UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

NADJIA JANDT FELLER

Geração de Testes de Aceitação a Partir de Modelos U2TP para Sistemas Web

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

Prof. Dr. Marcelo Soares Pimenta
Orientador

Porto Alegre, Março de 2015.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Feller, Nadjia Jandt

Geração de Testes de Aceitação a Partir de Modelos U2TP para Sistemas Web / Nadjia Jandt Feller. – 2015.

77 f.:il.

Orientador: Marcelo Soares Pimenta.

Dissertação (Mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2015.

1. Geração automatizada de testes. 2. Testes de aceitação para sistemas *web*. 3. U2TP. 4. Automação de testes. I. Pimenta, Marcelo Soares, orient. III. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitor de Pós-Graduação: Prof. Vladimir Pinheiro do Nascimento
Diretor do Instituto de Informática: Prof. Luís da Cunha Lamb
Coordenador do PPGC: Prof. Luigi Carro
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

AGRADECIMENTOS

Gostaria de agradecer a Daniel, Lauro, Mirian e Gustavo por todo o amor, apoio, dedicação e companheirismo, e pela compreensão nas ausências e crises.

Á minha família e aos meus amigos, pelos momentos felizes e tudo o que tenho a oportunidade de aprender com cada um.

Ao meu orientador, Marcelo Pimenta, pela orientação e apoio durante todo este período do mestrado, e por sempre ter acreditado neste trabalho.

Ao pessoal com quem já pude trabalhar nas equipes ELSA Brasil e LINDA Brasil, que me ajudaram a crescer profissional e pessoalmente. E todos do CPD da UFRGS, pela acolhida e pelos desafios propostos a cada dia.

RESUMO

A atividade de teste durante o desenvolvimento de *software* é fundamental para a busca da qualidade e da confiabilidade do *software*, encontrando defeitos, para que estes sejam removidos. Porém, apesar da sua importância, realizar testes se mostrou ser algo custoso, difícil e problemático quando feito de maneira inadequada.

Um novo paradigma para o desenvolvimento de testes de *software* é o teste dirigido por modelos (do inglês Model Driven Testing, MDT), que pode ser definido como teste de *software* onde os casos de teste são derivados de um modelo que descreve alguns aspectos do sistema sendo testado, como, por exemplo, comportamento. Essa descrição, geralmente utilizando diagramas UML e/ou seus perfis, pode ser processada para produzir um conjunto de casos de teste.

Especificações de *software* baseadas em cenários de uso utilizando diagramas UML adequados são consideradas expressivas e efetivas, pois descrevem os requisitos do sistema de uma maneira intuitiva e visual. Assim, podem ser utilizadas para a descrição de testes de aceitação, que validam se o sistema satisfaz os requisitos de usuário. Estas especificações também facilitam a automação deste tipo de teste. A automação de testes pode oferecer um ganho ao diminuir o tempo gasto com os testes e com isso diminuir o custo desta atividade.

Assim, este trabalho propõe uma abordagem para geração automatizada de testes de aceitação para sistemas *web* a partir de diagramas U2TP (UML 2.0 Testing Profile, o perfil de teste da UML), baseado no paradigma de desenvolvimento orientado por comportamento (BDD), obtendo cenários de aceitação e código de teste executável em um formato compatível com um *framework* para automação de testes de aceitação. Foi feita também a adoção desta abordagem em um contexto real de desenvolvimento, como exemplo de aplicação, através da realização de um experimento.

A utilização desta abordagem no ciclo de desenvolvimento de uma aplicação *web* traz algumas vantagens, como ser necessário gerar manualmente apenas o modelo de comportamento de cada funcionalidade da aplicação, (pois os demais artefatos são gerados automaticamente), consumindo menos tempo e estando menos sujeitos a erros, além de prevenir diferentes interpretações dos requisitos pelos *stakeholders*, desenvolvedores e testadores. O tempo despendido na especificação dos modelos é compensado pelo tempo economizado com a geração dos cenários e do código de testes.

Palavras-Chave: Geração automatizada de testes, Testes de aceitação para sistemas *web*, U2TP, Automação de testes.

Acceptance Tests Generation from U2TP Models for Web Applications

ABSTRACT

The testing activity throughout software development is fundamental to the pursuit of software quality and reliability, finding faults to be removed. However, despite its importance, software testing is often an underutilized phase in software development. Moreover, tests are proved to be expensive, difficult and problematic when not done in the appropriate way.

A new paradigm for software testing is model-driven testing (MDT), which can be defined as software testing where test cases are derived from a model that describes some aspects of the system being tested, such as behavior, for example. This description, often using UML diagrams and/or its profiles, can be processed to produce a set of test cases.

Software specifications based on usage scenarios expressed by appropriate UML diagrams are considered significant and effective, because they describe the system's requirements from an intuitive and visual perspective. Thus, they can be used for the description of acceptance tests, which validate that the system meets user requirements. These specifications also facilitate the automation of this kind of test. Test automation can decrease time spent on testing, thereby reducing the cost of this activity.

Thus, this work proposes an approach for automated generation of acceptance tests from U2TP (the UML 2.0 test profile) diagrams for web applications, based on behavior driven development (BDD) paradigm, obtaining acceptance scenarios and executable test code supported by an acceptance testing automation framework. This approach was applied on an actual development environment, by means of an experiment.

Using this approach in an web application development cycle has some advantages, such as being required only to manually generate the model of behavior of each application functionality (because other artifacts are generated automatically), thus being less time consuming and less prone to errors, and preventing different interpretations of requirements by stakeholders, developers and testers. The time spent at the models' specification is compensated by the time saved with the generation of scenarios and test code.

Keywords: Automated test generation, Web applications acceptance tests, U2TP, Test automation.

