unidades de um programa, que podem ser funções, procedimentos, métodos ou classes. Neste contexto espera-se que sejam identificados erros relacionados a algoritmos incorretos ou mal implementados, estruturas de dados incorretas, ou simples erros de programação [9]".

Segundo o autor Pfleeger [62], o processo para o teste de unidade é feito primeiramente examinando-se o código, em busca de defeitos no algoritmo, nos dados e na sintaxe. Em seguida compila-se o código eliminando os defeitos remanescentes. Por fim, geram-se os casos de teste. O teste de unidade também é conhecido como teste de caixa branca, pois, os casos de teste são baseados na implementação dos objetos e exercitam comandos e caminhos da estrutura destes objetos.

Os principais critérios de testes utilizados para a construção dos casos de teste de unidade são [62]:

- **Revisão do código:** Critério semelhante às revisões de requisitos;
- ***Walkthroughs* de código:** Tipo de revisão de código onde uma pessoa apresenta o código desenvolvido e a documentação correspondente para a equipe de revisão;
- **Inspeções de código:** Tipo de revisão de código semelhante a *Walkthrough*, sendo que é mais formal. Verifica se o código e a documentação estão de acordo com uma lista de aspectos predeterminados;
- **Técnica de prova formal:** Critério que se baseia na conversão do código em seu correspondente lógico;
- **Execução simbólica:** Execução simulada do código utilizando símbolos em vez das variáveis de dados;
- **Prova automática de teorema:** Desenvolve-se uma ferramenta que leia como entrada: os dados e as condições de entrada, os dados e as condições de saída e as linha de código do componente que será testado.

Dentro da parte de teste de unidade, em que o código pode ser testado minuciosamente, podemos destacar as seguintes abordagens [62]: Teste de comando, teste de ramificação, teste de caminho, teste de caminho definição-uso, teste de todos os usos, teste de uso de todos os predicados e de alguns usos computacionais, teste de todos os usos computacionais e de alguns usos de predicados.

Testes de Integração

Visa garantir que um ou mais componentes combinados (ou unidades) funcionam corretamente [47]. Os componentes integrados podem ser comerciais, reusáveis adaptados a determinado sistema ou componentes novos desenvolvidos. Para muitos sistemas de grande porte, provavelmente são usados os três tipos de componentes.

O teste de integração verifica se esses componentes funcionam realmente em conjunto, se são chamados corretamente e se transferem dados corretos no tempo correto por meio de suas interfaces [62]. Existem algumas abordagens para se trabalhar com o teste de integração. Algumas dessas abordagens são: integração *top-down*, integração *bottom-up*, teste de *big bang*, teste de integração estrutural e teste de *thread*.

Testes em Aplicações Orientadas a Objetos

Os sistemas Object Oriented (OO) são diferentes dos sistemas convencionais. Portanto, os Testes de *Software* também precisam ser aplicados de acordo com a necessidade desses sistemas. Alguns testes tradicionais podem ser adaptados para este contexto.

Para a realização de testes em sistemas orientados a objetos pode ser realizada uma mistura apropriada de técnicas e abordagens. Uma abordagem de trabalho genérica, desmembrada em três fases, partindo das classes individuais para depois a integração e interações é descrita pelo autor Pezzè em [61]:

- **Intraclasse:** Testa isoladamente as classes (teste unitário);
- **Interclasse:** Testa a integração entre as classes (teste de integração);
- **Sistema e aceitação:** Aplica os testes funcionais e de aceitação padrão para componentes maiores e todo o sistema.

Existem alguns critérios para se trabalhar com testes orientados a objetos, são eles:

- **Teste intraclasse:** Que compreende o teste unitário e de integração;
- **Teste com máquinas de estado:** Forma de teste para explorar sistematicamente os estados, que pode ser derivada de especificações de módulos;
- **Teste interclasse:** Que compreende o teste de integração;
- **Teste estrutural de classes**: Dividido em teste estrutural intraclasse e teste estrutural interclasse.

Testes de Aplicações Web

As aplicações *Web* são caracteristicamente dinâmicas e utilizam de diversas tecnologias como *eXtensible Markup Language (XML)*, *Web Services*, *Hypertext Transfer Protocol (HTTP)* entre outras. O uso dessas tecnologias contribui para o aumento da flexibilidade. Com isso, as aplicações *Web* tendem a se tornarem mais complexas.

Na atividade de teste, essa característica de dinamismo gera novos aspectos para serem abordados, tais como [9]:

- O usuário pode afetar a execução de um programa, dependendo da ação que ele tiver. Por exemplo, ele pode, ao clicar um simples botão mudar totalmente o contexto do programa;
- Constantes mudanças podem ocorrer no cliente e/ou no servidor. Em aplicações *Web* podem ocorrer mudanças dinâmica de dados;
- Na arquitetura de aplicações *Web* ocorre que *hardware* e *software* podem ser heterogêneos. Isso requer um esforço maior nas questões de garantir a interoperabilidade e compatibilidade que em uma aplicação tradicional talvez não requeira tanto esforço;
- Aplicações *Web* utilizam de várias tecnologias. Essas tecnologias tendem sempre a evoluir. Com isso, as ferramentas de testes também devem acompanhar esta evolução;
- A equipe de testes deve ser formada por pessoal com conhecimentos diversos devido à grande variedade de tecnologias envolvidas.

Existem algumas técnicas que foram adaptadas e estendidas para aplicações Web, dentre elas estão [9]:

- Teste estrutural de aplicações *Web* utiliza de técnicas adaptadas de testes em fluxo de dados como modelo de objetos, fluxo de controle e fluxo de dados e modelo de comportamento;
- Teste baseado em modelos de especificação tem a proposta de trabalhar com diagramas de classes, explorando a geração de casos de teste com base nos critérios de teste estrutural e funcional para aplicações Web;
- Teste baseado em defeitos trabalha com interação de componente Web através de mensagens XML e associado a um documento XML.

2.2 Computação em Nuvem

Podemos dizer que a Computação em Nuvem faz referência a infraestrutura e serviços que são disponibilizados na rede através de virtualização e acessados via *Internet*. Neste ambiente, as aplicações são executadas em servidores hospedados como serviços. Uma das vantagens da Computação em Nuvem é cortar custos operacionais e permitir que departamentos de TI se concentrem em projetos estratégicos [92] [66] [79].

O uso da palavra "nuvem"faz referência aos dois conceitos essenciais [79]:

- **Abstração:** A Computação em Nuvem abstrai os detalhes de implementação do sistema de usuários e desenvolvedores. Os aplicativos são executados em sistemas físicos que não estão especificados, os dados são armazenados em locais que são desconhecidas, a administração de sistemas é terceirizada por outros;
- **Virtualização:** A Computação em Nuvem virtualiza sistemas e compartilha de recursos. Sistemas de armazenamento podem ser fornecidos conforme a necessidade de uma infraestrutura centralizada, os custos são avaliados em uma medida base e os recursos podem ser escalados com facilidade.

Diferentes tipos de infraestruturas de nuvens podem ser implantadas, conforme Figura 2.1. Elas podem ser chamadas de modelos de serviço [79] [92]. As três principais modelos de serviço são:

- O *Software-as-a-Service (SaaS)* - É um ambiente operacional completo, com aplicações, gerenciamento e interface do usuário. Desde o aplicativo até a infraestrutura, a responsabilidade é do fornecedor;

- A *Platform-as-a-Service (PaaS)* - Fornece máquinas virtuais, sistemas operacionais, aplicativos, serviços, *frameworks* de desenvolvimento, operações e estruturas de controle. O cliente pode implantar suas aplicações na infraestrutura de nuvem ou usar aplicativos e ferramentas que são suportados pelo prestador de serviços PaaS;

- A *Infrastructure-as-a-Service (IaaS)* - Fornece máquinas virtuais, armazenamento virtual, infraestrutura virtual e recursos de *hardware*. O prestador de serviços IaaS gerencia toda a infraestrutura.

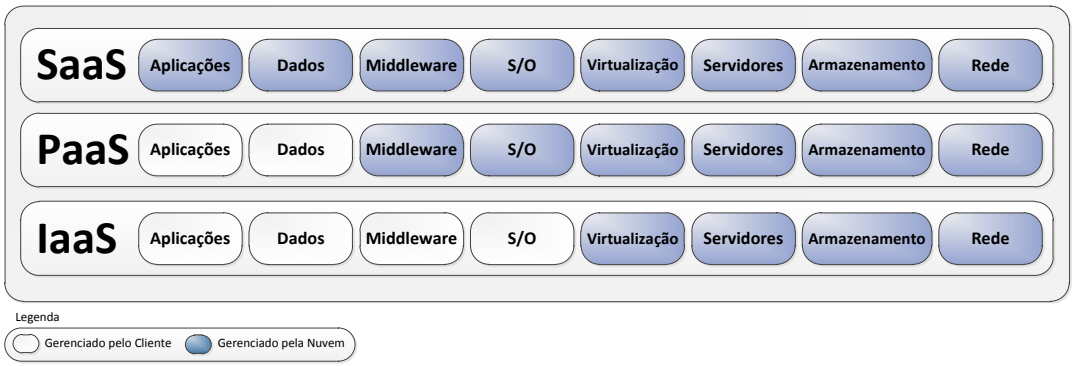

Figura 2.1: Modelo de Serviço das Nuvens - adaptado de [82]

Conforme observado na Figura 2.1, na parte inferior da pilha é o *hardware* ou infraestrutura que compreende a rede. À medida que se move para cima na pilha, cada modelo de serviço herda as capacidades do modelo de serviço inferior [79]. A IaaS possui os menores níveis de funcionalidade integradas e os mais baixos níveis de integração, e por sua vez, a SaaS possui os maiores níveis de funcionalidades integradas.

Como exemplos de prestadores de serviços, de acordo com a infraestrutura oferecida pela nuvem, podemos citar:

- **IaaS:** *Amazon Elastic Compute Cloud (EC2) [11], RackSpace [65], Open Eucalyptus [85], OpenStack [77]*;

- **PaaS:** *Google App Engine (GAE) [18], Windows Azure [4], CloudFoundry [13], OpenShift [56]*;

- **SaaS:** *Google Apps [20], SalesForce.com [70], Oracle On Demand [60].*

Dentro dos exemplos citados, as nuvens *Open Eucalyptus, OpenStack, CloudFoundry* e *OpenShift* são consideradas "nuvens abertas", ou seja, projetos desenvolvidos de nuvens com código aberto (*Open Source*) que o usuário pode utilizar sem custos.

2.2.1 SaaS e SOA

A nuvem SaaS e SOA são complementares. A característica em comum é o modelo de serviços [92]. SOA se caracteriza como um paradigma de arquitetura de *software* que define o uso de serviços para suportar os requisitos dos usuários de *software* [32]. SOA não é uma ferramenta ou um *framework* de trabalho. SOA pode ser considerada uma abordagem para a construção de arquiteturas de *software*. Esta arquitetura pode trazer diversos benefícios, como [46] a promoção da reutilização e capacidade de combinar serviços para a criação de novas aplicações compostas.

Nas aplicações em nuvem, SOA oferece um projeto arquitetônico para acessar diversos serviços através de um método flexível. Portanto, fornece uma capacidade para a aplicação evoluir. Assim, as aplicações se tornam mais configuráveis pelo usuário através da infraestrutura que SOA oferece para aplicações em nuvem [79].

2.2.2 Suporte de Teste em Computação em Nuvem

Suporte de teste para Computação em Nuvem é uma forma para testar Aplicações *Web* que utilizam este ambiente. Testes procuram simular ambientes do mundo real através de testes de carga e de teste de estresse [33]. Existem diversas vantagens quando se trabalha com teste em Computação em Nuvem [33] [28]:

- Redução de custos, pois a infraestrutura das nuvens pode dar suporte para criação de redes e várias máquinas para o ambiente de testes;

- Melhoria da eficiência dos testes, pois pode-se reduzir o tempo para construir um ambiente de teste, como máquinas, redes, a instalação do sistema operacional e instalação de várias ferramentas de teste;

- Realização de teste de desempenho mais realista, pois a infraestrutura oferece suporte para várias máquinas virtualizadas.

2.3 MDE

A complexidade do desenvolvimento de *software*, exigiu esforços para a busca de soluções na engenharia de *software* a fim de oferecer soluções racionais para controlá-la. A MDE é uma proposta complementar aos processos de desenvolvimentos tradicionais, como *waterfall*, espiral e prototipagem. MDE se concentra em modelos formais, mapeamentos e transformações para suportar o ciclo de vida dentro do desenvolvimento de *softwares* [36].

A MDE é uma abordagem de desenvolvimento de *software* que se concentra na criação de modelos que descrevem os elementos de um sistema [72] [94] e orientam sua implementação [94]. Podemos dizer que modelo é "uma descrição de um (ou parte de) sistema expresso em uma linguagem bem definida. Os modelos respeitam um formato preciso (uma sintaxe) e um significado (uma semântica). Tal descrição deve servir para uma interpretação automatizada por computador" [34] [7].

A criação de modelos é concebida através da utilização de uma linguagem de modelagem bem definida que apresente uma sintaxe e uma semântica para regulamentar a criação dos elementos e suas relações. Uma linguagem de modelagem é uma especificação formal bem definida que contém os elementos de base para construir modelos. Além disto, uma linguagem de modelagem é concebida dentro de um domínio limitado e com objetivos específicos [41].

A linguagem para criar modelos geralmente é descrita por um metamodelo. Metamodelo é "um modelo que define a linguagem para exprimir um modelo" [53]. Por sua vez, um metamodelo é definido por um metametamodelo. Um metametamodelo é "um modelo que define a linguagem para expressar metamodelos. A relação entre um metametamodelo e um metamodelo é similar à relação entre um metamodelo e um modelo" [53].

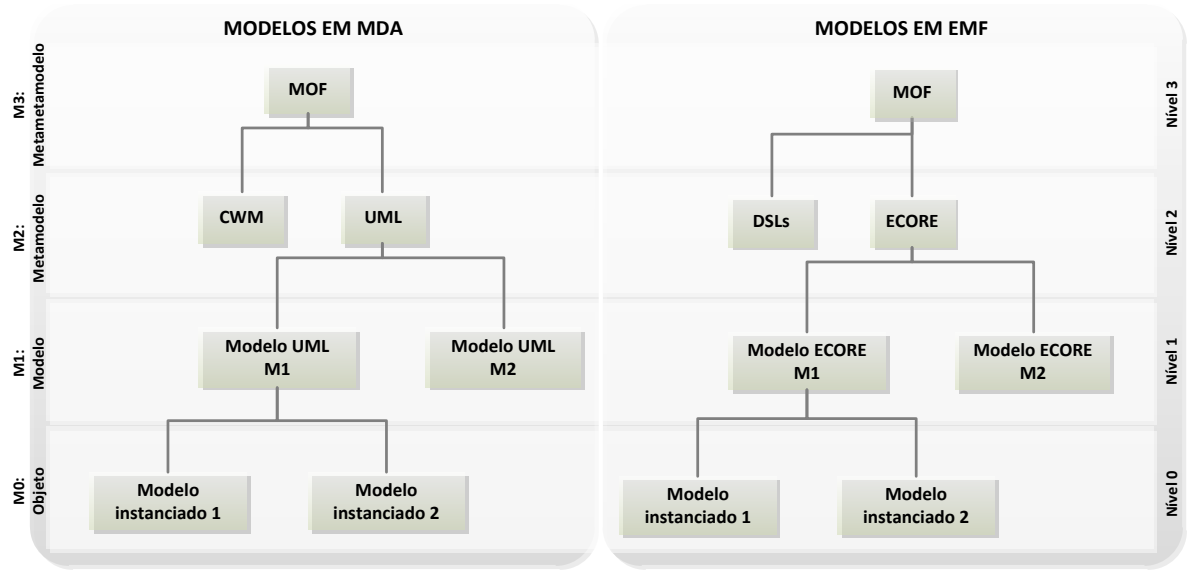

Figura 2.2: Arquitetura de quatro camadas - adaptado de [34]

Na Figura 2.2, a descrição dos 4 (quatro) níveis de camadas de modelagem é apresentada [34], contendo:

- M0 (objeto) - camada onde são representadas as instâncias do conceitos do mundo real. Tem-se como objetivo principal descrever os objetos em um domínio computacional em uma plataforma final;
- M1 (modelo) - camada onde se concentra os modelos do mundo real, representados conforme os conceitos definidos no metamodelo correspondente na camada M2. Observa-se, como função principal desta camada, a definição de um domínio da informação através de uma linguagem que descreva este modelo;
- M2 (metamodelo) - camada no qual se encontram os metamodelos. Como função principal desta camada de metamodelo, tem-se a definição de uma linguagem que especifique modelos;
- M3 (metametamodelo) - camada de mais alto nível, que constitui a base da arquitetura de metamodelagem. Nesta camada, define-se uma linguagem abstrata e um *framework* para representar metamodelos. Um metametamodelo define um modelo de mais alto nível de abstração que o metamodelo.

Existem algumas abordagens para se trabalhar com a MDE. A Arquitetura Dirigida por Modelos MDA do *Object Management Group (OMG)* [53], o EMF [12] e o Teste Dirigido por Modelos (MDT) da IBM [27] são exemplos de MDE.

2.3.1 MDA

A OMG definiu a MDA como um *framework* de desenvolvimento de *software*, e como uma iniciativa para a criação de padrões no desenvolvimento. A MDA utiliza artefatos que são modelos formais, ou seja, modelos que podem ser entendidos por computadores [34]. A MDA trabalha com a ideia de criar diferentes modelos em diferentes níveis de abstração e depois uni-los para formar uma implementação [45]. A MDA também propõe o aumento da produtividade e da reutilização através da separação de conceitos e abstração suportados por modelos [40].

Esta iniciativa com MDA apresenta certas contribuições, tais como [34]:

- **Produtividade:** a transformação do *Platform Independent Model (PIM)* para o *Platform Specific Model (PSM)* precisa ser realizado uma única vez. Depois, pode ser aplicado no desenvolvimento de diversos sistemas, reduzindo assim o tempo de desenvolvimento;
- **Portabilidade:** através dos mapeamentos entre modelos, um mesmo PIM pode ser automaticamente transformado em vários PSMs de diversas plataformas;
- **Interoperabilidade:** diferentes PSMs gerados a partir de um mesmo PIM podem ter relacionamentos entre si.

Dentro da MDA, alguns conceitos são importantes para o conhecimento do funcionamento desta abordagem. Dentre esses conhecimentos, destacam-se:

- **PIM** - Modelo de alto nível de abstração, independente de qualquer tecnologia de implementação. Neste modelo, todas as funcionalidades do sistema e restrições de negócio são modeladas;
- **PSM** - Modelo construído para uma plataforma específica, ou seja, descreve um sistema com conhecimento sobre uma determinada plataforma. Neste modelo,

detalhes da implementação são descritos em uma plataforma junto com as informações do PIM. O PSM é gerado a partir de um PIM através da transformação de modelos;

- **Definição de Transformação** - Descreve como um modelo fonte, em uma linguagem fonte (metamodelo fonte) poderá se transformar em um modelo alvo em uma linguagem alvo (metamodelo alvo);

- **Motor de Transformação** - Para executar uma definição de transformação é necessário existir um motor de transformação [34]. Uma transformação de modelos é "um processo de conversão de um modelo em outro modelo" [34]. Os modelos são refinados e novas informações são adicionados a eles através deste processo de conversão.

O processo de transformação é base da abordagem MDA, como podemos observar na Figura 2.3. Após sucessivas transformações, o código-fonte de um sistema de *software* é obtido como resultado final do processo de desenvolvimento.

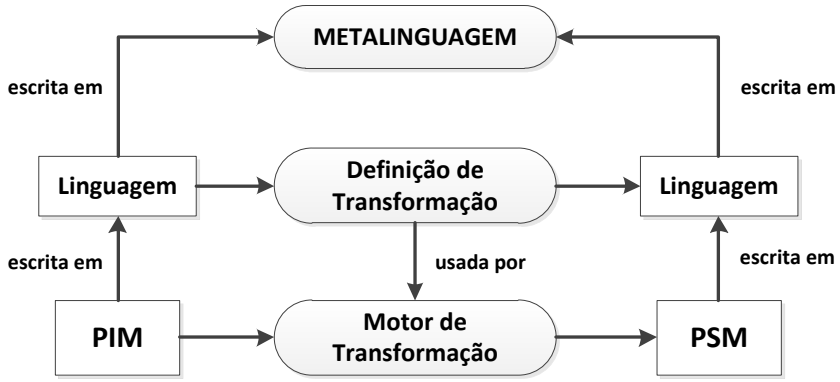

Figura 2.3: MDA - *Framework* básico [34]

A MDA divide a complexidade do sistema em três tipos de modelos: no primeiro, o *Computation Independent Model (CIM)*, que recai sobre os requisitos do sistema, não considerando sua estrutura ou processamento. No segundo, o PIM, mostra a lógica de negócios do sistema fora do contexto da uma plataforma específica. No terceiro, PSM são relacionados ao sistema para uma plataforma específica [35].

2.3.2 EMF

Como o próprio nome diz, EMF [84] é um *framework* de modelagem para Eclipse que facilita a geração de código para aplicações em Java baseado em modelos de classe simples. O EMF age como facilitador unificando as três importantes tecnologias: Java, XML e Unified Modeling Language (UML), conforme ilustra a Figura 2.4. E, independente de como o modelo EMF é fornecido, ou seja, modelos em XML ou UML, o resultado final será o mesmo.

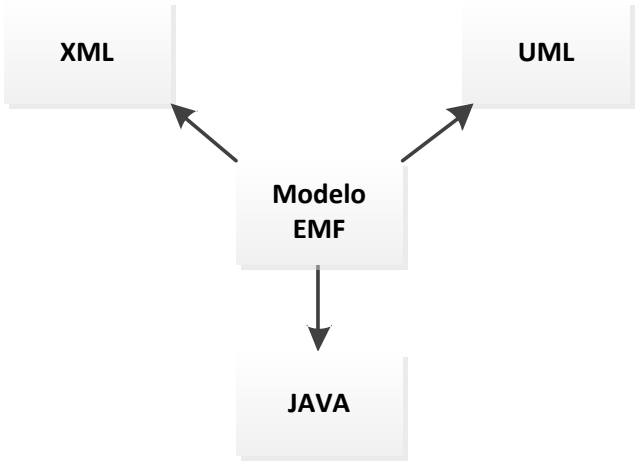

Figura 2.4: EMF - Unificação com Java, XML e UML [84]

Para representar os modelos em EMF, metamodelos em formato Ecore são utilizados. Através do EMF, interfaces Java podem ser geradas. Assim como outras formas de modelos podem ser geradas a partir de um modelo Ecore serializado, ou seja, modelos em XML Metadata Interchange (XMI). O XMI, considerado um padrão da OMG [51] para troca de informações baseado em XML, pode ser considerado como uma representação primária do metamodelo Ecore, ou padrão que torna possível a serialização de metadados usando XML.

2.4 Síntese

Neste capítulo, apresentou-se uma revisão dos principais conceitos das tecnologias envolvidas e utilizadas para o desenvolvimento do presente trabalho, com o intuito de favorecer um melhor entendimento do funcionamento do *framework* proposto.

Sobre Teste de *Software*, os principais termos utilizados dentro do ambiente de desenvolvimento de testes foram apresentados, os tipos de defeitos que são encontrados na fase de teste com uma classificação ortogonal para melhor reconhecimento do problema. Também foram apresentados os tipos de testes e os principais critérios utilizados para a criação dos casos de teste.

Importantes conceitos relacionados a Computação em Nuvem foram abordados. Apresentou-se as infraestruturas de acordo com tipo de serviço oferecido pela nuvem. Algumas relações entre SOA e SaaS foram abordadas. Nesta seção também foi relacionado o papel do suporte de teste dentro do ambiente da Computação em Nuvem.

Finalmente, a MDE e suas aplicações foram contextualizadas. Apresentou-se também os conceitos de MDA e seus benefícios como produtividade, portabilidade e interoperabilidade. E os objetivos e facilidades trazidas pelo *Framework* de Modelagem Eclipse (EMF) e suas aplicações fecharam este capítulo.

3 Estado da Arte

Este capítulo tem como objetivo descrever as principais abordagens para realizar Testes de *Software* em ambientes de Computação em Nuvem, utilizados no meio acadêmico, comercial e industrial.

A diferença entre especificação de correspondência e definição de transformação é descrita. Uma conceitualização de linguagens de transformação pesquisadas é apresentada.

A diferença entre as abordagens de Teste de *Software* para SaaS, PaaS e IaaS são relatadas através de uma apresentação resumida.

Trabalhos com relações próximas ao projeto desenvolvido, dentro de cada classificação de serviço (SaaS, PaaS e IaaS), são analisados e comparados, segundo alguns critérios estabelecidos, servindo base para o desenvolvimento desta dissertação.

3.1 Especificação de Correspondência e Definição de Transformação

A especificação de correspondência e a definição de transformação estão dentro dos principais assuntos que envolvem a MDE. Através deles, os modelos poderão ser transformados e códigos-fontes das aplicações são gerados.

De acordo com a visão de [38], os conceitos de mapeamento e transformação devem ser explicitamente distinguidos, e em conjunto, podem estar envolvidos no mesmo processo, denominado processo de transformação. Ainda segundo os autores [38], dentro do processo de transformação, a especificação de mapeamento precede a definição de transformação.

A especificação do mapeamento é uma definição das correspondências entre metamodelos [38], ou seja, um metamodelo para a construção de um PIM e outro para a construção de um PSM. A definição de transformação contém uma descrição para transformar um modelo para outro, ou seja, consiste na concepção e realização

de programas de transformação em uma linguagem de transformação segundo as recomendações da OMG [39]. Um modelo de transformação é gerado a partir de um modelo de correspondência e um programa de transformação em execução é baseado neste modelo de transformação [31].

Baseado nestas definições, podemos entender que a especificação de correspondência apresenta a lógica utilizada para estabelecer as relações entre dois metamodelos. E a definição de transformação contém regras precisas para transformar um modelo em um outro modelo.

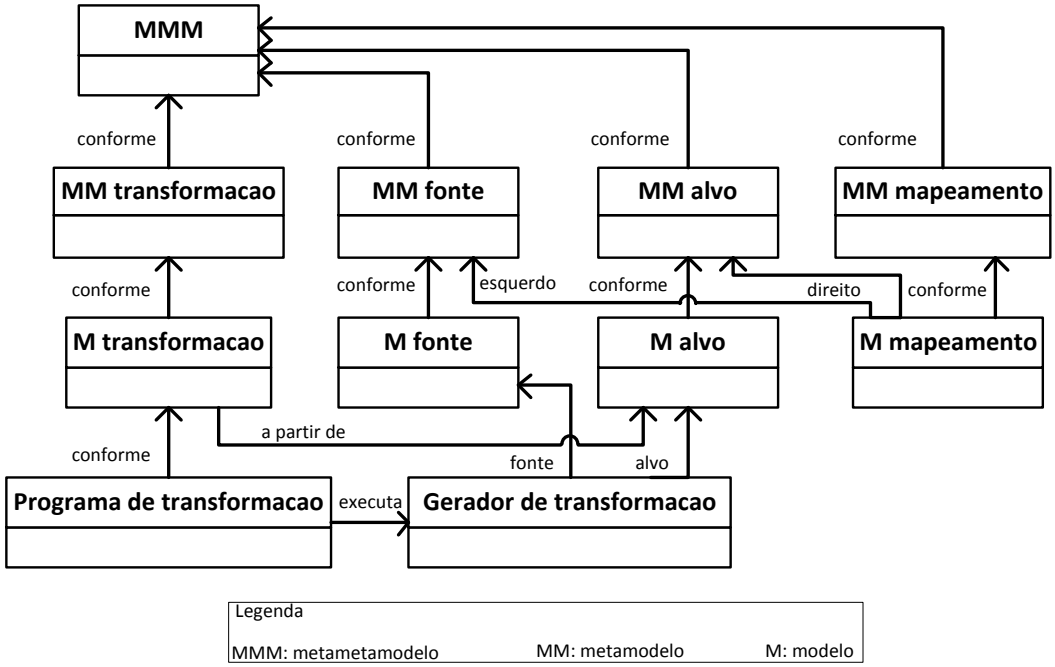

Figura 3.1: Uma proposta de arquitetura para transformação de modelos [38]

A arquitetura para a transformação de modelos com suas relações para cada entidade envolvida são apresentadas na Figura 3.1. Esta figura apresenta a separação entre o modelo de correspondência e modelo de transformação. As entidades desta arquitetura são listadas a seguir:

- MMM (metametamodelo): por exemplo, MOF ou Ecore;
- MM fonte e MM Alvo (metamodelo fonte e metamodelo alvo): por exemplo, o metamodelo UML;

- M fonte e M Alvo (modelo fonte e modelo alvo): por exemplo, um modelo de uma biblioteca em modelagem UML que posteriormente deve ser transformado em modelo Java;

- MM mapeamento (metamodelo de correspondência): utiliza linguagem para modelagem de correspondências entre os elementos de um metamodelo fonte e os elementos de um metamodelo alvo;

- M mapeamento (modelo de correspondência): modelo onde correspondências entre dois metamodelos estão contidas;

- MM transformação (metamodelo de transformação): utilizando de formalismo (através de uma linguagem de transformação), permitindo a criação precisa de transformações de um modelo fonte em um modelo alvo;

- M transformação (modelo de transformação): descreve a transformação de modelos;

- Programa transformação (programa de transformação): um programa executável para realizar a transformação;

3.1.1 Linguagens de Transformação de Modelos

Diversas linguagens para transformação de modelos foram desenvolvidas, dentre elas podemos citar a MOF QVT da OMG [55], o *MOFScript* do Eclipse Project [50], e a ATL da Universidade de Nantes [22]. Uma breve descrição de cada linguagem de transformação acima listado será relatado a seguir:

- ***Linguagem de transformação MOF QVT:*** MOF QVT é uma linguagem híbrida, ou seja, aceita construções declarativas e imperativas. É formada por três sublinguagens: núcleo *(Core)*, relações *(Relations)* e mapeamentos operacionais *(Operational Mappings)*, sendo as duas primeiras da arquitetura declarativa e a última imperativa dentro de sua arquitetura. Estes dois níveis da arquitetura da parte declarativa constitui o enquadramento para a execução da semântica para a parte imperativa da linguagem [55];

- ***Linguagem de transformação MOFScript:*** A ferramenta *MOFScript* é uma linguagem de transformação que implementa modelo *MOFScript* para texto. A ferramenta é desenvolvida sob forma de *plug-in* do Eclipse. É baseada em *EMF* e *Ecore*. O *MOFScript* suporta a análise, verificação e execução de scripts [50];

- ***Linguagem de transformação ATL:*** É uma linguagem de transformação tanto para um metamodelo como para uma sintaxe concreta textual. No campo da MDE, a ATL fornece aos desenvolvedores um meio de especificar a forma de produzir uma série de modelos alvo a partir de um conjunto de modelos fonte. ATL é uma linguagem híbrida. É composta de regras *(matched rule*) que definem como os elementos do modelo fonte são combinados para criar e inicializar os elementos do modelo alvo. A linguagem permite refatoração código através da definição de bibliotecas [22].

Para o desenvolvimento do trabalho da dissertação, a linguagem de transformação ATL foi utilizada, pois, conforme especificado anteriormente, é uma linguagem que realiza transformações modelo-a-modelo e modelo-a-texto. O ambiente de utilização da linguagem ATL se apresenta como forma de *plug-in* para o Eclipse, tornando-se de fácil utilização.

3.2 Teste de *Software* para Computação em Nuvem

Atualmente, muito se fala em relação à Teste de *Software* para ambientes de Computação em Nuvem [30] [28] [5]. Em [30], os autores exibiram alguns pontos de vista relacionados à este assunto. São eles:

"Os testes basicamente se alinham ao conceito da nuvem SaaS. Eles fornecem a capacidade de testar aproveitando os recursos da nuvem, trazendo assim os mesmos benefícios que a nuvem proporciona para os clientes."(Vinita Ananth, Director - APJ Região, HP Software-as-a-Service)

"Testes em nuvem utilizam ambientes de Computação em Nuvem e procuram simular mundo real do usuário através de carga ou estresse em sites de teste."(Nivedan Prakash)

"Os testes em nuvem é a resposta para o teste de desempenho que se origina dentro da infraestrutura de um cliente. Quando usamos testes de nuvem, nós tiramos proveito do

hardware e largura de banda que mais se aproxima da nossa observação e condições do mundo real."(R V Ramanan, presidente - Global Delivery e Chief Software Architect, Hexaware Technologies)

Existem quatro formas diferentes de se testar *software* para ambientes de Computação em Nuvem [30] [5]:

- **Teste de um SaaS na nuvem:** Assegura a qualidade de um *software* e seus serviços oferecidos na nuvem com base nos requisitos funcionais e não funcionais;
- **Teste de uma nuvem:** Valida a qualidade de uma nuvem, com uma visão externa, com base nos recursos previstos e serviços específicos da nuvem;
- **Teste interno de uma nuvem:** Verifica a qualidade de uma nuvem do ponto de vista interno, baseado na infraestrutura de uma nuvem e suas capacidades específicas. Este tipo de teste é realizado pelos fornecedores das nuvens, uma vez que possuem acessos a infraestrutura interna e as conexões entre os serviços, segurança, gerenciamento e monitoramento da nuvem;
- **Teste sobre a nuvem:** Testa o serviço baseado em aplicações das nuvens, incluindo nuvens privadas, públicas e híbridas. É um tipo de teste realizado geralmente por fornecedores de aplicativos.

3.3 Técnicas e Ferramentas de Implementações para Teste em Computação em Nuvem

Diversas técnicas e formas de testes do processo de desenvolvimento de *software* tradicional são adequados para Computação em Nuvem. Em [5], os autores citam algumas técnicas e ferramentas que são utilizadas para os testes nestes ambientes:

- **Simulação:** Para reduzir a complexidade e melhorar a qualidade dos testes, um simulador de nuvem é necessário para a realização de testes. Através da simulação, o testador pode se concentrar em problemas no nível de componente de nuvem em particular, e analisar o comportamento do sistema sob vários cenários;

- **Emulação de Serviço:** Os sistemas hospedados na nuvem podem utilizar serviços de diversos fornecedores. Técnicas convencionais têm sido adaptadas para emular as funcionalidades externas em nível de interface de serviço;

- **Teste Paralelo:** A programação paralela se torna uma prática natural para aplicações construídas sobre a nuvem. Testes podem se beneficiar desta prática em trabalhos paralelizados reduzindo tempo e custo;

- **Virtualização de ambientes:** Ao longo do ciclo de vida de testes do *software*, testes de regressão são necessários para manter ambientes de testes para várias versões do *software* e suas plataformas. As máquinas virtuais podem ajudar a aliviar e acelerar o processo e reduzir o custo do teste.

3.4 Teste para Infraestrutura como Serviço (IaaS)

IaaS é um modelo de nuvem em que o *hardware* é virtualizado. Neste modelo em particular, o fornecedor de serviço é proprietário dos equipamentos: servidores, armazenamento, infraestrutura de rede, e assim por diante. O desenvolvedor cria um *hardware* virtual sobre o qual desenvolve aplicações e serviços. Essencialmente, um fornecedor de nuvem IaaS cria um serviço de utilidade de *hardware* onde o usuário dispõe de recursos virtuais conforme o necessário [79].

No que se refere a testes, podemos considerar que a virtualização na Computação em Nuvem é uma contribuição importante das infraestruturas dos provedores de serviços. Esta infraestrutura possui a capacidade de consolidar vários serviços de uma nuvem em poucos servidores físicos, fornecendo assim, benefícios como a eficiência por meio da redução dos custos, espaço e consumo de energia.

3.4.1 Trabalhos relacionados com Testes em IaaS

Teste como Serviço sobre Nuvens [93]

O trabalho apresentado, o *Testing-as-a-service (TaaS)*, é uma plataforma de teste em um ambiente de nuvem para fornecer testes de *software* para usuários finais

[93]. Esta plataforma tem por objetivo economizar o custo com manutenção e esforço de atualização e serviços.

Este trabalho desenvolve um protótipo de TaaS, para ser utilizado em diversas nuvens, visando avaliar a escalabilidade da plataforma, distribuição de tempo no agendamento de tarefas de teste e processamento de teste sobre a nuvem. Esta plataforma adota técnica de Computação em Nuvem para construir as infraestruturas de recursos de forma elástica. Além disso, fornece vários tipos de testes de serviços aos usuários. Esta plataforma facilita os testadores na escolha do ambiente de testes, definindo também os testes de unidade [93].

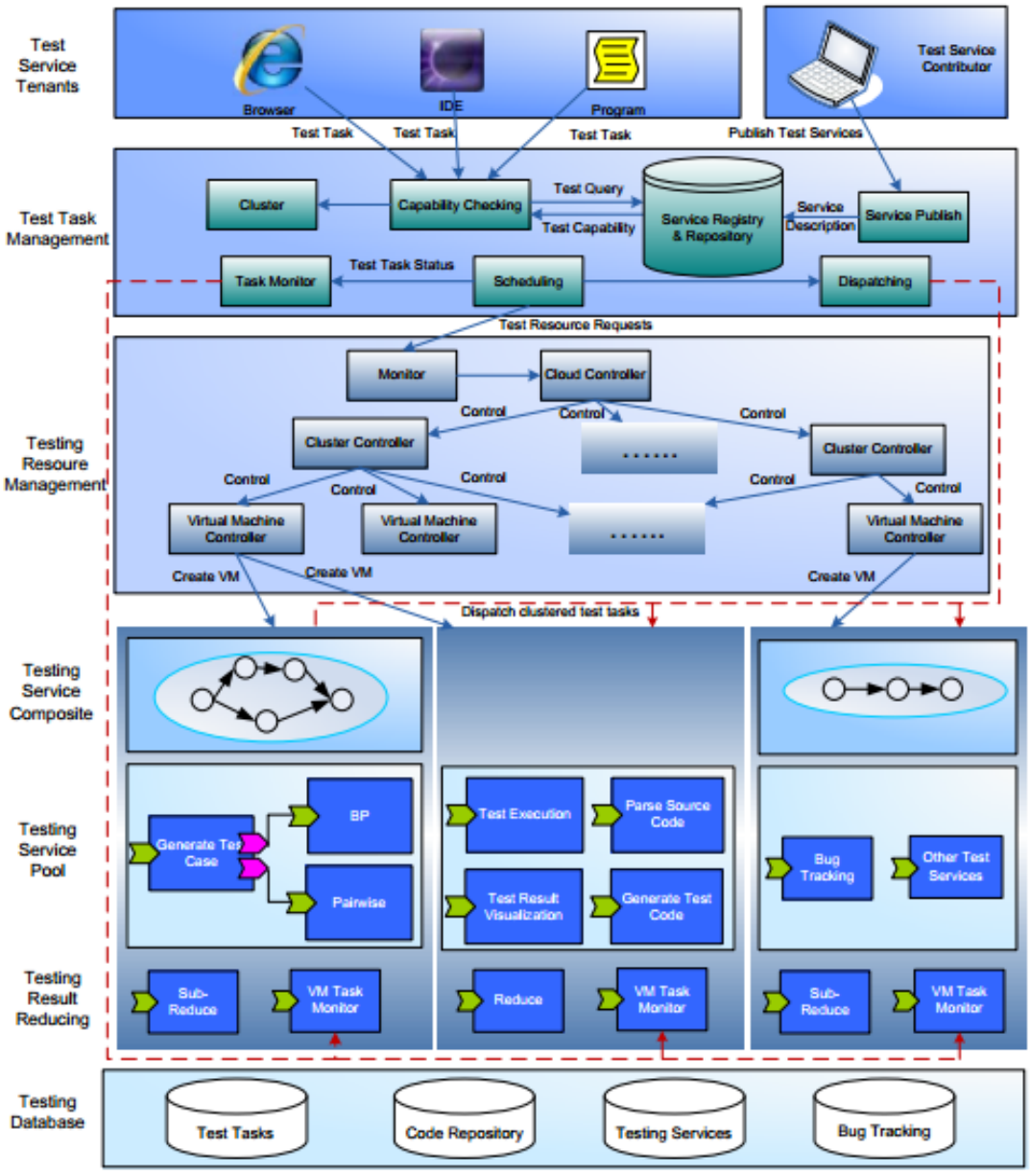

Figura 3.2: Arquitetura de um TaaS [93]

A Figura 3.2 ilustra uma arquitetura de uma nuvem TaaS com cinco camadas, propostas neste trabalho, sendo elas:

1. ***Testing Service Tenants:*** Camada onde o existe o serviço de teste do cliente e a camada de serviço do contribuinte de testes;
2. ***Test Task management:*** Camada de teste para o gerenciamento da tarefas de teste;
3. ***Testing Resource management:*** Camada responsável pela gestão de recursos da camada de teste;
4. ***Testing Service Composite, Service Pool, Result Reducing:*** Camada de teste que contém três componentes: A camada de onde se realiza composição de serviços para testes, a camada serviço de armazenamento de testes *(pool)* e a camada do serviço de redução de testes *(MapReduce) [89]*;
5. ***Testing Database:*** Camada base para dados de teste.

O trabalho trouxe diversas propostas [93]. Entre elas, propõe uma arquitetura em forma de infraestrutura e serviços, conforme especificado anteriormente, para a realização de diversos testes. Trabalha também com técnicas de ontologias, utilizando *Semantic Web Rules Language (SWRL)*, para a modelagem das regras de correspondência. As regras de correspondência foram utilizadas para construir propostas de diversas tarefas de testes, conforme listados a seguir:

- **Processo de gerenciamento de tarefas:** O trabalho propõe a utilização de técnicas de mineração de dados, para a realização de testes, conforme a solicitação de diversos usuários e explora algoritmos de escalonamento para alocar recursos, na realização de testes sobre as tarefas destes usuários;
- **Modelagem de tarefas de teste de capacidade:** Realiza a construção das tarefas de testes de capacidade, modeladas com classe *Web Ontology Language (OWL)* definida como *TestTask* que contém subclasses OWL, como *TestResource* que apresenta os requisitos de *hardware* e *software* para uma tarefa de teste;
- **Testes de *clustering*:** Tarefa para realizar uma avaliação do ambiente do sistema, para, por exemplo, verificar a necessidade de servidor *Web*. Os autores propõe

baseado em *K-Medoid* [26], para atribuição de tarefas para infraestrutura da nuvem;

- **Programação de tarefas de teste:** Tem como objetivo o encontro de uma função com as melhores soluções para a priorização da execução das tarefas de teste. Os autores trabalham considerando o tempo de espera e o valor de cada tarefa como seleção de critérios;

- **Testes de tolerância a falhas:** O trabalho propõe um algoritmo, baseado em heurísticas, para tarefas do tipo *backup*, podendo tolerar falhas simples em qualquer processador.