LISTA DE ABREVIATURAS E SIGLAS

AJAX	Asynchronous Javascript and XML
ATDD	Acceptance Test Driven Development
BDD	Behavior Driven Development
CORBA	Common Object Request Broker Architecture
DOM	Document Object Model
FIT	Framework for Integrated Test
IDE	Integrated Development Environment
IDL	Interactive Data Language
MBT	Model Based Testing
MDA	Model Driven Architecture
MDD	Model Driven Development
MDT	Model Driven Testing
OCL	Object Constraint Language
OMG	Object Management Group
SUT	System Under Test
TDD	Test Driven Development
UML	Unified Modeling Language
U2TP	UML 2.0 Testing Profile
XMI	XML Metadata Interchange
XML	Extensible Markup Language
XP	Extreme Programming
XPath	XML Path Language

LISTA DE FIGURAS

Figura 2.1 - Ciclo ATDD 19
Figura 2.2 - Fluxo BDD 20
Figura 2.3 - Arquitetura de Teste 26
Figura 2.4 - Comportamento de Teste 27
Figura 2.5 - Dados de Teste 29
Figura 2.6 - Conceitos de Tempo 30
Figura 2.7 - Arquitetura e Comportamento de Teste do Metamodelo baseado no MOF 31
Figura 2.8 - Grupo Dados de teste do Metamodelo baseado no MOF 32
Figura 4.1 - Visão Geral da Abordagem WATGUM 38
Figura 4.2 - Seção do Diagrama de Casos de Uso da Aplicação 41
Figura 4.3 - Diagrama de Estados para estória Acessar Sistema 42
Figura 4.4 - Modelo de Teste para a estória Acessar Sistema 44
Figura 4.5 - Arquivo XMI dos modelos de projeto 46
Figura 4.6 - Arquivo XMI do modelo de teste 47
Figura 4.7 - Formato do cenário de aceitação proposto 48
Figura 4.8 - Estrutura do aplicativo gerador de cenários de teste 49
Figura 4.9 - Cenários de aceitação gerados para a estória Acessar Sistema 50
Figura 4.10 - Código de teste gerado automaticamente 53
Figura 4.11 - Estrutura do aplicativo gerador de código de teste 54
Figura 4.12 - Relatório dos resultados da execução dos testes com JBehave 56
Figura 5.1 - Seção do Diagrama de Casos de Uso que inclui "Cadastro" 61
Figura 5.2 - Protótipo da Tela do Formulário de Cadastro de Participantes 62
Figura 5.3 - Seção do Diagrama de Casos de Uso que inclui "Busca por Participante" 63
Figura 5.4 - Protótipo de Tela de Busca de Participantes com os filtros originais 63
Figura 5.5 - Protótipo de Tela de Busca de Participantes com novo filtro de data de recepção 63
Figura 5.6 - Frequência das respostas à questão 1 do questionário 66

LISTA DE TABELAS

Tabela 4.1 - Mapeamento entre itens dos modelos de projeto e teste 43
Tabela 4.2 - Mapeamento de U2TP para XMI 45
Tabela 5.1 - Objetivo, questões e métricas para os experimentos propostos 59
Tabela 5.2 - Resultados obtidos por cada método para as métricas M1 e M2 para as tarefas T1, T2 e T3 64
Tabela 5.3 - Resultados do teste Mann-Whitney para cada tarefa e métricas M1 e M2 65
Tabela 5.4 - Resultados obtidos para as métricas M3 e M4 65
Tabela 5.5 - Resultado do teste Mann-Whitney para as métricas M3 e M4.................. 66

SUMÁRIO

RESUMO 4
ABSTRACT 5
LISTA DE ABREVIATURAS E SIGLAS 6
LISTA DE FIGURAS 7
LISTA DE TABELAS 8
1 INTRODUÇÃO 11
1.1 Contribuições 13
1.2 Objetivos 13
1.2.1 Objetivos Gerais 14
1.2.2 Objetivos Específicos 14
2 TESTE DE SOFTWARE: FUNDAMENTOS 15
2.1 Planejamento e Projeto de Testes 16
2.2 Testes de Aceitação 16
2.2.1 Automação de Testes de Aceitação 17
2.2.2 Desenvolvimento Guiado por Testes de Aceitação 18
2.2.3 Desenvolvimento Guiado por Comportamento 19
2.3 Testes de Aplicações Web 21
2.4 Testes Dirigidos por Modelos 22
2.4.1 UML 23
2.4.2 O Perfil de Teste da UML (U2TP) 24
2.4.2.1 Arquitetura de Teste 25
2.4.2.2 Comportamento de Teste 26
2.4.2.3 Dados de Teste 27
2.4.2.4 Conceitos de Tempo 29
2.4.2.5 Metamodelo MOF para Teste 30
2.4.3 Modelagem Ágil 32
3 TRABALHOS RELACIONADOS 34
3.1 Geração de Cenários de Teste a Partir de Modelos para Execução Manual 34
3.2 Geração de Cenários de Teste a Partir de Modelos para Execução Automatizada 35
3.3 Geração de Testes de Aceitação Automatizados para Aplicações Web 36
3.4 Síntese 36
4 WATGUM: UMA ABORDAGEM DE GERAÇÃO DE TESTES DE ACEITAÇÃO A PARTIR DE MODELOS U2TP PARA APLICAÇÕES WEB 38
4.1 Modelagem 39
4.1.1 Atividade A1: Modelagem de Aplicação 41
4.1.2 Atividade A2: Modelagem de Teste 42
4.2 Atividade A3: Exportação dos Diagramas para XMI 45
4.3 Geração do Teste de Aceitação 47
4.3.1 Atividade A4: Geração dos Cenários de Teste 47
4.3.2 Atividade A5: Geração do Código de Teste 50
4.4 Atividade A6: Execução dos Cenários de Teste 54
5 AVALIAÇÃO EMPÍRICA 57
5.1 Projeto do Experimento 57
5.1.1 Escopo, Contexto e Participantes 57

5.1.2 Objetivos, Hipóteses, Parâmetros e Variáveis ... 58
5.1.3 Tarefas e Instrumentos ... 59
5.1.4 Unidades Experimentais ... 61
5.2 Resultados e Análise ... 64
5.2.1 Eficiência das abordagens ... 64
5.2.2 Manutenibilidade das abordagens ... 65
5.2.3 Compreensão dos critérios de aceitação ... 67
5.2.4 Análise ... 67
5.2.5 Ameaças à Validade ... 67
6 CONCLUSÃO ... 69
6.1 Limitações do Trabalho ... 70
6.2 Trabalhos Futuros ... 71
REFERÊNCIAS ... 72

1 INTRODUÇÃO

A atividade de teste durante o desenvolvimento é fundamental para a busca da qualidade e da confiabilidade do *software*, encontrando defeitos, para que estes sejam removidos. Porém, a dificuldade intrínseca à atividade de teste, apoiada por diversos fatores como o seu custo, a escassez de bons profissionais e a dificuldade de implantação do processo de teste durante todo o ciclo de vida podem tornar o custo/benefício do teste de *software* impraticável quando não houver um bom planejamento desta atividade. O custo da verificação de *software* geralmente excede metade do custo geral do desenvolvimento e manutenção do mesmo (PEZZÈ, 2008). A atividade de teste é muito importante para o processo de desenvolvimento, pois ao detectar os erros mais rapidamente diminui-se o custo de manutenção, o tempo gasto com retrabalho e aumenta a confiança no *software* sendo desenvolvido.

Porém, apesar da sua importância, o teste do *software* é em geral uma fase subutilizada no processo de desenvolvimento de *software*. Além disso, realizar testes se mostrou ser algo custoso, difícil e problemático quando feito de maneira inadequada. Uma fonte de problemas são as especificações geralmente imprecisas, incompletas e ambíguas, o que faz com que falte uma base bem feita para o teste. Outra fonte de problemas é o fato de os testes serem realizados frequentemente como um processo manual e trabalhoso, sem uma efetiva automação, o que faz com que esteja sujeito a erros e que consuma muitos recursos. A fase de teste geralmente fica “esmagada” entre as datas de entrega de código, que mudam frequentemente, e as datas de entrega ao cliente, geralmente fixas (TRETMANS, 2003). Assim, muita sistematização e controle são necessários para que a atividade de teste deixe de ser uma tarefa totalmente *ad hoc* para se tornar uma atividade de engenharia com resultados efetivos e previsíveis (WAZLAWICK, 2013).

Teste dinâmico de *software* é um tipo de teste que visa executar o *software* de uma maneira controlada com duas metas distintas: demonstrar ao desenvolvedor e ao cliente que o *software* atende aos requisitos especificados; e descobrir falhas ou defeitos no *software,* identificando comportamento incorreto, não desejável ou em não conformidade com sua especificação (SOMMERVILLE, 2007).

Conforme os sistemas de *software* se tornam cada vez mais complexos, novos paradigmas são necessários para a sua construção. Um destes novos paradigmas é o desenvolvimento orientado a modelos (Model-Driven Development, MDD), que já demonstrou impacto na melhoria de qualidade do produto, aumento da produtividade e redução do *time-to-market* (BAKER, 2007) (LUMERTZ, 2009). Este paradigma introduz modelos rigorosos durante todo o processo de desenvolvimento, permitindo abstração e automação. Uma das linguagens utilizadas para isto é a UML (Unified Modeling Language), com a qual os requisitos e projeto do sistema são criados e

visualizados de uma maneira gráfica, permitindo melhor comunicação e validação. UML também permite a introdução de técnicas de automação de geração de *software*.

A partir do MDD surgiu uma nova abordagem, que une o Teste de Software com a Engenharia Dirigida por Modelos, sendo chamada de Teste Dirigido por Modelos (Model-Driven Testing, MDT) ou Teste Baseado em Modelos (Model-Based Testing, MBT). Neste trabalho trataremos MDT e MBT indiscriminadamente e adotaremos a abreviação MDT. MDT possui como principais vantagens em relação aos testes manuais: eficácia similar ou maior na descoberta de defeitos, redução do custo e do tempo dos testes e aumento da cobertura dos testes (BAKER, 2007).

MDT pode ser definido como teste de *software* onde os casos de teste são derivados totalmente, ou em parte, de um modelo que descreve alguns (se não todos) os aspectos do SUT (abreviação de *System Under Test*, o sistema sendo testado) (BAKER, 2007). Para os testes, um modelo fornece uma descrição comportamental do SUT. Essa descrição pode ser processada para produzir um conjunto de casos de teste que pode ser usado para determinar se o SUT está de acordo com uma propriedade desejável que é representada no modelo (BAKER, 2007).

A geração automática de testes a partir de modelos necessita de uma linguagem de especificação que tenha uma semântica, como Máquinas de Estado Finitas (representáveis em UML), por exemplo. O processo de geração automática de testes é muito similar para todos os formalismos (BAKER, 2007). Isto significa que uma especificação é traduzida em um modelo e o algoritmo de geração de teste busca por indícios que demonstrem certas propriedades do modelo. Estes indícios formam a base para o comportamento do teste.

Na literatura há poucas abordagens relacionadas a testes de sistema e aceitação que contam com o auxílio que UML oferece na criação de testes caixa-preta, fazendo com que o testador tenha acesso apenas ao que lhe interessa (BAKER, 2007). Por isto este trabalho tem foco nos testes em nível de aceitação. Teste de aceitação é uma extensão do teste de sistema, no qual os requisitos e os casos de teste associados com sua validação são desenvolvidos pelo cliente (ou em conjunto com este) (BAKER, 2007), ou seja, enquanto o teste de sistema faz a verificação do sistema (defeitos), o teste de aceitação faz sua validação (requisitos) (WAZLAWICK, 2013).

Especificações de *software* baseadas em cenários de uso utilizando diagramas UML adequados são consideradas expressivas e efetivas para serem utilizadas em testes de aceitação, pois descrevem os requisitos do sistema de uma maneira intuitiva e visual. O comportamento expressado nestas especificações representa os objetivos de teste, o que é útil para a geração de casos de teste (NAYAK, 2009). Estes testes devem ser mais robustos contra mudanças da estrutura interna (RUMPE, 2003). Apesar disso, um modelo de fluxo compreensível precisa ser derivado de um diagrama UML, convertendo o diagrama em uma representação mais adequada para a geração de casos de teste.