D-Cloud: Design de um ambiente de Teste de *Software* confiável para Sistemas Distribuídos utilizando a tecnologia de Computação em Nuvem [6]

Neste trabalho, uma proposta de um ambiente de Teste de *Software* é apresentada, chamada de D-Cloud [6]. Ela utiliza tecnologia de Computação em Nuvem e máquinas virtuais com o uso de injeção de falhas. D-Cloud possui um ambiente onde são executados vários testes automaticamente, de acordo com um determinado cenário. D-Cloud permite o teste de tolerância a falhas através do uso de máquina virtual.

Este ambiente foi construído usando o ambiente de *software Eucalyptus* [48] e *Quick EMUlator (QEMU)* [64] e uma linguagem de descrição de configuração do sistema. O cenário de injeção de falhas é escrito em XML. D-Cloud e permite que um usuário configure e teste um sistema distribuído sobre a nuvem, reduzindo o custo e o tempo de realização dos testes [6].

A Figura 3.3 ilustra os procedimentos de funcionamento do D-Cloud seguindo uma ordem de numeração. Os detalhes de cada numeração são listados a seguir [6]:

1. Um testador descreve um cenário de teste, arquivos de configuração e *scripts*;

2. O testador carrega o cenário de teste, com os arquivos de configuração, o *script* de teste e os arquivos para o *frontend* do D-Cloud. Se o testador usar imagens do Sistema Operacional (SO) personalizadas pelos testadores, são realizados *uploads* pelo testador, dessas imagens, para o *frontend* do D-Cloud;

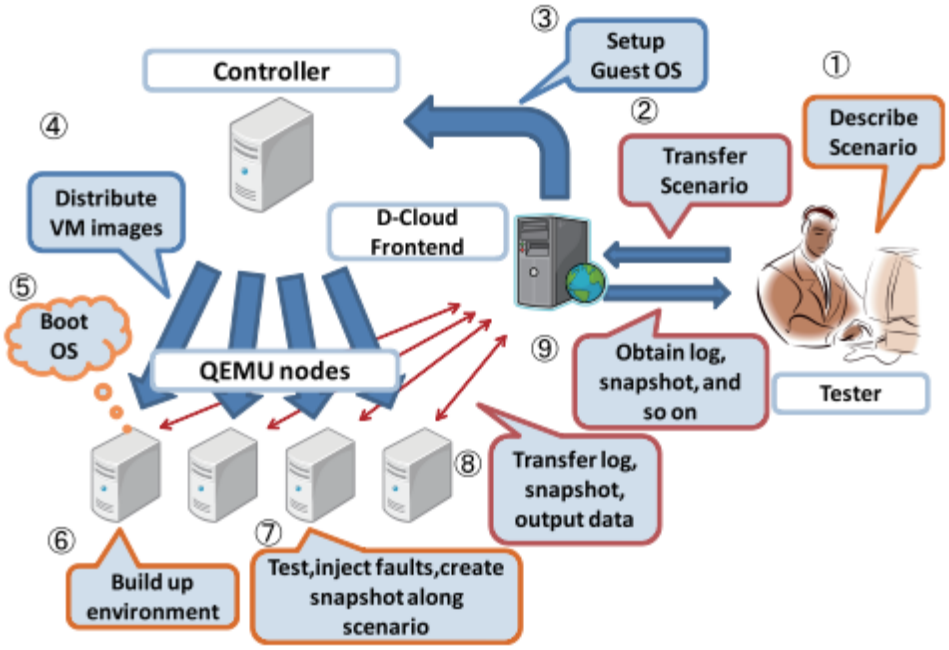

Figura 3.3: Funcionamento do D-Cloud [6]

3. São instruções para configurar o sistemas operacionais convidados, para o teste, no controlador do *Eucalyptus*. Se um testador tem suas próprias imagens de SO, o D-Cloud transfere as imagens do sistema operacional para o controlador do *Eucalyptus*. Caso contrário, o testador deve selecionar um sistema operacional pré-configurado, ou seja, uma imagem fornecida pelo *Eucalyptus* de sistema operacional para ser transferido;

4. O controlador seleciona um nó de QEMU disponível do *Eucalyptus* para iniciar os sistemas operacionais convidados. E, para transferir as imagens do SO, para os nós QEMU selecionados;

5. São iniciadas as imagens do SO nos nós QEMU selecionados;

6. O D-Cloud transfere os arquivos utilizados nos testes a serem executados para cada sistema operacional convidado. Em seguida, configura um ambiente de teste;

7. De acordo com o cenário de teste, o teste de sistema e injeção de falhas são executados e *snapshots* são capturados;

8. Após os testes do sistema, cada sistema operacional convidado transfere a saída de dados, registros e *snapshots* para o *frontend* do D-Cloud;

9. O testador, em seguida, transfere os dados de saída, com os *snapshots* e os registros obtidos através dos testes de sistema.

3.5 Teste para Plataforma como Serviço (PaaS)

Uma nuvem de PaaS fornece recursos necessários para construir aplicações e serviços a partir da *Internet*, sem a necessidade de baixar ou instalar o *software*. Pode-se dizer que PaaS é um centro especial de recursos de *Internet*, em forma de plataforma aberta, para fornecer serviço de *Application Programming Interface (API)* [42]. API - em português, Interface de Programação de Aplicativos - é o conjunto de padrões de programação que permite a construção de aplicativos e sua utilização de maneira não tão evidente para os usuários [87].

A partir da definição, pode-se dizer que teste em PaaS é o teste sobre uma plataforma hospedada. Assim, usuários de testes acessam os serviços hospedados remotamente, usando o navegador *Web*, para a realização de testes funcionais e não funcionais.

3.5.1 Trabalhos relacionados com Testes em PaaS

Uma abordagem para a Composição de Serviços e testes para Computação em Nuvem [90]

O trabalho apresentado visa facilitar a implementação da composição de serviços [90]. A técnica utilizada para esta composição se aproxima à composição do *Google Guice* [23] e da ferramenta *Spring* [80]. Este trabalho propõe uma ampliação com base em testar o fluxo de trabalho de composição de serviços. O processo de teste pode ser executado em paralelo. Um outro aspecto abordado neste trabalho é a utilização de ontologias de domínio para especificação das interfaces dos serviços e a aplicação do serviço de *MapReduce* [89] para acelerar o processo de teste.

Este trabalho propõe um mecanismo *MapReduce* para acelerar o processo de teste abordando as seguintes questões [90]:

- Permitir que o mecanismo de composição de serviços indique implementações de serviço específico;
- Desenvolver o mecanismo testes para a *Workflow* utilizado na composição de serviços;
- Técnica de teste de forma paralela para acelerar o processo de teste.

O fluxo de trabalho de testes na Composição de Serviço depende tanto do número de serviços de nuvem disponíveis e do tamanho de dados que os serviços de nuvem precisam lidar. Este trabalho segue um grupo de serviços para testar fluxos de trabalho. São eles [90]:

1. **Geração de oráculo para Composição de Serviços:** Um caso de teste é um par (entrada de teste, saída esperada). Mas, frequentemente, um resultado esperado é difícil de obter. Mecanismos de geração de oráculo de testes podem ser usados para determinar o resultado esperado;
2. **Teste de Unidade:** Uma vez que um caso de teste é formado com um oráculo, este pode ser usado para testar a aplicação do novo serviço. Os Casos de Teste também podem ser classificados para ajudar a nuvem e os usuários, na seleção de quais e quantos casos de teste mais relevantes serão realizados primeiramente;
3. **Teste de Integração:** Uma vez que o fluxo de trabalho for concluído, pode-se aplicar o seu cenário de utilização, como *scripts* de teste, para testar o fluxo de trabalho. Um uso de cenário para um serviço pode envolver outros serviços. Assim, este cenário utilizado proporciona uma relação entre estes dois serviços, ou seja, que a relação indica que esses dois serviços são ligados uns aos outros. Além disso, o uso de cenários para um fluxo de trabalho pode ser armazenado e utilizado posteriormente para a composição de serviço ou como base para *scripts* de teste;
4. **Teste contínuo:** É um processo de teste que pode ser aplicado durante as etapas de desenvolvimento e execução de composição de serviços. Em testes, se apresentam principalmente sob a forma de testes de regressão, e são aplicados durante o tempo de desenvolvimento da aplicação. Nos ambientes das nuvens, como novas aplicações podem ser compostas a partir de serviços existentes, o teste contínuo pode ser aplicado antes e após da composição de serviços, e

mesmo durante execução, como parte do monitoramento de serviços e/ou política de processos de execução;

5. **Geração de entrada orientada a metadados de teste:** Para entradas e saídas de interfaces de serviço, pode-se usar metadados para definir entradas de teste. Por exemplo, se o comprimento ID de usuário para um site deve ser de 64 bits, um teste simples de entradas podem ser gerados dos 64 bits randomicamente;

6. **Executar testes de processamento por nível de serviço:** Como o número de composição de serviços pode ser grande em uma nuvem, um grupo de teste pode ser aplicado com o serviço *MapReduce* para executar testes em paralelo. A idéia é usar diferentes combinações de implementações de serviço para o mesmo fluxo de trabalho. Os dados de entrada do fluxo de trabalho é obtido por desenvolvedores ou gerados a partir dos metadados.

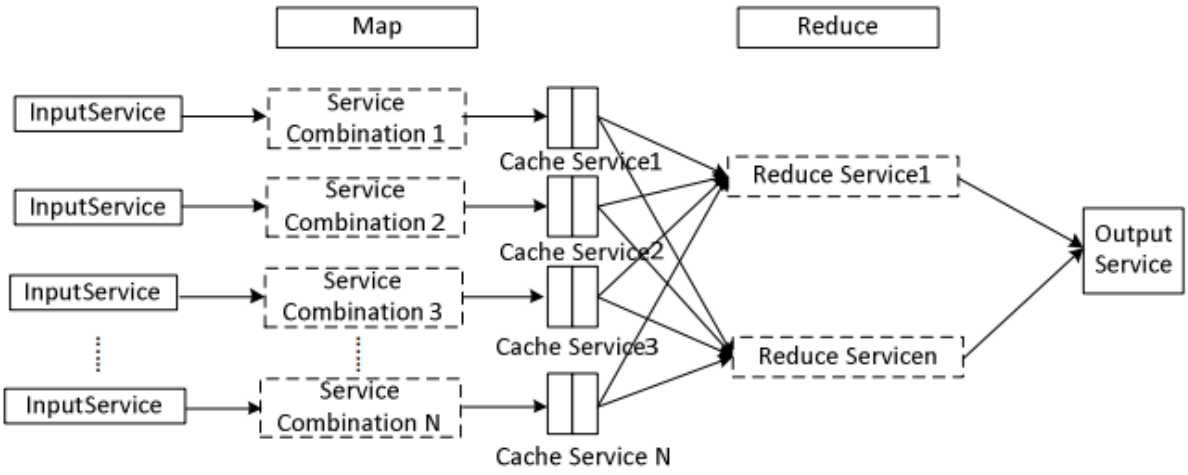

Figura 3.4: Serviço de geração de oráculo de teste com MapReduce [90]

Taxonomia e Racionalização de Requisitos para Teste de *Software* baseado em Infraestrutura de nuvem [68]

O trabalho apresenta uma taxonomia de padrões para testes para nuvem. Um estudo de caso foi desenvolvido e aplicado em uma Plataforma como Serviços (PaaS). Este domínio foi selecionado por não haver um extenso estudo sobre aplicações de testes nesta infraestrutura, bem como requisitos para apoiar tais testes [68].

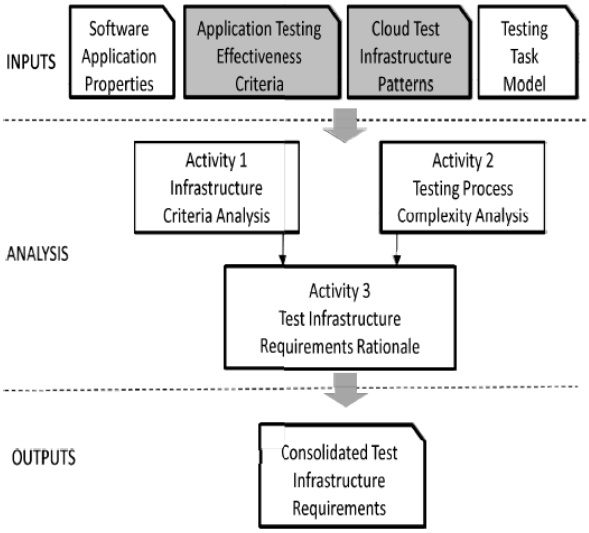

Figura 3.5: Infraestrutura de nuvem de teste para de processo de derivação requisitos [68]

As entradas de propriedades de aplicação do *software* descrevem o uso e consumo de recursos, enquanto o modelo de tarefa é uma coleção de todas as tarefas em um determinado processo de teste, conforme ilustrado na Figura 3.5.

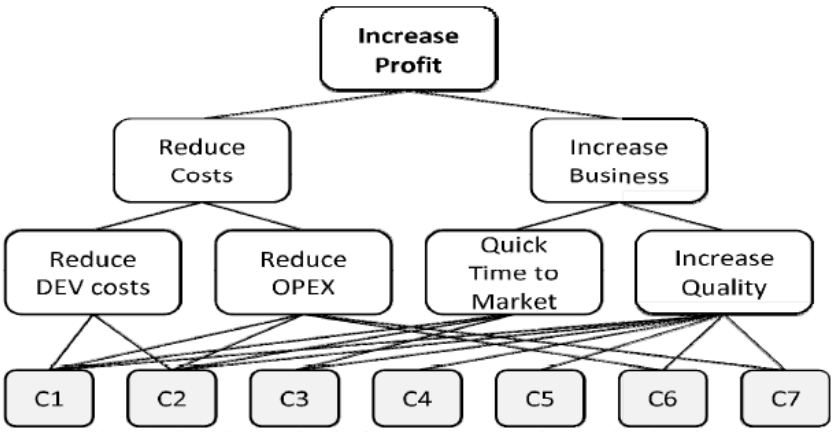

Figura 3.6: Derivação de 7 de critérios (C1-C7) para Testes de *Software* na aplicação dos objetivos de negócios [68]

A taxonomia proposta é formada por 5 padrões de teste e 7 critérios [68]. Os padrões servem como base para a montagem dos requisitos de testes que poderão ser aplicados em qualquer infraestrutura. Os cinco padrões são: Teste na nuvem, Suíte de teste na nuvem, Sistema em teste na nuvem, Teste da nuvem em vários locais e Intermediação de testes em vários locais.

Os critérios de testes são derivados de uma análise dos objetivos de negócio, relacionado com o Teste de *Software*. A análise de objetivos em engenharia de requisitos, envolve relacionar ou transformar os objetivos de negócio para requisitos

funcionais ("que os objetivos sejam alcançados") e requisitos não funcionais ("como os objetivos devem ser alcançados") [68].

Como representado na Figura 3.6, o trabalho apresentado relaciona 4 critérios que podem ser decompostos em 7 critérios (C1 - C7), para tornar o processo de testes eficaz. Os sete critérios relacionados são: Eficácia de custo (C1), simplicidade (C2), representação direcionada (C3), controlabilidade (C4), observabilidade (C5), previsibilidade (C6) e reprodutibilidade (C7).

3.6 Teste para *Software* como Serviço (SaaS)

Atualmente, o SaaS está se tornando um mecanismo dominante para o consumo de *software* direcionado a usuários finais [74]. Informalmente, o SaaS pode ser descrito como *software* implementado como um serviço, hospedado e acessado através da *Internet*, sem a necessidade de usuários para implantar e manter uma infraestrutura, ou seja, sem a necessidade do usuário instalar o *software* a ser utilizado no seu computador local.

Da perspectiva de um fornecedor de SaaS, os benefícios da utilização deste tipo de serviço, surgem com a economia gerada. Pois, através desse serviço, um grande número de clientes acessam o mesmo *software*, através de um serviço compartilhado, onde o *software* se encontra hospedado em um servidor [74].

3.6.1 Trabalhos relacionados com Testes em SaaS

Engenharia *Multi-Tenant* para *Software* como Serviço [74]

Este artigo, propõe um auxílio, através de modelagem de variabilidade, para a evolução do *software* para *multi-tenant* de um sistema em SaaS. Também trata dos desafios da engenharia, que o fornecedor do *software* precisa resolver, para a adaptação no ambiente de Computação em Nuvem.

Multi-tenancy permite que um único aplicativo seja emulado em múltiplas instâncias. Com o *multi-tenancy*, um único aplicativo pode ser compartilhado por mui-

tas organizações. No que se refere a testes, o trabalho comenta sobre duas questões a serem consideradas para este tipo de sistema [74]:

- A primeira é quando uma nova instância do aplicativo é aberta. Como devem ser realizados os testes, para verificar se as instâncias que já estão em utilização, não sejam afetados pelas alterações introduzidas por esta nova instância;
- Em segundo lugar, como testar, de forma eficiente, para que o sistema satisfaça as necessidades dos diferentes usuários que se conectam ao serviço.

Os autores [74] consideram a segunda questão como relevante para abordagem por eles trabalhadas. Dado um grande grau de semelhança que exista entre os usuários, quais recursos de teste significativos podem ser realizados, para o sistema ser totalmente testado, em todos os cenários aplicáveis.

Na primeira questão, os casos de teste são elaborados considerando que não existam no atual pacote de teste, mas que devem ser testados, uma vez que o novo usuário conectar ao serviço [74]. Testes são gerados para salientar as alterações do programa, ou seja, execução das mudanças e propagação dos efeitos das alterações na saída do programa. O problema geral dessa questão pode ser tratado através de duas etapas.

Na primeira etapa, uma análise de dependência de controle é feita para encontrar dados de entrada para executar a alteração. Na segunda etapa, modifica-se o caminho da alteração, atingindo os dados de entrada para garantir que as saídas geradas pelo programa sejam diferentes, com ou sem alteração [74]. Para alcançar a alteração, pode-se explorar as pré-condições das operações, em vez de executar um controle refinado análise de dependência. Finalmente, para propagar o efeito das alterações executadas, pode-se analisar as operações de pós-condições para encontrar um teste adequado.

A segunda questão relevante para testar sistemas de multiusuários é a elaboração de uma estratégia de teste que explore a semelhança entre usuários e estruturas de acordo com uma suíte de testes. Para esta estratégia, os autores propõem um teste baseado em árvores. Neste teste, conforme ilustra a Figura 3.7, o nó raiz de uma árvore de teste, capta o conjunto de casos de teste que precisam de ser testados para todos os usuários. Cada nó intermediário da árvore irá capturar um conjunto de casos

de teste que precisam ser testados para um subconjunto de usuários. Assim, um particionamento do conjunto de usuários é dado pela raiz até as folhas da árvore, pelo seus caminhos.

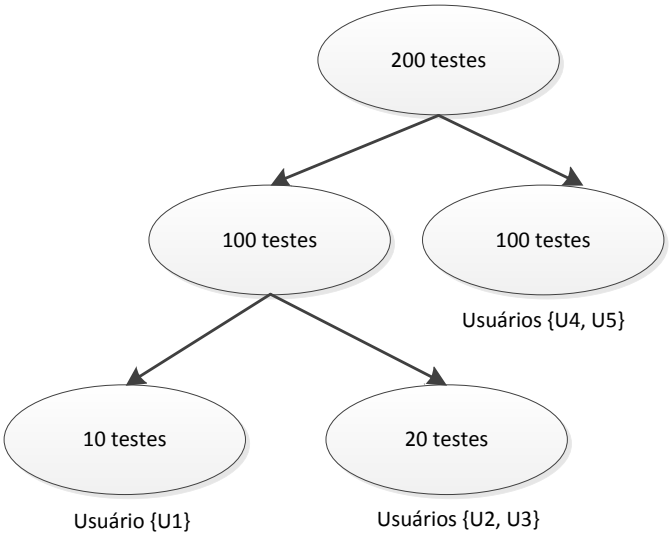

Figura 3.7: Árvore de teste para sistemas multiusuários em SaaS [74]

Automação de Testes em plataforma SaaS [44]

Este artigo ilustra as principais características do *framework* de teste Apex [71] junto com contribuições realizadas para melhorar a capacidade de testes em uma plataforma SaaS.

Código Apex é uma linguagem fornecida pela plataforma *Force.com* [70], caracterizada como *on-demand*, utilizada para personalizar e melhorar funcionalidades da aplicação construída [71]. Tal como acontece com a maioria das linguagens de programação, o código Apex fornece construções de linguagem regulares, como tipos de dados, variáveis, expressões, *loops*, etc. Além disso, o Código Apex também oferece as seguintes características que são mais específicas para o Plataforma *Force.com*: Ferramenta de pesquisa semelhante ao *Structured Query Language (SQL)*, *Triggers* e integrações com Serviços *Web* [44].

Como acontece com qualquer aplicativo de *software*, na plataforma *Force.com*, as aplicações requerem testes para garantir a qualidade. Assim, o código Apex fornece um *framework* para automatizar o processo de validação de testes. Uma

vez que todo o código Apex está na própria plataforma, os clientes não têm que se preocupar com o manutenção dessas suítes de automação de testes.

Os Testes em Apex são de métodos de classe que são executados sem inserção de dados. Os métodos Apex são conhecidos com a palavra chave *"TestMethod"* e são tratados como testes individuais. Estes métodos podem executar qualquer código Apex, como qualquer outro método regular Apex. Eles são tratados como testes quando são executados dentro de um contexto de testes [44].

Todos os testes escritos para uma aplicação em Apex podem ser executados usando a interface do usuário *Salesforce.com*. A interface do usuário fornece a opção executar todos os testes para uma classe específica ou para todas as classes.

Depois da execução dos resultados dos testes, os seguintes dados podem ser relatados:

- Número de testes executados;
- Número de falhas encontradas;
- Percentagem de cobertura de código;
- Informações de perfil para a execução do teste.

Avaliação de Desempenho e Escalabilidade em nuvens SaaS [17]

Este trabalho se concentra na avaliação do desempenho e medições de escalabilidade de *software* para SaaS. O trabalho propõe também, novos modelos em forma de gráficos e métricas para avaliar o desempenho do sistema SaaS e analisar a escalabilidade das nuvens [17].

Os autores comentam que existem três diferentes ambientes de teste na nuvem que eles classificam como níveis, conforme ilustrado na Figura 3.8. Os três níveis são [17]:

- *SaaS-level* - Onde uma aplicação individual (ou instância), em nuvem privada (ou pública), é avaliada e validada em uma nuvem para verificar o desempenho e escalabilidade do sistema SaaS;

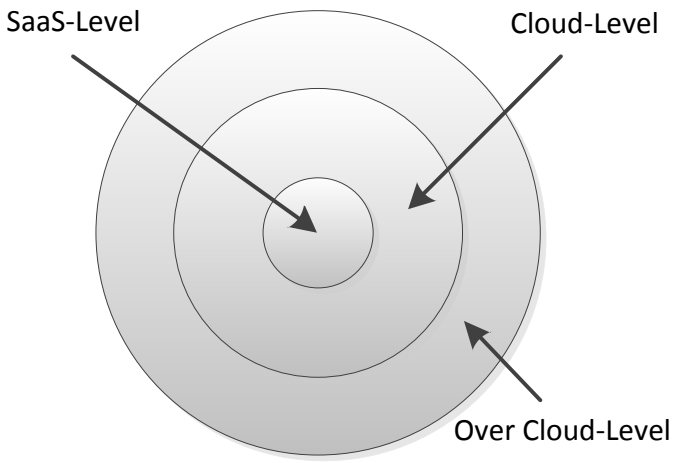

Figura 3.8: Avaliação de Performance e Escalabilidade em nuvens [17]

- *Cloud-level* - Várias instâncias de SaaS, em nuvem privada (ou pública), são avaliadas e validadas para desempenho e escalabilidade;
- *Over Cloud-level* - Aplicações em nuvem de infraestruturas híbridas, ou seja, nuvem que combina os modelos das nuvens públicas e privadas, são avaliadas e validadas de ponta a ponta para verificação de desempenho e escalabilidade.

Os autores utilizam do gráfico do tipo **Radar** para desenvolverem os modelos formais. Também desenvolveram quatro métricas para apoio na análise, monitoramento e avaliação de desempenho quantitativo e, também, parâmetros e indicadores para avaliação de escalabilidade.

A primeira métrica apresentada é de **Alocação de Recursos e Utilização**, pois existe forte interesse da parte dos fornecedores de nuvens em saber o custo benefício para avaliar a alocação de recursos e a utilização da nuvem [17]. Os autores utilizam um polígono baseado em modelo gráfico como base para definir uma atribuição de medidas de recursos genéricos e medidas de utilização para avaliar os aplicativos SaaS em nuvens.

Dentro desta métrica eles utilizam o **Medidor de Alocação para Recursos Uniformes**, denominado *CRAM (S, t)*, onde apresentam a quantidade total de alocações de recursos para *S* em uma nuvem no tempo avaliado, denominado *t*. E utilizam também a métrica de **Medição Uniforme de Recursos Alocados**, que apresenta e ava-

lia os recursos utilizados de uma nuvem de qualquer aplicativo SaaS dada durante a validação do sistema.

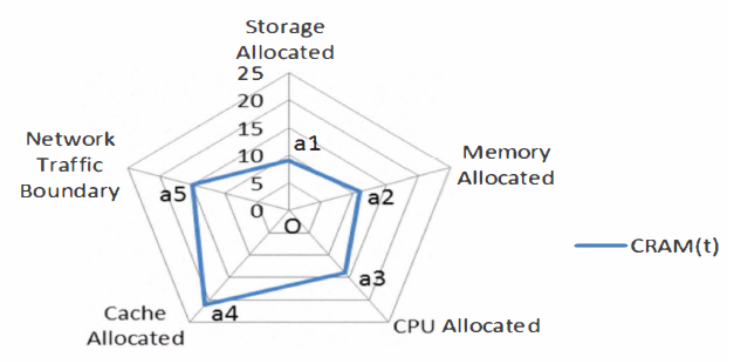

Figura 3.9: Medida de Alocação de Recursos Computacionais (CRAM) [17]

A Figura 3.9 ilustra um exemplo, retirado de [17] que descreve cinco tipos de recursos computacionais alocados para uma aplicação SaaS, incluindo CPU, *cache*, memoria, uso de rede e armazenamento em disco. Cada tipo de recursos de computação tem a sua própria unidade de alocação, no seu eixo correspondente no gráfico, como também, deve existir sua própria unidade de apresentação e escalas. Por exemplo, *MB* e *GB* para memoria. O polígono azul representa o total de recursos computacionais atribuídos *S* no tempo avaliado *t*.

A segunda métrica apresentada é a de **Desempenho do Sistema** [17], pois é comum para recolher e medir os indicadores de desempenho de um sistema SaaS, através de testes de desempenho e avaliação com base na *Quality of Service (QoS)* e no *Service-level Agreement (SLA)*. Um conjunto de parâmetros de desempenho inclui a velocidade do processamento (como tempo de resposta do usuário), e a utilização do sistema, através da confiabilidade e disponibilidade.

A terceira métrica apresentada no trabalho é a de **Medida de Capacidade de Sistema de Nuvens** [17], onde apresentam a as seguintes medidas:

- **Medida de Carregamento do Sistema (SLM)**, onde esta medida aplicada em um *software* pode ser definida como a sua capacidade para lidar com cargas de sistema com base nos recursos do alocados;
- **Medidor de Capacidade do Sistema (SCM)**, no qual é avaliado a capacidade da aplicação de um sistema SaaS implantado;

- **Eficiência de Capacidade do Sistema (SEC)**, que se torna útil e prático para os fornecedores e clientes de SaaS na questão de descoberta e eficiência da capacidade real do sistema, baseado na utilização dos recursos computacionais para uma aplicação. Esta necessidade torna-se óbvia e crítica devido para o custos dos recursos alocados.

E a quarta métrica definida pelos autores é a **Medida de Escalabilidade de Sistemas** [17]. Esta medida define um medidor de escalabilidade do sistema baseado na capacidade efetiva durante um dado período de tempo. Para cada aplicação SaaS, o fornecedor deve certificar-se de oferecer aos seus clientes os requisitos mínimos de desempenho sobre diversas cargas do sistema, baseado em seus recursos computacionais subjacentes em um ambiente de nuvem escalável.

3.7 Análise dos Trabalhos Relacionados

Nesta seção, itens que demonstraram relevância foram destacados entre os trabalhos relacionados. A análise destes trabalhos, baseados nestes itens avaliativos, e a comparação com a abordagem proposta nesta dissertação, são fundamentais para a definição das direções que servirão de base para avaliações, conclusões e considerações do presente trabalho.

Dentre os itens avaliados, citam-se:

1. O trabalho de testes em nível de infraestrutura, permitindo que avaliações sejam realizadas em nível de alocação de recursos de *hardware*, servidores, máquinas virtuais, serviços de controle de demanda, etc. Permitindo que um serviço, aplicativo ou mesmo uma infraestrutura, tenha sua eficiência garantida, quando disponibilizada aos usuários;

2. O trabalho de testes em nível de desenvolvimento, que permite os desenvolvedores e testadores de *software* tenham recursos adequados, dentro do ambiente de desenvolvimento da nuvem, para testar a aplicação ou serviço desenvolvido;

3. O trabalho de testes em nível de aplicação, que permite um melhor controle de qualidade nos serviços de aplicativos desenvolvidos e disponibilizados para os usuários finais, dentro do ambiente da Computação em Nuvem;

4. Ambiente interoperável, garantindo que os casos de teste elaborados sejam aplicados em diversos ambientes de Computação em Nuvem. E também, oferecendo uma expansão no processo de desenvolvimento de caso de teste;

5. Geração de código de testes, para facilitar os testadores na execução de testes que satisfaçam a qualidade do serviço disponibilizado, tanto em nível funcional como não funcional;

6. Técnicas de teste utilizadas, onde um ambiente de testes possa suportar variadas técnicas para garantir uma cobertura de testes eficiente para aplicação ou serviço desenvolvido;

7. Modelos de casos de teste, usados para suportar e organizar o processo de desenvolvimento de casos de teste para a Computação em Nuvem;

8. Critérios de testes utilizados para composição de casos de teste, utilizados para facilitar a composição dos testes que possam cobrir todo o cenário onde deve ser testado com o número mínimo de casos de teste.

Conforme apresentado na Tabela 3.1, os trabalhos analisados foram classificados de acordo com o tipo de serviço oferecido pela nuvem. Assim, dois trabalhos foram classificados como categoria de IaaS por proporem uma infraestrutura de ambiente de testes. Dois trabalhos foram classificados como categoria PaaS, pela abordagem na construção de casos de teste sob uma perspectiva de desenvolvimento de aplicação. E finalizando a seleção de trabalhos relacionados, três trabalhos foram classificados como categoria SaaS por terem o foco no desenvolvimento de casos de teste para aplicativos, em ambientes de utilização dos usuários finais, em relação aos serviços oferecidos pela Computação em Nuvem.

A Tabela 3.1 ilustra um comparativo, ressaltando os principais itens apontados entre os trabalhos pesquisados. A motivação para este levantamento de pontos de avaliação, deu-se pela necessidade de comparar características (ou critérios) e objetivos, com forte relação com o projeto do *framework* proposto nesta dissertação.

Na questão de trabalho em nível de desenvolvimento da aplicação, três trabalhos foram classificados nesta categoria: “Uma abordagem para a Composição de Serviços e testes para Computação em Nuvem” [90], “Taxonomia e Racionalização de

Requisitos para Teste de *Software* baseado em Infraestrutura de Nuvem" [68] e "Automação de Testes em plataforma SaaS" [71].

Pode-se observar que, nos trabalhos de "Uma abordagem para a Composição de Serviços e testes para Computação em Nuvem" [90] e "Automação de Testes em plataforma SaaS " [71], códigos de testes são gerados para serem aplicados diretamente no código do *software* desenvolvido.

Observou-se também que apenas no trabalho "Teste como Serviço sobre Nuvens" [93], os autores propuseram um ambiente de testes que possa ser utilizado em diversas nuvens, ou seja, a proposta de um ambiente interoperável. A interoperabilidade pode ser classificada como um item de grande relevância, pois um *framework* que possa ser adaptado à diversas plataformas de Computação em Nuvem pode trazer benefícios, como, por exemplo, a redução de custos no processo de Testes de *Software*.

Todos os trabalhos pesquisados apresentaram modelos ou forma de organização para construção de casos de teste. Pode ser considerado que a construção de uma DSL pode trazer benefícios, como por exemplo, o auxílio na construção de modelos de testes aplicados na Computação em Nuvem.

Portanto, a proposta de um *framework* para geração de casos de teste para ambientes de Computação em Nuvem, que tenha possibilidades de dar suporte à vários tipos e técnicas de teste podem trazer diversos benefícios, tais como: Redução do tempo para o desenvolvimento de testes, interoperabilidade, qualidade e eficiência nos casos de teste e automação da geração dos casos de teste, reduzindo intervenções humanas.

3.8 Síntese

Neste capítulo, apresentou-se o Estado da Arte, com o objetivo de relatar ao leitor a base da pesquisa bibliográfica realizada neste trabalho de dissertação, descrevendo as principais abordagens para realizar a geração de casos de teste para ambientes de Computação em Nuvem. Através dessa pesquisa, referente à Teste de *Software* para o ambiente proposto, foram encontradas várias vertentes e soluções.

Sete trabalhos relacionados sobre a geração de casos de teste para Computação em Nuvem foram apresentados. Uma tabela composta pelos itens avaliativos foi preenchida para cada abordagem presente nos trabalhos relacionados.

Encerrando o capítulo, uma análise destes trabalhos relacionados através dos itens avaliativos organizados em uma tabela comparativa foi realizada com o objetivo de fundamentar, com a pesquisa, o trabalho proposto nesta dissertação. Também, alguns trabalhos fortemente relacionados foram destacados como base de comparação e contribuição sobre o desenvolvimento da solução proposta.

	IaaS		PaaS		SaaS		
Itens Avaliados	Teste como Serviço sobre Nuvens [93]	D-Cloud: Design de um ambiente de Teste de Software para Sistemas Distribuídos [6]	Uma abordagem para a Composição de Serviços e testes para Computação em Nuvem [90]	Taxonomia e Racionalização de Requisitos para Teste baseado em Nuvem [68]	Engenharia Multi-Tenant para Software como Serviço [74]	Automação de Testes em plataforma SaaS [71]	Avaliação de Desempenho e Escalabilidade em nuvens SaaS [17]
Trabalha em nível de Infraestrutura	Sim	Sim	Não	Sim	Não	Não	Sim
Trabalha em nível de Desenvolvimento	Não	Não	Sim	Sim	Não	Sim	Não
Trabalha em nível de Aplicação	Sim	Não	Sim	Não	Sim	Sim	Sim
Ambiente interoperável	Sim	Não	Não	-	-	Não	-
Gera código fonte de teste	Não	Não	Sim	Não	Não	Sim	Não
Técnicas de testes utilizadas	Escalabilidade	Tolerância a Falhas	Funcional, Continuo e Integração	Funcional, Performance e escalabilidade	Funcional	Regressão, Funcional e Cobertura de Código	Escalabilidade e Performance
Modelo para formar Casos de Teste	Ontologias para modelagem de regras de correspondência das tarefas de teste	Cenários de configuração de testes	Ontologias de domínio para organizar e especificar serviços e interfaces	Taxonomia para criação de requisitos de teste	Modelos de variabilidades e Modelos de cenários baseados no raciocínio semântico	Model View Controller (MVC) e sObjects	Modelos Formais e métricas
Critérios de testes utilizados para composição de Casos de Teste	Não	Não	Não	Sim	Não	Não	Não

Tabela 3.1: Análise dos Trabalhos relacionados

4 Uma Abordagem para Teste de *Software* em Computação em Nuvem

Neste capítulo, um *framework* para geração de casos de teste baseado em MDE para Computação em Nuvem é apresentado. Este *framework* tem o objetivo de automatizar o processo de geração dos Casos de Teste para aplicativos desenvolvidos neste ambiente.

Uma metodologia é proposta para descrever as etapas a serem realizadas para a composição dos casos de teste. Um algoritmo é apresentado, como auxílio na geração automática de modelos de teste, a partir de um modelo de negócio da aplicação (PIM) e, na escolha de critérios de testes.

Fechando o capítulo, os metamodelos desenvolvidos para serem usados no *framework* são apresentados para comporem a solução proposta.

4.1 *Framework* para Geração de Casos de Teste em Computação em Nuvem

Nesta seção, uma solução para automatizar a geração de casos de teste e código de teste utilizando a MDE para sistemas em ambientes de Computação em Nuvem é apresentada. A abordagem MDE proposta para gerar casos de teste para Computação em Nuvem fornece um *framework*, uma metodologia, metamodelos e a utilização de um algoritmo de *matching*. A abordagem proposta é uma extensão e adaptação do *framework* apresentado em [81].

O *framework* é proposto com a finalidade de suportar a interoperabilidade e pode ser utilizado em diversas plataformas de Computação em Nuvem. Esta interoperabilidade é conseguida através de modelos. Este *framework* fornece a geração de testes funcionais e testes de tempo de resposta e estresse, mas podendo ser estendido posteriormente para outros tipos de testes.

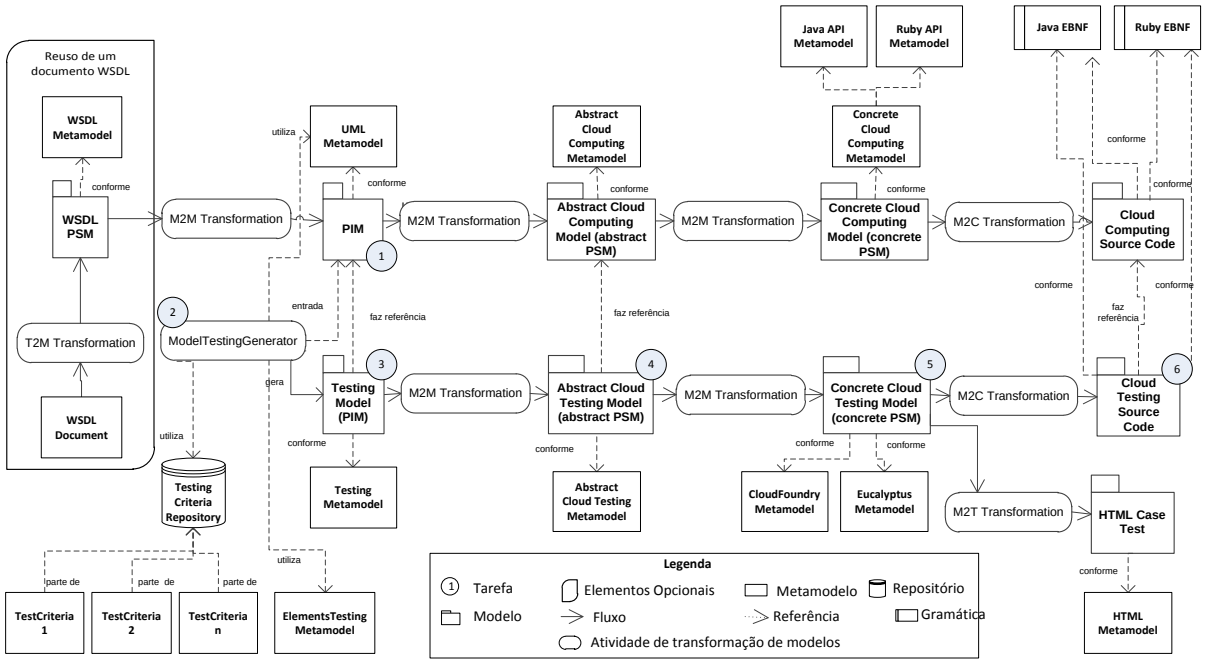

Figura 4.1: *Framework* para a geração de casos de teste para Computação em Nuvem

O *framework* suporta testes em nível de aplicação que podem ser utilizados em nuvens do tipo IaaS, PaaS e SaaS. Os testes realizados pelo *framework* são testes funcionais e testes não funcionais. O *framework* pode ser estendido para outros tipos de testes (e não somente em nível de aplicação), como também pode ser estendido para outras plataformas de nuvens. Esta extensão é obtida através da MDE com a construção de metamodelos para outras plataformas de nuvens.

O produto final gerado pelo *framework* é constituído de três tipos de arquivos. O primeiro são códigos de testes (conforme uma linguagem de programação) para os testes funcionais. O segundo é um arquivo em formato *shell*, contendo informações de conexão com a nuvem e casos de teste a serem aplicados para os testes não funcionais. E o terceiro, um relatório contendo as informações do caso de teste em formato *HyperText Markup Language (HTML)*. O *framework* realiza testes que podem ser aplicados nos principais modelos de serviços oferecidos pelas nuvens (IaaS, PaaS e SaaS).

Conforme ilustra a Figura 4.1, a geração de casos de teste é realizada a partir das informações de entradas obtidas pelo PIM da aplicação. Um algoritmo é proposto, conforme descrito na seção 4.3, para a criação automática dos modelos de teste independentes de plataforma, apoiando a escolha dos casos de teste que serão gerados e quais os modelos de critérios de testes utilizados a partir dos dados de entrada representados no PIM.

Após a construção dos modelos de testes independentes de plataforma, um PSM abstrato de teste para nuvem é concebido através da transformação de modelos tendo como entrada um PIM de teste. Após esta etapa, um PSM concreto, ou seja, um PSM para a nuvem específica (por exemplo, X, Y e Z) é gerado por uma transformação de modelos tendo como entrada o PSM abstrato. Em seguida, este modelo de teste pode ser transformado em código-fonte que está em conformidade com a plataforma da nuvem em específico através de uma transformação de modelo-à-texto, conforme apresentado na Figura 4.1.