A automação de testes pode oferecer um ganho ao diminuir o tempo gasto com os testes e com isso diminuir o custo total desta atividade. Esse ganho deve-se à capacidade de diminuição de testes manuais e redundantes, maximizando a confiabilidade dos testes e diminuindo o custo de repetição.

Assim, este trabalho propõe uma abordagem para geração automatizada de testes de aceitação para sistemas *web* a partir de diagramas U2TP, obtendo cenários de aceitação

e código de teste executável em um formato compatível com um *framework* para testes de aceitação que utiliza a abordagem de Desenvolvimento Orientado por Comportamento (Behavior Driven Development, BDD), neste caso, JBehave. Foi feita também a aplicação do processo em um ambiente de desenvolvimento real, como exemplo de aplicação da abordagem, através da realização de um experimento. Espera-se com isto facilitar e tornar mais intuitiva a geração de testes para validação dos requisitos de um sistema.

Escolhemos U2TP pois esta é uma linguagem padronizada para modelagem de arquitetura de teste (estrutura e configuração dos testes), comportamento de teste (aspectos dinâmicos, estímulos, vereditos e ações de teste) e dados de teste (estruturas e significado de valores que serão processados no teste) em UML (MLYNARSKI, 2010).

Este trabalho está organizado da seguinte maneira: no capítulo 2 são apresentados conceitos relacionados a Teste de Software relevantes para este trabalho; no capítulo 3 são apresentados os trabalhos existentes relacionados a este encontrados na literatura; no capítulo 4 é descrita a abordagem proposta, com suas etapas de modelagem, geração dos cenários e código de teste e execução dos casos de teste; no capítulo 5, a descrição de um experimento tratando da aplicação da abordagem proposta em um ambiente real de desenvolvimento; e no capítulo 6, as conclusões obtidas com este trabalho.

1.1 Contribuições

Com este trabalho foi possível obter algumas contribuições:

- Aumento da produtividade no desenvolvimento de aplicações *web*, diminuindo o esforço para desenvolvimento e execução dos testes, pois estes são especificados em uma linguagem de alto nível e gerados automaticamente;
- Através do uso dos modelos, os erros na especificação do projeto são descobertos cedo, diminuindo o custo decorrente da sua correção;
- Melhor definição de critérios de aceitação que devem ser satisfeitos pelo sistema;
- Mais fácil manutenção dos testes do sistema, pois alterações podem ser feitas diretamente nos modelos, e os artefatos são gerados de forma automática novamente;
- Utilização de conceitos e diagramas U2TP para geração de testes automatizados para sistemas *web* compatíveis com um *framework* reconhecido na indústria de desenvolvimento de *software*;
- Apresentar uma maneira de utilizar MDT em conjunto com BDD para automação de testes para sistemas *web*.

1.2 Objetivos

O objetivo deste trabalho é propor uma abordagem para geração de cenários de uso e código executável para testes de aceitação automatizados de sistemas *web*, utilizando como ponto de partida modelos do Perfil de Testes da UML (U2TP), e baseado no paradigma de BDD. Com isto, paradigmas de modelagem, desenvolvimento e teste que são bem aceitos pela comunidade e já demonstraram ter diversos benefícios

isoladamente podem ser integrados de maneira a auxiliar no desenvolvimento (e em particular, nos testes) de sistemas *web*.

1.2.1 Objetivos Gerais

Como objetivos gerais deste trabalho podemos citar:

- Propor uma abordagem para geração automatizada de artefatos de teste de aceitação para sistemas *web* a partir de modelos bem definidos;
- Gerar testes de aceitação de execução automatizada, que serão executados através de um *framework* adequado;
- Ajudar a diminuir o esforço e a aumentar a produtividade do desenvolvimento de testes para aplicações *web*;
- Auxiliar a melhorar a comunicação a diminuir ambiguidades entre equipe de desenvolvimento e *stakeholders* com relação aos requisitos de usuário.

1.2.2 Objetivos Específicos

Os objetivos específicos deste trabalho são:

- Avaliar a viabilidade de modelagem de testes de sistemas *web* utilizando o Perfil de Testes da UML 2.0 (U2TP);
- Propor alguns modelos que serão utilizados na definição do comportamento do sistema e da estrutura de teste, e suas propriedades;
- Realizar a exportação destes modelos para um formato computável e definir um mapeamento entre seus elementos e os elementos de teste;
- Propor uma solução para automatização da geração de cenários de aceitação para estes modelos;
- Propor uma solução para automatização da geração de código de teste de aceitação para esta modelagem;
- Avaliar e escolher um *framework* BDD que possibilite a execução automatizada dos cenários e código de teste gerados, e que forneça um relatório sobre o resultado da execução dos testes;
- Aplicar a solução proposta em um ambiente de desenvolvimento real para validação da abordagem apresentada.

2 TESTE DE SOFTWARE: FUNDAMENTOS

Neste capítulo serão apresentados os fundamentos de Teste de Software (em especial sobre testes de aceitação, testes de aplicações *web* e testes dirigidos por modelos) considerados necessários para o entendimento do trabalho.

É conhecido que o desenvolvimento de testes consome uma porção significativa do esforço geral necessário para o desenvolvimento de sistemas de *software*. Consequentemente, pesquisas sobre teste têm focado em como fazer com que o desenvolvimento e validação de suítes de teste possam ser feitos de maneira simples, mais rápida e menos custosa (SCHIEFERDECKER, 2003a). Bons testes são compreensíveis para os clientes, não são frágeis e testam apenas um único conceito de cada vez (PUGH, 2010).

É importante que sistemas sejam construídos de forma que os testes possam ser realizados adequadamente (RUMPE, 2003). O ideal é que existam, ao longo de todo o processo de desenvolvimento, atividades de validação e verificação. As atividades de validação têm como objetivo contrapor artefatos de *software* com o documento de requisitos, para assegurar que o *software* continue representando-os apropriadamente e também para assegurar que esses requisitos permaneçam atualizados durante todo o processo (checagem de adequação) (MARUCCI, 2002). Todas as mudanças de estado de um sistema devem ser testadas, e cada exceção deve ter seu próprio teste (PUGH, 2010).

Utilizando requisitos de teste e informações detalhadas de projeto, casos de tese, oráculos e *drivers* de teste podem ser desenvolvidos (BRIAND, 2001). Para determinar se um teste passou com sucesso, os resultados da execução devem ser verificados comparando com os resultados aceitos pela especificação (HECKEL, 2003).

O conjunto de estímulos e respostas esperadas pode ser chamado de propósito de teste. Um caso de teste é uma implementação de um propósito de teste (SCHIEFERDECKER, 2003a). Casos de teste podem ser criados a partir da fase de análise do sistema. É possível utilizar artefatos UML de análise para derivar requisitos de teste de sistema, que são uma especificação precisa de quais cenários de teste devem ser executados (BRIAND, 2001).

Um teste é passível de repetição e determinado, ou seja, para a mesma configuração, os mesmos resultados são obtidos (RUMPE, 2003). Teste de Regressão é feito durante a manutenção do *software*, com o propósito de estabelecer a confiança de que este continuará funcionando mesmo depois de ser alterado. Este tipo de teste necessita ser feito tanto para mudanças progressivas (mudanças na especificação e código) quanto para mudanças corretivas (apenas mudanças de código) (WINTER, 1998).

2.1 Planejamento e Projeto de Testes

O processo de teste de *software* é composto por atividades que têm por objetivo executar um programa a fim de revelar seus defeitos e avaliar sua qualidade. As principais atividades do processo de teste são: planejamento, projeto dos casos de teste, procedimento de teste, execução dos testes, avaliação dos resultados dos testes (BIASI, 2006).

O principal objetivo do planejamento de testes é definir informações sobre abrangência, abordagem, recursos e atividades de teste. Durante o desenvolvimento do plano de testes é necessário definir uma estratégia que contém:

- **Níveis de Teste:** O nível de teste é dependente da fase do processo de desenvolvimento de *software* em que o teste é aplicado. Os principais níveis são: Teste de Unidade (unidades de implementação), Teste de Integração (integração das unidades), Teste de Sistema (sistema de *software* funcional) e Teste de Aceitação (teste com relação às especificações do(s) usuário(s)).
- **Técnica de Teste:** A técnica de teste direciona a escolha de critérios para projetos de casos de teste. Os métodos de teste compreendem Teste Estrutural (caixa-branca) e Teste Funcional (caixa-preta). Os testes estruturais conhecem o código-fonte do programa e o testam internamente. Os testes funcionais testam o comportamento do sistema baseado nos requisitos, sem conhecimento da estrutura interna.
- **Critérios:** O critério de teste serve para orientar o testador na geração dos casos de teste. Os critérios de teste são dependentes do método utilizado. No teste caixa-branca podem ser feitos testes de caminhos, de condições, *loops*, etc. e no teste caixa-preta os testes podem ser de limites, partições de equivalência, entre outros.
- **Tipo de Teste:** Os tipos de teste representam as características do *software* a serem testadas. Alguns exemplos são Teste de Funcionalidade, Teste de Interface e Teste de Segurança.

Com base no planejamento e nas especificações do sistema são projetados os casos de teste. Para a atividade de elaboração desses casos de teste, é de grande ajuda o uso de uma linguagem de alto nível, clara e expressiva, a exemplo do perfil de teste da UML (U2TP).

2.2 Testes de Aceitação

Não se deve apenas entregar o *software*, mas entregar *software* que agregue valor de negócio (PUGH, 2010). Teste de Aceitação é uma atividade essencial no ciclo de desenvolvimento de *software*. Seu objetivo é validar que o *software* atende aos requisitos do cliente. Para isto, os testes de aceitação são dirigidos a (e correspondem a) cenários de uso, ou estórias de usuário. Testes de aceitação automatizados podem também servir como parte de uma suíte de testes de regressão, verificando que os requisitos do cliente não são violados pela evolução do *software* (NEGARA, 2012). Além disso, testes de aceitação asseguram que não há mal-entendidos entre os *stakeholders* e os desenvolvedores (KOUDELIA, 2011).

Como alguns dos benefícios dos testes de aceitação, podemos citar (KOUDELIA, 2011):

- Servem como requisitos de *software*;
- Podem ser implementados como especificações executáveis que verificam que o sistema sob teste se comporta como esperado e satisfaz os requisitos;
- Servem como um meio de comunicação sem ambiguidade entre desenvolvedores e os especialistas do domínio que fornecem os requisitos;
- Dão aos *stakeholders* a possibilidade de avaliar o *software* durante toda a fase de desenvolvimento, não apenas quando ele é considerado pronto;
- Auxiliam a estimar o verdadeiro estado de progresso de um projeto e indica quando o *software* está pronto;
- Facilitam a refatoração, sinalizando quando algo deixa de funcionar durante o processo de refatoração;
- Auxiliam desenvolvedores que não estão familiarizados com o *software* a explorá-lo através dos casos de uso executáveis;
- Levam à construção de *software* com boas características de testabilidade, como baixo acoplamento dos módulos.

Para aumentar a estabilidade dos testes de aceitação, é útil seguir alguns padrões para desenvolvimento de modelos de teste, que são similares a padrões de codificação (RUMPE, 2003):

- Em geral um teste de aceitação deve ser abstrato, sem tentar determinar todos os detalhes da parte sob teste do sistema;
- Um oráculo de teste não deve tentar determinar toda a saída e as estruturas de resultado, mas concentrar em detalhes importantes, ignorando valores de objetos e atributos que não são interessantes;
- Não se deve tentar observar interações internas durante a execução do sistema. Isto significa que diagramas são utilizados como *drivers* para testes de aceitação, se concentrando em interações nas camadas mais alto nível do sistema.