O primeiro PSM pode ser considerado como uma plataforma abstrata *(abstract platform)* [2], ou também denominado de Linguagem de Implementação Independente de Ambiente *(Implementation Language Environment Independent)* [52]. O segundo PSM pode ser considerado como uma plataforma concreta *(concrete platform)* [2] ou também denominado de Linguagem de Implementação Dependente de Ambiente Específico *(Implementation Language Environment Specific)* [52]. A ideia de plataforma abstrata e concreta foi feita pela OMG [52] em 2001 e novamente refeito em 2004 [2].

No *framework* proposto, os testes funcionais e não funcionais (teste de estresse e tempo de resposta) são gerados para a nuvem de PaaS, *CloudFoundry* [15], conforme ao metamodelo concreto da nuvem *CloudFoundry* e conforme o metamodelo concreto da nuvem *Eucalyptus*, que constitui em uma nuvem de IaaS [86].

A ideia do *framework* proposto é suportar testes nos três principais serviços de nuvem: SaaS, PaaS e IaaS. Essa tarefa pode ser realizada através dos metamodelos construídos, presentes na seção 4.4, e através dos metamodelos que poderão ser futuramente acrescentados, pois o *framework* suporta este tipo de extensão.

Inicialmente, o *framework* suporta a geração de casos de teste em formato *jUnit*. O *jUnit* foi escolhido por ser um *framework* bastante conhecido e também porque a linguagem Java é amplamente utilizada na construção de aplicativos para plataformas de nuvens. Mas, nada impede que o *framework* possa ser estendido para outros *frameworks* de teste, como por exemplo o *Test::Unit* [69] para testes funcionais em *Ruby* e o *jMeter* [3] para testes de performance.

A mesma figura apresentada nesta seção, referente ao *framework* proposto, pode ser vista de forma ampliada no Anexo 8.9.

4.2 Metodologia para Geração de Casos de Teste em Computação em Nuvem

Uma metodologia é proposta, conforme a Figura 4.2, para auxiliar a abordagem de construção de casos de teste para sistemas em ambientes de Computação em Nuvem.

Conforme mencionado na Seção 1.2, detectou-se uma ausência de um padrão de desenvolvimento de teste para plataformas de Computação em Nuvem, embora se tenha observado a existência de algumas abordagens específicas para aplicacao em determinada plataforma.

A metodologia proposta parte da escolha do usuário entre três opções. A primeira opção é reutilizar um serviço, ou seja, testar uma aplicação considerando somente a utilização de serviços como um consumidor. A segunda opção é testar uma aplicação considerando que os serviços sejam construídos a partir do zero. A terceira opção é híbrida, pois permite utilizar um serviço existente (através de sua descrição em WSDL) ou um novo serviço em desenvolvimento.

Se a aplicação inclui somente a utilização dos serviços existentes, então ocorre um processo de extração das informações do serviço descritos no seu WSDL para criar um PSM e, depois, para gerar parte do PIM. Como os sistemas no ambiente de Computação em Nuvem geralmente são desenvolvidos sob a arquitetura SOA, a utilização do WSDL se faz necessária, pois descrevem os serviços que são consumidos ou produzidos para um sistema de *software*. Se a aplicação a ser testada usa serviços novos, então a metodologia parte direto da construção do PIM da aplicação.

A próxima etapa da metodologia é a validação do PIM, ou seja, verificar se o PIM está correto. Caso não esteja correto, uma checagem será realizada para realização das correções necessárias. Após a criação e validação do PIM, as atividades são divididas em duas linhas: geração de código do sistema e geração do código de teste do sistema. Em ambas as linhas são aplicadas definições de transformações descritas em ATL. Essas transformações construídas no *framework* são realizadas de forma semiautomática com a utilização das ferramentas MT4MDE e SAMT4MDE [37] [39].

Na parte da geração de casos e códigos de teste, um modelo de teste independente de plataforma é criado. Para a criação deste modelo, um algoritmo auxilia

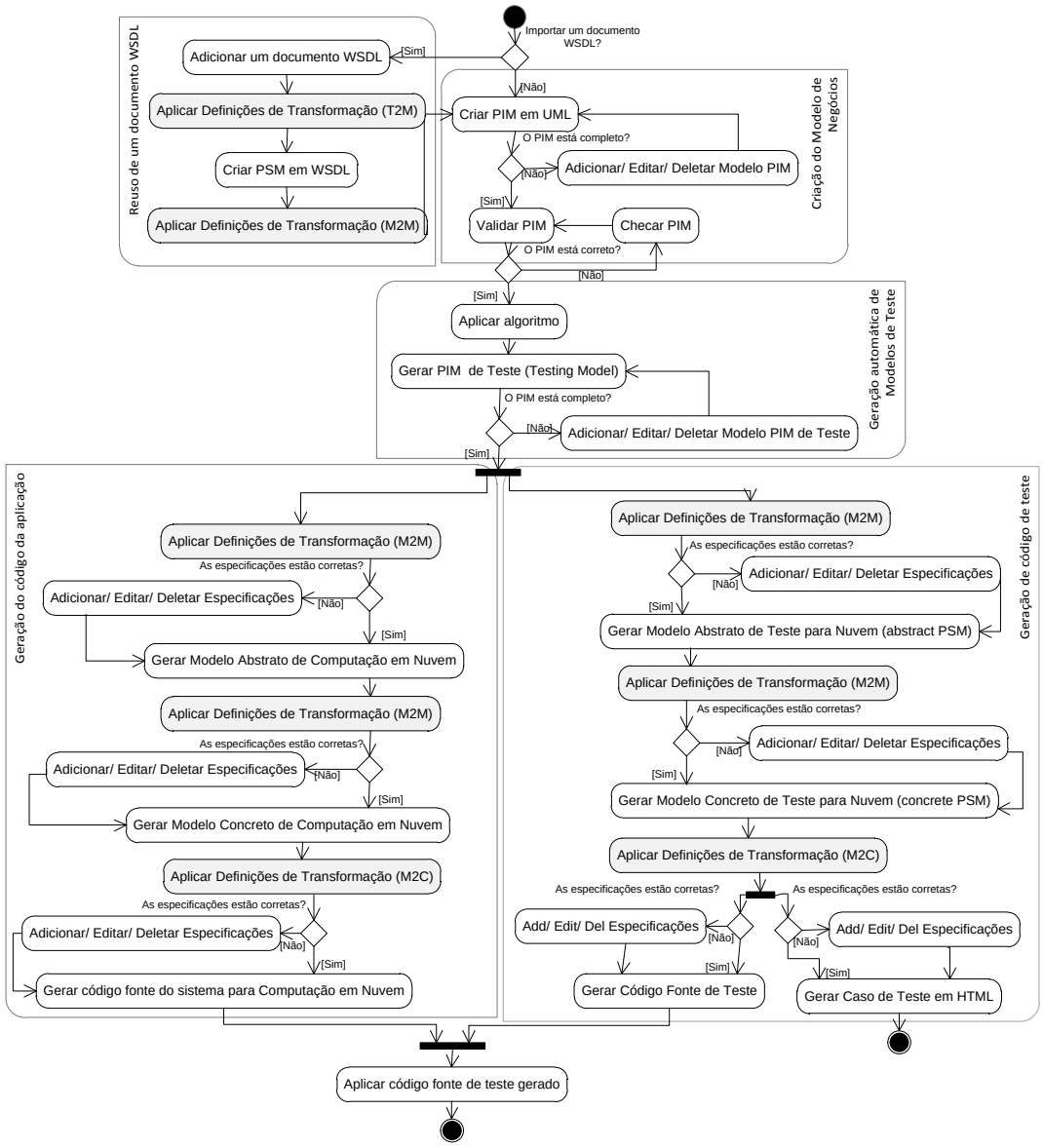

Figura 4.2: Metodologia para geração de casos de teste para Computação em Nuvem

na escolha dos critérios de testes para constituir os casos de teste da aplicação. Após a criação do modelo de teste, os PSMs são gerados através de transformações de modelos tendo como entrada o modelo de teste independente de plataforma. Finalmente, novas definições de transformação são aplicadas para a geração do código fonte do sistema e do código fonte de teste. Um relatório em formato *HTML* também é gerado contendo a descrição do Caso te Teste.

4.3 Algoritmo para Geração Automática de Modelos de Teste

No presente trabalho de dissertação, um algoritmo foi proposto para auxiliar na construção dos modelos de testes e para a geração de casos de teste para ambientes de Computação em Nuvem.

Como ilustrado na Figura 4.3, observa-se três principais etapas de funcionamento do algoritmo. A primeira etapa consiste em extrair informações do modelo de negócio, a segunda etapa consiste em instanciar os modelos de critérios de teste e a terceira etapa consiste em instanciar o modelo de casos de teste do *framework*. Cada etapa está detalhada nas seções a seguir.

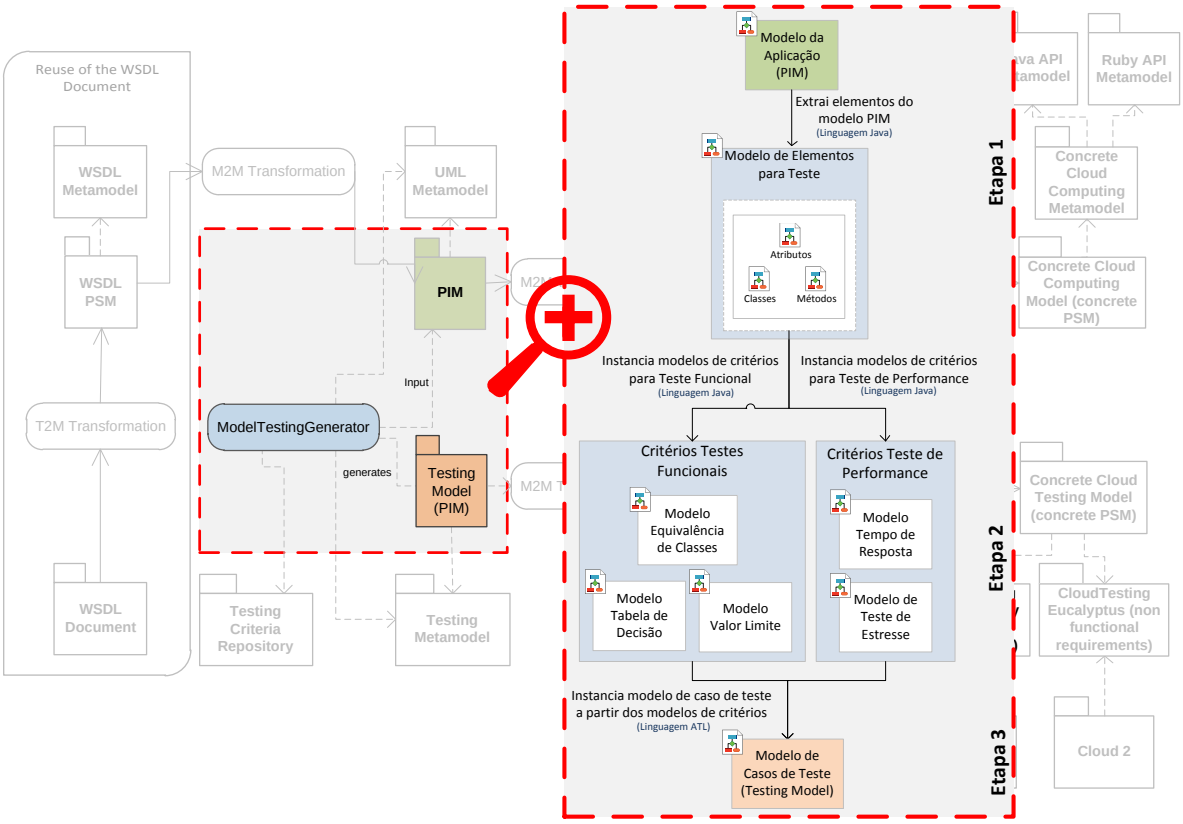

Figura 4.3: Modelos e etapas presentes no Algoritmo de Geração de casos de teste

4.3.1 Etapa 1 - Extração do Modelo de Negócios - PIM

Na primeira etapa, o algoritmo navega e coleta as informações pertinentes junto ao modelo de negócios (PIM) da aplicação e armazena-os em um modelo espe-

cífico. Os critérios de busca por informações, que ajudam na criação automática do modelo de teste, estão pré-definidos no algoritmo.

Através da execução deste algoritmo, os elementos do PIM da aplicação são identificados e separados. Estes elementos são armazenados em outro modelo, chamado de *TestingElements*, que foi escrito conforme o metamodelo *TestingElements-Metamodel*. A descrição detalhada deste metamodelo pode ser encontrada na seção 4.4.2. A Figura 4.4 apresenta um diagrama contendo as atividades da primeira etapa do algoritmo. As principais etapas estão listadas a seguir:

- **Navegar em modelo PIM** - O algoritmo realiza uma leitura no modelo PIM que foi previamente construído, contendo as informações referentes a aplicação desenvolvida (classes, serviços, métodos, etc);
- **Realizar leitura e separação de componentes do PIM** - Nesta atividade, são separados os principais componentes para a composição dos critérios de teste. O algoritmo busca primeiramente pelas classes, em seguida métodos, parâmetros e atributos;
- **Ler e separar Classes** - Consiste em identificar as classes que compõem o modelo de negócio. Essas classes poderão também, ser a representação de um serviço utilizado (através da leitura de um WSDL, na fase anterior do *framework*). Esta identificação servirá para a composição, por exemplo, dos Testes Funcionais;
- **Identificar Métodos** - Assim como as classes, os métodos identificados servirão para compor quais os métodos e/ou chamadas de serviço serão testados;
- **Identificar Parâmetros** - Uma vez reconhecido os métodos, o algoritmo verificará quais os parâmetros irão compor o método a ser testado;
- **Identificar Atributos** - É necessário o reconhecimento dos atributos de uma classe para a realização de testes, como por exemplo um teste de validação de um tipo de atributo (*string*, *integer*, etc);
- **Identificar Valor Mínimo e Valor Máximo** - Estes valores deverão ser catalogados, tanto dos parâmetros como dos atributos, para a aplicação de critérios de testes funcionais, como por exemplo, Análise do Valor Limite e Equivalência de Classes;

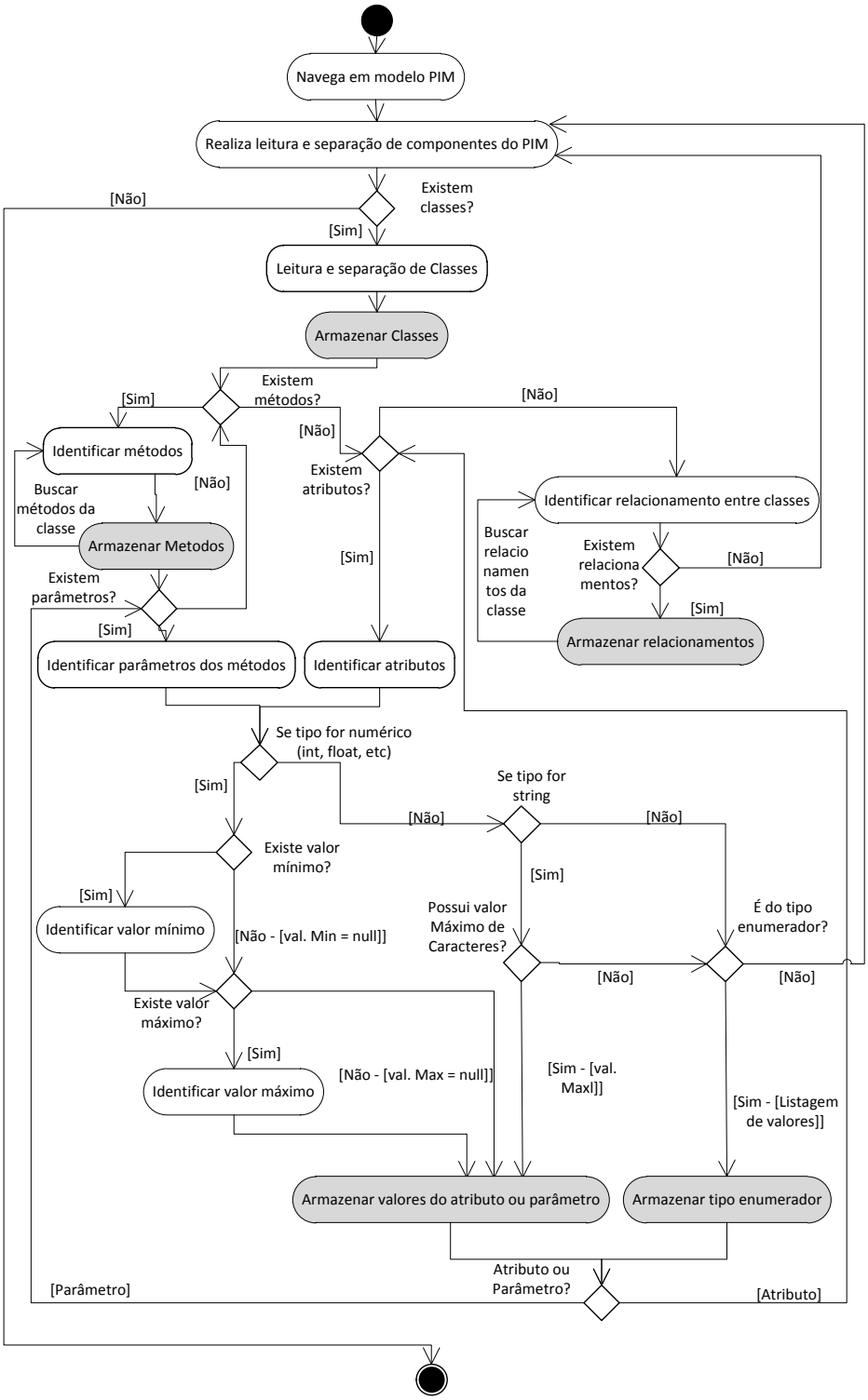

Figura 4.4: Metodologia da Etapa 1 do Algoritmo de Geração de casos de teste

- **Identificar relacionamento entre Classes** - Esta atividade se faz necessária para a verificação de dependências entre Classes e/ou Serviços.

4.3.2 Etapa 2 - Instanciação dos modelos de Critérios de Teste

Esta etapa consiste na leitura do modelo de elementos de teste *TestingElements Metamodel* e a instanciação dos modelos de critérios de testes adequados para a aplicação.

Os critérios de testes são aplicados para realizar uma cobertura nos principais pontos a serem testados em uma aplicação, contribuindo na redução do número de testes [47]. No *framework* proposto eles auxiliam na construção automática dos modelos de casos de teste.

Para a construção deste *framework*, cinco metamodelos de critérios de testes foram utilizados: Equivalência de Classes, Valor Limite e Tabela de Decisão pertencentes aos critérios que formam o Teste Funcional. Também foram construídos os metamodelos de Tempo de Resposta e Teste de Estresse, pertencentes ao grupo de critérios dos Testes de Performance.

Na Figura 4.5, um diagrama de atividades da segunda etapa de geração automática de casos de teste é apresentado. As principais atividades são:

- **Realiza busca no repositório por Metamodelos de Critérios** - É necessário, nesta etapa, que metamodelos referentes a critérios de teste estejam armazenados em repositório para a construção de modelos a serem utilizados na próxima etapa do algoritmo;

- **Realiza a contagem de Metamodelos de Critérios** - O algoritmo realiza a contagem para poder percorrer todos os critérios existentes no repositório;

- **Escolhe Metamodelo de Critérios** - Depois de realizada a soma da quantidade de metamodelos de critérios existentes no repositório, o algoritmo escolhe qual o metamodelo será instanciado;

- **Separa elementos de Teste** - O algoritmo percorrerá o modelo de elementos de teste e separa o elemento, ou conjunto de elementos, para aplicar as regras de critérios de teste. Existem regras para escolha do elemento correto. Por exemplo,

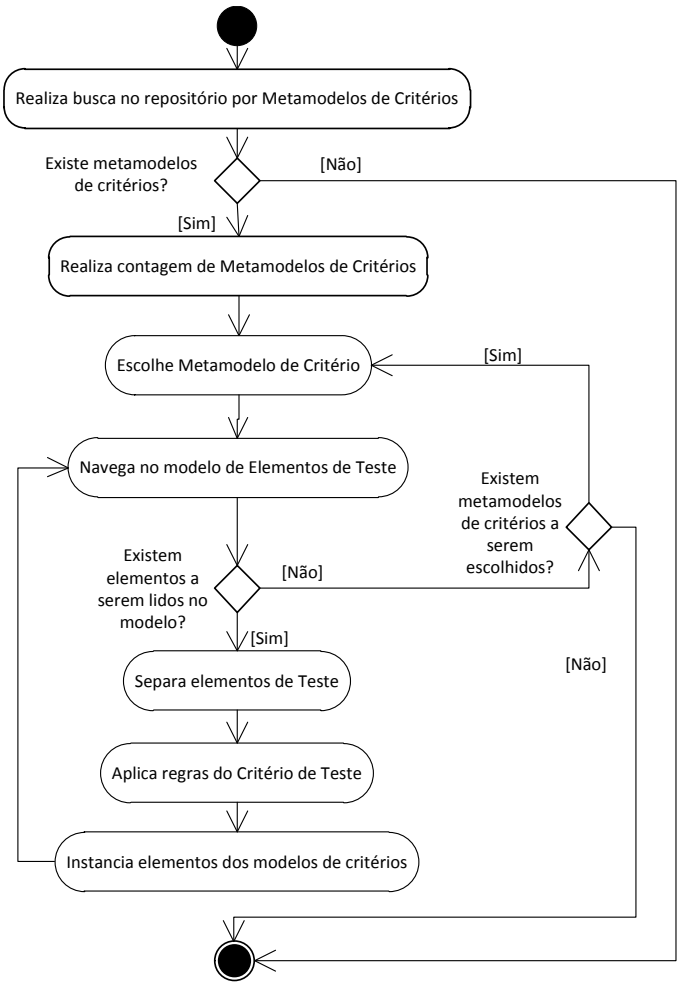

Figura 4.5: Metodologia da Etapa 2 do Algoritmo de Geração de casos de teste

um parâmetro de um método, não será testado sozinho. Para realização de teste com este parâmetro, será necessário conhecer o elemento que representa a classe a ser testada e o elemento que representa o nome do método;

- **Aplica regras de Critérios de Teste** - Escolhido o elemento ou o conjunto de elementos, são executadas regras, escritas de acordo com o tipo do critério de teste a ser aplicado para a instanciação do modelo de critérios de teste.

Observa-se que no presente trabalho de dissertação, o algoritmo percorrerá por todos os critérios existentes no repositório. Futuramente, poderia ser construído uma abordagem para escolha adequada dos critérios, de acordo com modelo de negócio da aplicação (modelo PIM), para não haver a necessidade de percorrer por todos os metamodelos de critérios contidos no repositório.

4.3.3 Etapa 3 - Instanciação do Modelo de casos de teste do *Framework*

Uma vez instanciados os modelos de critérios de teste, a terceira etapa do algoritmo consiste na instanciação do modelo de casos de teste, que foi escrito conforme o metamodelo construído *Testing Metamodel*. A descrição detalhada do *Testing Metamodel* se encontra na seção 4.4.3. A Figura 4.6 ilustra as atividades a serem realizadas

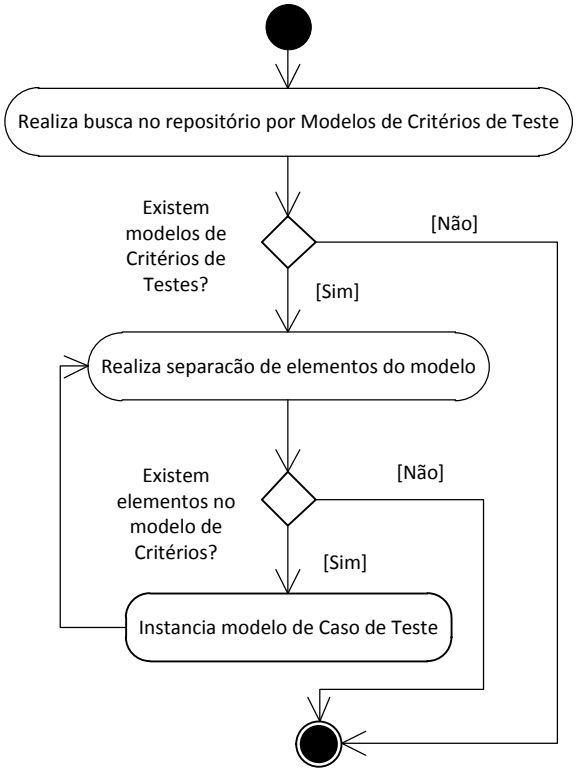

Figura 4.6: Metodologia da Etapa 3 do Algoritmo de Geração de casos de teste

nesta fase. Sendo as principais:

- **Separação dos elementos do modelo** - O algoritmo realiza a separação do elemento, ou conjunto de elementos para a instanciação do modelo de caso de teste;
- **Instanciação do modelo de Caso de Teste** - Após a separação dos elementos do modelo de critérios de teste, os mesmos irão compor os casos de teste a serem aplicados.

Um modelo de caso de teste pode conter um ou vários casos de teste, ou seja, podem ser instanciados vários modelos de casos de teste. Na abordagem proposta no

framework, procurou agrupar os casos de teste de requisitos funcionais em um modelo e os casos de teste de requisitos não funcionais em outro modelo de teste.

4.4 Metamodelos Propostos

Em MDE, metamodelos são essenciais para a criação de modelos e para transformações entre modelos [55]. Os metamodelos propostos nesta dissertação visam garantir a melhoria na qualidade dos testes de *software* para Computação em Nuvem. Transformações de modelos podem gerar um modelo de teste a partir de um outro modelo.

4.4.1 Metamodelos de Critérios de Teste

Nesta seção, os metamodelos de critérios de teste são apresentados. Estes metamodelos contribuem na redução do número de casos de teste para uma aplicação [47] e auxiliam na construção automática de casos de teste no *framework* proposto.

Para a construção deste *framework*, foram utilizados cinco metamodelos de critérios: Equivalência de Classes, Valor Limite e Tabela de Decisão pertencentes aos critérios que formam o Teste Funcional. Também foram construídos os metamodelos de Tempo de Resposta e Teste de Estresse, pertencentes ao grupo de critérios dos Testes de Performance.

Existem diversos tipos de Testes de *Software* (Funcional, Regressão, Performance, etc) e junto com eles os critérios que os compõe. Posteriormente outros metamodelos de critérios de teste poderão ser construídos e integrados ao *framework* proposto.

Metamodelos de Critérios de Teste: *EquivalenceClass Metamodel* (Equivalência de Classes)

Este critério trabalha com dedução lógica [47]. A redução de casos de teste é obtida através da divisão do domínio de entrada (por exemplo, uma regra de negócio) em subdomínios de elementos equivalentes.

Um metamodelo foi construído para representar e utilizar este critério na construção de casos de teste, conforme ilustrado na Figura 4.7. Os elementos principais deste metamodelo são:

- *NameElement* (Nome do Elemento): É a generalização dos elementos de testes. Os principais elementos de testes possui um nome que o identifica;
- *CriteriaEquivalenceClass* (Critério de Equivalência de Classe): Elemento principal do metamodelo. Representa qual o tipo de equivalência a ser testada (Discreta ou Contínua);
- *ObjectTesting* (Objeto Testado): Elemento que representa qual objeto (classe ou serviço) a ser testado no sistema;
- *SubObjectTesting* (Sub Objeto Testado): Este elemento possui informações do que será testado do *ObjetcTesting*, representado pelos itens no *enumeration SubObject*;
- *Value* (Valor): Representa os valores utilizados na aplicação dos testes.

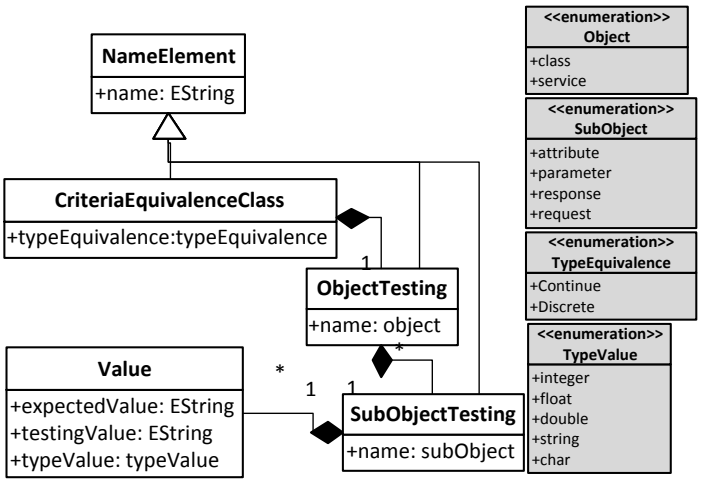

Figura 4.7: Metamodelo de Critério de Teste: Equivalência de Classes

Uma classe de equivalência pode ser considerada **Contínua**, quando possui valores contínuos, ou seja, a classe é composta por números racionais Q limitada pela quantidade de casas decimais [47]. E uma classe de equivalência pode ser considerada **Discreta**, quando percebe-se que esta classe possui um conjunto de valores discretos e únicos, ou seja, esta classe é composta por números inteiros que fazem parte do universo desejado [47].

Metamodelos de Critérios de Teste: *BoundaryValue Metamodel* (Valor Limite)

Este critério de teste tem por objetivo reduzir o número de casos de teste, testando os valores de fronteiras das classes [47].

Na Figura 4.8, é apresentado o metamodelo do critério de Valor Limite. A classe *CriteriaBoundaryValue* representa o critério de teste e, assim como o critério de Equivalência de Classe, representa também o tipo (Discreto ou Contínuo). E a classe *Value* representa os valores utilizados na aplicação dos testes.

Podemos observar que o metamodelo de Classe de Equivalência e o metamodelo de Valor Limite são idênticos. Porém, a diferenciação entre eles se dá na forma de aplicação, ou seja, as regras (ou combinações) de aplicação são diferentes. Para um melhor entendimento, estas combinações podem ser vistas no capítulo seguinte, na Seção 5.1.2.

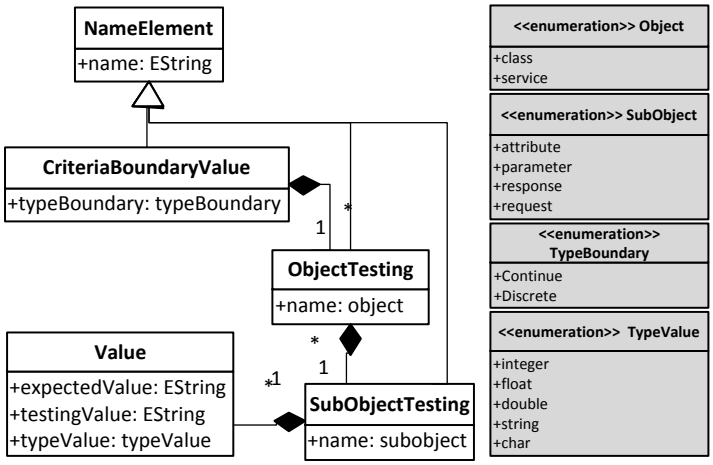

Figura 4.8: Metamodelo de Critério de Teste: Análise de Valor Limite

Metamodelos de Critérios de Teste: *DecisionTable Metamodel*

Este critério monta um conjunto de casos de teste que representam regras e conjunto de ações envolvidas a partir de uma tabela construída de acordo com as especificações do objeto a ser testado. É utilizado quando existem várias regras e uma possa existir interferência entre elas [47]. Conforme representado na Figura 4.9 os elementos principais deste metamodelo são:

- *CritériaDecisionTable* (Critério Tabela de Decisão): Elemento que representa o critério de teste e ela contém condições e ações;

- *Condition* (Condição): Este elemento traz uma condição especificada pelo Engenheiro de Teste que deve existir no *software* a ser testado;
- *Action* (Ação): Elemento que representa o que deve ser esperado da condição;
- *ValueCondition* e *ValueAction* (Valor de Condição e Valor de Ação): As condições e ações possuem valores, estes valores são representados por um valor esperado e o seu tipo (*string*, *integer*, *double*, etc).

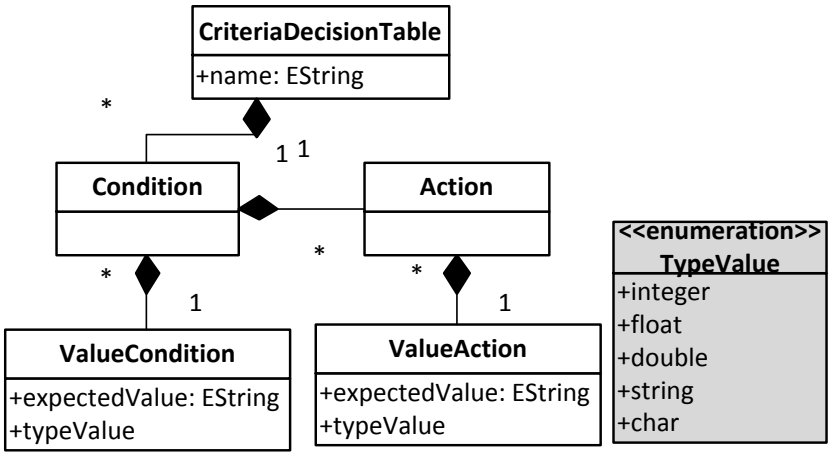

Figura 4.9: Metamodelo de Critério de Teste: Tabela de Decisão

Metamodelos de Critérios de Teste: *ResponseTime Metamodel* (Tempo de Resposta)

Este critério tem por objetivo calcular o tempo de resposta de um serviço ou método chamado. Este critério pertence ao Teste de Performance. Na Figura 4.10, é apresentado o critério de Tempo de Resposta, onde são encontrados os seguintes elementos:

- *CriteriaResponseTime* (Critério de Tempo de Resposta): Elemento principal do metamodelo que possui informações sobre a aplicação a ser testada;
- *Method* (Método): Elemento que representa o método ou uma requisição de serviço a ser testado;
- *Parameter* (Parâmetro): Representa os parâmetros do que está sendo testado, com seu valor e tipo (*string*, *float*, *integer*, etc);

- *TimeValue* (Valor de Tempo): Elemento que possui informações sobre o tempo de base para o teste (tempo testado e tempo esperado).

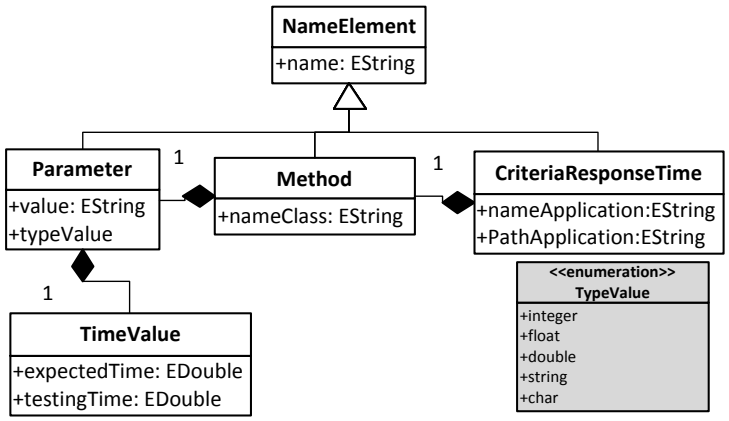

Figura 4.10: Metamodelo de Critério de Teste: Tempo de Resposta

Metamodelos de Critérios de Teste: *Stress Metamodel*(Teste de Estresse)

O critério de estresse, também pertencente aos critérios que compõem o Teste de Performace, é particularmente relevante para sistemas distribuídos baseado em uma rede de processadores [78]. No *framework* é utilizado para verificar quantas solicitações de um aplicativo podem ser instanciadas num ambiente de nuvem sem que haja perda de tempo no processo. Na Figura 4.11, é representado o metamodelo de Teste de Estresse com os elementos:

- *CriteriaStress* (Critério de estresse): Elemento principal do metamodelo, responsável pelas informações da aplicação a ser testada, onde está localizado a aplicação e o número de instâncias a serem alocadas;

- *TimeValue* (Valor de Tempo): Elemento que possui os valores de tempo que servirão de parâmetros para os testes.

4.4.2 Metamodelo de Perfil de Teste: *TestingElements*

No *framework* proposto, um metamodelo que contém a descrição, classificação e separação dos elementos de testes é utilizado para facilitar a instanciação dos metamodelos de critérios de testes. Esta separação se faz necessária para agilizar o

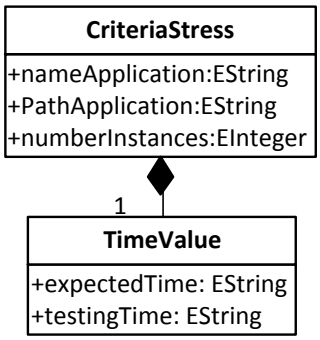

Figura 4.11: Metamodelo de Critério de Teste: Tempo de Estresse

processo de geração automática de casos de teste a partir do modelo de negócio da aplicação.

O metamodelo foi elaborado para separar elementos para testar métodos e atributos de classes e também separar elementos que representam serviços *Web* utilizados ou chamados em uma aplicação. A Figura 4.12, ilustra o metamodelo de elementos de teste que é composto pelos seguintes elementos:

- *NameElement* (Nome do Elemento): É a generalização dos elementos de testes. Os principais elementos de testes tem um nome que o identifica;
- *TestingElements* (Elemento de Teste): É o elemento principal do metamodelo de Elementos de Teste, este elemento possui o nome da aplicação e o caminho da aplicação utilizados para composição dos *scripts* de teste;
- *TimeApplication* (Tempo da Aplicação): Elemento que contém informações para o teste de tempo de resposta de uma aplicação;
- *ClassServiceAssociation* (Associação Classe e Serviço): Elemento responsável por coletar relacionamento entre as classes do diagrama de negócio da aplicação;
- *Class* (Classe): Neste elemento, os grupos de objetos e comportamentos do sistema a serem testados são descritos. Utilizado para a composição dos testes funcionais;
- *Method* (Método): Elemento que representa o método a ser testado ou uma sequência de declarações para executar uma ação;

- *TimeMethod* (Tempo do Método): Este elemento possui informações relacionados para realização de testes de tempo de resposta de um método;

- *Parameter* (Parâmetro): Elemento representa um conjunto de valores ou de características que determinam as funções do método a ser testado no sistema;

- *ValueParameter* (Valor do Parâmetro): Elemento que compõe o elemento *Parameter*. Este elemento possui o tipo do parâmetro a ser testado: se *string*, *integer*, *float*, etc. Este elemento contém ainda os elementos *MinValueParameter*, *MaxValueParameter* e *ExpectedValueParameter*, que, dependendo do tipo do parâmetro, coletará informações de valores mínimo, máximo e esperado;

- *Attribute* (Atributo): Este elemento possui informações relacionados para realização de testes de tempo de resposta de um método;

- *ValueAttribute* (Valor do Atributo): Este elemento possui o tipo do atributo a ser testado: se *string*, *integer*, *float*, etc. Este elemento contém ainda os elementos *MinValueAttribute*, *MaxValueAttribute* e *ExpectedValueAttribute*, que, dependendo do tipo do parâmetro, coletará informações de valores mínimo, máximo e esperado;

- *Service* (Serviço): Elemento que possui informações relativas a serviço utilizado ou chamado em uma aplicação;

- *TimeService* (Tempo do Serviço): Neste elemento, informações de teste de tempo de resposta do serviço são coletados;

- *ServiceResponseOperation* (Serviço de Operação de Resposta) e *ServiceRequestOperation* (Serviço de Operação de Requisição): Elementos responsáveis por separar os serviços de resposta e requisição de um serviço.

Pode-se observar, que existem elementos neste metamodelo nos quais representam informações relacionadas aos valores mínimo, máximo e esperado. Esta separação de valores é importante para a composição dos critérios de testes, como por exemplo, o critério de teste de Valor Limite pertencente a um teste Funcional.

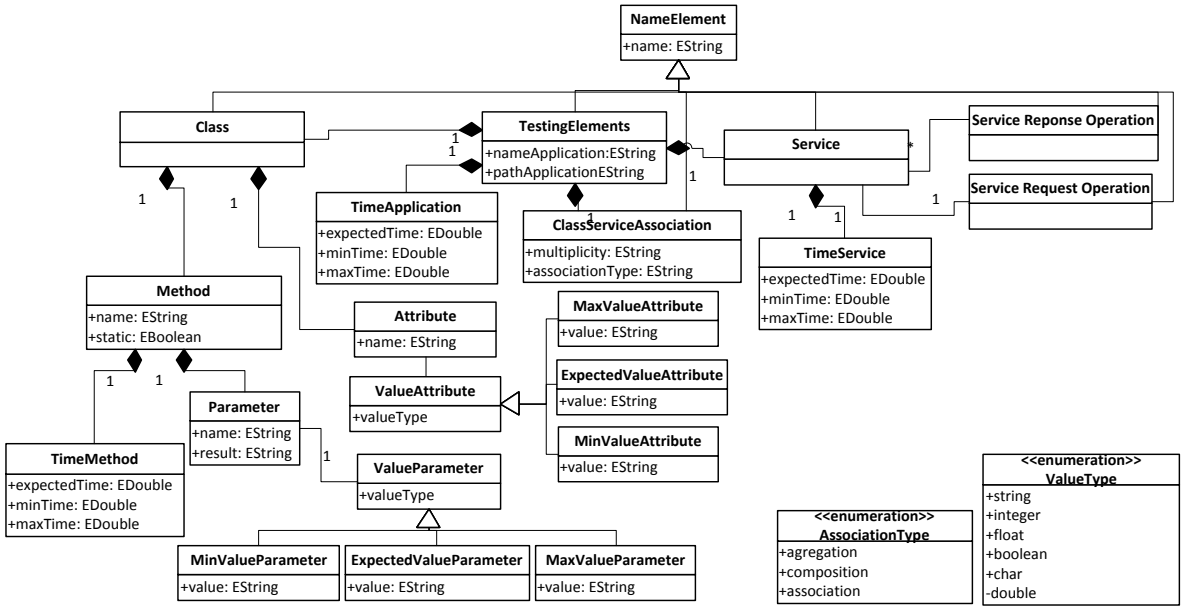

Figura 4.12: Metamodelo de Elementos de Teste

4.4.3 Metamodelo de Teste Independente de Plataforma: *TestingMetamodel*

O metamodelo de *TestingMetamodel* foi adaptado do trabalho de [81] e baseado nos testes funcionais, testes de integração e alguns critérios de teste de performance. Critérios de testes foram adicionados no metamodelo para escolha dos testes a serem aplicados no *software*.

Os testes funcionais, conhecidos como testes de caixa preta, tem o objetivo de encontrar discrepâncias entre o que um programa faz e o que deveria fazer [61]. Dados de entrada são utilizados, uma função é executada e um resultado de saída é esperado.

Os testes de integração visam garantir que um ou mais componentes funcionam realmente em conjunto, se são chamados corretamente e se transferem dados corretos no tempo correto por meio de suas interfaces [62]. A abordagem dos testes de integração é baseada na integração *Top Down*.

Os testes de performance são baseados nos critérios de teste de segurança, que são relacionados com disponibilidade, integridade e confidencialidade. E, os testes de tempo, são relacionados ao tempo que o sistema leva para atender uma funcionalidade [62].

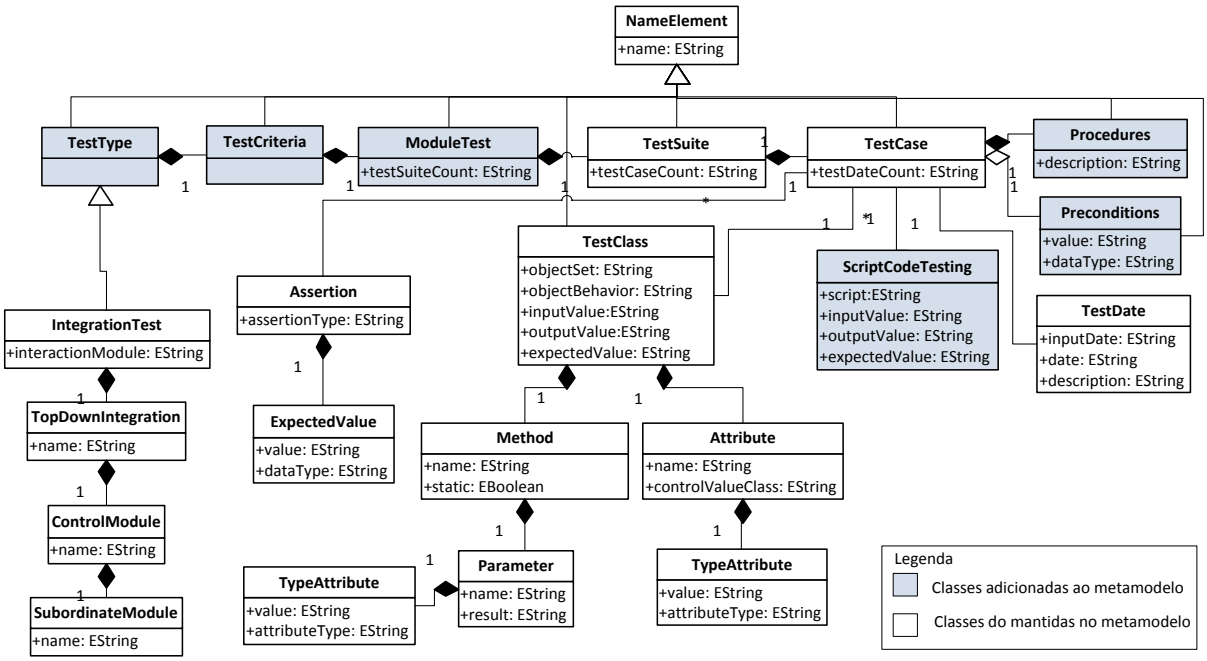

Figura 4.13: Metamodelo de Teste (Testing Metamodel)

Na Figura 4.13, *Testing Metamodel* é apresentado. Conforme a legenda, classes foram mantidas do metamodelo original (as classes em cor branca) e foram adicionadas novas classes (em azul) para atender a geração de testes para ambientes de Computação em Nuvem. Os elementos principais deste metamodelo são:

- *NameElement* (Nome do Elemento): Este elemento é generalização dos componentes de testes. Este elemento tem um nome que o identifica;
- *TestType* (Tipo de Teste): Este elemento é o principal do metamodelo, ele contém os tipos dos testes a serem aplicados;
- *IntegrationTest* (Teste de Integração): Elemento que possui a descrição da execução dos testes de integração, estabelecendo os módulos a serem testados no sistema;
- *TestCriteria* (Critério de Teste): Elemento que possui o critério de teste que será executado;
- *ModuleTest* (Módulo de Teste): Elemento que apresenta o módulo ou fases onde serão executados os testes;
- *TestCase* (Caso de Teste): Este elemento apresenta os casos de teste para a execução dos testes de *software*. Este elemento tem a função de conter um conjunto de

condições e variáveis que são determinadas pelo Engenheiro de Teste a fim de determinar se a aplicação apresenta alguma falha. Alguns elementos contribuem para a formação dos casos de teste, como:

- *TestClass* (Teste de Classe): Neste elemento, os grupos de objetos e comportamentos do sistema a ser testados são descritos;
- *ScriptCodeTesting* (Código de Script de Teste): Elemento adicionado ao metamodelo com o objetivo de receber informações de testes a serem executados na forma de scripts. Este elemento foi construído desta forma (na forma de scripts) com a finalidade de suportar testes funcionais e não funcionais;
- *Procedures* (Procedimentos): Apresenta descrições dos procedimentos a serem executados nos casos de teste. Este elemento tem o objetivo principal de fornecer dados para o relatório de testes do *framework* proposto;
- *Preconditions* (Pré-condições): Elemento que contém informações relativas à condições que sejam necessárias para a execução dos casos de teste. Elemento utilizado também para fornecer dados ao relatório de testes.

4.4.4 Metamodelo de Teste de Nuvem Abstrata: *Abstract CloudTesting*

O metamodelo de Teste de Nuvem Abstrata é uma extensão do metamodelo proposto *Testing Metamodel*. Além dos elementos específicos para os casos de teste, elementos relacionadas à nuvem em que serão aplicados os testes são representadas (elementos na cor cinza).

Os elementos que se referem à nuvem podem ser utilizados para representar os elementos básicos de identificação do serviço de uma nuvem para a construção dos casos de teste. Este conjunto de elementos servem principalmente para composição dos testes de requisitos não funcionais. Pode-se observar na parte superior da Figura 4.14 que existem os seguintes elementos específicos para o serviço da nuvem:

- *AbstractCloudTestModel* (Modelo Abstrato de Teste de Nuvem): Elemento que recebe o nome da nuvem no qual os casos de teste serão gerados;

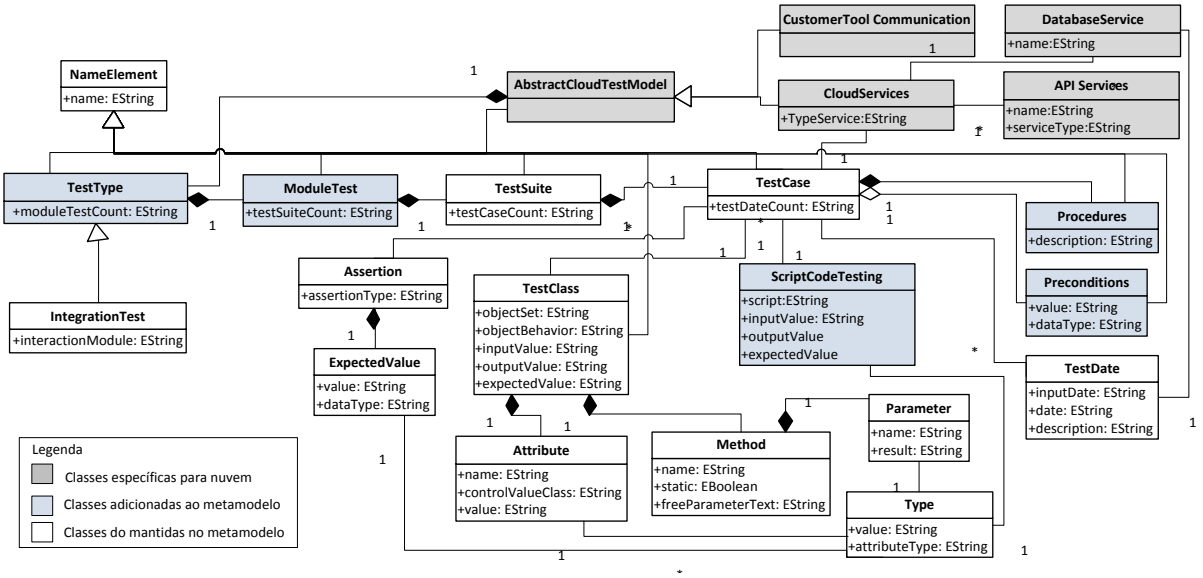

Figura 4.14: Metamodelo de Teste de Nuvem Abstrata

- *CloudServices* (Serviço da Nuvem): Elemento que representa o tipo de serviço que a nuvem oferece: SaaS, PaaS ou IaaS;

- *API Services* (Serviço de API): Elemento que representa o tipo de API que envolve o caso de teste no caso de uma nuvem do tipo PaaS. Estas APIs são disponibilizadas pelas nuvens para que o desenvolvedor possa utilizá-la em sua aplicação;

- *DataBaseService* (Serviço de Banco de Dados): Elemento que representa o serviço de banco de dados utilizado na aplicação a ser testada;

- *CustomerToolCommunication* (Ferramenta de Comunicação do Cliente): Elemento responsável por representar a ferramenta de comunicação do cliente para acesso a nuvem escolhida.

4.4.5 Metamodelo de Teste de Nuvem Concreta: *CloudFoundry*

O metamodelo proposto *CloudFoundry* foi baseado no metamodelo *xUnit* do trabalho de [29] [81], incluindo as características da plataforma *CloudFoundry*.

A nuvem *CloudFoundry* é do tipo PaaS, portanto, os testes a serem gerados para esta nuvem são em nível de funcionalidade da aplicação. Dentre os elementos que representam funções específicas da nuvem, existe o elemento *CustomerToolCommunication* que contém os comandos da ferramenta *VMC* [88], específica desta nuvem. Esta

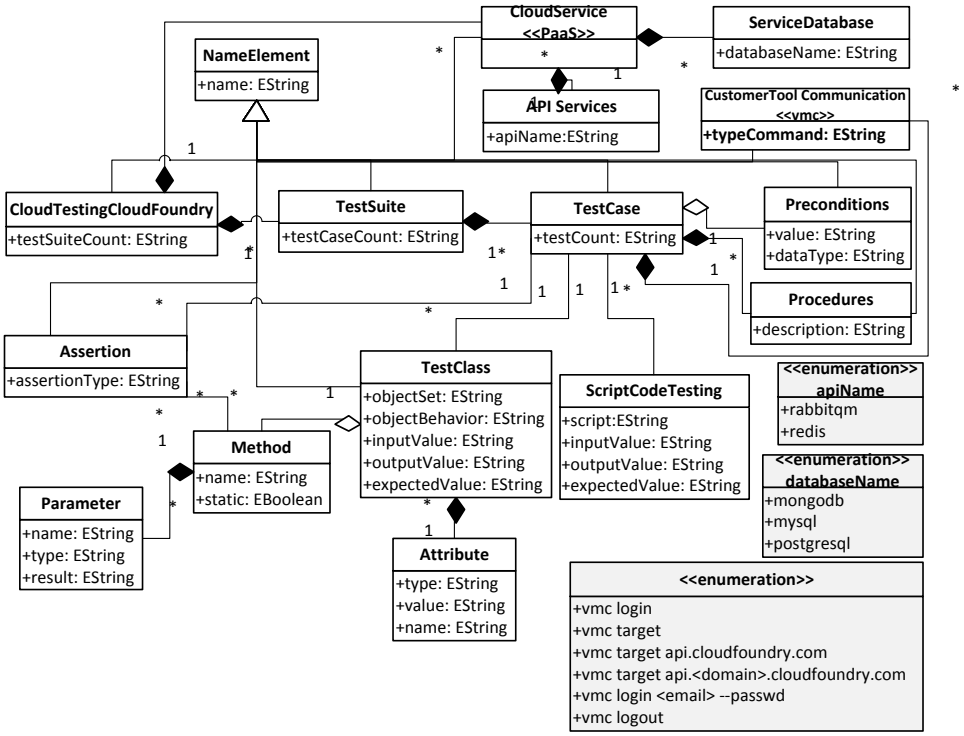

Figura 4.15: Metamodelo de Teste de Nuvem Concreta: CloudFoundry

ferramenta de comunicação é necessária para a execução de serviços e comunicação do cliente com esta nuvem.

O elemento principal deste metamodelo é a classe *CloudTestingCloudFoundry* que contém o elemento *TestSuite*. A classe *TestSuíte* contém o elemento *TestCase*. O elemento *TestCase* possui o *Assertion* e o *TestClass*. O *Assertion* contém *ExpectedValue*. Este elemento armazena os valores esperados na condição do caso de teste. O *TestClass* representa a classe que será testada com seus atributos e métodos. Um método por sua vez possui o objeto *Parameter*. Através das classes propostas neste metamodelo, pode-se modelar testes funcionais e não funcionais . A Figura 4.15 apresenta o metamodelo para teste *CloudFoundry*, onde destacam-se os *enumerations* que contém informações específicas da plataforma *CloudFoundry*:

- *apiName* (Nome da API): Possui os nomes das APIs disponibilizadas pela nuvem *CloudFoundry* para que o desenvolvedor utilize em sua aplicação;
- *databaseName* (Nome do Banco de Dados): Possui o nome dos Bancos de Dados disponibilizados pela nuvem;

- *typeCommand* (Tipo de Comando): São os tipos de instruções que o cliente ou administrador da nuvem precisa utilizar para submeter uma aplicação na nuvem, executar esta aplicação, conectar a nuvem, etc.

4.4.6 Metamodelo de Teste de Nuvem Concreta: *Eucalyptus*

O metamodelo proposto para a plataforma *Eucalyptus* fornece o suporte para a criação de modelos em formato de *scripts shell*. O metamodelo foi desenvolvido para dar suporte a testes funcionais e não funcionais. A utilização de *scripts* facilita a aplicação dos testes, porque eles podem ser aplicados diretos na nuvem, ou seja, em nível de infraestrutura da nuvem. Estes *scripts* de testes podem também serem construídos com a finalidade de serem executados em cada instância de máquina, na nuvem *Eucalyptus* que possua o sistema a ser testado.

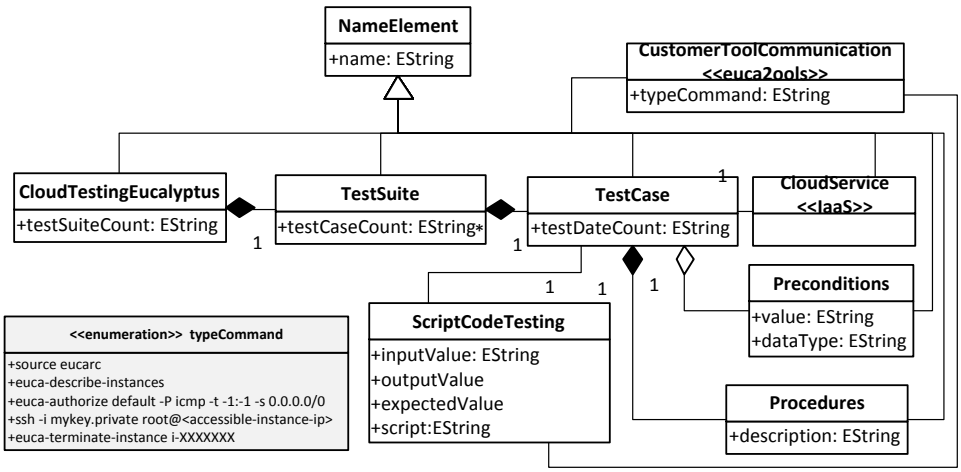

Figura 4.16: Metamodelo de Teste de Nuvem Concreta: Eucalyptus

Conforme ilustra a Figura 4.16, o metamodelo é apresentado contendo como elemento principal o *CloudTestingEucalyptus*. Este elemento contém o *TesteSuite*. Por sua vez, o *TesteSuite* contém o elemento *TestCase*, onde os casos de teste e os *scripts* que compõem estes testes são modelados. O *TestCase* possui o elemento *ScriptCodeTesting* responsável pelos scripts a serem executados. Neste metamodelo, dois elementos são específicos da nuvem *Eucalyptus*:

- *CloudService* (Serviço da Nuvem): Elemento que contém informações, caso um serviço a seja testado (geralmente em testes não funcionais);

- *CustomerToolCommunication* (Ferramenta de Comunicação do Cliente): Elemento que contém os comandos da ferramenta *Euca2ools* necessários para a comunicação do cliente com as instâncias ou serviços da nuvem. Os comandos do *Euca2ools* [86] [24] utilizados no metamodelo, compõe os itens do *enumeration typeCommand* utilizado no elemento *ScriptCodeTesting*.

4.4.7 Metamodelo de Computação em Nuvem Abstrata: *Abstract Cloud Computing*

Na Figura 4.17, apresenta-se a proposta de um metamodelo de Computação em Nuvem abstrata. A ideia é representar os principais serviços que uma nuvem pode disponibilizar. Os principais elementos representados neste metamodelo são: PaaS, SaaS e IaaS, onde:

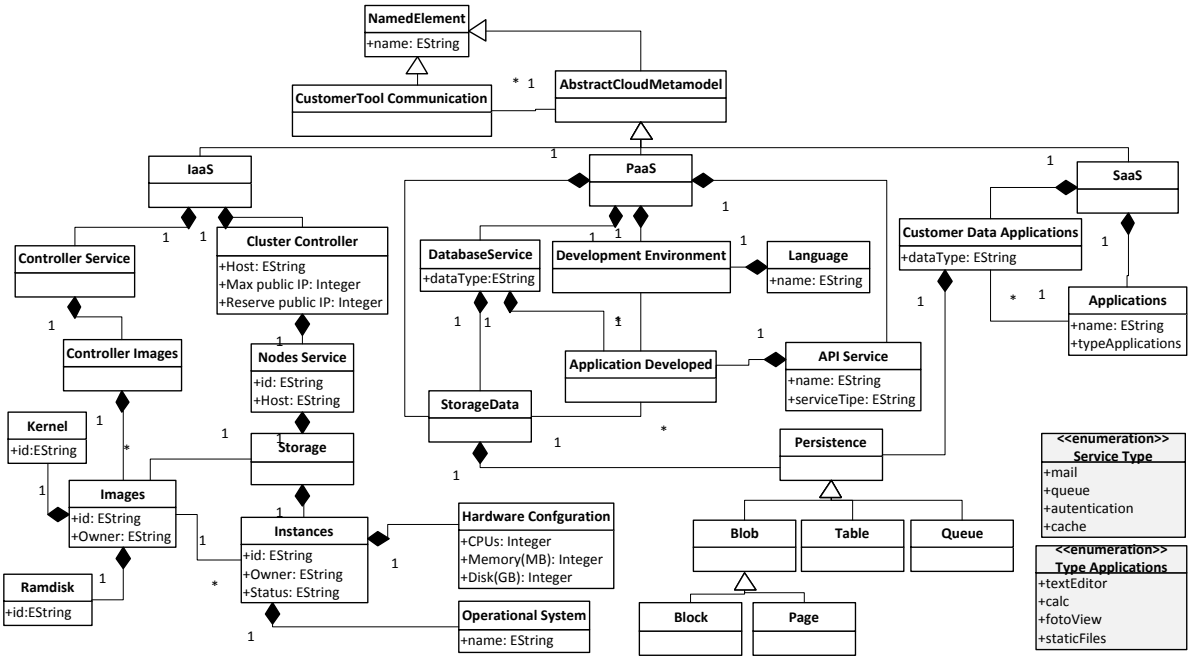

Figura 4.17: Metamodelo de Computação em Nuvem Abstrata

- IaaS: É o elemento que representa as nuvens de IaaS. Este elemento contém *ControllerService* e *ClusterController*;

  - *ControllerService* (Serviço Controlador): Elemento responsável por gerenciar os serviços disponibilizados, prover uma interface com os usuários, e armazenamento das imagens que serão instanciadas pelos usuários;

  - *ControllerImages* (Controlador de Imagens): Elemento que contém as imagens representadas pela classe *Image*;
  - *Storage* (Armazenamento): Elemento que contém as instâncias dos clientes serão armazenadas no elemento *Storage* que está contido em *NodesService* que pertence a *ClusterController*;
  - *ClusterController* (Controlador de *Cluster*): é responsável pelo gerenciamento das instâncias dos clientes;

- PaaS: É o elemento que representa as nuvens de PaaS. Este elemento contém por sua vez os seguintes elementos:

  - *DevelopmentEnvironment* (Ambiente de Desenvolvimento): Elemento que representa o ambiente responsável para a construção das aplicações;
  - *ApplicationDeveloped* (Aplicações Desenvolvidas): Este elemento representa as aplicações dos clientes;
  - *APIService* (Serviço de API): Elemento que representa as APIs que a nuvem disponibiliza para que os clientes possam utilizar o serviço na aplicação criada;
  - *DatabaseService* (Serviço de Banco de Dados): Representa o serviço de Banco de Dados que a aplicação utilizará para o armazenamento e gerenciamento de dados;

- SaaS: É o elemento que representa as nuvem de SaaS que possui os seguintes elementos:

  - *Applications* (Aplicações): Elemento que representa as aplicações disponibilizadas nas nuvens;
  - *CustomerDataApplications* (Dados de Aplicações dos Clientes): Elemento que contém informações responsáveis pelo armazenamento de dados gerados pelas aplicações utilizadas pelos clientes;
  - *TypeApplication* (Tipo de Aplicação): Elemento que representa os diferentes tipos de aplicações que o cliente pode utilizar neste tipo de nuvem.

A proposta do Metamodelo de Computação em Nuvem Abstrata tem a finalidade de compor o conjunto de metamodelos utilizados no *framework* proposto, ou seja, este metamodelo somente foi apresentado, mas não foi utilizado na implementação do *framework*. Neste trabalho, procurou-se focar no desenvolvimento de casos de teste e não na geração do código da aplicação.

4.5 Síntese

Este capítulo apresentou a proposta de um *framework* para geração de casos de teste baseado em MDE para Computação em Nuvem. Uma abordagem foi desenvolvida e apresentada. Esta abordagem utilizou Teste de *Software* para Computação em Nuvem e MDE para a geração automática de casos de teste. No *framework* apresentado, os níveis de abstração dos elementos da abordagem foram separados e detalhados.

Também, a proposta de uma metodologia para o funcionamento do *framework* proposto foi exibida. Um algoritmo para a geração automática de casos de teste foi apresentado com o objetivo de compor o *framework* e auxiliar no processo de criação de casos de teste mais eficientes e com menos intervenção humana. Uma descrição e diagramas de atividades foram projetados para ilustrar detalhadamente o funcionamento do algoritmo. Finalizando este capítulo, metamodelos desenvolvidos, que fazem parte da solução, foram apresentados e detalhados.

5 Implementação do Protótipo do *Framework* para Casos de Teste em Computação em Nuvem

Neste Capítulo, a implementação do *framework* para suporte de testes em Computação em Nuvem é apresentada. O *framework* foi desenvolvido com o auxílio do EMF *(Eclipse Modeling Framework)* que consiste em um *framework* facilitador da geração de código, apoiado na construção de aplicações Java [59] baseadas em simples definições de modelos. Utilizou-se também a ferramenta do EMF, chamada *GenModel*, para gerar os *plug-ins* em Eclipse para criação e edição dos modelos utilizados no trabalho.

Um algoritmo implementado em linguagem Java para automatizar a geração de casos de teste foi implementado, juntamente com a combinação de elementos do metamodelo de elementos de teste apoiado aos metamodelos de critérios de teste, desenvolvidos neste *framework*. Finalmente, a utilização das ferramentas (MT4MDE) e (SAMT4MDE) que auxiliaram na composição da implementação do *framework*.

5.1 Implementação do Algoritmo para geração automática de Modelos de Teste

Um algoritmo para a geração automática de casos de teste foi desenvolvido, conforme apresentado na Seção 4.3. Este algoritmo tem o objetivo de facilitar a criação de casos de teste baseado no conceito de criação de metamodelos de critérios de teste, conforme descrito na Seção 4.4.1.

Este algoritmo possui três etapas bem definidas. A primeira etapa consiste na leitura de um modelo de negócio, conforme um diagrama de classes UML e instanciação de um modelo de elemento de testes, conforme um metamodelo de elementos de testes (descrito na Seção 4.4.2). A segunda etapa consiste na classificação dos elementos de teste instanciados no modelo de elementos de teste e a geração dos modelos de critérios de teste conforme os metamodelos de critérios (descritos na Seção 4.4.1). A

terceira etapa do algoritmo consiste na geração dos casos de teste, ou seja, a transformação de modelos a partir dos modelos de critérios de testes gerados na Etapa 2 do algoritmo para um modelo de caso de teste, conforme o metamodelo de casos de teste para Nuvens (descrito na Seção 4.4.3)

5.1.1 Implementação da leitura do Modelo de Negócios da Aplicação

A primeira Etapa do algoritmo foi implementada com o EMF porque este *framework* do Eclipse suporta a criação de modelos. Sendo assim, foi construído o metamodelo de Elementos de Teste (conforme Seção 4.4.2) no EMF, no formato Ecore (linguagem de metamodelagem do EMF).

A partir do metamodelo *TestingElements.ecore*, criou-se um modelo gerador (*TestingElements.genmodel*) que é utilizado na segunda Etapa do algoritmo. Ou seja, a partir da instanciação do Metamodelo de Elementos de Teste é gerado o Modelo de Elementos de Teste, em formato XMI, como ilustra a Figura 5.1. Este modelo foi lido com o código implementado em Java e com a utilização da API denominada *Simple API for XML (SAX)* [57].

O Modelo de Elemento de Teste, baseado no modelo de negócios da aplicação e instanciado a partir da ferramenta do EMF foi salvo no repositório do *framework* (*Testing Elements Repository*), conforme ilustrado na Figura 5.2.

5.1.2 Implementação da Classificação dos Elementos de Teste

Para o desenvolvimento da segunda Etapa, utilizou-se a linguagem Java para o processamento dos modelos de Elementos de Teste, gerado em formato XMI, junto com o *SAX* [57] para leitura do documento XMI e o *Streaming API for XML (StAX)* [83] para a escrita de um documento XMI.

O processamento de XMI e XML em linguagem Java pode ser feito de várias maneiras. Existem diversas APIs que permitem este processamento (leitura e escrita de XMI e XML) [21] [75], sendo que, dentre as mais conhecidas, pode-se destacar as APIs *Simple API for XML SAX* [57] e *Document Object Model (DOM)* [10]. Estas APIs foram desenvolvidas para permitir o acesso dos programadores às informações contidas em

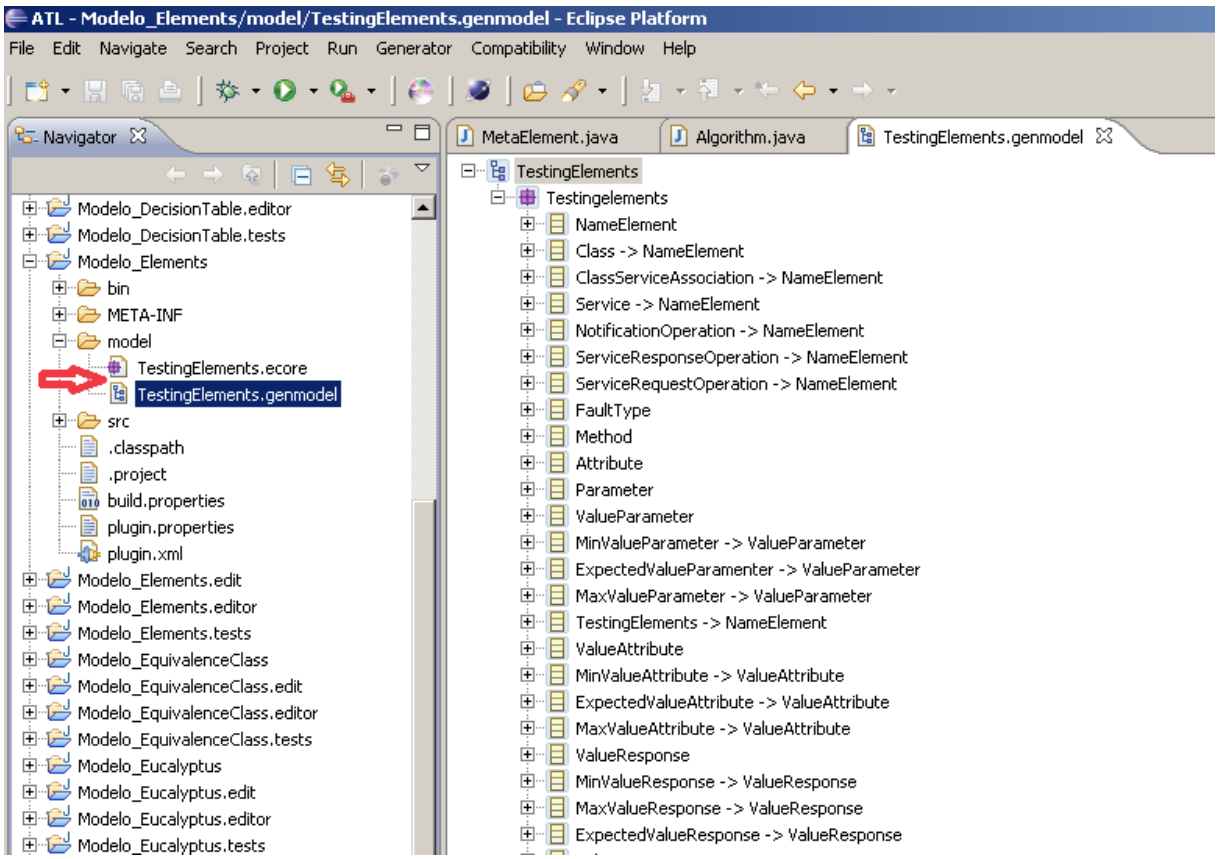

Figura 5.1: De Ecore para GenModel

arquivos XML, sem a necessidade de recorrerem a um interpretador da linguagem de programação [75].

No entanto, a abordagem utilizada para a realização do processo varia de uma API para a outra. O DOM permite o acesso à informação através de uma estrutura hierárquica (árvores), onde todo o documento é carregado em memoria. Já o SAX, permite o acesso à informação através de sequências de eventos. Os arquivos XMI que necessitam ser lidos e que foram utilizados no *framework* podem apresentar tamanhos grandes. Uma leitura onde é necessário carregar toda a informação, se torna uma solução não recomendada em nível de desempenho. Por este motivo, a API escolhida para a implementação da leitura dos arquivos foi o SAX. E o StAX foi utilizado para a escrita dos documentos XMI, por apresentar certa facilidade em sua utilização.

Classes Utilizadas e Desenvolvidas no Algoritmo

Classes foram desenvolvidas na construção do algoritmo de geração de casos de teste, conforme exibido na Figura 5.3. *ArrayList* [58] foram utilizados para a

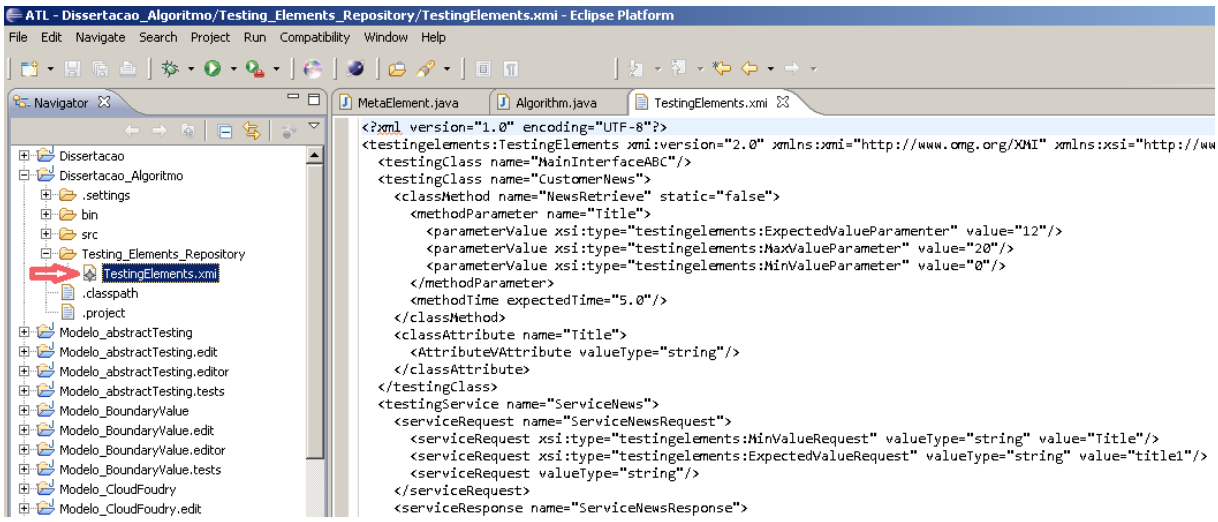

Figura 5.2: Repositório de Elementos de Teste

organização dos elementos que compõem os critérios de testes instanciados nos modelos. Uma descrição detalhada das principais é apresentado para melhor entendimento do algoritmo.

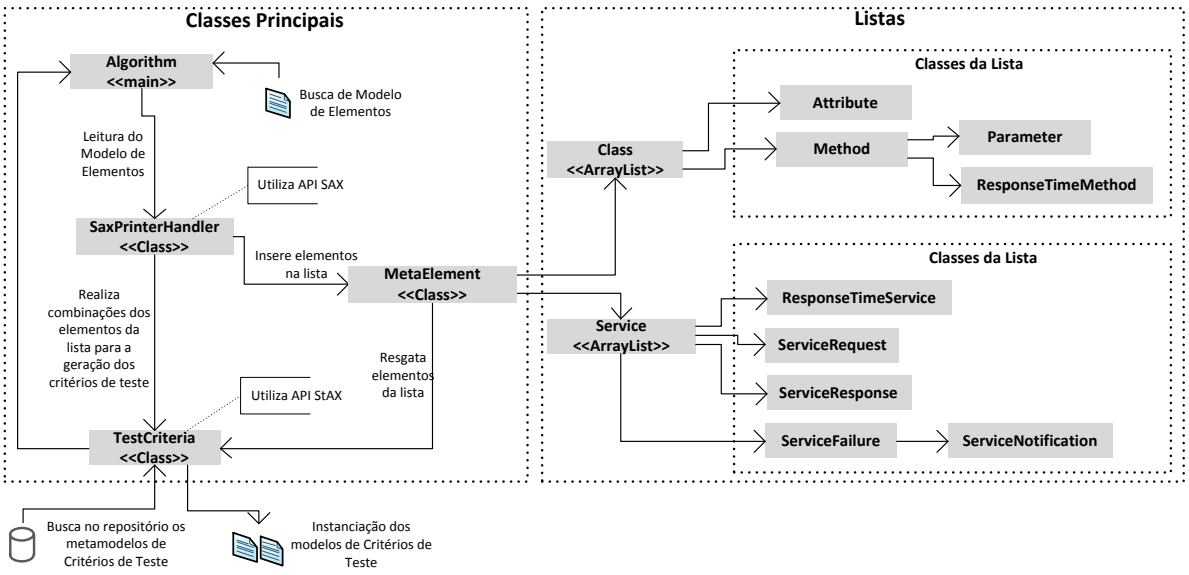

Figura 5.3: Organização e funcionalidades das classes no Algoritmo.

Classe Algorithm

Classe principal do algoritmo, responsável por fazer a chamada da leitura do arquivo XMI e chamada da classe *SaxPrinterHandler*. Um fragmento do código desta classe é apresentado a seguir.

Código 5.1: Fragmento de código da classe *Algorithm*

```
1 public static void main(String[] args) throws Exception {
2
3  XMLReader xmlReader = new SAXParser();
4  SAXPrintHandler handler = new SAXPrintHandler();
5  xmlReader.setContentHandler(handler);
6  xmlReader.setErrorHandler(handler);
7  FileReader reader = new FileReader("arquivo_teste.xmi");
8  xmlReader.parse(new InputSource(reader));
```

O trecho de código do algoritmo realiza a chamada de leitura do arquivo em XMI na linha 7 (sete), conforme exibido na Listagem 5.1.

Classe SaxPrinterHandler

Classe com métodos responsáveis pela leitura dos elementos do modelo e adição destes elementos aos *ArrayList* correspondentes. O método *startElement* realiza esta separação dos elementos e instancia os objetos de acordo com a classificação dentro do modelo de elementos e envia para o método *endElement* que realiza a adição destes objetos nos *ArrayList*. Um trecho da classe *SaxPrinterHandler* é exibido.

Código 5.2: Fragmento de código da classe *SaxPrinterHandler*

```
1 public void startElement(String uri, String name, String qName, Attributes atts) {
2  String verificaTipo = null;
3  int nroInstancias = 0;
4  if (atts.getLength() > 0) {
5   String attribs = " ";
6   int attrib;
7   for ( attrib = 0; attrib < atts.getLength(); ++attrib){
8
9    if (name.equals("testingClass")){
10         nomeClasse.setNome(atts.getValue(attrib));
11      attribs += atts.getLocalName(attrib) + ": '" +  atts.getValue(attrib) + "' ";}
12
13   if (name.equals("classMethod")){
14       if (atts.getLocalName(attrib).equals("name")){
15        nomeMetodo.setNome(atts.getValue(attrib));
16       }  }
17
18   if(name.equals("methodParameter")){
19       parametro = new Parametros();
20       parametro.setNome(atts.getLocalName(attrib));
21   }
22
23
24 public void endDocument() {
```

```
25

26 CriteriosDeTeste escritaMetamodelosClasse = new CriteriosDeTeste();
27 escritaMetamodelosClasse.contagemCriterios(metaModelo);}
```

Observa-se entre as linhas de código 7 (sete) a 21 (vinte e um), da Listagem 5.2, que o algoritmo faz uma comparação visando detectar as *tags* do modelo. Por exemplo, na linha 9 (nove), o algoritmo busca pela *tag* chamada *testingClass*. Se na leitura do XMI esta *tag* é encontrada, o objeto *nomeClasse* recebe o conteúdo referente ao rótulo pertencente à esta *tag*.

Classe TestCriteria

Classe que faz as combinações dos elementos a partir dos metamodelos de critérios de teste existentes no repositório e instancia os modelos de critérios de teste. O método *contagemCriterios* realiza a contagem e captura o nome do critério de teste e faz a chamada dos métodos para a separação dos elementos (conforme elementos de classes, serviços e de aplicação). Nesta classe também foram criados métodos para a instanciação dos critérios de teste, de acordo com o tipo e valor do elemento a ser testado e testes de tempo para aplicação, tempo de resposta de serviços e métodos de classes. Um exemplo de método para a instanciação dos critérios de teste pode ser visto no Fragmento 5.3, entre as linhas 3 (três) e 20 (vinte) exibido a seguir.

Código 5.3: Fragmento de código da classe *TestCriteria*

```
1 public void separaElementosClasses (MetaElemento metaElemento) {
2 ...
3  metaElementoIterator = metaElemento.listaClasse.iterator();
4  while (metaElementoIterator.hasNext()) {
5
6   Classe objClasse = (Classe) metaElementoIterator.next();
7   classe = objClasse.getNome();
8   if (!(metaElementoIterator == null)){
9
10   ArrayList<Metodos> metodos = new ArrayList<Metodos>();
11   metodos = objClasse.getMetodos();
12   Iterator<Metodos> metodoiIterator = metodos.iterator();
13   while(metodoiIterator.hasNext()){
14    if (!(metodoiIterator == null)){
15     Metodos objMetodos = metodoiIterator.next();
16     metodo = objMetodos.getNome();
17
```

```
18      ArrayList<TempoRepostaMetodo> tempoRepostaMetodos = new ArrayList<TempoRepostaMetodo>()↩
            ;
19      tempoRepostaMetodos = objMetodos.getTempoRespostaMetodo();
20      Iterator<TempoRepostaMetodo> tempoMetodoIterator= tempoRepostaMetodos.iterator();
21
22
23
24 public void ClassificaTestesTempoAplicacao(){
25
26  int numeroTestado, aux;
27  if(numeroInstancias > 0){
28
29   aux = numeroInstancias -1;
30   numeroTestado = aux;
31   stress(numeroInstancias, numeroTestado, nomeAplicacao, caminhoAplicacao, numeroInstancias)↩
         ;
32
33
34
35 public void boundaryValue(String objeto, String subObjeto, String valorEsperado, String ↩
      valorTestado, String TipoTeste, String TipoValor) {
36 ...
37    XMLOutputFactory factory = XMLOutputFactory.newInstance();
38    XMLStreamWriter writer ;
39    try {
40                writer = factory.createXMLStreamWriter(streamOut);
41         writer.writeStartDocument("Cp1252", "1.0");
42
43         writer.writeStartElement("boundaryvalue:CriteriaBoundaryValue");
44         writer.writeAttribute("xmi:version", "2.0");
45         writer.writeAttribute("name", objeto);
46
47         writer.writeStartElement("criteriaObject");
48         writer.writeAttribute("name", objeto);
49         writer.writeAttribute("typeBoundary", TipoTeste);
```

Classe MetaElement

Classe que contém o *ArrayList* de Classes responsável pelo armazenamento das informações referentes à Classes, Atributos, Métodos, Parâmetros e Tempo de resposta de um Método. A classe *MetaElement* também contém o *ArrayList* de Serviços responsável por armazenar as informações, tais como: Nome do Serviço, Serviço de Resposta, Serviço de Requisição e Serviço de Notificação. Fragmentos do código da classe desenvolvida é listado a seguir.