2.2.1 Automação de Testes de Aceitação

A automação de testes pode oferecer um ganho ao diminuir o tempo gasto com os testes e com isso diminuir o custo total desta atividade. Esse ganho deve-se à capacidade de uma diminuição de testes manuais e redundantes, maximizando a confiabilidade dos testes e diminuindo o custo de repetição. Testes automatizados asseguram uma baixa taxa de defeitos e progresso contínuo, enquanto testes manuais podem rapidamente levar os testadores à exaustão (RUMPE, 2003).

Testes de aceitação podem ser automatizados. A automatização de todos os testes de aceitação forma uma definição executável do produto. Geralmente testes de aceitação são realizados como testes caixa-preta. O sistema como um todo é configurado como se estivesse em ambiente de produção. Os testes são executados na camada mais alta do sistema (geralmente a interface de usuário) e envolvem todos os componentes da aplicação. Efetivamente, os testes de aceitação são suscetíveis a encontrar a maioria dos erros técnicos e lógicos. Entretanto, executar a funcionalidade através da interface de

usuário é algo relativamente lento, o que prejudica a velocidade de *feedback*. Como a automação exige um baixo nível de ambiguidade, há um risco menor de falta de entendimento entre o desenvolvedor e o cliente (MAJCHRZAK, 2012).

O objetivo do teste é verificar que o sistema se comportará apropriadamente quando uma sequência de ações de usuário ocorrer. Um *script* de teste é definido como uma sequência completa de ações necessárias para criar um cenário de uso completo para o sistema, desde o *setup*, durante todas as ações, até a finalização. Um *script* de teste pode ser decomposto em uma série de primitivas de teste individuais que realizam ações específicas. As primitivas serão combinadas em uma sequência específica para prover um *script* de teste que verifica um cenário de uso único para o produto (APFELBAUM, 1997). Um *driver* de teste aplica os casos de teste ou suítes de teste em uma aplicação (HECKEL, 2003).

Tradicionalmente, automação de teste se refere à automatização da execução do teste, e às vezes à análise de teste. Isto abre caminho para a automação completa de teste, onde o sistema sob teste e suas especificações são os únicos requisitos necessários (TRETMANS, 2003).

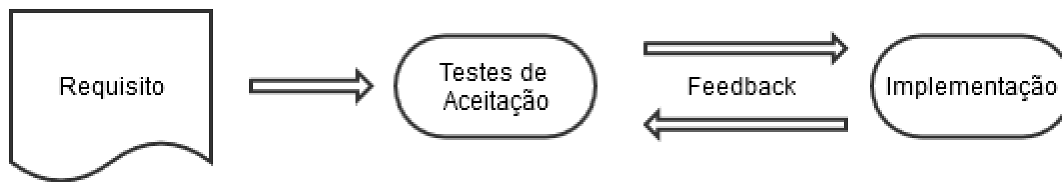
2.2.2 Desenvolvimento Guiado por Testes de Aceitação

Em projetos ágeis, os requisitos são tipicamente capturados como estórias de usuário. Elas definem apenas o escopo de um requisito, não a especificação de fato. A estória de usuário com maior prioridade é escolhida e especificada em forma de um ou mais testes de aceitação (MAJCHRZAK, 2012), que idealmente devem ser escritos antes das unidades de *software* que pretendem testar (WAZLAWICK, 2013). Cenários de estórias e casos de uso sugerem testes de aceitação (PUGH, 2010).

Desenvolvimento Guiado por Testes de Aceitação (Acceptance Test Driven Development, ATDD) é um tipo de Desenvolvimento Guiado por Testes (Test Driven Development, TDD) onde o processo de desenvolvimento é orientado pelos testes de aceitação que são usados para representar os requisitos dos *stakeholders*. ATDD auxilia os desenvolvedores a transformar requisitos em casos de teste e permite verificar a funcionalidade de um sistema. Um requisito é satisfeito se todos os testes ou critérios de aceitação associados a ele são satisfeitos. Em ATDD os testes de aceitação podem ser automatizados. TDD e ATDD são amplamente adotados pela indústria, pois estas técnicas melhoram a qualidade e a produtividade do *software* (SOLÍS, 2011).

Como o nome sugere, ATDD é uma prática orientada a testes, significando que cada ciclo de desenvolvimento começa com a implementação de um teste de aceitação e não da funcionalidade em si (Figura 2.1 (PUGH, 2010)). Apesar do uso de TDD ser uma maneira comum de programar em métodos ágeis, os testes não necessitam sempre preceder a construção do *software*. O conceito essencial é combinar programação e teste em um processo uniforme (KOUDELIA, 2011). Conforme cada estória é criada, é necessário listar seus critérios de aceitação. Estes critérios serão expandidos em testes de aceitação específicos para esta estória (PUGH, 2010).

Figura 2.1 - Ciclo ATDD



Os testes de aceitação pertencem ao cliente; são escritos em conjunto entre cliente, desenvolvedores e testadores; referenciam "o quê" e não "como"; e são expressos na linguagem do problema de domínio, com concisão, precisão e sem ambiguidade. É vital escrever testes de aceitação compreensíveis e legíveis. Como os clientes estão ativamente envolvidos na análise de requisitos, deve ser permitido a eles entender os testes de aceitação. Os testes de aceitação são o meio de comunicação central em ATDD (MAJCHRZAK, 2012).

ATDD possui três objetivos principais. Primeiro, provê um meio de comunicação compartilhada para melhorar a troca de informações entre testadores, desenvolvedores e especialistas do domínio (quem fornece os requisitos de *software*). ATDD não tenta apenas fazer a informação fluir mais facilmente entre estes grupos, mas também dentro de cada grupo. Segundo, ATDD fornece instrumentos para armazenar documentação funcional de *software* que se mantém atualizada durante todo o ciclo de desenvolvimento. Finalmente, ATDD implica que o sistema sendo construído constantemente atenda aos requisitos através de teste automatizado e permaneça funcionando durante as constantes refatorações (KOUDELIA, 2011). Outros objetivos incluem: descobrir requisitos ambíguos e possíveis lacunas mais cedo no processo de desenvolvimento; criar um registro do entendimento entre negócio e desenvolvimento; e dar retorno com relação à qualidade do *software* sendo desenvolvido (PUGH, 2010).