Código 5.4: Fragmento de código da classe *MetaElement*

```
1 public class MetaElement {
2  Collection<Class> listaClasse = new ArrayList();
3  Collection<Service> listaServico = new ArrayList();
4
5  static String caminhoAplicacao = null;
6  private static String nomeAplicacao = null;
```

Combinações de Elementos dos Modelos de Critérios de Teste

Conforme descrito na Seção 5.1.2, combinações foram realizadas para a composição dos casos de teste. Estas combinações é o resultado da junção dos elementos obtidos a partir do modelo de Elementos de Teste e dos metamodelos de Critérios de Teste armazenados no repositório do *framework*.

Cada metamodelo de critério de teste possui a sua lógica de funcionamento de acordo com o critério de teste específico. Por exemplo, o critério de teste *Equivalence Class* tem como objetivo reduzir o número de casos de teste. Esta redução é obtida através da divisão do domínio de entrada (por exemplo, uma regra de negócio) em subdomínios de elementos equivalentes. Um exemplo de utilização deste critério de teste é exibido:

Regras de negócios de idade para fazer inscrição em concurso

Regras	**Classes de Equivalências Válidas**	**Classes de Equivalências Inválidas**
Com idade entre 0 a 15 não pode fazer inscrição no concurso	0 >= VE <= 15	VE < 0 e VE > 15
Com idade entre 16 a 55 pode fazer inscrição no concurso	16 >= VE <= 55	VE < 16 e VE > 55
Com idade entre 56 a 120 não pode fazer inscrição no concurso	56 >= VE <= 120	55 < VE e 120 > VE

Tabela 5.1: Regras de negócio e classes de equivalências

A Tabela 5.1 possui três colunas sendo que a primeira coluna, chamada **Regras**, representa as regras de negócio do exemplo proposto, onde o valor a ser testado seria a **idade**. Na segunda coluna são representadas as **classes de equivalências válidas** para cada regra sendo que **VE** representa os **valores de entrada**. E na terceira coluna representa as **classes de equivalências inválidas**. Para elaborar os casos de teste, primeiramente escolhe-se uma regra para ser testada. A cor em vermelho destaca o intervalo do qual serão gerados os casos de teste, conforme Tabela 5.2:

Regras	Classes de Equivalências Válidas	Classes de Equivalências Inválidas
Com idade entre 0 a 15 não pode fazer inscrição no concurso	0 >= VE <= 15	VE < 0 e VE > 15
Com idade entre 16 a 55 pode fazer inscrição no concurso	16 >= VE <= 55	VE < 16 e VE > 55
Com idade entre 56 a 120 não pode fazer inscrição no concurso	56 >= VE <= 120	55 < VE e 120 > VE

Tabela 5.2: Valores a serem testados

A partir da Tabela 5.2 pode-se notar a existência de **três** conjuntos formados pelos elementos entre os intervalos: valores abaixo e valores acima. Desta forma, considerando que os elementos de mesmo conjunto afetam o *software* de forma semelhante, podemos reduzir os casos de teste utilizando a técnica de equivalência de classes. Pode-se então reduzir os casos de teste em apenas **3 valores** ao invés de ser testados **120 valores** (de 0 a 120) conforme Tabela 5.3.

Entradas	Saídas
0	Não pode fazer inscrição
17	Pode fazer inscrição
57	Não pode fazer inscrição

Tabela 5.3: Entradas dos casos de teste

No *framework* proposto nesta dissertação, utilizou-se 5 (cinco) critérios de teste. Através destes 5 (cinco) casos de teste, 35 (trinta e cinco) combinações foram detectadas. Estas combinações podem ser vistas na Listagem 5.5, conforme o trecho da documentação da classe implementada *TestCriteria*.

Código 5.5: Combinações detectadas dos critérios de teste

```
1 //*****************************************************************************
2 //Classe para instanciar os critérios de testes em XMI
3 //Recebe arraylist e instancia modelos
4 //*****************************************************************************
5
6 //*****************************************************************************
7 //Para o framework proposto, cinco metamodelos de critérios de testes foram utilizados:
8 //Analise do valor limite, Classes de Equivalência, Tabela de Decisão, Stress
9 // e Tempo de resposta.
10
```

```
11 //De acordo com a classificação e separação dos elementos, os criterios são instanciados.
12 //-------------------------------------------------------------------------------------
13 //Para: Análise do valor limite
14 // Se tipo do valor = integer, double e float
15 // Para continuo aplicar valores  de testes sequencias e para discreto valores não ↩
       sequenciais
16 //
17 //Se existe MAXVALUE - teste menor MAXVALUE = V
18 //Se existe MAXVALUE - teste igual MAXVALUE = V
19 //Se existe MAXVALUE - teste maior MAXVALUE = F
20 //
21 //Se existe MINVALUE - teste menor MINVALUE = F
22 //Se existe MINVALUE - teste igual MINVALUE = V
23 //Se existe MINVALUE - teste maior MINVALUE = V
24 //
25 //Se existe EXPECVALUE - teste menor EXPECVALUE = V
26 //Se existe EXPECVALUE - teste igual EXPECVALUE = V
27 //Se existe EXPECVALUE - teste maior EXPECVALUE = V
28 //
29 //Para: Análise do valor limite
30 // Se tipo do valor = string
31 // Para continuo aplicar valores  de testes sequencias
32 //
33 //Se existe EXPECVALUE - teste = EXPECVALUE = V
34 //Se existe EXPECVALUE - teste <> EXPECVALUE = V
35 //
36 //-------------------------------------------------------------------------------------
37 //Para: Classes de equivalência
38 //Se tipo valor = integer, double e float
39 //
40 //EXPECVALUE recebe integer - TESTINGVALUE recebe string = F
41 //EXPECVALUE recebe double - TESTINGVALUE recebe string = F
42 //EXPECVALUE recebe float - TESTINGVALUE recebe string = F
43 //
44 //Se tipo valor = char
45 //
46 //EXPECVALUE recebe char - TESTINGVALUE recebe string = F
47 //
48 //Se tipo valor = integer, double e float
49 // Para continuo aplicar valores  de testes sequencias e para discreto valores não ↩
       sequenciais
50 //
51 //Se existe MAXVALUE - teste menor MAXVALUE = V
52 //Se existe MAXVALUE - teste maior MAXVALUE = F
53 //
54 //Se existe MINVALUE - teste menor MINVALUE = F
55 //Se existe MINVALUE - teste maior MINVALUE = V
56 //
57 //-------------------------------------------------------------------------------------
58 //Para: Tempo de Resposta
59 //Se existir tempo esperado metodo
```

```
60 //
61 //Se existe TIMEEXPECVALUE  - teste maior TIMEEXPECVALUE = F
62 //Se existe TIMEEXPECVALUE  - teste igual TIMEEXPECVALUE = V
63 //Se existe TIMEEXPECVALUE  - teste menor TIMEEXPECVALUE = V
64 //
65 //Se existir tempo esperado servico
66 //
67 //Se existe TIMEEXPECVALUE  - teste maior TIMEEXPECVALUE = F
68 //Se existe TIMEEXPECVALUE  - teste igual TIMEEXPECVALUE = V
69 //Se existe TIMEEXPECVALUE  - teste menor TIMEEXPECVALUE = V
70 //
71 //-------------------------------------------------------------------------------------
72 //Para: Stress
73 //Se existir tempo esperado  e numero de instancias
74 //
75 //Se existe TIMEEXPECVALUE e NUMBERINSTANCES - teste maior TIMEEXPECVALUE = F
76 //Se existe TIMEEXPECVALUE e NUMBERINSTANCES - teste igual TIMEEXPECVALUE = V
77 //Se existe TIMEEXPECVALUE e NUMBERINSTANCES - teste menor TIMEEXPECVALUE = V
78 //
79 //-------------------------------------------------------------------------------------------------↩

80 //Para: Tabelas de decisão
81 //(No framework este critério será utilizado somente quando existe tempo esperado e numero ↩
       de instancias)
82 //
83 //Se existe TIMEEXPECVALUE e NUMBERINSTANCES - teste igual TIMEEXPECVALUE e teste igual ↩
       NUMBERINSTANCES= V
84 //Se existe TIMEEXPECVALUE e NUMBERINSTANCES - teste igual TIMEEXPECVALUE e teste menor ↩
       NUMBERINSTANCES= V
85 //
86 //Se existe TIMEEXPECVALUE e NUMBERINSTANCES - teste maior TIMEEXPECVALUE e teste maior ↩
       NUMBERINSTANCES = F
87 //Se existe TIMEEXPECVALUE e NUMBERINSTANCES - teste maior TIMEEXPECVALUE e teste igual ↩
       NUMBERINSTANCES= F
88 //Se existe TIMEEXPECVALUE e NUMBERINSTANCES - teste maior TIMEEXPECVALUE e teste menor ↩
       NUMBERINSTANCES= F
89 //
90 //Se existe TIMEEXPECVALUE e NUMBERINSTANCES - teste menor TIMEEXPECVALUE e teste igual ↩
       NUMBERINSTANCES= V
91 //Se existe TIMEEXPECVALUE e NUMBERINSTANCES - teste menor TIMEEXPECVALUE e teste menor ↩
       NUMBERINSTANCES= V
92 //
93 //*************************************************************************************↩
```

Uma destas combinações foi utilizada no método *ClassificaTestesTipoValor*, pertencente à classe *TestCriteria*, conforme exibido na Listagem 5.6, entre as linhas 6 (seis) e 15 (quinze).

Código 5.6: Utilização das combinações dos critérios de teste

```
1
2 public void ClassificaTestesTipoValor(String objTestado, String subObjTestado, String ↩
      valorEsperado, String valorMin, String ValorMax, String TipoParametro){
3
4 ...
5
6  if (TipoParametro.equals("double")){
7
8   doubleAux = Double.parseDouble(ValorMax) - 1.0;
9   boundaryValue(objTestado, subObjTestado, ValorMax, Double.toString(doubleAux), continuo, ↩
        TipoParametro);
10
11  doubleAux2 = Double.parseDouble(ValorMax) - 3.0;
12  boundaryValue(objTestado, subObjTestado, ValorMax, Double.toString(doubleAux2), discreto, ↩
        TipoParametro);
13
14  doubleAux2 = Double.parseDouble(ValorMax) - 5.0;
15  boundaryValue(objTestado, subObjTestado, ValorMax, Double.toString(doubleAux2), discreto, ↩
        TipoParametro);
16
17 ...
```

5.1.3 Implementação da Instanciação dos Modelos de Critérios de Teste

A terceira parte do algoritmo foi implementada utilizando definições de transformação descritas na linguagem ATL. Foram utilizados 5 (cinco) metamodelos de entrada e 1 (um) metamodelo de saída. Os metamodelos de entrada utilizados foram:

- *BoundaryValue.ecore*;
- *EquivalenceClass.ecore*;
- *DecisionTable.ecore*;
- *ResponseTime.ecore*;
- *Stress.ecore*.

E o metamodelo de saída utilizado foi o *TestingMetamodel.ecore*. A tela que contém as configurações para execução da definição de transformação é apresentada

conforme a Figura 5.4. As regras de transformação teve como entrada 5 (cinco) modelos que foram escritos conforme os metamodelos de critérios de teste: *BoundaryValue*, *EquivalenceClass*, *ResponseTime*, *DecisionTable* e *Stress*. Os modelos instanciados por estes metamodelos, constituem em um caso de teste a ser executado na aplicação.

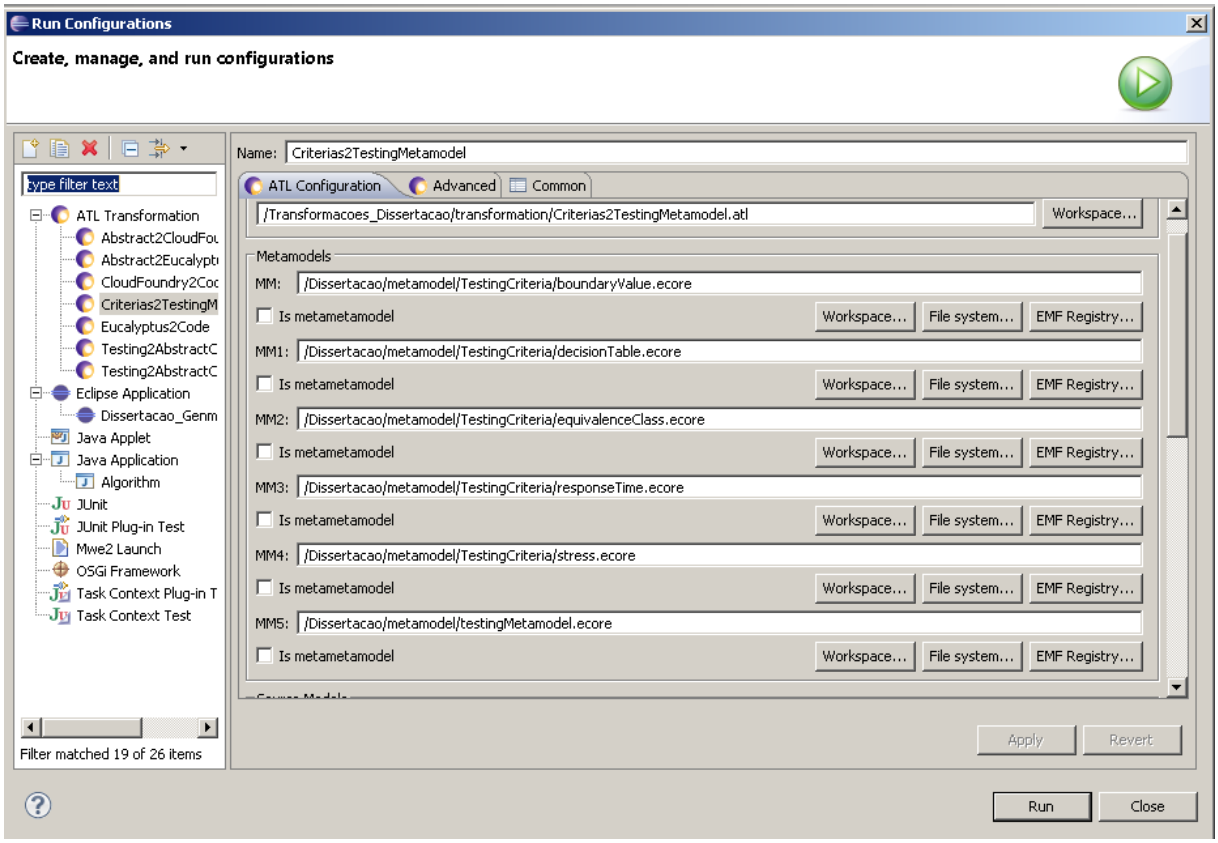

Figura 5.4: Configuração da definição de transformação de Critérios de Teste para o modelo *TestingMetamodel*

O algoritmo executado na etapa anterior (etapa 2), instancia um modelo de critério para cada caso de teste que atenda às combinações de elementos, conforme explicado na Seção 5.1.2. O modelo de casos de teste, escrito conforme o metamodelo *Testing Metamodel*, constituirá na união de todos os casos de teste instanciados pelos modelos de critérios de testes, ou seja, um único modelo escrito conforme o *Testing Metamodel* conterá todos os casos de teste a serem executados posteriormente na aplicação. Um fragmento do arquivo em formato *ATL*, contendo estas definições de transformação pode ser visto a seguir:

Código 5.7: Fragmento das definilções de transformação: *Criteria2TestingMetamodel*

```
1 module Criterias2TestingMetamodel;
```

```
2 create OUT : MM5 from IN : MM, IN1 : MM1, IN2 : MM2, IN3 : MM3, IN4 : MM4;
3
4 rule CriteriaBoundaryValue{
5   from MM:MM!CriteriaBoundaryValue
6   to t1:MM5!TestType (
7         name<- 'Boundary Value  ' + MM.name  ),
8    t2:MM5!TestCriteria (
9         name<- 'Functional'  ),
10   t3:MM5!ModuleTest (
11        name<- MM.name,
12 testSuiteCount<- '1' ),
13 t4:MM5!TestSuite (
14        name<- MM.name,
15 testCaseCount<- '1' ),
16 t5:MM5!TestCase (
17        name<- MM.typeBoundary,
18 testDateCount<- '1' )}
19
20 rule CriteriaBoundaryValue2TestClass{
21      from MM:MM!ObjectTesting
22      to MM5:MM5!TestClass (
23       name<- MM.name )}
```

5.2 Implementação da Geração dos casos de teste para Computação em Nuvem

Para a implementação da geração de casos de teste proposto no *framework*, metamodelos de testes específicos para nuvem foram desenvolvidos, conforme apresentado na Seção 4.4. Junto com estes metamodelos, definições de transformação foram especificadas para que os códigos dos casos de teste pudessem ser gerados. As ferramentas (MT4MDE) e (SAMT4MDE) foram utilizadas no *framework* para apoiar essa construção das definições de transformação, facilitando a criação desses códigos.

5.2.1 Utilização das ferramentas MT4MDE e SAMT4MDE

As ferramentas MT4MDE e SAMT4MDE, desenvolvidas no trabalho de [37] [39], serviram para apoiar a construção das definições de transformação utilizadas neste *framework*, conforme ilustrado na Figura 5.5. Essas ferramentas suportam a gera-

ção semiautomática das especificações de correspondências e automatiza a geração de definições de transformações.

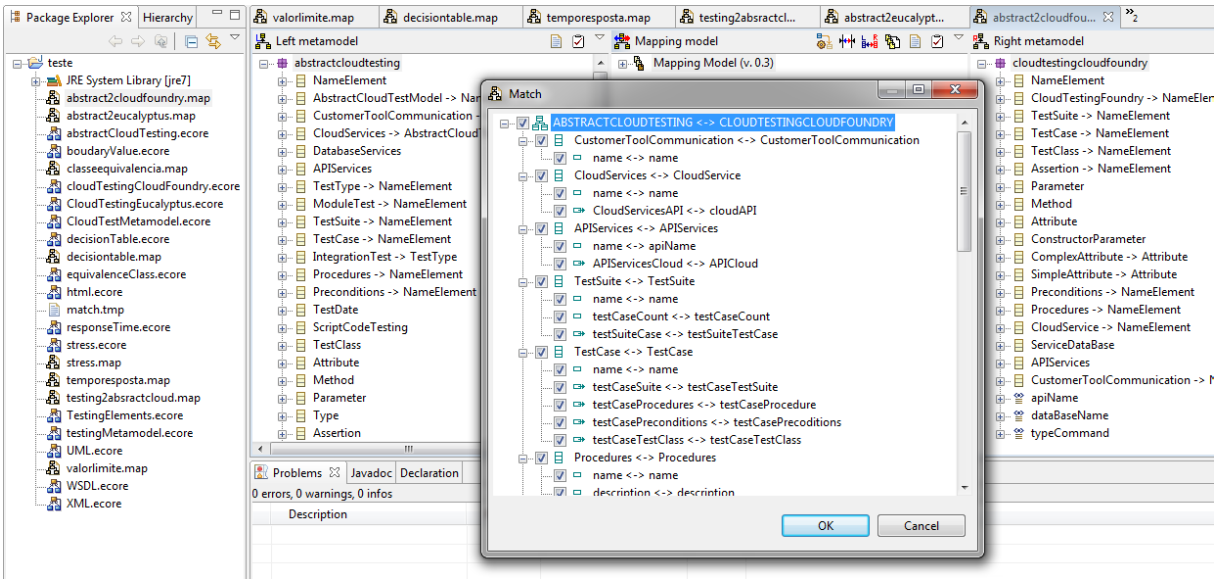

Figura 5.5: Utilização das ferramentas (MT4MDE) e (SAMT4MDE).

Ferramenta de correspondências para MDE (MT4MDE)

MT4MDE *(Mapping Tool for Model Driven Engineering)* é uma ferramenta que possibilita a criação das especificações de correspondências entre modelos. Esta ferramenta determina uma possível especificação de correspondência e a permissão para a geração de definições de transformação em linguagem ATL a partir de um modelo de correspondência.

Ferramenta de geração semiautomática de correspondências para MDE (SAMT4MDE)

A ferramenta SAMT4MDE *(Semi-Automatic Matching Tool for MDE)* é uma ferramenta que se integra a ferramenta MT4MDE. Esta união proporciona a funcionalidade de determinar as correspondências entre dois metamodelos de forma semiautomática. A SAMT4MDE permite a criação de um modelo de correspondência baseado nas correspondências [37] [39].

Definições de Transformações utilizadas no *Framework*

Para construção dos casos de teste para ambientes de Computação em Nuvem, diversas definições de transformações em linguagem ATL foram construídas. Através das definições de transformação, modelos foram manipulados para obtenção dos casos de teste. As definições de transformação realizadas neste trabalho foram:

- ***Criterias2TestingMetamodel*** - Definições de transformação utilizadas para a implementação da terceira parte do algoritmo, conforme Seção 5.1, responsável pela instanciação do modelo de teste. Nessas definições, 5 (cinco) metamodelos de critérios de testes foram utilizados como entrada: *BoundaryValue Metamodel*, *DecicionTable Metamodel*, *Stress Metamodel*, *EquivalenceClass Metamodel* e *TimeResponse Metamodel*. E para a saída, o modelo foi gerado conforme o metamodelo *Testing Metamodel*;

- ***TestingMetamodel2AbstractCloudTestingPaaS*** - Definições de transformação responsáveis por instanciar o modelo de casos de teste para uma nuvem abstrata, ou seja, além das informações relativas aos casos de teste, informações para qual tipo de nuvem será aplicados (neste caso, já direcionando para nuvem do tipo PaaS) são adicionadas. Utilizou-se como entrada o metamodelo *Testing Metamodel* e como saída o *AbstractCloudTesting Metamodel*;

- ***TestingMetamodel2AbstractCloudTestingIaaS*** - Definições de transformação semelhantes às definições de *TestingMetamodel2AbstractCloudTestingPaaS*. E neste caso, já direcionando para nuvem do tipo IaaS. Utilizou-se como entrada o metamodelo *Testing Metamodel* e como saída o *AbstractCloudTesting Metamodel*;

- ***AbstractCloudTesting2CloudTestingCloudFoundry*** - Estas definições de transformação são especificas para gerar o modelo de casos de teste para ser aplicado na nuvem *CloudFoundry*. O modelo gerado recebe informações relativas aos casos de teste acrescido das informações para ser aplicados na nuvem em específico, por exemplo, os comandos para a comunicação aos serviços da nuvem. O metamodelo de entrada nesta definição de transformação foi o *AbstractCloudTesting* e o metamodelo de saída utilizado foi o *CloudTestingCloudFoundry*;

- ***AbstractCloudTesting2CloudTestingEucalyptus*** - As definições de transformação utilizadas para gerar o modelo de casos de teste específico para nuvem *Eucalyp-*

tus. Assim como o modelo da nuvem *CloudFoundry*, os modelos gerados a partir desta definição de transformação, receberão os comandos específicos da nuvem *Eucalyptus*. O metamodelo de entrada utilizado foi o *AbstractCloudTesting* e o metamodelo de saída foi o *CloudTestingEucalyptus*;

- ***CloudFoundry2Code*** e ***Eucalyptus2Code*** - Definições de transformação responsáveis pela transformação de modelo-a-texto. Nestas transformações, os códigos dos casos de teste foram gerados para serem aplicados nas nuvens CloudFoundry e Eucalyptus. Esses códigos se apresentam em formato de *script* em *jUnit* e em *script shell*. Os metamodelos de entrada utilizados foram: *CloudTestingCloudFoundry* e *CloudTestingEucalyptus*.

As definições de transformação descritas no *framework*, podem ser vistas nos Anexos 8.1, 8.2, 8.3, 8.4, 8.5, 8.6, e 8.7 deste trabalho.

5.3 Síntese

Neste Capítulo, apresentou-se a implementação do *framework* para geração de casos de teste para Computação em Nuvem, com a finalidade de testar código-fonte de aplicações geradas em ambientes de Computação em Nuvem. Para a implementação deste *framework*, desenvolveu-se metamodelos específicos de critérios de testes e metamodelos específicos para nuvem.

Um algoritmo foi desenvolvido e foi adicionado ao *framework* com a proposta de automatizar o processo de criação de casos de teste. Esta automatização foi conseguida através da combinação da leitura dos elementos de teste, instanciados a partir do Modelo de Elementos de Teste, junto com a combinação dos elementos que compõem os critérios de teste, apresentados também em forma de metamodelos.

Um *plug-in* composto por duas ferramentas (MT4MDE e SAMT4MDE) auxiliou na geração das definições de transformação utilizadas neste *framework*, tornando a geração de casos de teste mais segura e com menor tendência a erros.

6 Exemplo Ilustrativo

Neste Capítulo, dois exemplos ilustrativos são apresentados com o objetivo de levar o leitor a um melhor entendimento do *framework* e demonstrar as funcionalidades por ele oferecidas. Um breve explicativo sobre o que foi realizado nos testes é apresentado. Em seguida, o primeiro exemplo é exibido de forma detalhada. Este exemplo consiste na geração de casos de teste para um serviço de notícias de um sistema *Web*. O segundo exemplo é exposto de forma simplificada e consiste em um sistema de controle de Recados, também como sistema *Web*.

6.1 Etapas executadas nos Exemplos

Nesta seção, apresenta-se a sequência de etapas realizadas nos exemplos ilustrativos. Um diagrama de blocos é apresentado para melhor entendimento, conforme ilustrado na Figura 6.1.

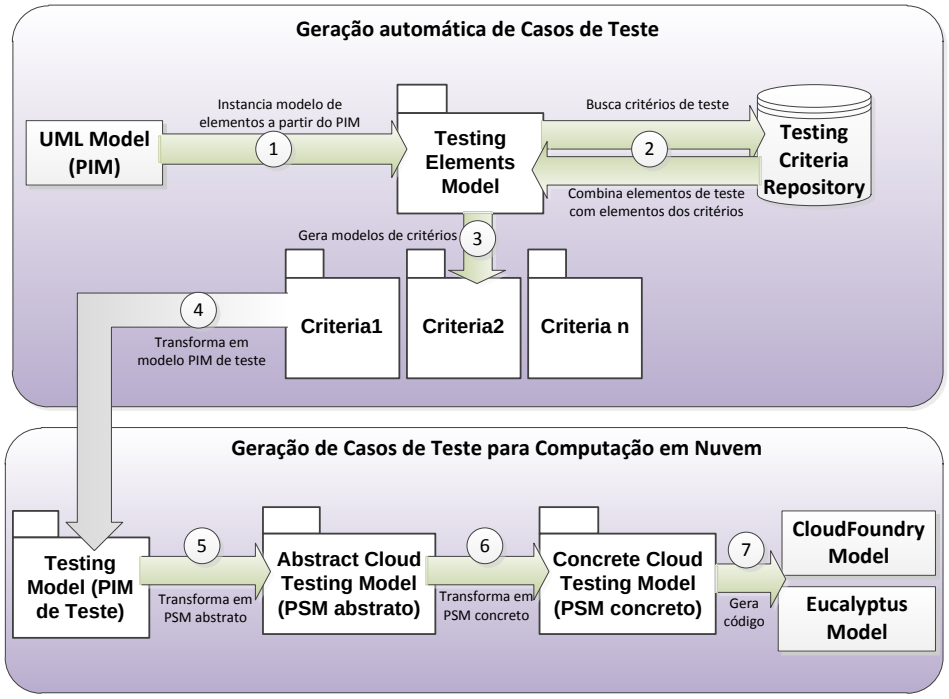

Figura 6.1: Etapas executadas nos Exemplos Ilustrativos

Pode-se perceber que o *framework* possui duas fases distintas. A primeira é a geração automática de critérios de teste. A segunda é a geração de casos de teste para plataforma de Computação em Nuvem. O primeiro exemplo ilustrativo seguirá todas as sequências de fases implementadas no protótipo do *framework* e o segundo exemplo começará a partir da segunda fase.

O objetivo é mostrar ao leitor que o *framework* suporta a criação de casos de teste sob duas formas. A primeira é a leitura de um modelo de negócio da aplicação e a geração de critérios de teste a partir dos metamodelos de critérios existentes no repositório do *framework*. A segunda forma suporta a geração direta do modelo de teste (PIM de teste), de forma manual, a partir dos casos de teste especificados pelo usuário.

6.2 Exemplo 1 - Serviço de Notícias de uma Página *Web*

Para demonstrar as funcionalidades do *framework* proposto, apresenta-se um exemplo ilustrativo que consiste na construção de casos de teste para o serviço de notícias de uma página *Web*. Este serviço permite que o cliente o acesse e este as retorne, contendo o título e o texto da notícia.

Sistemas como este apresentado no exemplo ilustrativo são compostos por funcionalidades distintas ou componentes funcionais, caracterizando a arquitetura em Três Camadas [16]. Cada uma das camadas da arquitetura trata de questões específicas e sua separação é facilmente identificável.

- Camada de Apresentação (*client tier*): responsável pela interface com o usuário. Pode incluir a adaptação da apresentação da aplicação para diferentes dispositivos, através de formatos como HTML, XML, etc. As classes dessa camada utilizam os serviços oferecidos pela camada de negócios;
- Camada Lógica (*middle tier*): responsável por implementar a lógica de negócio da aplicação. Nela estão todas as classes inerentes ao domínio da aplicação;
- Camada de Gerenciamento de Dados (*resource tier*): responsável pela persistência e acesso aos dados da aplicação que pode ser representado por um ou mais SGBDs.

No caso do exemplo ilustrativo a ser mostrado, será aplicado testes em nível de camada lógica da aplicação (*middle tier*).

O aplicativo testado no exemplo foi desenvolvido para ser utilizado na nuvem *CloudFoundry* e para a nuvem *Eucalyptus*, ou seja, o mesmo aplicativo estará em nuvens diferentes, o que demonstra a interoperabilidade conseguida através dos modelos desenvolvidos neste *framework*.

Os testes são em nível de aplicação e de formas diferentes. Para o aplicativo desenvolvido para a *CloudFoundry* são aplicados testes funcionais. E para a nuvem *Eucalyptus* será aplicado testes não funcionais e teste de tempo de resposta.

O *framework* pode gerar testes funcionais e não funcionais para as duas plataformas (*Eucalyptus* e *CloudFoundry*). No exemplo ilustrativo, aplicou-se os testes funcionais em uma nuvem e os não funcionais em outra nuvem, para demonstrar a interoperabilidade da ferramenta. O mesmo PIM (de negócios e de teste) pode ser transformado em modelos para plataformas específicas.

A Figura 6.2 ilustra os casos de uso básicos da página *Web* que simula o acesso a vários serviços em forma de diagrama de caso de uso em UML do exemplo ilustrativo.

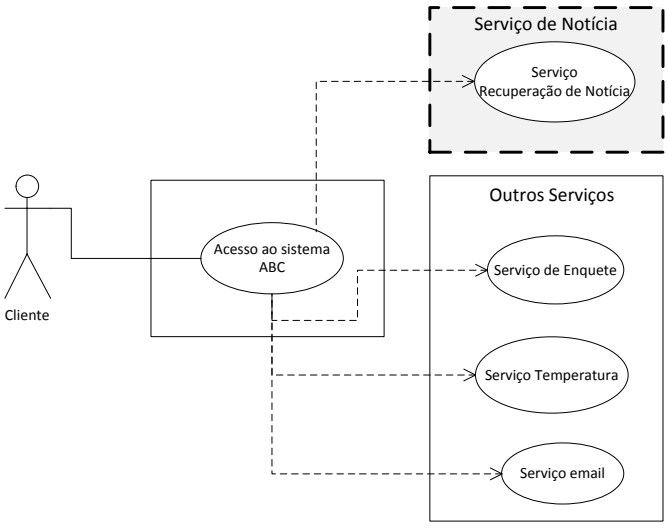

Figura 6.2: Diagrama de Caso de Uso do exemplo ilustrativo Serviço de Notícias

A sequência de passos para a realização do exemplo ilustrativo usando o *framework* proposto, pode ser observada na Figura 6.1 através das etiquetas enumeradas.

Código 6.1: Fragmento de WDSL para o exemplo ilustrativo

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:axis2="http://ws.↩
      apache.org/axis2" xmlns:ns1="http://org.apache.axis2/xsd" xmlns:wsaw="http://www.w3.org↩
      /2006/05/addressing/wsdl" xmlns:http="http://schemas.xmlsoap.org/wsdl/http/" xmlns:ns0="↩
      http://ws.apache.org/axis2xsd" xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:soap="↩
      http://schemas.xmlsoap.org/wsdl/soap/" xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/↩
      " xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/" targetNamespace="http://ws.↩
      apache.org/axis2">
3     <wsdl:documentation>News</wsdl:documentation>
4  ...
5
6
7     <wsdl:message name="getTituloResposta">
8         <wsdl:part name="parameters" element="ns0:getTitleResponse"/>
9     </wsdl:message>
10    <wsdl:message name="getTextoResposta"/>
11    <wsdl:message name="getTextoResposta">
12        <wsdl:part name="parameters" element="ns0:getTextResponse"/>
13    </wsdl:message>
14    <wsdl:message name="RecuperacaoNoticiaRequicicao"/>
15    <wsdl:message name="RecuperacaoNoticiaResposta">
16        <wsdl:part name="parameters" element="ns0:RecuperacaoNoticia"/>
17    </wsdl:message>
18    <wsdl:message name="setTituloRequisicao">
19        <wsdl:part name="parameters" element="ns0:setTitulo"/>
20    </wsdl:message>
```

Inicialmente, este exemplo considera o teste sobre um serviço *Web* de notícias. Portanto, conforme as funcionalidades iniciais do *framework* proposto, um WSDL é lido e instanciado em um modelo WSDL (“*Reuso de um documento WSDL*”, na Figura 6.1), escrito conforme um metamodelo WSDL. Um fragmento do WSDL utilizado neste exemplo pode ser visto na Listagem 6.1.

Depois, este modelo é transformado em um Diagrama de Classes UML para se obter parte do PIM da aplicação através de definições de transformação. Este Diagrama UML pode ser modificado pelo usuário, adicionando funcionalidades à aplicação, além do serviço utilizado (neste caso, o serviço de notícias). Este diagrama de classes é apresentado na seção seguinte.

6.2.1 Modelo Independente de Plataforma (PIM)

Na sequência das atividades do *framework*, um Modelo Independente de Plataforma (PIM) de um sistema foi gerado, com acesso à diversos serviços, como *email*, notícias e temperatura. Parte deste modelo foi gerado a partir da leitura do WSDL do serviço de recebimento de notícias. Funcionalidades além do serviço de notícias foram adicionadas pelo usuário, conforme apresentado na Figura 6.3.

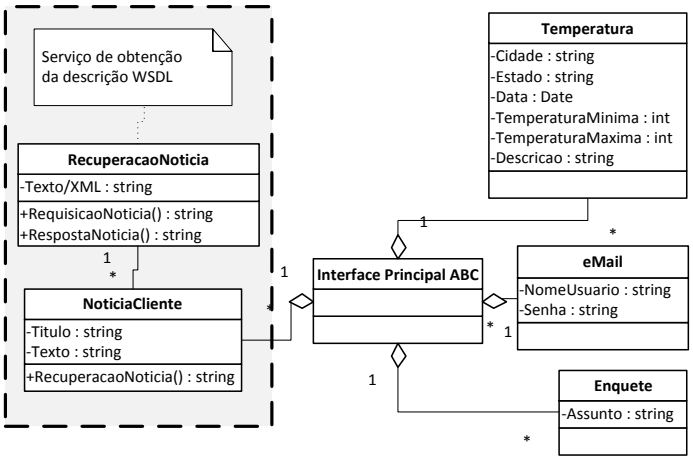

Figura 6.3: PIM em UML para o exemplo ilustrativo de Serviço de Notícias

Este PIM para a página *Web* possui as seguintes classes que servirão de informações para gerar os casos de teste:

- *NoticiasCliente*, classe que exibirá na página principal do sistema as notícias recebidas;
- *RecuperacaoNotícia*, classe que representa o acesso ao serviço de notícias através dos métodos: *RequisicaoNoticia* e *RespostaNoticia*. Esta classe representa o WSDL para a realização de testes sobre o serviço disponível por terceiros.

As demais classes também são exibidas:

- *Interface Principal ABC*, que representa a página principal da página *Web*;
- *eMail* que permite o acesso ao serviço de email para os clientes cadastrados;
- *Enquete* que exibe uma enquete para os clientes responderem;
- *Temperatura*, classe que exibe as informações sobre clima, temperatura mínima e máxima por cidade.

6.2.2 Geração automática de casos de teste pelo *Framework*

Neste exemplo ilustrativo, aplicou-se o algoritmo para geração de casos de teste. Instanciou-se o modelo de Elementos de Teste (etapa 1, Figura 6.1 relativo à parte a ser testada, ou seja, a parte pontilhada da Figura 6.3 que compreende o diagrama de classes apresentado neste exemplo. Um fragmento deste modelo de Elementos de Teste são apresentados na Listagem 6.2:

Código 6.2: Fragmento do modelo de Elementos de Teste em formato XMI

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <testingelements:TestingElements xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" ↩
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:testingelements="http://↩
      testingelements/1.0" name="InterfaceABC" PathApplication="" numberInstances="1">
3   <testingClass name="NoticiaCliente">
4     <classMethod name="RecuperacaoNoticia" static="true">
5       <methodParameter name="Titulo">
6         <parameterValue xsi:type="testingelements:ExpectedValueParamenter" valueType="string↩
              " value="Titulo1"/>
7         <parameterValue xsi:type="testingelements:MinValueParameter" valueType="string" ↩
              value="Titulo1"/>
8         <parameterValue xsi:type="testingelements:MaxValueParameter" valueType="string" ↩
              value="Titulo1"/>
9       </methodParameter>
10    </classMethod>
11   </testingClass>
12  <testingService name="RecuperacaoNoticia">
13    <serviceRequest name="RequisicaoNoticia">
14      <serviceRequest valueType="string"/>
15      <serviceRequest xsi:type="testingelements:ExpectedValueRequest" valueType="string" ↩
            value="Titulo1"/>
16      <serviceRequest xsi:type="testingelements:MinValueRequest" valueType="string" value="↩
            Titulo1"/>
17      <serviceRequest xsi:type="testingelements:MaxValueRequest" valueType="string" value="↩
            Titulo1"/>
18    </serviceRequest>
```

Com o modelo de Elementos de Teste instanciado, aplicou-se o algoritmo de geração de casos de teste (etapa 2, Figura 6.1). Observa-se que, na Figura 8.2, chamou-se a execução do algoritmo pelo EMF. No *Console* da ferramenta, observa-se os casos de teste gerados pela execução do algoritmo.

Com a execução do algoritmo sobre as 2 (duas) classes testadas: *RecuperacaoNoticia* e *NoticiaCliente*, gerou-se 6 (seis) casos de teste (etapa 3, Figura 6.1). A Figura

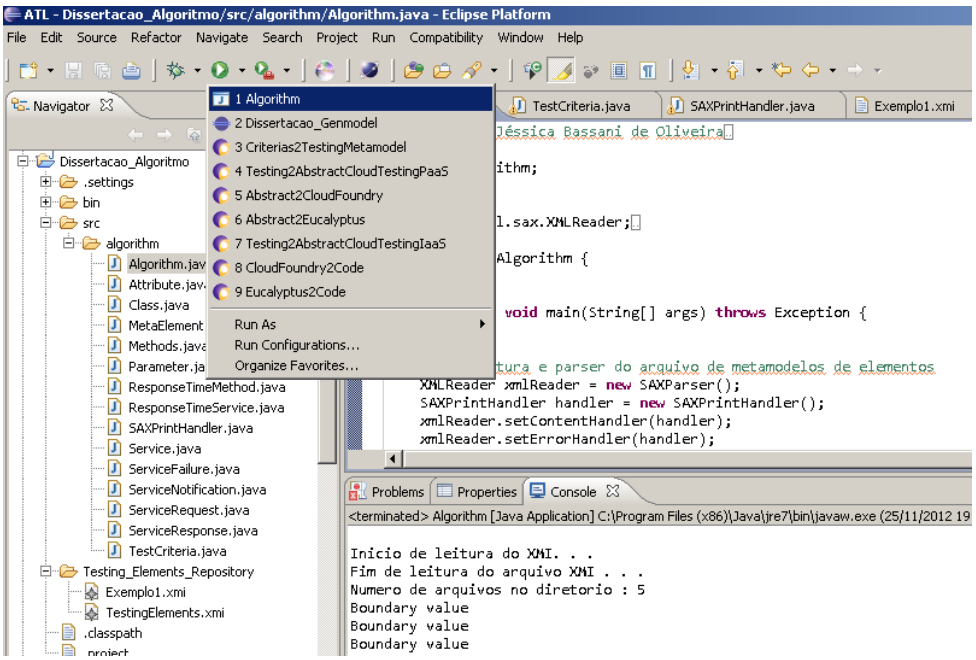

Figura 6.4: Execução do algoritmo de geração de casos de teste

6.5 exibe a lista de casos de teste gerados a partir da execução do algoritmo, bem como um dos casos de teste construído.

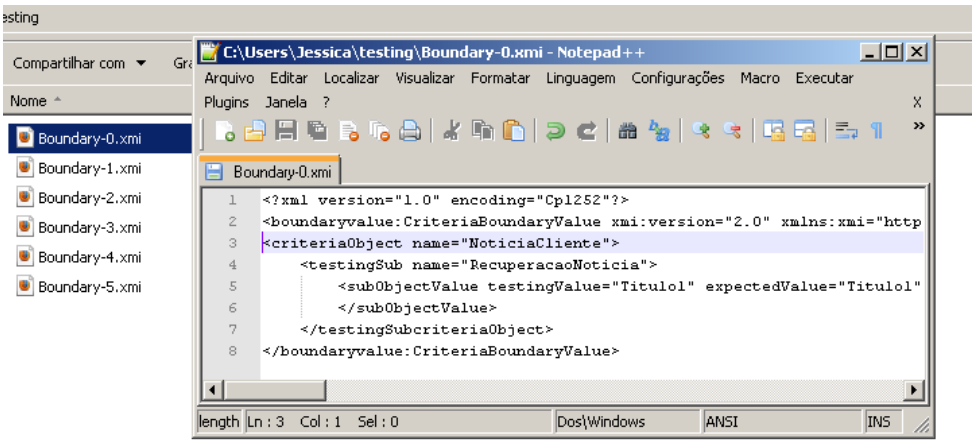

Figura 6.5: Modelos de Critérios de Teste gerados pelo algoritmo

Para demonstrar o restante das funcionalidades do *Framework*, utilizou-se apenas 2 (dois) dos casos de teste gerados pela ferramenta. Essas funcionalidades serão apresentadas nas seções a seguir.