ATDD tenta eliminar a ambiguidade apresentando os requisitos como exemplos de ações do usuário sendo aplicadas ao *software*. Estes exemplos descrevem exatamente como os estados do sistema mudam de um para outro, ou seja, quais valores de entrada são necessários para iniciar a transição de estados e quais valores de saída vão validar o resultado. Esta abordagem se baseia na ideia básica de que requisitos expressos como exemplos concretos e objetivos com valores de entrada e saída definidos deixam menos espaço para mal-entendidos, em comparação às estórias abstratas e subjetivas escritas em linguagem natural (KOUDELIA, 2011).

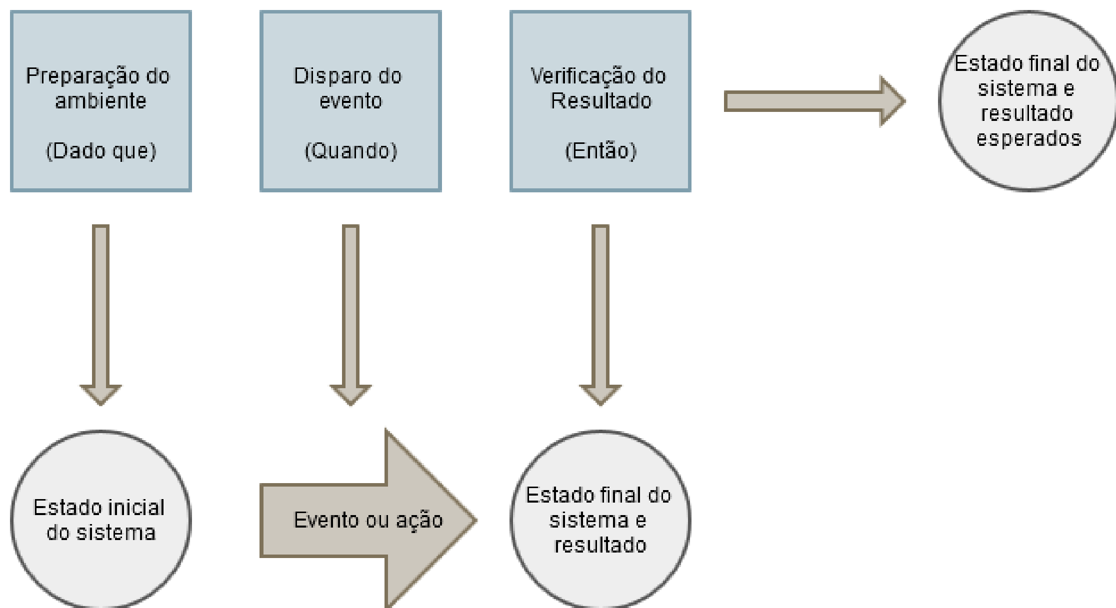
2.2.3 Desenvolvimento Guiado por Comportamento

Desenvolvimento Guiado por Comportamento (Behavior Driven Development, BDD) geralmente é tratado como uma evolução de TDD e ATDD. BDD é focado em definir especificações granulares do comportamento do sistema-alvo, de maneira que possa ser automatizado. O objetivo principal de BDD é conseguir especificações executáveis de um sistema. BDD se baseia em ATDD, mas em BDD os testes são claramente escritos e de fácil compreensão, pois BDD provê uma linguagem ubíqua que auxilia os *stakeholders* a especificar seus testes (SOLÍS, 2011). Há vários *frameworks*

que suportam BDD, como JBehave (JBEHAVE, 2013), Cucumber (HELLESOY, 2013) e RSpec (RSPEC, 2013).

North (2006) propôs o termo Desenvolvimento Guiado por Comportamento (BDD) e introduziu a primeira ferramenta BDD, JBehave, que foi criada por ele mesmo. JBehave enfatiza comportamento sob teste através da utilização de um vocabulário específico. Especificações BDD normalmente seguem a notação "Dado que - Quando - Então": Dado que o sistema sobre teste está em um estado "A", Quando evento "X" ocorre, Então o estado do sistema se torna "B" (KOUDELIA, 2011). BDD cria um contexto para popular um estado conhecido no sistema, então um evento ocorre neste contexto, que realiza o verdadeiro comportamento do cenário. Finalmente passa o controle para os resultados definidos para a estória (Figura 2.2) (PUGH, 2010).

Figura 2.2 - Fluxo BDD



Uma estória de usuário pode ter diferentes versões em diferentes contextos. As instâncias específicas de uma estória de usuário são chamadas de cenários. Cenários devem descrever contextos específicos e resultados da estória de usuário, que devem ser fornecidos pelos clientes. Cenários em BDD são utilizados como critérios de aceitação (SOLÍS, 2011). O comportamento de uma estória é representado pelos seus critérios de aceitação. Se o sistema satisfaz todos os critérios de aceitação, então está se comportando da maneira desejada. E se não satisfaz, não está correto (NORTH, 2006).