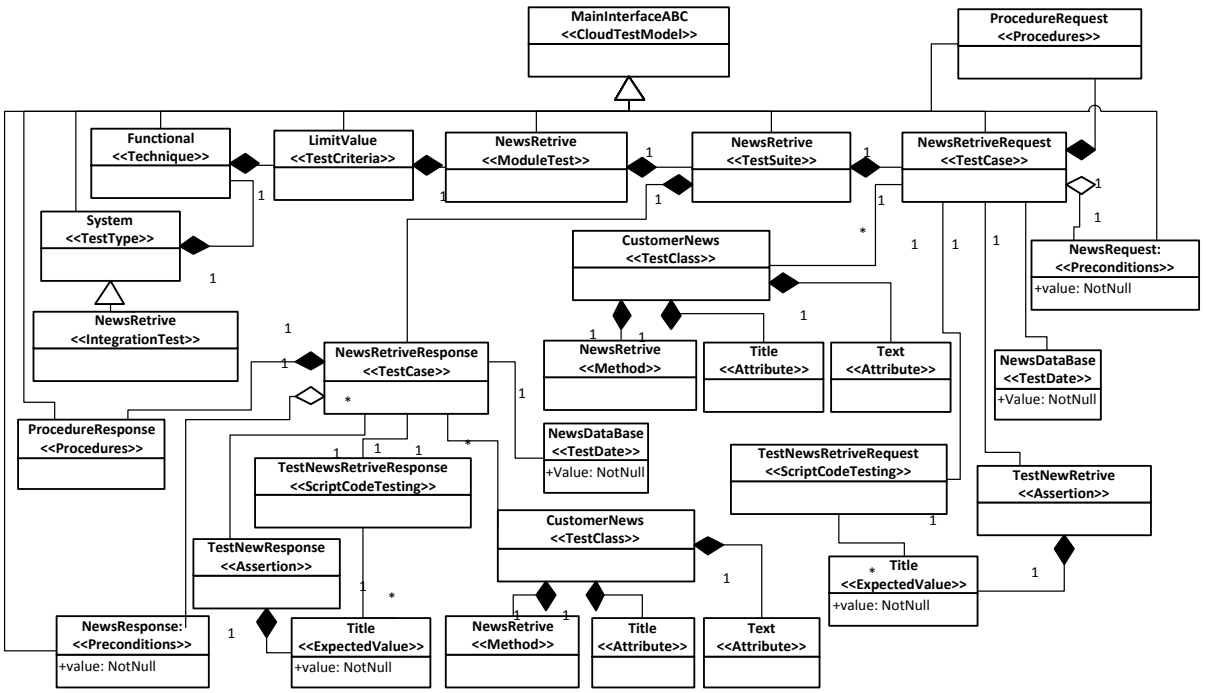

Figura 6.6: Modelo de Teste - PIM

6.2.3 Modelo de Teste Independente de Plataforma (PIM)

O modelo de teste (PIM) foi construído a partir da aplicação do algoritmo proposto neste *framework* (etapa 4, Figura 6.1). O algoritmo realizou a leitura do PIM da aplicação e os casos de teste foram elaborados de acordo com os critérios de testes apropriados para o modelo de negócio. Estes critérios de testes se encontram em forma de metamodelos e estão armazenados num repositório do *framework*. O modelo de negócio contém a lógica do negócio da aplicação e o modelo de teste é específico para a modelagem de testes [81].

Código 6.3: Fragmento de código ATL para geração do Modelo Abstrato da Nuvem

```
1 module testingMetamodel2abstractCloudTestingPaaS;
2 create OUT:abstractCloudTesting from IN:testingMetamodel;
3
4 entrypoint rule Cloud(){
5   to
6      t : abstractCloudTesting!AbstractCloudTestModel (
7        name <- 'CloudFoundry'
8      ),
9
10     t1 : abstractCloudTesting!CloudServices1 (
11       typeService<- 'PaaS'
12     )
```

```
13 }
14
15
16 rule TestType2TestType1{
17       from testtype:testingMetamodel!TestType
18       to testtype1:abstractCloudTesting!TestType1
19       (
20             name<- testtype.name,
21             moduleTestCount<- '1' )
22 }
```

Observa-se na Listagem 6.3, que a nuvem onde será aplicado os testes já é informada. A regra *entrypoint rule Cloud*, instancia no Modelo de Teste de Nuvem Abstrata que os testes serão aplicados para a nuvem *CloudFoundry*. Informações específicas da nuvem *CloudFoundry* serão instanciadas nas transformações de modelos seguintes.

Com a aplicação do algoritmo apresentado na seção 6.2.2, o modelo de teste foi criado de acordo com o metamodelo *CloudTesting* apresentado na Figura 6.6. Neste modelo, dois casos de teste são exibidos para demonstrar as funcionalidades deste *Framework*. O primeiro caso de teste se refere ao método *RequisicaoNoticia* e o segundo caso de teste se refere ao método *RespostaNoticia*.

6.2.4 De PIM para PSM

Com a obtenção do modelo de teste, definições de transformações foram aplicadas para se obter o modelo de testes abstrato para nuvem. Este modelo é descrito conforme o metamodelo *AbstractCloudTesting*, apresentado na Figura 4.14. As definições de transformação foram escritas na linguagem ATL utilizando-se as ferramentas MT4MDE e SAMT4MDE propostas em [39] [40].

A Figura 6.7 exibe o modelo instanciado a partir do metamodelo *AbstractCloudTesting* (etapa 4 do *framework*, Figura 6.1). Neste metamodelo, a nuvem para a qual são gerados os casos de teste está definida, bem como o serviço que ela oferece.

O metamodelo de testes abstrato para nuvem contém as informações relativas aos casos de teste, obtidas do modelo de testes (etapa 5, Figura 6.1), acrescido de informações referente à nuvem. Estas informações referentes à nuvem servem principalmente para a aplicação dos testes de requisitos não funcionais, pois os testes para estes tipos de requisitos serão executados sob a forma de *script*. Logo, este *script* contém

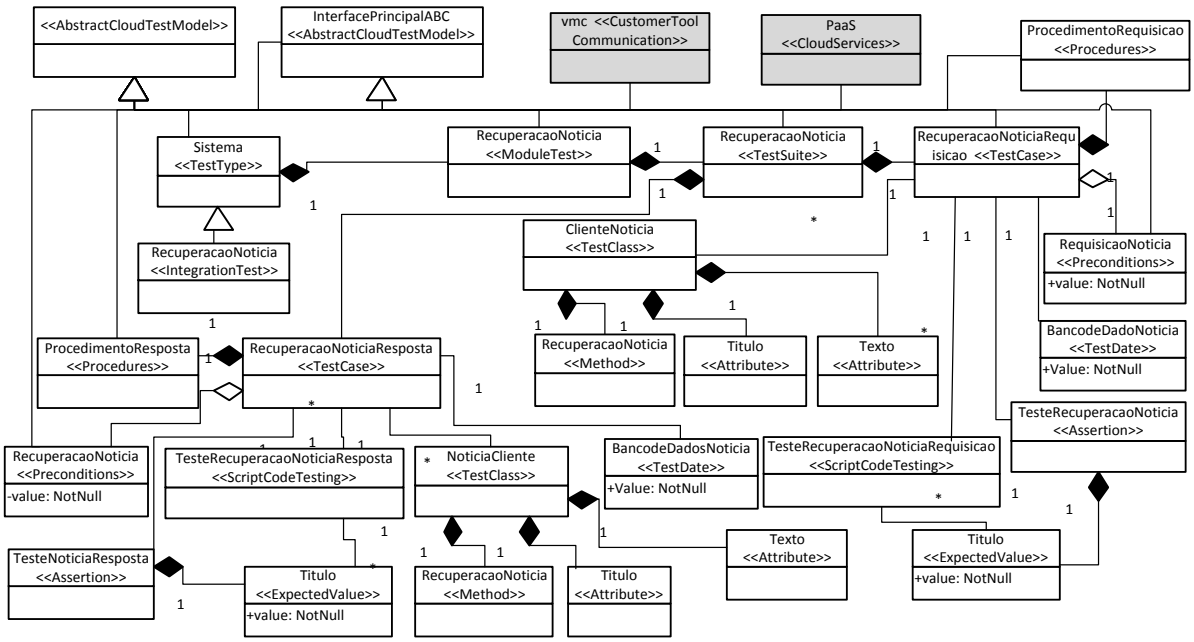

Figura 6.7: Modelo de Teste de Nuvem Abstrata - PSM

informações relativas à conexão com a nuvem e/ou serviço da nuvem para a execução dos testes. Um fragmento das definições de transformação do modelo de teste para o modelo de teste abstrato de uma nuvem podem ser vistas na Listagem 6.3.

6.2.5 De PSM abstrato para PSM concreto

Dentro do fluxo de atividades do *framework*, o modelo de teste abstrato deve ser transformado para um modelo de teste concreto, ou seja, para uma plataforma de nuvem específica (etapa 6 do *framework*, Figura 6.1). E, conforme o nosso exemplo ilustrativo, este modelo de teste abstrato foi transformado para dois modelos concretos: *CloudFoundry* e *Eucalyptus*.

Para a realização desta atividade, transformações entre os modelos foram feitas para a nuvem *Eucalyptus* conforme o metamodelo *Eucalyptus Metamodel* e para a nuvem *CloudFoundry* conforme o metamodelo desenvolvido *CloudFoundry Metamodel*. Um fragmento com definições de transformações para a nuvem *Eucalyptus* pode ser visto na Listagem 6.4.

Código 6.4: Fragmento de código ATL para a geração do PSM abstrato para o PSM *Eucalyptus*

```
1 module abstractcloudtesting2cloudtestingeucalyptus;
2 create OUT:cloudtestingeucalyptus from IN:abstractcloudtesting;
```

```
3 rule ScriptCodeTesting12ScriptCodeTesting{
4       from scriptcodetesting1:abstractcloudtesting!ScriptCodeTesting1
5       to scriptcodetesting:cloudtestingeucalyptus!ScriptCodeTesting
6       (
7             script<- scriptcodetesting1.script,
8             inputValue<- scriptcodetesting1.inputValue,
9             outputValue<- scriptcodetesting1.output,
10            expectedValue<- scriptcodetesting1.expectedValue,
11            scriptTestCase<- scriptcodetesting1.scriptCodeTestCase,
12            scriptCustomer<- scriptcodetesting1.scriptCustomer )
13 }
```

Observa-se neste fragmento de código que os testes serão aplicados em forma de *script* na nuvem *Eucalyptus* através da regra *ScriptCodeTesting12ScriptCodeTesting*.

Um fragmento com definições de transformações para a nuvem *CloudFoundry* pode ser visto na Listagem 6.10.

Código 6.5: Fragmento de código ATL para a geração do PSM abstrato para PSM *CloudFoundry*

```
1 module abstractcloudtesting2cloudtestingcloudfoundry;
2 create OUT:cloudtestingcloudfoundry from IN:abstractcloudtesting;
3 rule CustomerToolCommunication12CustomerToolCommunication{
4       from customertoolcommunication1:abstractcloudtesting!CustomerToolCommunication1
5       to customertoolcommunication:cloudtestingcloudfoundry!CustomerToolCommunication
6       (
7             name<- customertoolcommunication1.name )
8 }
```

Neste fragmento de código, observa-se informações relativas à forma de comunicação com a nuvem *CloudFoundry* são definidas através da regra *CustomerToolCommunication12CustomerToolCommunication*.

A Listagem 6.6 apresenta um fragmento do modelo de teste para plataforma *CloudFoundry* em formato XMI.

Código 6.6: Teste para Plataforma *CloudFoundry* em formato XMI

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <cloudtestingcloudfoundry:CloudTestingFoundry xmi:version="2.0" xmlns:xmi="http://www.omg.↩
      org/XMI" xmlns:cloudtestingcloudfoundry="http://cloudtestingcloudfoundryl/1.0" name="↩
      InterfacePrincipalABC" testSuiteCount="1">
3   <cloudTestSuite name="RecuperacaoNoticia" testCaseCount="1">
```

```
4     <testSuiteTestCase name="TesteRecuperacaoNoticiarequisicao">
5       <testCaseAssertion name="AssertionTesteRecuperacaoNoticia" assertionType="Equals" ↩
            assertionMethod="//@cloudTestSuite.0/@testSuiteTestCase.0/@testCaseTestClass.1/↩
            @testClassMethod.0" order="1" expectedValue="News 1" dataType="string"/>
6       <testCaseAssertion name="AssertionTesteRecuperacaoNoticia2" assertionType="Equals" ↩
            assertionMethod="//@cloudTestSuite.0/@testSuiteTestCase.0/@testCaseTestClass.0/↩
            @testClassMethod.0" order="1" expectedValue="News 2" dataType="String"/>
7       <testCaseTestClass name="RecuperacaoNoticia2">
```

A Figura 6.8 ilustra o fragmento do modelo de teste conforme o metamodelo *CloudFoundry* em forma de árvore, construído com o *Eclipse Modeling Framework* (EMF).

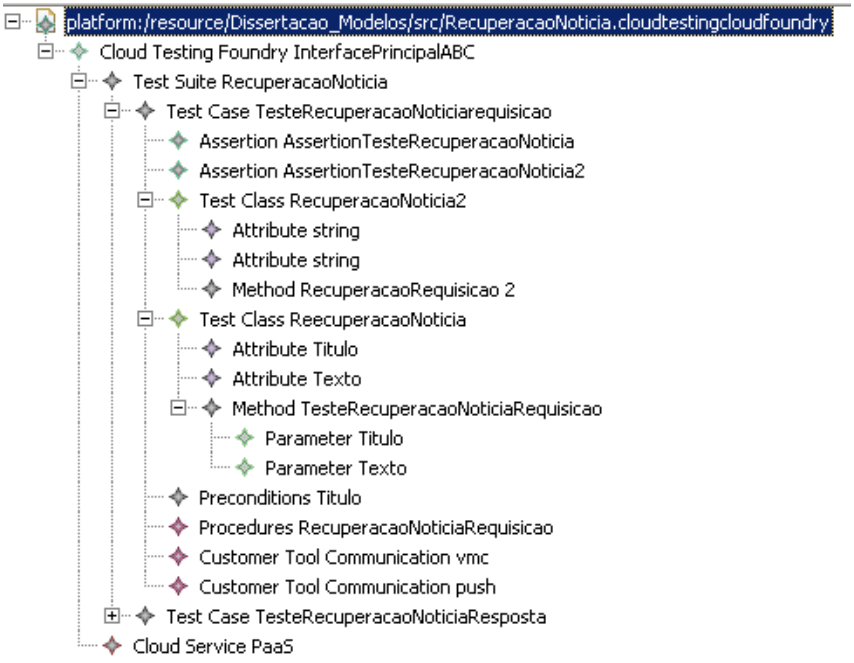

Figura 6.8: Teste para Plataforma *CloudFoundry* em formato de árvore (*genmodel* no EMF)

A Figura 6.9 ilustra o fragmento do modelo de teste conforme o metamodelo *Eucalyptus* em forma de árvore, construído com o *Eclipse Modeling Framework* (EMF).

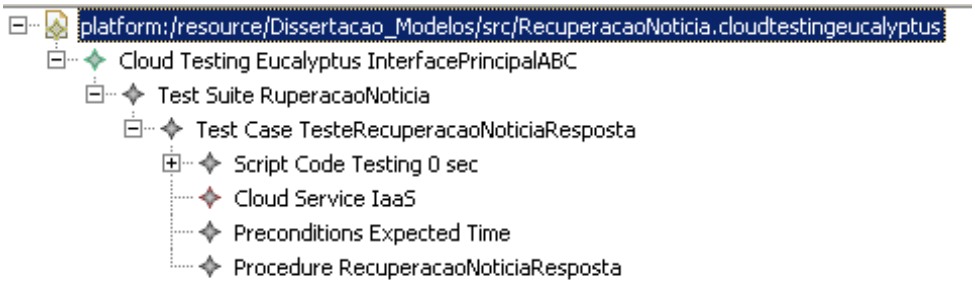

Figura 6.9: Teste para Plataforma *Eucalyptus* em formato de árvore (*genmodel* no EMF)

6.2.6 De PSM para Código

Dando sequência nas transformações existentes no *framework*, códigos de testes em Java são gerados conforme a EBNF da linguagem Java que comumente é usada para especificar a gramática de linguagens formais, incluindo linguagens de programação [1].

Para esta última atividade do exemplo ilustrativo, construiu-se definições de transformação de modelo para código (etapa 7, Figura 6.1). A Listagem 6.7 exibe o trecho de código na linguagem ATL para fazer a transformação de modelos PSM da nuvem *CloudFoundry* para o código de teste.

Código 6.7: Fragmento de código ATL do PSM *CloudFoundry* para a geração do código jUnit

```
1 query CloudFoundry2Code = cloudTestingCloudFoundry!NameElement.allInstances()->
2 select(e | e.oclIsTypeOf(cloudTestingCloudFoundry!CloudTestingFoundry)or e.oclIsTypeOf(↩
      cloudTestingCloudFoundry!TestSuite)  or
3 e.oclIsTypeOf(cloudTestingCloudFoundry!TestCase) or e.oclIsTypeOf(cloudTestingCloudFoundry!↩
      CustomerToolCommunication))->
4 collect(x | x.toString().writeTo('C:/Temp/FuncionalTest/' + x.name + '.java')); -- Query ↩
      Template
5
6 uses library_CloudFoundry2Code;
7
8 ----------------------------------------------------
9
10 library library_CloudFoundry2Code; -- Library Template
11
12 helper context cloudTestingCloudFoundry!Method def: callsMethod(): String =
13 'new '+ self.freeParameter + '().'
14 --+ self.name + '('+
15 if self.methodParameter->isEmpty() then ''
16 else
17 self.methodParameter->iterate( i ; acc:String=''| acc +
18 if acc='' then ''
19 else ','
20 endif + i.name)
21 endif +')';
22
23
24 helper context cloudTestingCloudFoundry!Assertion def: getNameAssertion() : String =
25 if self.assertionType='Equals' then 'assertEquals'
26 else if self.assertionType='True' then 'assertTrue'
27 else if self.assertionType='False' then 'assertFalse'
28 else if self.assertionType='NotNull' then 'assertNotNull'
29 else 'undefinedAssertionType' endif endif endif endif;
```

```
30
31
32 helper context cloudTestingCloudFoundry!Assertion def: generateAssertionType() : String =
33 '\n' + '\t\t'+ self.getNameAssertion() +
34 '(' + self.expectedValue +
35 ', '+ self.assertionMethod.callsMethod() + ');';
```

Observa-se no fragmento de código apresentado, que a regra *cloudTesting-CloudFoundry!Assertion* possui o tipo de condição de testes a ser aplicado, como por exemplo a condição *Equals*, para comparar o valores de teste.

Com a aplicação destas definições de transformação, o código de teste foi gerado para a ser aplicado na nuvem *CloudFoundry* e um arquivo em formato *script Shell* foi gerado para ser aplicado na nuvem *Eucalyptus*.

O código de teste foi escrito conforme a linguagem Java, para a realização de testes do tipo Funcional. Podemos observar que um código de teste escrito em Java para a uma aplicação em nuvem *CloudFoundry* é semelhante a um código de teste para uma aplicação Java local. Devido ao fato que, para o desenvolvimento de uma aplicação em nuvem, seja ela do tipo Iaas, SaaS ou PaaS, a aplicação é desenvolvida localmente e depois submetida a uma nuvem. A nuvem *CloudFoundry* e o ambiente do *Google App Engine*, por exemplo, disponibilizam um *plug-in* para o Eclipse [14] [19], onde o usuário baixa e instala em sua máquina, desenvolve o código localmente, de acordo com a linguagem específica, testa localmente e depois que submete o aplicativo desenvolvido para o ambiente de nuvem. Assim um código de testes *jUnit* para uma aplicação local e para uma aplicação para um ambiente de computação em nuvem tornam-se semelhantes.

A Listagem 6.8 exibe o trecho de código em *jUnit* gerado através das transformações de modelos com o *framework* para a plataforma de nuvem específica da *CloudFoundry*. Este código possui os testes funcionais a serem executados sobre o aplicativo desenvolvido para esta plataforma.

Código 6.8: Código de teste em jUnit

```
1 public class NewsRetrieve extends TestCase {
2  TestSuite suite = new TestSuite("RecuperacaoNoticia");
3  suite.addTestCase(TesteRecuperacaoNoticiaRequisicao.class);
4  suite.addTestCase(TesteRecuperacaoNoticiaResposta.class);
5  return suite;
```

```
6 }
7
8 public class TesteRecuperacaoNoticiaRequisicao extends junit.framework.TestCase{
9  @Test
10  public void AssertionTesteRecuperacaoNoticia(){
11   assertEquals(Noticia 1, new TesterecuperacaoNoticiarequisicao().(Titulo,Texto));
12  }
13
14  @Test
15  public void AssertionTesteRecuperacaoNoticia2(){
16   assertEquals(News 2, new RecuperacaoRequisicao2().(Titulo));
17  }
18 }
19
20 public class TesteRecuperacaoNoticiaResposta extends junit.framework.TestCase{
21  @Test
22  public void AssertionTesteNoticiaResposta(){
23   assertEquals(Noticia 1, new TesteRecuperacaoNoticiaResposta().(Titulo));
24  }
25 }
```

E, finalmente, a Listagem 6.9 exibe informações de um arquivo no formato de *script Shell*, com um teste de Tempo de Resposta para ser aplicado na plataforma de nuvem específica da *Eucalyptus*.

Código 6.9: *Script* de teste para Plataforma *Eucalyptus*

```
1 #!/bin/sh
2
3 #Teste para ser executado em nuvem Eucalyptus com Centos 5.7
4 #Teste de tempo de resposta
5
6 echo "Initial testing"
7
8 #Verifica instancias disponíveis, caso esteja pending
9 #coloca em modo running
10 euca-describe-instances
11 euca-authorize default -P icmp -t -1:-1 -s 0.0.0.0/0
12 ssh -i mykey.private root@<accessible-instance-ip>
13
14 #Chama o programa construído em java para testar tempo de resposta
15 exec_program_responseTime
16
17
18 #Termino de acesso a uma instancia
19 euca-terminate-instance i-XXXXXXX
20 echo "Final testing"
```

Observa-se que no exemplo ilustrativo foram construídos *scripts* de códigos de testes para a aplicação. O exemplo focou-se na parte inferior do *framework*, conforme a Figura 4.1. Os metamodelos da parte superior foram desenvolvidos, como, por exemplo, o *Abstract Cloud Computing Metamodel*. Porém, não foram mostrados neste trabalho por não ser o foco o desenvolvimento de código da aplicação.

6.3 Exemplo 2 - Sistema de Recados

Apresenta-se um segundo exemplo, de forma mais simplificada, que consiste na construção de casos de teste para um sistema de recados, onde se é utilizado dois serviços *Web*, um serviço de inserção de recado e um serviço de busca do recado armazenado.

Assim como no exemplo anterior, os casos de teste gerados foram desenvolvidos para serem utilizados na nuvem *CloudFoundry* (testes funcionais) e na nuvem *Eucalyptus* (testes não funcionais), ou seja, o mesmo PIM (de negócios e de teste) pode ser transformado em modelos para plataformas específicas.

Na Figura 6.10, representada por um diagrama de caso de uso em UML, ilustra-se os casos de uso do sistema que simula a inserção e a listagem de um recado.

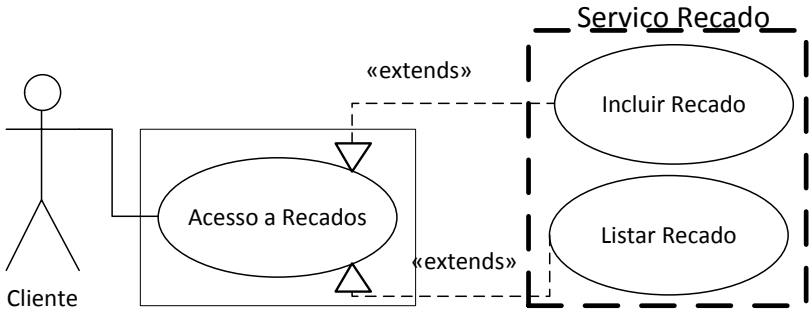

Figura 6.10: Diagrama de Caso de Uso do exemplo ilustrativo do Serviço de Recado

6.3.1 Modelo Independente de Plataforma (PIM)

Do mesmo modo do exemplo anterior, inicialmente é construído um modelo de negócios, conforme um Diagrama de Classes UML para se obter o PIM da aplicação. O diagrama de classes é apresentado na seção seguinte, conforme a Figura 6.11.

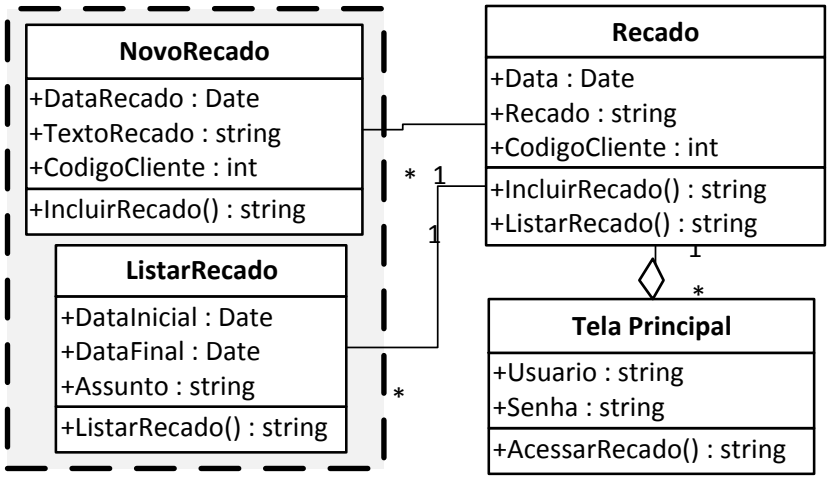

Figura 6.11: PIM em UML para o exemplo ilustrativo Sistema de Recado

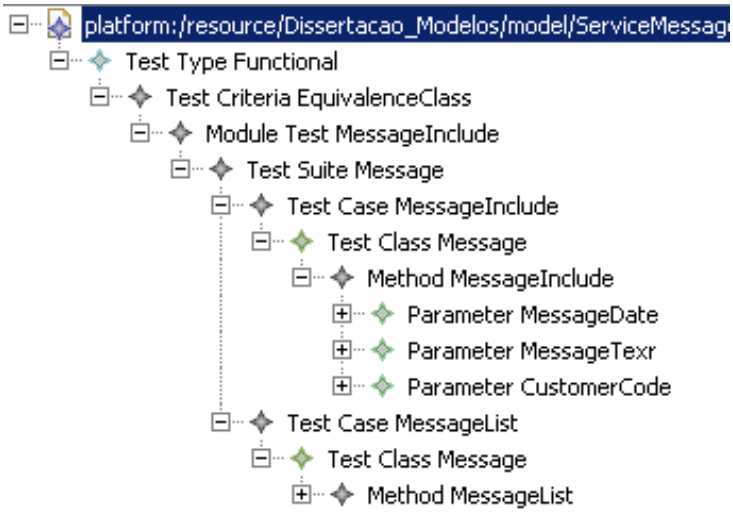

Figura 6.12: Modelo de teste para exemplo de Sistema de Recado - PIM

6.3.2 Modelo de Teste Independente de Plataforma (PIM)

Diferente do exemplo anterior, os casos de teste foram diretamente elaborados, sem o auxílio do algoritmo de geração de casos de teste, ou seja, instanciou-se diretamente o modelo de teste conforme o metamodelo *CloudTesting*. O modelo de teste (PIM) foi criado, conforme apresentado, em forma de árvore, na Figura 6.12. Dois casos de teste são elaborados para demonstração neste exemplo ilustrativo. O primeiro caso de teste se refere ao método *InclusaoRecado* e o segundo caso de teste se refere ao método *ListagemRecado*.

6.3.3 Do modelo PIM para PSM

Seguindo as etapas do *framework*, definições de transformações foram aplicadas para se obter o modelo de testes abstrato para nuvem (etapa 5, Figura 6.1).

6.3.4 Do PSM abstrato para PSM concreto

Nesta etapa, o modelo de teste abstrato é transformado para um modelo de teste concreto (etapa 6, Figura 6.1). Para a realização desta atividade, transformações entre os modelos foram feitas para a nuvem *CloudFoundry* conforme o metamodelo desenvolvido *CloudFoundry Metamodel*. Um fragmento com definições de transformações para a nuvem *CloudFoundry* pode ser visto na Listagem 6.10.

Código 6.10: Fragmento de código ATL para geração do PSM *CloudFoundry*

```
1  rule CustomerToolCommunication12CustomerToolCommunication{
2        from customertoolcommunication1:abstractcloudtesting!CustomerToolCommunication1
3        to customertoolcommunication:cloudtestingcloudfoundry!CustomerToolCommunication
4        (
5              name<- customertoolcommunication1.name  )}
6
7  rule CloudServices12CloudService{
8        from cloudservices1:abstractcloudtesting!CloudServices1
9        to cloudservice:cloudtestingcloudfoundry!CloudService
10       (
11             name<- cloudservices1.name,
12             cloudAPI<- cloudservices1.CloudServicesAPI )}
```

Observa-se na listagem apresentada que informações relativas ao serviço da nuvem são mapeadas, por exemplo, informações da classe *CustomerToolCommunication*, relativo à linguagem para comunicação com a nuvem.

6.3.5 Do modelo PSM para Código

A última etapa consiste na geração dos códigos de testes em Java (etapa 7, Figura 6.1). A Listagem 6.11 exibe o trecho de código em *jUnit*, gerado através das transformações de modelos para a plataforma de nuvem específica da *CloudFoundry*. Este código gerado possui testes a serem aplicados sobre os serviço de *Listagem de Recados.*

Código 6.11: Código de teste jUnit para exemplo 2

```
1 public class MessageList extends junit.framework.TestCase{
2
3  @Test
4  public void AssertionTestMessageList(){
5
6 assertEquals(01-01-2012, new MessageList().(DataInicial));
7 }
8
9 @Test
10  public void AssertionTestMessageList2(){
11
12 assertEquals(Message, new MessageList().(Assunto));}
13
14
15 }
```

6.4 Avaliação dos Resultados

O desenvolvimento de testes utilizando o protótipo do *Framework* para Geração de casos de teste para plataformas de Computação em Nuvem foi feito para demonstrar a veracidade dos metamodelos desenvolvidos, bem como da metodologia proposta. Procurou-se também demonstrar a automatização conseguida pela aplicação do algoritmo para a construção dos Critérios de Teste.

Os itens avaliados entre os trabalhos relacionados (conforme Seção 3.7) são retomados para a comparação e análise com a proposta deste trabalho de dissertação. Estes itens avaliados acrescentam contribuições, além dos objetivos especificados no início deste trabalho. Como ilustra a Tabela 6.1, uma nova coluna (com fonte azul), referente ao trabalho desenvolvido foi inserida para fins de comparação com os trabalhos relacionados.

Dentre os **oito** itens avaliados que se caracterizam por possuir uma forte relação a abordagem proposta, detectou-se que a proposta apresentada neste trabalho atingiu **sete** destes itens, que são:

1. *"Trabalha em nível de infraestrutura"*: O trabalho de testes em nível de infraestrutura neste *framework* foi conseguido por dois motivos. O primeiro foi o desenvolvimento do metamodelo de critério de teste que abrangeu o teste de *stress*. Este

teste se caracteriza por verificar quantas instâncias de uma aplicação podem ser abertas e suportadas pela nuvem, durante um espaço determinado de tempo. O segundo foi o desenvolvimento do metamodelo para a nuvem *Eucalyptus* (nuvem do tipo IaaS). Assim, os testes nesta plataforma foram aplicados em forma de *script*. O diferencial do *framework* neste item pode ser percebido em relação dos tipos de testes a serem aplicados. Os trabalhos que conseguiram atingir este objetivo [93] [6] [68] [17], se mostraram limitados nos tipos de testes, enquanto o *framework* proposto pode ser estendido para diversos tipos de testes a partir da construção dos metamodelos de critérios;

2. *"Trabalha em nível de desenvolvimento"*: O trabalho em nível de desenvolvimento foi conseguido pela criação de metamodelos de critérios de teste para o teste do tipo funcional, e assim, códigos em *jUnit* podem ser gerados pelo *framework*. Desenvolveu-se também um metamodelo específico para a plataforma *CloudFoundry* que constitui em uma nuvem do tipo PaaS. Uma das vantagens do *framework* em relação aos demais trabalhos que atingiram este objetivo, é que o *framework* pode gerar casos de teste para diversas nuvens PaaS através de um único PIM, enquanto os demais trabalhos [90] [68] [71] se limitam a uma plataforma;

3. *"Ambiente Interoperável"*: Através da construção dos metamodelos para nuvem abstrata e os metamodelos para nuvens concretas, conseguiu-se um ambiente interoperável em nível de modelos. O *framework* ainda pode ser estendido através da construção de metamodelos para outras plataformas de nuvem. Somente um trabalho apresentou características de interoperabilidade [93]. Porém, este trabalho relacionado se limita a interoperabilidades entre nuvens IaaS. Pois a proposta deste trabalho relacionado é a construção de um ambiente de testes sobre uma nuvem de infraestrutura. O *framework* proposto consegue atingir um nível de interoperabilidade maior através da manipulação de modelos, atingindo assim outros modelos de serviço de nuvens, tais como PaaS, IaaS, SaaS.

4. *"Geração de código de testes"*: Através das transformações (modelo-à-texto) do modelo de nuvem concreta para código, conseguiu-se a geração de códigos de testes funcionais e não funcionais. Dois trabalhos relacionados conseguiram a geração de casos de teste [90] [71]. Porém, um diferencial do *framework* construído é a geração destes casos de teste a partir da leitura do modelo de negócio da aplicação;

5. *"Técnicas de teste utilizadas"*: O *framework* trabalha com testes funcionais, através de três critérios de teste (Valor Limite, Equivalência de Classes e Tabelas de Decisão) e testes não funcionais, através dos critérios de teste (Estresse e Tempo de Resposta). Os casos de teste foram gerados através do emprego destas técnicas com o auxilio de um algoritmo para automatizar a geração de casos de teste e metamodelos destes critérios. O *framework* tem suporte para ser estendido para demais critérios de teste, garantindo assim que outras técnicas de testes sejam suportadas. Todos os trabalhos, visto que se tratam de teste de *software*, apresentaram um tipo de técnica de teste que foi gerada ou aplicada. Uma das vantagens do *framework* desenvolvido em relação aos demais trabalhos é que ele pode ser estendido para outras técnicas de teste funcionais e não funcionais;

6. *"Modelos de casos de teste"*: Modelos de casos de teste foram conseguidos através da utilização da abordagem de Engenharia Dirigida por Modelos para construção de modelos utilizados neste *framework*. Aplicou-se também uma metodologia para realizar as transformações entre os modelos e obter os casos de teste. Todos os trabalhos relacionados apresentaram algum tipo de modelo, seja modelo de teste ou modelo de aplicação de um teste. Porém, uma das vantagens do *framework* em relação aos trabalhos apresentados, é que os modelos construídos neste *framework* são em formato *.ecore* do EMF. Isso faz com que os modelos sejam serializados, ou seja, em formato XMI. E o XMI constitui num padrão de troca de informações em XML [51];

7. *"Critérios de testes utilizados para composição de casos de teste"*: Critérios de testes foram utilizados através da construção de metamodelos específicos de critérios. Assim, através de um número mínimo de casos de teste, conseguiu-se uma cobertura do cenário a ser testado, de acordo com um determinado tipo de teste. Apenas um trabalho relacionado cita a utilização de critérios para construção de testes [68]. Porém, estes critérios não são critérios das técnicas de testes existentes na literatura. Estes critérios foram definidos no trabalho para tornar o processo de teste eficaz. Enquanto o *framework* desenvolvido trabalha com critérios de testes pertencentes aos testes desenvolvidos e relatados na literatura;

Observa-se através dos itens avaliados, apresentados na Tabela 6.1, que o *framework* desenvolvido nesta dissertação conseguiu reunir diversos itens considera-

dos relevantes para o desenvolvimento de casos de teste para plataformas de Computação em Nuvem. O trabalho relacionado que mais conseguiu reunir características relevantes foi o trabalho "Taxonomia e Racionalização de Requisitos para Teste baseado em Nuvem [68]", mesmo assim, não conseguiu atingir a quantidade de itens contemplados pelo *framework* desenvolvido.

Além dos itens avaliados que tiveram resultados considerados positivos, outros itens também foram observados com o uso do *framework* para geração de casos de teste nos dois exemplos ilustrativos utilizados. Acredita-se em uma redução significativa em relação ao esforço e ao tempo da equipe de testes.

Em relação à redução de esforço, este item é notado pela utilização dos modelos desenvolvidos neste *framework*. Pois, um caso de teste independente de plataforma é transformado em caso de teste para uma plataforma de nuvem específica, neste caso as nuvens *Eucalyptus* e *CloudFoundry*. Percebeu-se a redução de esforço pois, com um único PIM de teste podemos gerar casos de teste para nuvens distintas. Atualmente, o *framework* gera casos de teste somente para as nuvens *Eucalyptus* e *CloudFoundry*. Mas, como relatou-se na Seção 4.1, o *framework* desenvolvido neste trabalho pode ser estendido para outras plataformas de nuvem, através da construção do metamodelo da nuvem específica.

Ainda em relação ao esforço, observou-se nestes exemplos ilustrativos uma redução de possíveis injeções de erros durante o mapeamento entre os metamodelos fonte e alvo. Pois, através das ferramentas MT4MDE e SAMT4MDE [37] [39] possibilitou-se a semiautomatização do processo da construção das definições de transformação (modelo-à-modelo) utilizadas no *framework*.

Quanto ao tempo, observa-se que a aplicação do algoritmo para a criação dos Critérios de Teste agiliza e reduz o tempo da equipe de teste, além de, praticamente cobrir os possíveis erros a serem encontrados na aplicação, em relação aos tipos de testes demonstrados neste *framework* (teste funcional, tempo de resposta e estresse).

No presente trabalho, não avaliou-se o tempo da aplicação de um teste em si, como por exemplo, a execução de um teste de performance diretamente num ambiente de nuvem. O tempo ao qual se é referido, é o tempo de todo o processo de teste (definição de plano de teste, construção dos casos de teste, execução dos casos de teste, etc). Por isso, acredita-se na hipótese de redução de tempo, pois o algoritmo de geração

de casos de teste proposto neste trabalho, consegue construir modelos de casos de teste de forma automática a partir da leitura de um modelo de negócio de uma aplicação.

Ainda em relação à questão da geração dos casos de teste de forma automática a partir da leitura de um modelo de negócio da aplicação, nos trabalhos relacionados apresentados nesta dissertação, não detectou-se algum tipo de tarefa que se assemelhasse à proposta trazida neste trabalho. Também, em formas tradicionais para o processo de desenvolvimento de testes conhecidos na literatura [62], [47] [78], como para o teste de *software* para plataformas de Computação em Nuvem [5] [6] [95], percebeu-se que os casos de teste não são criados de forma automática.

Além disso, existem no mercado diversas ferramentas de automação de teste, como a *SeleniumHQ* [73], *TestComplete* [76], *Silk Test* [8], *jCompanyQA* [63], etc. Porém, pelos estudos realizados sobre o funcionamento destas ferramentas, percebeu-se que a geração automática de casos de teste não é contemplada, ou seja, o usuário necessita realizar o planejamento e criação dos testes a serem aplicados.

Para uma melhor avaliação e uma medição mais precisa dos itens relatados anteriormente, seria interessante a realização de um experimento com usuários para avaliar as funcionalidades disponibilizadas através do *framework*. Também seria viável a realização de um comparativo em relação ao esforço de trabalho empregado e a qualidade de testes gerados. Este comparativo deve ser realizado entre o trabalho gerado pelo *framework* proposto e o trabalho realizado pela equipe de testes.

6.5 Síntese

No capítulo que se encerra, apresentou-se exemplos para demonstrar as funcionalidades do *Framework* de Geração de casos de teste para plataformas de Computação em Nuvem. Assim, comprovou-se a veracidade dos metamodelos e metodologia desenvolvidos neste *framework*.

Dois exemplos ilustrativos foram apresentados. O primeiro exemplo foi exibido de forma mais detalhada e consistia na geração de casos de teste para um serviço de notícias de um sistema *Web*. E o segundo exemplo apresentado de forma simplificada, consistia em um sistema de controle de Recados, também como sistema *Web*.

Apresentou-se também a aplicação do algoritmo para geração dos Critérios de Teste no primeiro exemplo ilustrativo, mostrando as facilidades por ele proporcionadas. Finalmente, uma avaliação sobre os resultados obtidos com a utilização do protótipo foi apresentada, onde retomou-se a tabela de trabalhos relacionados, confrontando o trabalho desenvolvido nesta dissertação com os demais trabalhos, apresentando assim uma comparação que auxiliou na avaliação de resultados.

	IaaS		PaaS		SaaS			Proposta
Itens Avaliados	Teste como Serviço sobre Nuvens [93]	D-Cloud: Design de um ambiente de Teste de Software para Sistemas Distribuídos [6]	Uma abordagem para a Composição de Serviços e testes para Computação em Nuvem [90]	Taxonomia e Racionalização de Requisitos para Teste baseado em Nuvem [68]	Engenharia Multi-Tenant para Software como Serviço [74]	Automação de Testes em plataforma SaaS [71]	Avaliação de Desempenho e Escalabilidade em nuvens SaaS [17]	Uma abordagem baseada em MDE para Suportar o Teste de Software na Plataforma de Computação em Nuvem
Trabalha em nível de Infraestrutura	Sim	Sim	Não	Sim	Não	Não	Sim	Sim
Trabalha em nível de Desenvolvimento	Não	Não	Sim	Sim	Não	Sim	Não	Sim
Trabalha em nível de Aplicação	Sim	Não	Sim	Não	Sim	Sim	Sim	Não
Ambiente interoperável	Sim	Não	Não	-	-	Não	-	Sim
Gera código fonte de teste	Não	Não	Sim	Não	Não	Sim	Não	Sim
Técnicas de testes utilizadas	Escalabilidade	Tolerância a Falhas	Funcional, Continuo e Integração	Funcional, Performance e escalabilidade	Funcional	Regressão, Funcional e Cobertura de Código	Escalabilidade e Performance	Funcional, Tempo de Resposta e Estresse
Modelo para formar Casos de Teste	Ontologias para modelagem de regras de correspondência das tarefas de teste	Cenários de configuração de testes	Ontologias de domínio para organizar e especificar serviços e interfaces	Taxonomia para criação de requisitos de teste	Modelos de variabilidades e Modelos de cenários baseados no raciocínio semântico	Model View Controller (MVC) e sObjects	Modelos Formais e métricas	Uso do EMF para a construção de modelos)
Critérios de testes utilizados para composição de Casos de Teste	Não	Não	Não	Sim	Não	Não	Não	Sim

Tabela 6.1: Comparação e Análise dos Trabalhos relacionados

7 Conclusões e Trabalhos Futuros

Neste capítulo, os objetivos alcançados neste trabalho e suas limitações são apresentados e rediscutidos. Recomendações de trabalhos futuros, bem como as contribuições científicas, tecnológicas e sociais que o presente trabalho proporcionou são expostas ao leitor nas seções a seguir.

7.1 Conclusões do Trabalho

Este trabalho apresentou uma proposta de um *framework* para geração de casos de teste para plataformas de Computação em Nuvem. Uma metodologia foi proposta para entendimento das etapas relacionadas ao *framework*. Metamodelos foram desenvolvidos. Dentre eles, os metamodelos de critérios de teste serviram para suportar a criação de modelos de teste e as definições de transformação de modelos suportaram o processo de criação de casos de teste.

Um metamodelo de teste independente de plataforma foi adaptado para dar suporte a testes funcionais e não funcionais. Um metamodelo para testes em plataformas de Computação em Nuvem Abstrata foi desenvolvido. Outros metamodelos para plataformas de Computação em Nuvem Concreta foram produzidos, tais como, metamodelo para *Eucalyptus* e metamodelo para *CloudFoundry*. E um algoritmo foi proposto para automatizar a criação de casos de teste a partir da construção de critérios de teste. Estes metamodelos e algoritmo constituíram na construção dos casos de teste, no qual foi o foco deste trabalho.

Um metamodelo abstrato para plataformas de computação em nuvem, independente de tecnologia, foi construído. Este metamodelo possui as principais características das nuvens, separadas por tipos de serviço. Ou seja, o metamodelo apresenta os principais serviços das nuvens do tipo: IaaS, PaaS e SaaS. Este metamodelo constituiu a parte da geração de códigos referentes à plataformas de nuvens do *framework*.

Um protótipo foi construído, abrangendo a parte da geração de testes, a fim de prover a construção de casos de teste para plataformas de Computação em Nuvem.

Utilizou-se as ferramentas MT4MDE e SAMT4MDE para auxiliar na construção das definições de transformação entre os modelos utilizados neste *framework*. Finalmente, exemplos ilustrativos foram apresentados para demonstrar a utilização dos metamodelos e do algoritmo desenvolvido.

7.2 Objetivos Alcançados

Considerando a proposta do *framework*, juntamente com a metodologia, metamodelos, e o protótipo apresentado e a utilização de MT4MDE e SAMT4MDE, e a apresentação dos exemplos ilustrativos, acredita-se que os seguintes resultados foram alcançados:

- Redução no tempo para testes: Dado um modelo de negócio da aplicação, o *framework* proposto permite criar o modelo de teste a partir da construção do modelo de elementos de teste e aplicar as definições de transformação já implementadas neste *framework*. Neste processo, observa-se a intervenção humana somente na instanciação do modelo de elementos. A proposta do *framework* forneceu um suporte ao teste de *software* que reduz o trabalho do desenvolvedor e o auxilia a evitar erros devido a fadiga;

- Redução no tempo da entrega de um aplicativo: Através da redução de tempo gasto pelas equipes de teste através da utilização do *framework* e seu protótipo, os aplicativos tendem a ser entregues em um espaço de tempo menor;

- Diminuição da intervenções humanas no código-fonte: Como o campo de abrangência dos testes se torna amplo, devido aos códigos de testes serem gerados de uma forma mais adequada, juntamente com o auxílio do algoritmo desenvolvido no *framework*, então, a qualidade de *software* é assegurada, e a intervenção humana é menos utilizada para descobrir *bugs* e retirar defeitos de uma aplicação;

- Melhoria na qualidade do *software*: Como os códigos de teste são gerados automaticamente, o tempo na construção de casos de teste reduz. Assim, o teste pode ser mais amplo, pois a demanda de tempo é menor para a construção e aplicação de casos de teste, consequentemente, garantindo melhor qualidade no *software* desenvolvido.

7.3 Limitações

Embora os objetivos tenham sido atingidos, há limitações que são apresentadas como a seguir:

- O *framework* não consegue realizar automaticamente a leitura de um arquivo WSDL e construir modelos de negócios da aplicação, ainda é necessário realizar estas especificações de forma manual;
- O *framework* não realiza a leitura de um modelo de negócios em forma de arquivo XMI, ou seja, a leitura de um diagrama de classes UML ou outro tipo de diagrama que representasse o modelo de negócios. Ainda, é necessário a instanciação direta do modelo de Elementos de Teste;
- As definições de transformação são executadas somente com o motor de transformação em ATL, faltando a possibilidade de utilização de outros motores de transformação tais como, MOFScript [50], MOF QVT [54], etc;
- Ausência de um teste experimental com usuários. Por exemplo, disponibilizar o *framework* para uma equipe de testes, onde seria realizado um comparativo entre a utilização da ferramenta para construção dos casos de teste e a realização dos casos de teste da forma tradicional utilizada pela equipe;
- A aplicabilidade do *framework* foi apenas verificada para modelos de negócios simples. Um estudo de caso mais complexo permitiria explorar melhor o *framework* e verificar outras limitações.

7.4 Contribuições

Nesta conclusão, dividiu-se as contribuições em categorias designadas como contribuições científicas, tecnológicas e sociais.

7.4.1 Científicas

As contribuições científicas são as seguintes:

- A proposta de uma abordagem, baseada em MDE, para geração de casos de teste para plataformas de Computação em Nuvem;
- Proposta de um *framework*, baseado em MDE, com auxílio das ferramentas MT4MDE e SAMT4MDE, com o objetivo de gerar códigos de testes aplicáveis a diferentes plataformas de nuvem;
- Proposta de um algoritmo para auxiliar na construção dos casos de teste, baseado na leitura do modelo de negócios e na construção de critérios de teste;
- Estudo e avaliação da aplicabilidade da Engenharia Dirigida por Modelos para testar aplicações construídas para plataformas de Computação em Nuvem;
- Publicação de artigo *Model Driven Testing for Cloud Computing* que foi aceito na conferência CISSE 2012 e que será publicado em 2013 pela *Springer Verlag* em "*Lecture Notes in Electrical Engineering - Innovations and Advances in Computer, Information, Systems Sciences, and Engineering*", ISSN: 1876-1100;
- A submissão do artigo "*An Approach based in Model Driven Engineering to Support Testing in Cloud Computing*"em um jornal científico.

7.4.2 Tecnológicas

Dentre as contribuições tecnológicas, pode-se citar:

- Desenvolvimento do *framework* proposto como *plug-in* para Eclipse, incluindo (EMF), para construção das definições de transformação entre os modelos utilizados na proposta do *framework* e também para implementação do algoritmo proposto na abordagem para geração de casos de teste;
- Criação dos metamodelos *TestingElements*, *TestingModel*, *AbstractTestingModel*, *CloudFoundry*, *Eucalyptus* e metamodelos de critérios de teste (*BoundaryValue*, *EquivalenceClass*, *DecisionTable*, *Stress* e *TimeResponse*) e geração dos seus respectivos modelos de correspondência;

7.4.3 Sociais

Dentre as contribuições sociais, pode-se citar:

- A formação de recursos humanos, incluindo alunos de graduação, em Engenharia de *Software*, especificamente na área de Engenharia Dirigida por Modelos;
- Disseminação do conhecimento na área de Teste de *Software* em Computação em Nuvem.

7.5 Trabalhos Futuros

Os seguintes trabalhos futuros podem ser citados como propostas:

- Aperfeiçoar o *plug-in* para Eclipse, com protótipo do *framework*, visando a importação de modelos PIM (lógica do negócio) a partir de documentos XMI, e a geração automática do modelo PIM a partir da reutilização do documento WSDL;
- Construção de *wizards*, visando facilitar a utilização do *framework*, sem precisar recorrer das telas de configuração do *workspace*;
- Extensão do *framework* pra abranger outros tipos de testes através do desenvolvimento de metamodelos de Critérios de Testes não abordados neste trabalho;
- Implementação da parte do algoritmo que se refere à leitura de um modelo de negócio em formato XMI, para a instanciação do modelo de Elementos de Teste;
- Desenvolvimento de um metamodelo para a lógica e combinação de elementos utilizada para construção dos Critérios de Testes. Com a criação deste metamodelo, a lógica aplicada pode ser retirada do código em Java, facilitando assim a extensão da ferramenta;
- Estender o metamodelo de Elementos de Teste para dar suporte à leitura de outros diagramas como diagramas de sequência e diagrama de atividades;
- Desenvolvimento de metamodelos para outras plataformas de Computação em Nuvem. Atualmente, o *framework* gera casos de teste apenas para as nuvens *CloudFoundry* e *Eucalyptus*;
- Realização de testes experimentais do *framework* com uma equipe de testes para uma melhor avaliação e sugestão de melhorias.

Referências Bibliográficas

[1] Incorporating language processing into java applications: a javacc tutorial. *Software, IEEE 21*, 4 (july-aug. 2004), 70 –77.

[2] ALMEIDA, J., DIJKMAN, R., VAN SINDEREN, M., AND PIRES, L. On the notion of abstract platform in mda development. In *Enterprise Distributed Object Computing Conference, 2004. EDOC 2004. Proceedings. Eighth IEEE International* (sept. 2004), pp. 253 – 263.

[3] APACHE. Apache JMeter. Disponível em `http://jmeter.apache.org/`, Acesso em 18-04-2012.

[4] AZURE, W. Windows Azure. Disponível em `http://www.windowsazure.com/pt-br/`, Acesso em 25-02-2012.

[5] BAI, X., LI, M., CHEN, B., TSAI, W.-T., AND GAO, J. Cloud testing tools. 1 –12.

[6] BANZAI, T., KOIZUMI, H., KANBAYASHI, R., IMADA, T., HANAWA, T., AND SATO, M. D-cloud: Design of a software testing environment for reliable distributed systems using cloud computing technology. 631 –636.

[7] BEZERRA, E. D. C. Composição dinâmica de serviços web semânticos utilizando abordagens da engenharia dirigida por modelos. Master's thesis, Universidade Federal do Maranhão, 2011.

[8] BORLAND. Silk Test. Disponível em `http://www.borland.com/products/silktest/`, Acesso em 02-08-2012.

[9] DELAMARO, M. E., MALDONADO, J. C., AND JINO, M. *Introdução ao teste de software*. 2007.

[10] DOM, A. API DOM - Package org.w3c.dom. Disponível em: `http://docs.oracle.com/javase/1.4.2/docs/api/org/w3c/dom/package-summary.html` Acesso em: 20/07/2012.

[11] E2C. Amazon Elastic Compute Cloud (Amazon EC2). Disponível em http://aws.amazon.com/pt/ec2/, Acesso em 20-02-2012.

[12] ECLIPSE TOOLS PROJECT. *Eclipse Modeling Framework (EMF) version 2.0*, June 2004. Available at http://www.eclipse.org/emf.

[13] FOUNDRY, C. Cloud Foundry. Disponível em http://www.cloudfoundry.com/, Acesso em 10-03-2012.

[14] FOUNDRY, C. Cloud Foundry. Installing the Cloud Foundry Integration Extension for Eclipse or STS. Disponível em http://docs.cloudfoundry.com/tools/STS/configuring-STS.html, Acesso em 29-04-2012.

[15] FOUNDRY, C. *Cloud Foundry Documentation Get Started*, 04 2012. Avaliable at http://docs.cloudfoundry.com/getting-started.html.

[16] FRANKEL, D. S. *Model Driven Architecture: Applying MDA to Enterprise Computing*. Wiley and OMG Press, 2003.

[17] GAO, J., PATTABHIRAMAN, P., BAI, X., AND TSAI, W. Saas performance and scalability evaluation in clouds. 61 –71.

[18] GOOGLE. Google App Engine. Disponível em http://appengine.google.com/start, Acesso em 25-02-2012.

[19] GOOGLE. Google Developers. Google Plugin for Eclipse. Disponível em https://developers.google.com/eclipse/docs/download, Acesso em 29-04-2012.

[20] GOOGLEAPPS. Google Apps. Disponível em http://www.google.com/a/help/intl/pt-BR/users/privacy.html, Acesso em 29-04-2012.

[21] GROSE, T. J., DONEY, G. C., AND BRODSKY, S. A. *Mastering XMI: Java Programming with XMI, XML, and UML*. Publisher: John Wiley, 2002.

[22] GROUP, A., LINA, AND INRIA. *ATL: Atlas Transformation Language - ATL User Manual version 0.7*, February 2006.

[23] GUICE, G. Google Guice. Disponível em http://code.google.com/p/google-guice/, Acesso em 30-03-2012.

[24] GUIDE, E. U. Euca2ools User Guide. Disponível em `http://open.eucalyptus.com/wiki/Euca2oolsGuide`, Acesso em 27-03-2012.

[25] HALPERT, B. *Auditing Cloud Computing - A Security and Privacy Guide*. Wiley Corporate, 2011.

[26] HAN, J., AND M.KAMBER. *"Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, 2000.

[27] IBM HAIFA RESEARCH LABORATORY. *Model Driven Testing Tools - Overview*, September 2003.

[28] INÇKI, K., ARI, I., AND SOZER, H. A survey of software testing in the cloud. In *Software Security and Reliability Companion (SERE-C), 2012 IEEE Sixth International Conference on* (june 2012), pp. 18 –23.

[29] JAVED, A., STROOPER, P., AND WATSON, G. Automated generation of test cases using model-driven architecture. 3.

[30] JERRY GAO, XIAOYING BAI, W.-T. T. Cloud testing- issues, challenges, needs and practice. *Software Engineering : An International Journal (SEIJ) 01* (2011).

[31] JOSÉ DE SOUSA JR, DENIVALDO LOPES, D. B. C. Z. A. Step forward in semi-automatic metamodel matching: Algorithms and tool. *11th International Conference on Enterprise Information Systems LNBIP* (2009).

[32] JOSUTTIS, N. M. *SOA in pratice*. Alta Books, 2008.

[33] JUN, W., AND MENG, F. Software testing based on cloud computing. 176 –178.

[34] KLEPPE, A., WARMER, J., AND BAST, W. *MDA Explained: The Model Driven Architecture: Practice and Promise*, 1st ed. Addison-Wesley, August 2003.

[35] LAMANCHA, B., USAOLA, M., AND DE GUZMAN, I. Model-driven testing in software product lines. 511 –514.

[36] LIMA, B., DE SOUSA, J. G. J., AND LOPES, D. Using mda to support hypermedia document sharing. In *Proceedings of the International Conference on Software Engineering Advances* (Washington, DC, USA, 2007), ICSEA '07, IEEE Computer Society, pp. 65–.

[37] LOPES, D., HAMMOUDI, S., AND ABDELOUAHAB, Z. Schema Matching in the Context of Model Driven Engineering: From Theory to Practice. *Proceedings of the International Conference on Systems, Computing Sciences and Software Engineering (SCSS 2005)* (December 2005).

[38] LOPES, D., HAMMOUDI, S., BÉZIVIN, J., AND JOUAULT, F. Generating Transformation Definition from Mapping Specification: Application to Web Service Platform. *The 17th Conference on Advanced Information Systems Engineering (CAiSE'05)*, LNCS 3520 (June 2005), 309–325.

[39] LOPES, D., HAMMOUDI, S., BÉZIVIN, J., AND JOUAULT, F. Mapping Specification in MDA: From Theory to Practice. *First International Conference INTEROP-ESA'2005 Interoperability of Enterprise Software and Applications* (February 2005).

[40] LOPES, D., HAMMOUDI, S., JR., J. G. S., AND BONTEMPO, A. P. Metamodel Matching: experiments and comparison. *In IEEE International Conference on Software Engineering Advances (ICSEA 06)* (2006).

[41] LOPES, D. C. P. *Introdução a Engenharia Dirigida por Modelos.* I Escola Regional de Computação Ceará Maranhão Piauí (ERCEMAPI), 2007.

[42] LV, C., LI, Q., LEI, Z., PENG, J., ZHANG, W., AND WANG, T. Paas: A revolution for information technology platforms. 346 –349.

[43] MACHADO, P., RAMALHO, F., LIMS, H., AND ALVES, E. Integrando desenvolvimento e testes dirigidos por modelos.

[44] MATHEW, R., AND SPRAETZ, R. Test automation on a saas platform. 317 –325.

[45] MELLOR, S. J., SCOTT, K., UHL, A., AND WEISE, D. *MDA Distilled: Principles of Model-Driven Architecture*, 1st ed. Addison-Wesley, March 2004.

[46] MIRCEA, M. Soa, bpm and cloud computing: Connected for innovation in higher education. 456 –460.

[47] MOLINARI, L. *Functional Testing of Software*. Visual Books, 2008.

[48] NURMI, D., WOLSKI, R., GRZEGORCZYK, C., OBERTELLI, G., SOMAN, S., YOUSEFF, L., AND ZAGORODNOV, D. The eucalyptus open-source cloud-computing system. 124 –131.

[49] OBJECT MANAGEMENT GROUP. *MDA guide version 1.0.1n*. Object Management Group, June 2003. http://www.omg.org/cgi-bin/doc?omg/03-06-01.

[50] OLDEVIK, J. *MOFScript User Guide. Version 0.6 (MOFScript v 1.1.11),*, 2nd ed. Eclipse Project, http://www.eclipse.org/gmt/mofscript/doc/MOFScript-User-Guide.pdf, November 2006. http://www.eclipse.org/gmt/mofscript/doc/MOFScript-User-Guide.pdf.

[51] OMG. Object Management Group. Disponível em `http://www.omg.org/`, Acesso em 17-05-2012.

[52] OMG. *Model Driven Architecture (MDA)- document number ormsc/2001-07-01*, July 2001.

[53] OMG. *MDA Guide Version 1.0.1, Document Number: omg/2003-06-01*. OMG, June 2003.

[54] OMG. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification - Final Adopted Specification*, July 2007. Available at http://www.omg.org/docs/ptc/07-07-07.pdf.

[55] OMG. *Meta Object Facility (MOF) specification - version 2.4.1, formal/08-07-11*, August 2011.

[56] OPENSHIFT. OpenShift. Disponível em `https://openshift.redhat.com/app/`, Acesso em 29-04-2012.

[57] ORACLE. API SAX - Package org.xml.sax. Disponível em: `http://docs.oracle.com/javase/1.4.2/docs/api/org/xml/sax/package-summary.html` Acesso em: 20/07/2012.

[58] ORACLE. Class ArrayList. Disponível em: `http://docs.oracle.com/javase/6/docs/api/java/util/ArrayList.html` Acesso em: 30/07/2012.

[59] ORACLE. Java Language and Virtual Machine Specifications. Disponível em: `http://docs.oracle.com/javase/specs/` Acesso em: 12/04/2012.

[60] ORACLEONDEMAND. Oracle Cloud Services. Disponível em http://www.oracle.com/br/products/ondemand/index.html, Acesso em 01-05-2012.

[61] PEZZÈ, M., AND YOUNG, M. *Software Analisys and Testing*. Bookman, 2008.

[62] PFLEEGER, S. L., AND ATLEE, J. *Software Engineering: Theory and Practice*, 3rd ed. Prentice-Hall, 2005.

[63] POWERLOGIC. jCompany QA. Disponível em http://www.powerlogic.com.br/powerlogic/ecp/comunidade.do?app=portal&pg=541&tax=0, Acesso em 02-08-2012.

[64] QEMU. *QEMU, Open Source Processor Emulator*, December 2011. Available at http://www.qemu.org/.

[65] RACKSPACE. Rackspace Hosting. Disponível em http://www.rackspace.com/cloud/, Acesso em 22-02-2012.

[66] REESE, G. *Cloud Application Architectures - Building Applications and Infrastructure in the Cloud*. O'Reilly Media, 2009.

[67] RINGS, T., GRABOWSKI, J., AND SCHULZ, S. On the standardization of a testing framework for application deployment on grid and cloud infrastructures. 99 –107.

[68] ROBINSON, P., AND RAGUSA, C. Taxonomy and requirements rationalization for infrastructure in cloud-based software testing. 454 –461.

[69] RUBYDOC. Test::Unit. Disponível em http://www.ruby-doc.org/stdlib-1.9.3/libdoc/test/unit/rdoc/Test/Unit.html, Acesso em 18-04-2012.

[70] SALESFORCE. SalesForce. Disponível em http://www.salesforce.com/br/?ir=1, Acesso em 01-04-2012.

[71] SALESFORCE.COM. Force.com apex code developer's guide - version 25.0, June 2012. Disponível em http://www.salesforce.com/us/developer/docs/apexcode/index.htm, Acesso em 22-06-2012.

[72] SCHMIDT, D. C. *Model-Driven Engineering*. IEEE Computer, February 2006.

[73] SELENIUM. SeleniumHQ. Disponível em http://seleniumhq.org/, Acesso em 02-08-2012.

[74] SENGUPTA, B., AND ROYCHOUDHURY, A. Engineering multi-tenant software-as-a-service systems. 15–21.

[75] SILVA, F. P. D. Recuperação automática da modelagem comportamental com aplicações ao ensaio baseado em modelos. Master's thesis, Universidade Nova de Lisboa, 2008.

[76] SMARTBEAR. TestComplete. Disponível em http://smartbear.com/products/qa-tools/automated-testing-tools, Acesso em 02-08-2012.

[77] SOFTWARE, O. C. Openstack Cloud Software. Disponível em http://openstack.org/, Acesso em 25-02-2012.

[78] SOMMERVILLE, I. *Software Engineering*, 8st ed. Pearson, 2007.

[79] SOSINSKY, B. *Cloud Computing Bible*. Wiley Publishing, 2011.

[80] SOURCE, S. Spring Source. Disponível em http://www.springsource.org/, Acesso em 30-03-2012.

[81] SOUSA, H., LOPES, D., ABDELOUAHAB, Z., HAMMOUDI, S., AND CLARO, D. B. Building test cases through model driven engineering. *International Joint Conferences on Computer, Information, and Systems Sciences, and Engineering* (2008).

[82] SOUZA, V. Channel9. O que é computação na nuvem?. Disponível em http://channel9.msdn.com/posts/O-que-e-computao-na-nuvem, Acesso em 02-03-2012.

[83] STAX, A. API StAX. Disponível em: http://docs.oracle.com/cd/E17802_01/webservices/webservices/docs/1.6/tutorial/doc/SJSXP3.html Acesso em: 25/07/2012.

[84] STEINBERG, D., BUDINSKY, F., PATERNOSTRO, M., AND MERKS, E. *EMF: Eclipse Modeling Framework*, 2rd ed. Addison-Wesley Professional, 2008.

[85] SYSTEM, E. Open Eucalyptus. Disponível em http://open.eucalyptus.com/, Acesso em 22-02-2012.

[86] SYSTEMS, E. *Eucalyptus Usage Guide 3.0.2*. Eucalyptus Systems, 05 2012.

[87] TEC MUNDO. Disponível em: `http://www.tecmundo.com.br/programacao/1807-o-que-e-api-.htm`, Acesso em: 21/06/2012.

[88] THE COMMAND-LINE INTERFACE, V. I. VMC. Disponível em `http://docs.cloudfoundry.com/tools/vmc/installing-vmc.html`, Acesso em 27-03-2012.

[89] TSAI, W. T., AND ET AL. Optimal service replication strategies for mapreduce on the cloud. *The International Symposium on Autonomous Decentralized System (ISADS),* (2011).

[90] TSAI, W.-T., ZHONG, P., BALASOORIYA, J., CHEN, Y., BAI, X., AND ELSTON, J. An approach for service composition and testing for cloud computing. 631 –636.

[91] VEENENDAAL, E. V. *GLOSSÁRIO PADRÃO DE TERMOS UTILIZADOS EM TESTE DE SOFTWARE*, versão 2.1br ed. BSTQB, 2010.

[92] VELTE, A. T., VELTE, T. J., AND ESLENPETER, R. *Computação em Nuvem. Uma abordagem prática*. IEEE Computer, 2010.

[93] YU, L., TSAI, W.-T., CHEN, X., LIU, L., ZHAO, Y., TANG, L., AND ZHAO, W. Testing as a service over cloud. 181 –188.

[94] ZENDAGUI, B. A model-driven engineering approach for the observation needs specification. 67 –69.

[95] ZHANG, L., CHEN, Y., TANG, F., AND AO, X. Design and implementation of cloud-based performance testing system for web services. In *Communications and Networking in China (CHINACOM), 2011 6th International ICST Conference on* (aug. 2011), pp. 875 –880.

8 Anexos

8.1 Anexo A - Regras de Transformação de Critérios para *TestingMetamodel*

```
1 -- @path MM=/Dissertacao/metamodel/TestingCriteria/boundaryValue.ecore
2 -- @path MM5=/Dissertacao/metamodel/testingMetamodel.ecore
3 -- @path MM4=/Dissertacao/metamodel/TestingCriteria/stress.ecorediag
4 -- @path MM3=/Dissertacao/metamodel/TestingCriteria/responseTime.ecorediag
5 -- @path MM2=/Dissertacao/metamodel/TestingCriteria/equivalenceClass.ecorediag
6 -- @path MM1=/Dissertacao/metamodel/TestingCriteria/decisionTable.ecorediag
7 module Criterias2TestingMetamodel;
8 create OUT : MM5 from IN : MM, IN1 : MM1, IN2 : MM2, IN3 : MM3, IN4 : MM4;
9
10 rule CriteriaBoundaryValue{
11   from MM:MM!CriteriaBoundaryValue
12   to t1:MM5!TestType (
13         name<- 'Boundary Value  ' + MM.name  ),
14    t2:MM5!TestCriteria (
15         name<- 'Functional'  ),
16    t3:MM5!ModuleTest (
17         name<- MM.name,
18  testSuiteCount<- '1' ),
19 t4:MM5!TestSuite (
20         name<- MM.name,
21  testCaseCount<- '1' ),
22 t5:MM5!TestCase (
23         name<- MM.typeBoundary,
24  testDateCount<- '1' )
25 }
26
27 rule CriteriaBoundaryValue2TestClass{
28       from MM:MM!ObjectTesting
29       to MM5:MM5!TestClass (
30        name<- MM.name )
31 }
32
33 rule CriteriaDecisionTable{
34       from MM1:MM1!CriteriaDecisionTable
35       to t:MM5!TestType (
36        name<- 'Decision Table  ' + MM1.name  ),
37 t1:MM5!TestCriteria (
```

```
38         name<- 'Non Functional'  ),
39 t2:MM5!ModuleTest (
40         name<- 'Decision Table Test',
41  testSuiteCount<- '1' ),
42 t3:MM5!TestSuite (
43         name<- 'Application Test',
44  testCaseCount<- '1' ),
45 t4:MM5!TestCase (
46         name<- 'Application Test',
47  testDateCount<- '1' )
48 }
49
50 rule ValueCondition2ScriptCodeTesting{
51       from MM1:MM1!ValueCondition
52       to MM5:MM5!ScriptCodeTesting (
53         inputValueInstances<- MM1.expectedValue,
54          script<- '#!/bin/sh echo "Testing decision table" #Captura data e hora do sistema ↩
                qdo iniciou o teste date>initialdate.txt #Chama o numero de instâncias a serem ↩
                executadas exec_programa1 #Captura data e hora do sistema qdo finalizou o teste↩
                 date>finaldate.txt echo "End testing stress"',)}
55
56 rule ValueAction2ScriptCodeTesting{
57       from MM1:MM1!ValueAction
58       to MM5:MM5!ScriptCodeTesting  (
59       inputValueApplications<- MM1.expectedValue )
60 }
61
62 rule CriteriaEquivalenceClass2TestType{
63       from MM2:MM2!CriteriaEquivalenceClass
64       to t1:MM5!TestType (
65        name<- 'Equivalence Class  ' + MM2.name   ),
66 t2:MM5!TestCriteria  (
67        name<- 'Functional' ),
68 t3:MM5!ModuleTest  (
69         name<- MM2.name,
70  testSuiteCount<- '1'),
71 t4:MM5!TestSuite  (
72        name<- MM2.name,
73  testCaseCount<- '1'),
74 t5:MM5!TestCase
75       (
76         name<- MM2.typeEquivalence,
77  testDateCount<- '1' )
78 }
79
80 rule CriteriaEquivalenceClass2TestClass{
81       from MM2:MM2!ObjectTesting
82       to MM5:MM5!TestClass
83       (
84       name<- MM2.name )
85 }
```

```
86
87 rule CriteriaResponseTime2TestType{
88       from MM3:MM3!CriteriaResponseTime
89       to t:MM5!TestType
90       (
91        name<- 'Response Time  ' + MM3.name  ),
92 t1:MM5!TestCriteria
93       (
94        name<- 'Response Time' ),
95 t2:MM5!ModuleTest
96       (
97         name<- MM3.name,
98  testSuiteCount<- '1' ),
99 t3:MM5!TestSuite
100      (
101        name<- MM3.name,
102 testCaseCount<- '1' ),
103 t4:MM5!TestCase
104      (
105        name<- MM3.name,
106 testDateCount<- '1' )
107 }
108
109 rule Method2TestClass{
110 from MM3:MM3!Method
111 to MM5:MM5!TestClass
112      (
113 name<- MM3.name )
114 }
115
116 rule TimeValue2ScriptCodeTesting{
117       from MM3:MM3!TimeValue
118       to MM5:MM5!ScriptCodeTesting
119       (
120         expectedValue<- MM3.expectedTime,
121  script<- '#!/bin/sh #Teste de tempo de resposta echo "Initial testing" #Chama o programa ↩
       construído em java para testar tempo de resposta exec_program_responseTime echo "Final ↩
       testing")
122 }
123
124 rule CriteriaStress2TestType{
125       from MM4:MM4!CriteriaStress
126       to t:MM5!TestType
127       (
128        name<- 'Stress Test  ' + MM4.nameApplication   ),
129 t1:MM5!TestCriteria
130      (
131       name<- 'Stress Criteria' ),
132 t2:MM5!ModuleTest
133      (
134        name<- 'Test Application',
```

```
135  testSuiteCount<- '1' ),
136 t3:MM5!TestSuite
137       (
138         name<- MM4.nameApplication,
139  testCaseCount<- '1'),
140 t4:MM5!TestCase
141       (
142         name<- MM4.nameApplication,
143  testDateCount<- '1' )
144 }
145
146 rule TimeValueStress2ScriptCodeTesting{
147       from MM4:MM4!TimeValue
148    to MM5:MM5!ScriptCodeTesting
149       (
150         script<- '#!/bin/sh #Teste de stress echo "Testing stress start" #Captura data e ↩
              hora do sistema qdo iniciou o teste date>initialdate.txt #Chama o numero de ↩
              instâncias a serem executadas exec_programa1 #Captura data e hora do sistema qdo↩
               finalizou o teste date>finaldate.txt echo "End testing stress"',
151  expectedValue <- MM4.expectedTime )
152 }
```

8.2 Anexo B - Regras de Transformação de *TestingMetamodel* para *AbstractCloudTesting* - PaaS

```
1  module testingMetamodel2abstractCloudTestingPaaS;
2  create OUT:abstractCloudTesting from IN:testingMetamodel;
3
4
5  entrypoint rule Cloud(){
6    to
7       t : abstractCloudTesting!AbstractCloudTestModel (
8         name <- 'CloudFoundry'
9       ),
10
11      t1 : abstractCloudTesting!CloudServices1 (
12        typeService<- 'PaaS'
13      )
14 }
15
16
17 rule TestType2TestType1{
18       from testtype:testingMetamodel!TestType
19       to testtype1:abstractCloudTesting!TestType1
20       (
21             name<- testtype.name,
22             moduleTestCount<- '1'
23    )
24 }
25
26
27
28 rule ModuleTest2ModuleTest1{
29       from moduletest:testingMetamodel!ModuleTest
30       to moduletest1:abstractCloudTesting!ModuleTest1
31       (
32             name<- moduletest.name,
33             testSuiteCount<- moduletest.testSuiteCount,
34             moduleTestSuite<- moduletest.moduleTestSuite
35       )
36 }
37
38
39
40 rule TestSuite2TestSuite1{
41       from testsuite:testingMetamodel!TestSuite
42       to testsuite1:abstractCloudTesting!TestSuite1
43       (
44             name<- testsuite.name,
```

```
45              testCaseCount<- testsuite.testCaseCount,
46              testSuiteModule<- testsuite.testSuiteModule,
47              testSuiteCase<- testsuite.testSuiteTestCase
48       )
49 }
50
51
52 rule TestCase2TestCase1{
53       from testcase:testingMetamodel!TestCase
54       to testcase1:abstractCloudTesting!TestCase1
55       (
56              name<- testcase.name,
57              testCaseSuite<- testcase.testCaseTestSuite,
58              testCaseProcedures<- testcase.testCaseProcedures,
59              testCasePreconditions<- testcase.testCasePreconditions,
60              testCaseTestDate<- testcase.testCaseTestDate,
61              testCaseScript<- testcase.testCaseScriptCode,
62              testCaseTestClass<- testcase.testCaseTestClass
63       )
64 }
65
66
67 rule Procedures2Procedures1{
68       from procedures:testingMetamodel!Procedures
69       to procedures1:abstractCloudTesting!Procedures1
70       (
71              name<- procedures.name,
72              description<- procedures.description,
73              proceduresTestCase<- procedures.proceduresTestCase
74       )
75 }
76
77
78 rule Preconditions2Preconditions1{
79       from preconditions:testingMetamodel!Preconditions
80       to preconditions1:abstractCloudTesting!Preconditions1
81       (
82              name<- preconditions.name,
83              value<- preconditions.value,
84              dataType<- preconditions.dataType,
85              preconditionsTestCase<- preconditions.preconditionsTestCase
86       )
87 }
88
89
90 rule TestDate2TestDate1{
91       from testdate:testingMetamodel!TestDate
92       to testdate1:abstractCloudTesting!TestDate1
93       (
94              date<- testdate.inputDate,
95              imputDate<- testdate.date,
```

```
96              date<- testdate.date,
97              description<- testdate.description,
98              testDateT<- testdate.testDateTestCase
99        )
100 }
101
102
103 rule ScriptCodeTesting2ScriptCodeTesting1{
104       from scriptcodetesting:testingMetamodel!ScriptCodeTesting
105       to scriptcodetesting1:abstractCloudTesting!ScriptCodeTesting1
106       (
107             script<- scriptcodetesting.script,
108             inputValue<- scriptcodetesting.inputValue,
109             output<- scriptcodetesting.outputValue,
110             expectedValue<- scriptcodetesting.expectedValue,
111             scriptCodeTestCase<- scriptcodetesting.scriptCodeTestCase
112       )
113 }
114
115
116 rule TestClass2TestClass1{
117       from testclass:testingMetamodel!TestClass
118       to testclass1:abstractCloudTesting!TestClass1
119       (
120             name<- testclass.name,
121             objectSet<- testclass.objectSet,
122             objectBehavior<- testclass.objectBehavior,
123             inputValue<- testclass.inputValue,
124             outputValue<- testclass.outputValue,
125             expectedValue<- testclass.expectedValue,
126             testClassTestCase<- testclass.testClassTestCase,
127             testClassMethod<- testclass.testClassMethod,
128             testClassAttribute<- testclass.testClassAttribute
129       )
130 }
131
132
133 rule Method2Method1{
134       from method:testingMetamodel!Method
135       to method1:abstractCloudTesting!Method1
136       (
137             name<- method.name,
138             static<- method.static,
139             methodTestClass<- method.methodTestClass,
140             methodParameter<- method.methodParameter
141       )
142 }
143
144
145 rule Attribute2Attribute1{
146       from attribute:testingMetamodel!Attribute
```

```
147
148
149
150                 <- attribute.name,
151             controlValueClass<- attribute.controlValueClass,
152             attributeTestClass<- attribute.attributeTestClass,
153             attributeType<- attribute.attributeType
154       )
155 }
156
157 rule TypeAttribute2TypeAttribute1{
158       from typeattribute:testingMetamodel!TypeAttribute
159       to typeattribute1:abstractCloudTesting!TypeAttribute1
160       (
161             value<- typeattribute.value,
162             valueType<- typeattribute.valueType,
163             typeAttribute<- typeattribute.typeAttribute
164       )
165 }
166
167 rule Parameter2Parameter1{
168       from parameter:testingMetamodel!Parameter
169
170
171
172                 <- parameter.name,
173             result<- parameter.result,
174             parameterMethod<- parameter.parameterMethod,
175             parameterType<- parameter.parameterType
176       )
177 }
178
179
180 rule TypeParameter2TypeParameter1{
181      from typeparameter:testingMetamodel!TypeParameter
182
183
184
185                  <- typeparameter.value,
186             valueType<- typeparameter.valueType,
187             typeParameter<- typeparameter.typeParameter
188       )
189 }
190
191 rule Assertion2Assertion1{
192    from assertion:testingMetamodel!Assertion
193
```

```
194
195
196                         <- assertion.assertionType,
197             assertionExpectedValue<- assertion.assertionExpectedValue
198       )
199 }
200
201
202 rule ExpectedValue2ExpectedValue1{
203       from expectedvalue:testingMetamodel!ExpectedValue
204       to expectedvalue1:abstractCloudTesting!ExpectedValue1
205       (
206             value<- expectedvalue.value,
207             dataType<- expectedvalue.dataType,
208             expectedValueAssertion<- expectedvalue.expectedValueAssertion
209       )
210 }
211
212
213 rule IntegrationTest2IntegrationTest1{
214       from integrationtest:testingMetamodel!IntegrationTest
215       to integrationtest1:abstractCloudTesting!IntegrationTest1
216       (
217             name<- integrationtest.name,
218             interactionModule<- integrationtest.interactionModule
219       )
220 }
```

8.3 Anexo C - Regras de Transformação de *TestingMetamodel* para *AbstractCloudTesting* - IaaS

```
1 module testingMetamodel2abstractCloudTestingIaaS;
2 create OUT:abstractCloudTesting from IN:testingMetamodel;
3
4
5 entrypoint rule Cloud(){
6   to
7      t : abstractCloudTesting!AbstractCloudTestModel (
8        name <- 'Eucalyptus'
9      ),
10
11     t1 : abstractCloudTesting!CloudServices1 (
12       typeService<- 'IaaS'
13     )
14 }
15
16
17 rule TestType2TestType1{
18       from testtype:testingMetamodel!TestType
19       to testtype1:abstractCloudTesting!TestType1
20       (
21             name<- testtype.name,
22             moduleTestCount<- '1'
23    )
24 }
25
26
27
28 rule ModuleTest2ModuleTest1{
29       from moduletest:testingMetamodel!ModuleTest
30       to moduletest1:abstractCloudTesting!ModuleTest1
31       (
32             name<- moduletest.name,
33             testSuiteCount<- moduletest.testSuiteCount,
34             moduleTestSuite<- moduletest.moduleTestSuite
35       )
36 }
37
38
39
40 rule TestSuite2TestSuite1{
41       from testsuite:testingMetamodel!TestSuite
42       to testsuite1:abstractCloudTesting!TestSuite1
43       (
44             name<- testsuite.name,
```

```
45              testCaseCount<- testsuite.testCaseCount,
46              testSuiteModule<- testsuite.testSuiteModule,
47              testSuiteCase<- testsuite.testSuiteTestCase
48       )
49 }
50
51
52 rule TestCase2TestCase1{
53       from testcase:testingMetamodel!TestCase
54       to testcase1:abstractCloudTesting!TestCase1
55       (
56              name<- testcase.name,
57              testCaseSuite<- testcase.testCaseTestSuite,
58              testCaseProcedures<- testcase.testCaseProcedures,
59              testCasePreconditions<- testcase.testCasePreconditions,
60              testCaseTestDate<- testcase.testCaseTestDate,
61              testCaseScript<- testcase.testCaseScriptCode,
62              testCaseTestClass<- testcase.testCaseTestClass
63       )
64 }
65
66
67 rule Procedures2Procedures1{
68       from procedures:testingMetamodel!Procedures
69       to procedures1:abstractCloudTesting!Procedures1
70       (
71              name<- procedures.name,
72              description<- procedures.description,
73              proceduresTestCase<- procedures.proceduresTestCase
74       )
75 }
76
77
78 rule Preconditions2Preconditions1{
79       from preconditions:testingMetamodel!Preconditions
80       to preconditions1:abstractCloudTesting!Preconditions1
81       (
82              name<- preconditions.name,
83              value<- preconditions.value,
84              dataType<- preconditions.dataType,
85              preconditionsTestCase<- preconditions.preconditionsTestCase
86       )
87 }
88
89
90 rule TestDate2TestDate1{
91       from testdate:testingMetamodel!TestDate
92       to testdate1:abstractCloudTesting!TestDate1
93       (
94              date<- testdate.inputDate,
95              imputDate<- testdate.date,
```

```
96              date<- testdate.date,
97              description<- testdate.description,
98              testDateT<- testdate.testDateTestCase
99        )
100 }
101
102
103 rule ScriptCodeTesting2ScriptCodeTesting1{
104       from scriptcodetesting:testingMetamodel!ScriptCodeTesting
105       to scriptcodetesting1:abstractCloudTesting!ScriptCodeTesting1
106       (
107             script<- scriptcodetesting.script,
108             inputValue<- scriptcodetesting.inputValue,
109             output<- scriptcodetesting.outputValue,
110             expectedValue<- scriptcodetesting.expectedValue,
111             scriptCodeTestCase<- scriptcodetesting.scriptCodeTestCase
112       )
113 }
114
115
116 rule TestClass2TestClass1{
117       from testclass:testingMetamodel!TestClass
118       to testclass1:abstractCloudTesting!TestClass1
119       (
120             name<- testclass.name,
121             objectSet<- testclass.objectSet,
122             objectBehavior<- testclass.objectBehavior,
123             inputValue<- testclass.inputValue,
124             outputValue<- testclass.outputValue,
125             expectedValue<- testclass.expectedValue,
126             testClassTestCase<- testclass.testClassTestCase,
127             testClassMethod<- testclass.testClassMethod,
128             testClassAttribute<- testclass.testClassAttribute
129       )
130 }
131
132
133 rule Method2Method1{
134       from method:testingMetamodel!Method
135       to method1:abstractCloudTesting!Method1
136       (
137             name<- method.name,
138             static<- method.static,
139             methodTestClass<- method.methodTestClass,
140             methodParameter<- method.methodParameter
141       )
142 }
143
144
145 rule Attribute2Attribute1{
146       from attribute:testingMetamodel!Attribute
```

```
147
148
149
150                 <- attribute.name,
151             controlValueClass<- attribute.controlValueClass,
152             attributeTestClass<- attribute.attributeTestClass,
153             attributeType<- attribute.attributeType
154       )
155 }
156
157 rule TypeAttribute2TypeAttribute1{
158       from typeattribute:testingMetamodel!TypeAttribute
159       to typeattribute1:abstractCloudTesting!TypeAttribute1
160       (
161             value<- typeattribute.value,
162             valueType<- typeattribute.valueType,
163             typeAttribute<- typeattribute.typeAttribute
164       )
165 }
166
167 rule Parameter2Parameter1{
168       from parameter:testingMetamodel!Parameter
169
170
171
172                 <- parameter.name,
173             result<- parameter.result,
174             parameterMethod<- parameter.parameterMethod,
175             parameterType<- parameter.parameterType
176       )
177 }
178
179
180 rule TypeParameter2TypeParameter1{
181      from typeparameter:testingMetamodel!TypeParameter
182
183
184
185                  <- typeparameter.value,
186             valueType<- typeparameter.valueType,
187             typeParameter<- typeparameter.typeParameter
188       )
189 }
190
191 rule Assertion2Assertion1{
192    from assertion:testingMetamodel!Assertion
193
```

```
194
195
196                         <- assertion.assertionType,
197             assertionExpectedValue<- assertion.assertionExpectedValue
198       )
199 }
200
201
202 rule ExpectedValue2ExpectedValue1{
203       from expectedvalue:testingMetamodel!ExpectedValue
204       to expectedvalue1:abstractCloudTesting!ExpectedValue1
205       (
206             value<- expectedvalue.value,
207             dataType<- expectedvalue.dataType,
208             expectedValueAssertion<- expectedvalue.expectedValueAssertion
209       )
210 }
211
212
213 rule IntegrationTest2IntegrationTest1{
214       from integrationtest:testingMetamodel!IntegrationTest
215       to integrationtest1:abstractCloudTesting!IntegrationTest1
216       (
217             name<- integrationtest.name,
218             interactionModule<- integrationtest.interactionModule
219       )
220 }
```

8.4 Anexo D - Regras de Transformação de *AbsractTestingMetamodel* para *CloudFoundry*

```
1  module abstractcloudtesting2cloudtestingcloudfoundry;
2  create OUT:cloudtestingcloudfoundry from IN:abstractcloudtesting;
3
4
5  rule CustomerToolCommunication12CustomerToolCommunication{
6        from customertoolcommunication1:abstractcloudtesting!CustomerToolCommunication1
7        to customertoolcommunication:cloudtestingcloudfoundry!CustomerToolCommunication
8        (
9              name<- customertoolcommunication1.name
10       )
11 }
12
13
14 rule CloudServices12CloudService{
15       from cloudservices1:abstractcloudtesting!CloudServices1
16       to cloudservice:cloudtestingcloudfoundry!CloudService
17       (
18             name<- cloudservices1.typeService,
19             cloudAPI<- cloudservices1.CloudServicesAPI
20       )
21 }
22
23
24 rule APIServices12APIServices{
25       from apiservices1:abstractcloudtesting!APIServices1
26       to apiservices:cloudtestingcloudfoundry!APIServices
27       (
28             apiName<- apiservices1.name,
29             APICloud<- apiservices1.APIServicesCloud
30       )
31 }
32
33
34 rule TestSuite12TestSuite{
35       from testsuite1:abstractcloudtesting!TestSuite1
36       to testsuite:cloudtestingcloudfoundry!TestSuite
37       (
38             name<- testsuite1.name,
39             testCaseCount<- testsuite1.testCaseCount,
40             testSuiteTestCase<- testsuite1.testSuiteCase
41       )
42 }
43
44
```

```
rule TestCase12TestCase{
	from testcase1:abstractcloudtesting!TestCase1
	to testcase:cloudtestingcloudfoundry!TestCase
	(
		name<- testcase1.name,
		testCaseTestSuite<- testcase1.testCaseSuite,
		testCaseProcedure<- testcase1.testCaseProcedures,
		testCasePrecoditions<- testcase1.testCasePreconditions,
		testCaseTestClass<- testcase1.testCaseTestClass
	)
}


rule Procedures12Procedures{
	from procedures1:abstractcloudtesting!Procedures1
	to procedures:cloudtestingcloudfoundry!Procedures
	(
		name<- procedures1.name,
		description<- procedures1.description,
		procedureTestCase<- procedures1.proceduresTestCase
	)
}


rule Preconditions12Preconditions{
	from preconditions1:abstractcloudtesting!Preconditions1
	to preconditions:cloudtestingcloudfoundry!Preconditions
	(
		name<- preconditions1.name,
		Value<- preconditions1.value,
		DataType<- preconditions1.dataType,
		preconditionsTestCase<- preconditions1.preconditionsTestCase
	)
}


rule TestClass12TestClass{
	from testclass1:abstractcloudtesting!TestClass1
	to testclass:cloudtestingcloudfoundry!TestClass
	(
		name<- testclass1.name,
		objectSet<- testclass1.objectSet,
		objectBehavior<- testclass1.objectBehavior,
		inputValue<- testclass1.inputValue,
		outputValue<- testclass1.outputValue,
		expectedValue<- testclass1.expectedValue,
		testClassTestCase<- testclass1.testClassTestCase,
		testClassAttribute<- testclass1.testClassAttribute,
		testClassMethod<- testclass1.testClassMethod
	)
}
```

```
96
97
98 rule Attribute12ComplexAttribute{
99       from attribute1:abstractcloudtesting!Attribute1
100      to complexattribute:cloudtestingcloudfoundry!ComplexAttribute
101      (
102            value<- attribute1.controlValueClass,
103            attributeValue<- attribute1.value,
104            value<- attribute1.value,
105            attributeTestClass<- attribute1.attributeTestClass
106      )
107 }
108
109
110 rule Attribute12SimpleAttribute{
111      from attribute1:abstractcloudtesting!Attribute1
112      to simpleattribute:cloudtestingcloudfoundry!SimpleAttribute
113      (
114            value<- attribute1.controlValueClass,
115            attributeValue<- attribute1.value,
116            value<- attribute1.value,
117            attributeTestClass<- attribute1.attributeTestClass
118      )
119 }
120
121
122 rule Method12Method{
123      from method1:abstractcloudtesting!Method1
124      to method:cloudtestingcloudfoundry!Method
125      (
126            static<- method1.static,
127            name<- method1.name,
128            methodTestClass<- method1.methodTestClass,
129            methodParameter<- method1.methodParameter
130      )
131 }
132
133
134
135
136 rule Parameter12Parameter{
137      from parameter1:abstractcloudtesting!Parameter1
138      to parameter:cloudtestingcloudfoundry!Parameter
139      (
140            name<- parameter1.name,
141            result<- parameter1.result,
142            parameterMethod<- parameter1.parameterMethod
143      )
144 }
145
146
```

```
147 rule Assertion12Assertion{
148       from assertion1:abstractcloudtesting!Assertion1
149       to assertion:cloudtestingcloudfoundry!Assertion
150       (
151             assertionType<- assertion1.assertionType,
152             order<- assertion1.order
153       )
154 }
```

8.5 Anexo E - Regras de Transformação de *AbsractTestingMetamodel* para *Eucalyptus*

```
1  module abstractcloudtesting2cloudtestingeucalyptus;
2  create OUT:cloudtestingeucalyptus from IN:abstractcloudtesting;
3
4
5  rule CustomerToolCommunication12CustomerToolCommunication{
6        from customertoolcommunication1:abstractcloudtesting!CustomerToolCommunication1
7        to customertoolcommunication:cloudtestingeucalyptus!CustomerToolCommunication
8        (
9              name<- customertoolcommunication1.name,
10             customerScript<- customertoolcommunication1.customerScript
11       )
12 }
13
14
15 rule CloudServices12CloudService{
16       from cloudservices1:abstractcloudtesting!CloudServices1
17       to cloudservice:cloudtestingeucalyptus!CloudService
18       (
19             name<- cloudservices1.typeService
20       )
21 }
22
23
24 rule TestSuite12TestSuite{
25       from testsuite1:abstractcloudtesting!TestSuite1
26       to testsuite:cloudtestingeucalyptus!TestSuite
27       (
28             name<- testsuite1.name,
29             testCaseCount<- testsuite1.testCaseCount,
30             testSuiteTestCase<- testsuite1.testSuiteCase
31       )
32 }
33
34
35 rule TestCase12TestCase{
36       from testcase1:abstractcloudtesting!TestCase1
37       to testcase:cloudtestingeucalyptus!TestCase
38       (
39             name<- testcase1.name,
40             testCaseTestSuite<- testcase1.testCaseSuite,
41             testCaseProcedure<- testcase1.testCaseProcedures,
42             testCasePreconditions<- testcase1.testCasePreconditions,
43             testCaseScript<- testcase1.testCaseScript
44       )
```

```
45 }
46
47
48 rule Procedures12Procedure{
49       from procedures1:abstractcloudtesting!Procedures1
50       to procedure:cloudtestingeucalyptus!Procedure
51       (
52             name<- procedures1.name,
53             description<- procedures1.description,
54             procedureTestCase<- procedures1.proceduresTestCase
55       )
56 }
57
58
59 rule Preconditions12Preconditions{
60       from preconditions1:abstractcloudtesting!Preconditions1
61       to preconditions:cloudtestingeucalyptus!Preconditions
62       (
63             name<- preconditions1.name,
64             value<- preconditions1.value,
65             dataType<- preconditions1.dataType,
66             preconditionTestCase<- preconditions1.preconditionsTestCase
67       )
68 }
69
70
71 rule ScriptCodeTesting12ScriptCodeTesting{
72       from scriptcodetesting1:abstractcloudtesting!ScriptCodeTesting1
73       to scriptcodetesting:cloudtestingeucalyptus!ScriptCodeTesting
74       (
75             script<- scriptcodetesting1.script,
76             inputValue<- scriptcodetesting1.inputValue,
77             outputValue<- scriptcodetesting1.output,
78             expectedValue<- scriptcodetesting1.expectedValue,
79             scriptTestCase<- scriptcodetesting1.scriptCodeTestCase,
80             scriptCustomer<- scriptcodetesting1.scriptCustomer
81       )
82 }
```

8.6 Anexo F - Regras de Transformação de *Eucalyptus* para Código

```
query Eucalyptus2Code = CloudTestingEucalyptus!NameElement.allInstances()->
select(e | e.oclIsTypeOf(CloudTestingEucalyptus!ScriptCodeTesting) or e.oclIsTypeOf(↩
    CloudTestingEucalyptus!TestCase))->
collect(x | x.toString().writeTo('C:/Temp/NonFunctionalTest/' + x.name + '.sh')); -- Query ↩
    Template

uses library_Eucalyptus2Code;


-------------------------------------------------------------------------
library library_Eucalyptus2Code;  -- Library Template


helper context CloudTestingEucalyptus!ScriptCodeTesting def: toString() : String =
'-- Euca2ools \n' +

self.scriptCustomer->iterate(i; acc:String=''| acc + i.typeCommand +'\n') +

self.script
;


helper context CloudTestingEucalyptus!TestCase def: toString() : String =
' ' +
self.testCaseScript.toString() +
'\n'
;
```

8.7 Anexo G - Regras de Transformação de *CloudFoundry* para Código

```
query CloudFoundry2Code = cloudTestingCloudFoundry!NameElement.allInstances()->
select(e | e.oclIsTypeOf(cloudTestingCloudFoundry!CloudTestingFoundry)or e.oclIsTypeOf(cloudTestingCloudFoundry!TestSuite)  or
e.oclIsTypeOf(cloudTestingCloudFoundry!TestCase) or e.oclIsTypeOf(cloudTestingCloudFoundry!CustomerToolCommunication))->
collect(x | x.toString().writeTo('C:/Temp/FuncionalTestMessage/' + x.name + '.java')); -- Query Template

uses library_CloudFoundry2Code;


-------------------------------------------------------------------------------------------------------------------------


library library_CloudFoundry2Code; -- Library Template

helper context cloudTestingCloudFoundry!Method def: callsMethod(): String =
'new '+ self.freeParameter + '().'
--+ self.name
+ '('+
if self.methodParameter->isEmpty() then
''
else
self.methodParameter->iterate( i ; acc:String=''| acc +
if acc='' then ''  else ',' endif + i.name)
endif +')';



helper context cloudTestingCloudFoundry!Assertion def: getNameAssertion() : String =
if self.assertionType='Equals' then 'assertEquals'
else if self.assertionType='True' then 'assertTrue'
else if self.assertionType='False' then 'assertFalse'
else if self.assertionType='NotNull' then 'assertNotNull'
else 'undefinedAssertionType' endif endif endif endif;



helper context cloudTestingCloudFoundry!Assertion def: generateAssertionType() : String =
'\n' + '\t\t'+ self.getNameAssertion() +
'(' + self.expectedValue +
', '+ self.assertionMethod.callsMethod() + ');';


```

```
41
42 helper context cloudTestingCloudFoundry!Assertion def: toString() : String =
43 '\t@Test\n'+
44 '\tpublic void ' +
45 self.name + '(){\n' +
46 self.generateAssertionType() + '\n\t}\n';
47
48
49
50 helper context cloudTestingCloudFoundry!TestCase def: toString() : String =
51 'public class ' +
52 self.name +
53 ' extends junit.framework.TestCase{\n\n' +
54 self.testCaseAssertion->iterate(i; acc:String=''| acc + i.toString()+'\n') +
55 '\n}\n'
56 ;
57
58
59
60
61 helper context cloudTestingCloudFoundry!TestSuite def: toString() : String =
62 'public class ' +
63 self.name + ' ' +
64 'extends' + ' ' +
65 'TestCase' + ' ' +
66 '{\n\n'+
67 '\tTestSuite suite = new TestSuite("'+ self.name +'");\n'+
68 self.testSuiteTestCase->iterate(i; acc:String=''| acc + '\tsuite.addTestCase('+ i.name+'.↩
       class' + ');\n') +
69 '\treturn suite;\n' + '\n}\n\n'
70
71
72 helper context cloudTestingCloudFoundry!ScriptCodeTesting def: toString() : String =
73 '-- vmc\n' +
74
75 self.scriptCustomer->iterate(i; acc:String=''| acc + i.typeCommand +'\n') +
76
77 self.script;
78
79
80
81 helper context cloudTestingCloudFoundry!TestCase def: toString() : String =
82 ' ' +
83 self.testCaseScript.toString() +
84 '\n';;
```

8.8 Anexo H - Geração de casos de teste com o Algoritmo

Neste anexo, um terceiro exemplo ilustrativo de geração de casos de teste com o algoritmo construído no protótipo é apresentado. Neste exemplo, um modelo de negócios com métodos, cujo o tipo dos parâmetro é *double* e *float*, é apresentado. A Figura 8.1, exibe trecho do modelo de negócios instanciado a partir do modelo de

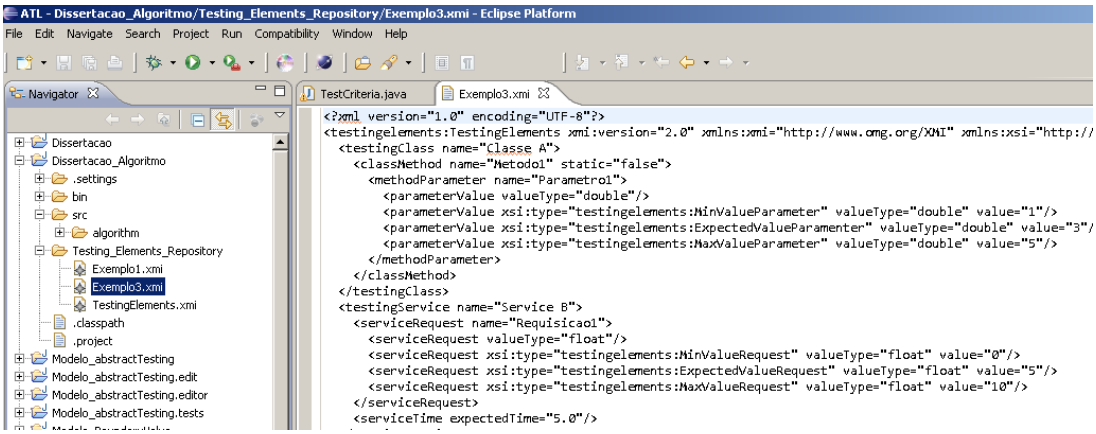

Figura 8.1: Modelo de negócio do terceiro exemplo ilustrativo

Elementos de Teste. Observa-se que existe o "*Parametro1*" do tipo *double*, que possui os valores mínimo, máximo e esperado (*MinValueParameter*, *ExpectedValueParamenter* e *MaxValueParameter*). Nota-se também que existe a requisição de serviço "*Requisicao1*" do tipo *float*, que também possui os valores mínimo, máximo e esperado.

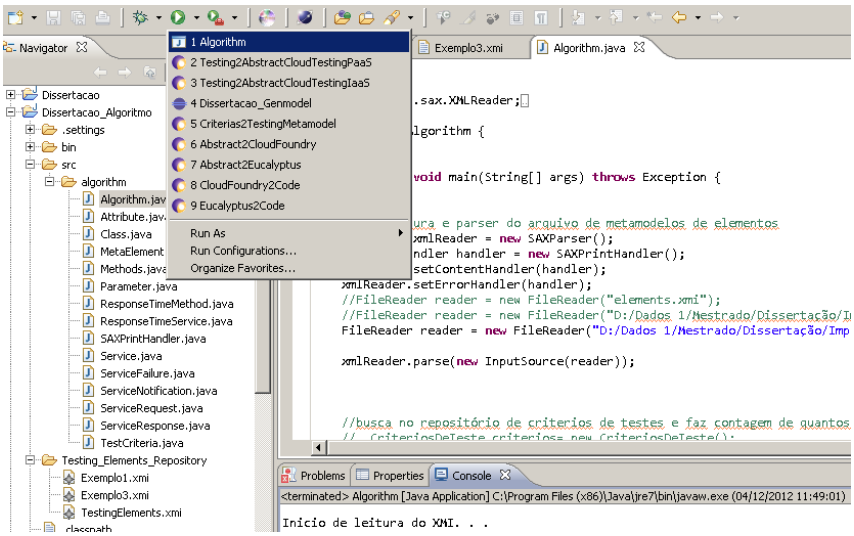

Figura 8.2: Aplicação do algoritmo para gerar Casos de Teste do terceiro exemplo ilustrativo

Conforme exibido na Figura 8.2, o algoritmo foi aplicado sobre o modelo de negócio. Com a aplicação do algoritmo para geração de casos de teste, apresentado neste *framework*, sobre este modelo, gerou-se 54 (cinquenta e quatro) casos de teste.

Dentre os casos de teste gerados, exibe-se o caso de teste da Listagem 8.1, gerado de acordo com o critério de Análise do Valor Limite, caracterizado como um teste do grupo de testes funcionais.

Observa-se nesta listagem que um caso de teste foi gerado para a "*classe A*", onde o tipo de teste é "contínuo", o método testado é o "*Metodo 1*" , do tipo *double*, cujo valor testado é 4.0 e o valor esperado é 5.

Código 8.1: Caso de Teste gerado para Análise do Valor Limite

```
1 <?xml version="1.0" encoding="Cp1252"?>
2 <boundaryvalue:CriteriaBoundaryValue xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" ↩
      xmlns:boundaryvalue="http://boundaryvalue/1.0" name="Classe A" typeBoundary="Continue">
3  <criteriaObject name="Classe A"><testingSub name="Metodo1"><subObjectValue testingValue="↩
       4.0" expectedValue="5" typeValue="double">
4  </subObjectValue>
5 </testingSub></criteriaObject>
6 </boundaryvalue:CriteriaBoundaryValue>
```

Outro caso de teste gerado neste exemplo é o caso de teste com o critério de Tempo de Resposta, conforme exibido na Listagem 8.2.

Observa-se nesta listagem que o teste é aplicado sobre o "*Service B*", com o tempo de teste em 6.0 e o tempo esperado para 5.0.

Código 8.2: Caso de Teste gerado para Tempo de Resposta

```
1 <?xml version="1.0" encoding="Cp1252"?>
2 <responsetime:CriteriaResponseTime xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" ↩
      xmlns:responsetime="http://responsetime/1.0" name="Response Time Testing">
3 <criteriaMethod name="Service B" nameClass="Service B">
4  <criteriaTime expectedTime="5.0" testingTime="6.0">
5  </criteriaTime>
6 </criteriaMethod>
7 </responsetime:CriteriaResponseTime>
```

A ideia neste exemplo ilustrativo é de mostrar ao leitor que, a partir de um modelo de negócio cujo os tipos dos parâmetros são do tipo *integer*, *float* e *double*, que

o número de casos de teste gerados pelo algoritmo aumenta. Este aumento ocorre porque com um modelo com esse tipo de parâmetro e com os valores (mínimo, máximo e esperado), atendem mais requisitos das combinações da lógica empregada no algoritmo, conforme descrito na Seção 5.1.2.

8.9 Anexo I - *Framework* Ampliado

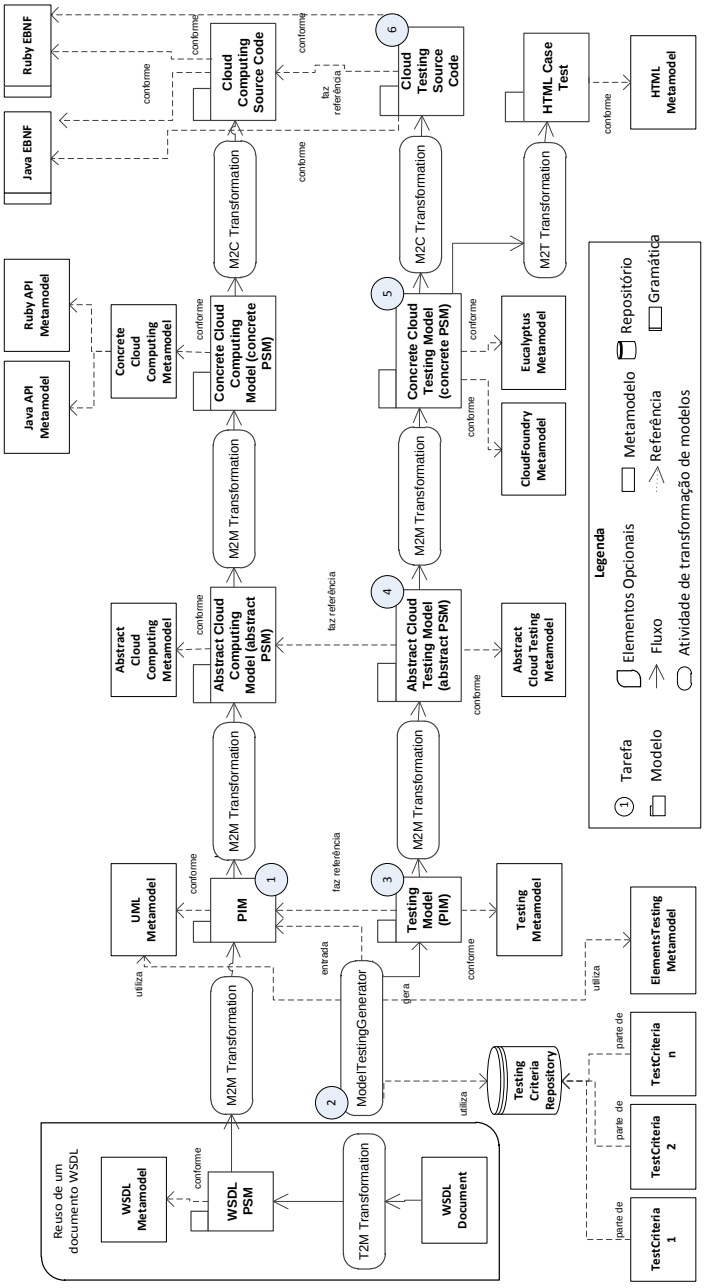

Figura 8.3: *Framework* - Ampliado

8.10 Anexo J - Implementação em Java da Classe *SaxPrintHandler*

```
1 //*********************************************************************************************↩

2 //Copyright: Jéssica Bassani de Oliveira
3 //Programa de pós-graduação em Engenharia da Eletricidade
4 //Classe que trabalha junto ao parser para a leitura do XMI
5 //Classe implementada para receber e separar os elementos em um arraylist
6 //*********************************************************************************************↩

7 package algorithm;
8
9 import java.util.ArrayList;
10 import java.util.Collection;
11 import java.util.Iterator;
12 import java.util.List;
13
14 import org.xml.sax.helpers.DefaultHandler;
15 import org.xml.sax.Attributes;
16
17 public class SAXPrintHandler extends DefaultHandler {
18
19 MetaElement metaModelo = new MetaElement();
20
21
22 Class nomeClasse = new Class() ;
23 Methods nomeMetodo = new Methods() ;
24 ServiceRequest nomeRequisicao= new ServiceRequest() ;
25 ServiceResponse nomeResposta= new ServiceResponse() ;
26 ServiceNotification nomeNotificacao= new ServiceNotification() ;
27 ServiceFailure mensagemFalha = new ServiceFailure();
28
29 Parameter parametro = new Parameter();
30
31 Attribute nomeAtributo;
32 Attribute atributo;
33 ResponseTimeMethod tempoEsperado = new ResponseTimeMethod();
34 ResponseTimeService tempoEsperadoServico = new ResponseTimeService();
35
36 Service nomeServico= new Service();
37
38 String valorParametro;
39 String valorMinParametro;
40 String valorMaxParametro;
41
42 String valorAtributo;
```

```
String valorMinAtributo;
String valorMaxAtributo;



//*****************************************************************************************************

public void startDocument() {
 System.out.println("\n"+"Inicio de leitura do XMI. . . ");
}


//*****************************************************************************************************

public void endDocument() {
 System.out.println("Fim de leitura do arquivo XMI . . . ");
 //encerrando a leitura e adição dos elementos na lista, é chamado metodo para instanciar ↩
     criterios de teste
 TestCriteria escritaMetamodelosClasse = new TestCriteria();
 escritaMetamodelosClasse.contagemCriterios(metaModelo);
}


//*****************************************************************************************************

public void startElement(String uri, String name, String qName, Attributes atts) {

 String verificaTipo = null;
 int nroInstancias = 0;

 if (atts.getLength() > 0) {
  String attribs = " ";
  int attrib;

  for ( attrib = 0; attrib < atts.getLength(); ++attrib){
   //para detalhes da aplicacao

   if (name.equals("TestingElements")){
    metaModelo.setCaminhoAplicacao("nada");
    metaModelo.setNumeroInstancias(0);
    metaModelo.setNomeAplicacao("nada");

     if ((atts.getLocalName(attrib).equals("PathApplication"))){
     metaModelo.setCaminhoAplicacao(atts.getValue(attrib));
    }
    if ((atts.getLocalName(attrib).equals("numberInstances"))){
        nroInstancias =  Integer.parseInt(atts.getValue(attrib));
     metaModelo.setNumeroInstancias(nroInstancias);
    }

    if ((atts.getLocalName(attrib).equals("name"))){
```

```
90      metaModelo.setNomeAplicacao(atts.getValue(attrib));
91     }
92    }
93
94    //para tempo de aplicacao
95     if (name.equals("testingTime")){
96      metaModelo.setTempoAplicacao(Double.parseDouble(atts.getValue(attrib)));
97     }
98
99
100   //para as classes
101   if (name.equals("testingClass")){
102    nomeClasse.setNome(atts.getValue(attrib));
103     attribs += atts.getLocalName(attrib) + ": '" +  atts.getValue(attrib) + "' ";
104   }
105
106   //para métodos
107   if (name.equals("classMethod")){
108       if (atts.getLocalName(attrib).equals("name")){
109        nomeMetodo.setNome(atts.getValue(attrib));
110       }
111   }
112
113   //para parametros dos métodos
114   if(name.equals("methodParameter")){
115       parametro = new Parameter();
116       parametro.setNome(atts.getLocalName(attrib));
117   }
118
119   //para valor esperado do parametro
120   if(name.equals("parameterValue")){
121    if (atts.getValue(attrib).equals("testingelements:ExpectedValueParamenter")){
122     if (!(atts.getValue(2) == null)){
123      parametro.setValorEsperado(atts.getValue(2));
124     } else {
125      if (!(atts.getValue(1) == null)){
126       parametro.setValorEsperado(atts.getValue(1));
127      }
128      }
129    }else{
130
131     if (atts.getLocalName(attrib).equals("valueType")){
132      parametro.setTipo(atts.getValue(attrib));
133     }
134    }
135   }
136     //para valor minimo do parametro
137   if(name.equals("parameterValue")){
138    if (atts.getValue(attrib).equals("testingelements:MinValueParameter")){
139     if (!(atts.getValue(2) == null)){
140      parametro.setValorMin(atts.getValue(2));
```

```
141      }else {
142       if (!(atts.getValue(1) == null)){
143        parametro.setValorMin(atts.getValue(1));
144       }
145       }
146     }else{
147      if (atts.getLocalName(attrib).equals("valueType")){
148       parametro.setTipo(atts.getValue(attrib));
149      }
150     }
151    }
152
153    //para valor maximo do parametro
154     if(name.equals("parameterValue")){
155       if (atts.getValue(attrib).equals("testingelements:MaxValueParameter")){
156        if (!(atts.getValue(2) == null)){
157         parametro.setValorMax(atts.getValue(2));
158        }
159        else {
160         if (!(atts.getValue(1) == null)){
161         parametro.setValorMax(atts.getValue(1));
162        }
163       }
164      }else{
165       if (atts.getLocalName(attrib).equals("valueType")){
166        parametro.setTipo(atts.getValue(attrib));
167       }
168      }
169     }
170
171     //para o tempo de resposta do metodo
172     if(name.equals("methodTime")){
173      Double tempo = Double.parseDouble((atts.getValue(attrib)));
174         tempoEsperado.setTempoEsperado(tempo);
175     }
176
177
178     //para Atributos
179     if (name.equals("classAttribute")){
180       nomeAtributo = new Attribute();
181       nomeAtributo.setNome(atts.getValue(attrib));
182
183     //para valor esperado do atributo
184     if(name.equals("AttributeVAttribute")){
185      if (atts.getValue(attrib).equals("testingelements:ExpectedValueAttribute")){
186       atributo.setValorEsperado(atts.getValue(1));
187      }
188     }else{
189      if (atts.getValue(attrib).equals("valueType")){
190        atributo.setTipo(atts.getLocalName(attrib));
191       }
```

```
192       if (name.equals("value")){
193        atributo.setTipo(atts.getLocalName(attrib));
194       }
195
196      }
197
198     //para valor minimo do atributo
199     if(name.equals("AttributeVAttribute")){
200      if (atts.getValue(attrib).equals("testingelements:MinValueAttribute")){
201       if (!(atts.getValue(1) == null)){
202        atributo.setValorMin(atts.getValue(1));
203       }
204      }else{
205       if (name.equals("valueType")){
206        atributo.setTipo(atts.getLocalName(attrib));
207       }else{
208
209       }if (name.equals("value")){
210        atributo.setTipo(atts.getLocalName(attrib));
211       }
212      }
213     }//fim atributo
214
215     //para valor maximo do atributo
216     if(name.equals("AttributeVAttribute")){
217      if (atts.getValue(attrib).equals("testingelements:MaxValueAttribute")){
218       atributo.setValorMax(atts.getValue(1));
219      }else{
220       if (name.equals("valueType")){
221        atributo.setTipo(atts.getLocalName(attrib));
222       }else{
223
224       }if (name.equals("value")){
225        atributo.setTipo(atts.getLocalName(attrib));
226       }
227      }
228       }
229     }
230
231   //se for serviço
232   if (name.equals("testingService")){
233    nomeServico.setNome(atts.getValue(attrib));
234   }
235
236
237   //para o tempo de resposta do servico
238   if(name.equals("serviceTime")){
239    Double tempoServico = Double.parseDouble((atts.getValue(attrib)));
240       tempoEsperadoServico.setTempoEsperado(tempoServico);
241   }
242
```

```
243  //serviceRequest
244  if (name.equals("serviceRequest")){
245       if (atts.getLocalName(attrib).equals("name")){
246        nomeRequisicao.setNome(atts.getValue(attrib));
247    }
248  }
249  //para valor esperado do parametro da requisição
250  if(name.equals("serviceRequest")){
251   if (atts.getLocalName(attrib).equals("valueType")){
252    nomeRequisicao.setTipo(atts.getValue(attrib));
253    //System.out.println("Tipo do REQUISICAO = "+(atts.getValue(attrib)));
254   }
255   //System.out.println("VALOR do RESPOSTA = "+(atts.getValue(attrib)));
256   if (atts.getValue(attrib).equals("testingelements:ExpectedValueRequest")){
257    if (atts.getLocalName(1).equals("value")){
258     nomeRequisicao.setValorEsperado(atts.getValue(1));
259     //System.out.println("VALOR do REQUISICAO = "+(atts.getValue(1)));
260    } else {
261     nomeRequisicao.setValorEsperado(atts.getValue(2));
262     //System.out.println("VALOR do REQUISICAO = "+(atts.getValue(2)));
263    }
264   }
265  }
266
267  //para valor minimo do parametro da requisição
268  if(name.equals("serviceRequest")){
269   if (atts.getLocalName(attrib).equals("valueType")){
270    nomeRequisicao.setTipo(atts.getValue(attrib));
271    //System.out.println("Tipo do REQUISICAO = "+(atts.getValue(attrib)));
272   }
273   //System.out.println("VALOR do REQUISICAO = "+(atts.getValue(attrib)));
274   if (atts.getValue(attrib).equals("testingelements:MinValueRequest")){
275     if (atts.getLocalName(1).equals("value")){
276     nomeRequisicao.setValorMin(atts.getValue(1));
277     //System.out.println("VALOR do REQUISICAO = "+(atts.getValue(1)));
278    } else {
279
280     nomeRequisicao.setValorMin(atts.getValue(2));
281     //System.out.println("VALOR do RESPOSTA = "+(atts.getValue(2)));
282
283        }
284   }
285  }
286
287  //para valor maximo do parametro da requisição
288  if(name.equals("serviceRequest")){
289   if (atts.getLocalName(attrib).equals("valueType")){
290    nomeRequisicao.setTipo(atts.getValue(attrib));
291    //System.out.println("Tipo do REQUISICAO = "+(atts.getValue(attrib)));
292   }
293   //System.out.println("VALOR do REQUISICAO = "+(atts.getValue(attrib)));
```

```
294    if (atts.getValue(attrib).equals("testingelements:MaxValueRequest")){
295     if (atts.getLocalName(1).equals("value")){
296      nomeRequisicao.setValorMax(atts.getValue(1));
297     } else {
298      nomeRequisicao.setValorMax(atts.getValue(2));;
299     }
300    }
301   }
302
303   //serviceResponse
304     if (name.equals("serviceResponse")){
305           if (atts.getLocalName(attrib).equals("name")){
306            nomeResposta.setNome(atts.getValue(attrib));
307           }
308     }
309
310     //para valor esperado do parametro da resposta
311     if(name.equals("serviceResponse")){
312      if (atts.getLocalName(attrib).equals("valueType")){
313       nomeResposta.setTipo(atts.getValue(attrib));
314       //System.out.println("Tipo do RESPOSTA = "+(atts.getValue(attrib)));
315      }
316
317      if (atts.getValue(attrib).equals("testingelements:ExpectedValueResponse")){
318       if (atts.getLocalName(1).equals("value")){
319        nomeResposta.setValorEsperado(atts.getValue(1));
320
321       } else {
322        nomeResposta.setValorEsperado(atts.getValue(2));
323        //System.out.println("VALOR do RESPOSTA = "+(atts.getValue(2)));
324       }
325      }
326     }
327
328     //para valor minimo do parametro da resposta
329     if(name.equals("serviceResponse")){
330      if (atts.getLocalName(attrib).equals("valueType")){
331       nomeResposta.setTipo(atts.getValue(attrib));
332       //System.out.println("Tipo do RESPOSTA = "+(atts.getValue(attrib)));
333      }
334
335      if (atts.getValue(attrib).equals("testingelements:MinValueResponse")){
336       if (atts.getLocalName(1).equals("value")){
337        nomeResposta.setValorMin(atts.getValue(1));
338        //System.out.println("VALOR MINN  do RESPOSTA = "+(atts.getValue(1)));
339       } else {
340        nomeResposta.setValorMin(atts.getValue(2));
341       // System.out.println("VALOR MiNN do RESPOSTA = "+(atts.getValue(2)));
342       }
343      }
344     }
```

```
345
346     //para valor maximo do parametro da resposta
347     if(name.equals("serviceResponse")){
348      if (atts.getLocalName(attrib).equals("valueType")){
349       nomeResposta.setTipo(atts.getValue(attrib));
350       //System.out.println("Tipo do RESPOSTA = "+(atts.getValue(attrib)));
351      }
352      if (atts.getValue(attrib).equals("testingelements:MaxValueResponse")){
353       if (atts.getLocalName(1).equals("value")){
354        nomeResposta.setValorMax(atts.getValue(1));
355       } else {
356        nomeResposta.setValorMax(atts.getValue(2));
357       }
358      }
359     }
360
361     //serviceOperation
362     if (name.equals("serviceOperation")){
363            if (atts.getLocalName(attrib).equals("name")){
364            nomeNotificacao.setNome(atts.getValue(attrib));
365        }
366      }
367
368     //notificationFault
369     if (name.equals("notificationFault")){
370            if (atts.getLocalName(attrib).equals("message")){
371            mensagemFalha.setMensagem(atts.getValue(attrib));
372           }
373     }
374   }
375  }
376  else
377   System.out.println("Sem atributos ou métodos.");
378 }
379
380 //*****************************************************************************************************************

381 public void endElement(String uri, String name, String qName) {
382
383  if (name.equals("testingClass")){
384   metaModelo.listaClasse.add(nomeClasse);
385  }
386
387  if (name.equals("classMethod")){
388     nomeClasse.setMetodos(nomeMetodo);
389     Methods objMetodos = new Methods();
390     objMetodos = nomeClasse.getObjMetodo();
391  }
392
393  if (name.equals("methodTime")){
394   nomeMetodo.setTempoRespostaMetodo(tempoEsperado);
```

```
395   ResponseTimeMethod objTempoMetodo = new ResponseTimeMethod();
396   objTempoMetodo = nomeMetodo.getObjTempoMetodo();
397   }
398
399  if(name.equals("parameterValue")){
400    nomeMetodo.setParameter(parametro);
401    Parameter objParametros = new Parameter();
402    objParametros = nomeMetodo.getObjParameter();
403
404    if (parametro !=null){
405     valorParametro = parametro.getValorEsperado();
406     if (!(valorParametro == null)){
407
408     }
409     valorMaxParametro = parametro.getValorMax();
410     if (!(valorMaxParametro == null)){
411     }
412     valorMinParametro = parametro.getValorMin();
413     if (!(valorMinParametro == null)){
414     }
415    }
416   }
417
418
419   if(name.equals("classAttribute")){
420    if (atributo !=null){
421     valorAtributo = atributo.getValorEsperado();
422     if (!(valorAtributo == null)){
423     }
424
425     valorMaxAtributo = atributo.getValorMax();
426     if (!(valorMaxAtributo == null)){
427     }
428
429     valorMinAtributo= atributo.getValorMin();
430     if (!(valorMinAtributo == null)){
431     }
432    }
433   }
434
435
436   if (name.equals("classAttribute")){
437      nomeClasse.setAttribute(nomeAtributo);
438      Attribute objAtributos = new Attribute();
439      objAtributos = nomeClasse.getObjAtributo();
440       if (objAtributos != null){
441      }
442   }
443
444
445   if (name.equals("testingService")){
```

```
446    metaModelo.listaServico.add(nomeServico);
447   }
448
449
450   if (name.equals("serviceRequest")){
451    nomeServico.setRequisicao(nomeRequisicao);
452    ServiceRequest objRequisicao = new ServiceRequest();
453       objRequisicao = nomeServico.getObjRequisicao();
454    }
455
456   if (name.equals("serviceResponse")){
457    nomeServico.setResposta(nomeResposta);
458    ServiceResponse objResposta = new ServiceResponse();
459       objResposta = nomeServico.getObjResposta();
460    }
461
462
463     if (name.equals("serviceTime")){
464      nomeServico.setTempoRespostaServico(tempoEsperadoServico);
465      ResponseTimeService objTempoServico = new ResponseTimeService();
466      objTempoServico = nomeServico.getObjTempoResposta();
467      }
468
469
470     if (name.equals("serviceOperation")){
471      nomeServico.setNotificacao(nomeNotificacao);
472      ServiceNotification objNotificacao = new ServiceNotification();
473          objNotificacao = nomeServico.getObjNotificacao();
474      }
475 }
476
477 //*************************************************************************************************************

478 public void characters(char ch[], int start, int length) {
479  String chars = new String(ch, start, length);
480 }
481 }
```