Seis características principais de BDD podem ser citadas (SOLÍS, 2011):

- **Linguagem ubíqua:** ou seja, uma linguagem cuja estrutura vem de um modelo de domínio. Isto permite que clientes e desenvolvedores falem a mesma linguagem sem ambiguidade;
- **Processo iterativo de decomposição:** Clientes precisam que algo com valor de negócio seja realizado por um projeto de *software*. O comportamento do sistema será derivado a partir dos resultados de negócio que se pretende

produzir. Destes resultados de negócio são extraídos conjuntos de funcionalidades. Uma funcionalidade é realizada por uma estória de usuário. As instâncias específicas de uma estória de usuário são chamadas de “cenários”. Cenários devem descrever contextos específicos e resultados da estória de usuário, o que deve ser provido pelos clientes;

- **Descrição em texto livre com *templates* de estórias e cenários de usuário:** Descrições de estórias de usuário e *templates* de cenários devem ser escritos na linguagem ubíqua definida para o projeto. Estas descrições são mapeadas para testes diretamente, o que significa que os nomes das classes e métodos também devem ser escritos nesta linguagem ubíqua;
- **Teste de aceitação automatizado com regras de mapeamento:** Cenários são traduzidos para testes que vão guiar a implementação. Um cenário é composto de diversos passos, que representam contextos, eventos e ações. Em BDD todos os cenários são executados automaticamente, o que significa que os critérios de aceitação devem ser importados e analisados automaticamente. Classes que implementam os cenários vão ler especificações em texto e executá-las. Regras de mapeamento proveem um padrão para mapear cenários para código. No *framework* JBehave, uma estória de usuário é um arquivo contendo um conjunto de cenários. O nome do arquivo é mapeado para uma classe de estória de usuário. Cada passo do cenário é mapeado para um método de teste localizado usando uma anotação descrevendo este passo, e um método de teste;
- **Código compreensível de especificação orientado a comportamento:** BDD sugere que o código deve fazer parte da documentação do sistema, o que está alinhado com os valores de métodos ágeis. As especificações devem fazer parte do código;
- **Orientação a comportamento em diferentes fases:** Desde o planejamento inicial, passando pela análise, implementação e teste, todas as fases de desenvolvimento são orientadas a comportamento. Testes de aceitação automatizados são uma parte integral da implementação da abordagem BDD.

O *template* de representação dos critérios de aceitação deve permitir que a estória seja dividida em fragmentos que podem ser automatizados, mas também não pode ser restritivo demais à análise. Eles são então descritos em termos de cenários. Os fragmentos do cenário (contexto, evento e resultados) possuem granularidade suficiente para serem representados diretamente em código. JBehave define um modelo de objeto que permite o mapeamento direto dos fragmentos do cenário para classes Java (NORTH 2006).

2.3 Testes de Aplicações Web

A evolução rápida e a crescente popularidade das tecnologias da Web 2.0 recentemente impulsionaram o desenvolvimento de aplicações *web* dinâmicas complexas. Teste de *software*, uma tarefa já complexa, é ainda mais desafiadora no contexto das aplicações *web* modernas, devido às suas características. Várias abordagens sugerem a automação dos testes de aplicações *web* dinâmicas através da exploração dos seus comportamentos. Alguns destes métodos não são adequados para a

tarefa de testes de aceitação, pois geram casos de teste baseados na análise exaustiva do estado das estruturas DOM (Document Object Model) dinâmicas de uma aplicação *web*, sem controlar o comportamento pretendido. Outros *frameworks* e métodos têm sido propostos que podem ser adotados para testes de aceitação, entretanto requerem expertise substancial em programação ou não possuem soluções completas de teste para aplicações *web* baseadas em AJAX. Outro problema que precisa ser tratado envolve a especificação do comportamento esperado da aplicação *web* e um mecanismo que o compare com o real comportamento desta (NEGARA, 2012).

Testes efetivos de uma aplicação *web* devem se basear em estratégias de teste bem definidas, definindo heurísticas ou algoritmos para criar casos de teste a partir de modelos de teste, especificando o escopo dos testes, definindo uma sequência de atividades de teste e outras decisões de processo (DI LUCCA, 2002).

Geralmente a execução de um caso de teste para uma aplicação *web* pode ser realizada de acordo com um processo que inclui os seguintes passos: inicialização do ambiente que será testado; execução do caso de teste em si; retornar a aplicação ao estado inicial, liberando recursos para evitar degradação do teste após sucessivas execuções; e avaliação dos resultados obtidos com relação aos esperados (DI LUCCA, 2002).

2.4 Testes Dirigidos por Modelos

O teste de *software* em geral é a primeira parte do ciclo de desenvolvimento de *software* que é omitida quando há uma limitação de tempo para entrega. Em outras palavras, os desenvolvedores podem não ter tempo suficiente para criar os casos de teste e testar o *software* ao terminar a codificação. A geração automatizada de casos de teste a partir de modelos pode ajudar a resolver estes problemas. Ela não apenas auxilia os desenvolvedores a testar o sistema quando eles finalizam a programação, mas também incentiva os desenvolvedores a implementar o *software* como foi definido na sua especificação (JEEVARATHINAM, 2010).

O uso de modelos para descrever o comportamento de um sistema é uma grande vantagem para equipes de desenvolvimento. Modelos podem ser utilizados de diversas maneiras durante o ciclo de vida do produto, incluindo: melhorar a qualidade das especificações, geração de código, análise de confiabilidade e geração de testes. Modelagem é um meio econômico para capturar conhecimento sobre um sistema e reusar este crescimento conforme o sistema cresce (APFELBAUM, 1997).

Enquanto a automação de testes baseada em *scripts* tenta melhorar a eficiência da execução dos testes e dos testes de regressão através da execução automatizada dos testes, testes dirigidos por modelos focam também em melhorar a eficiência do projeto de teste. Em particular, MDT trata das seguintes tarefas: a geração de casos de teste a partir de modelos (geralmente especificados em UML), de acordo com critérios definidos; a geração de oráculos de teste para determinar os resultados esperados; e a execução de testes em ambientes de teste, possivelmente também gerados a partir de modelos (MLYNARSKI, 2010).

Desenvolver uma especificação em forma de modelos é um meio muito efetivo de: descobrir defeitos no sistema; definir os cenários de uso de um sistema de maneira rápida; preservar o investimento para *releases* futuros ou sistemas similares. O processo de desenvolver um modelo pode ser feito em uma série de pequenos passos

incrementais (APFELBAUM, 1997). Os modelos, se criados durante a fase de definição de requisitos, podem prevenir diferentes interpretações dos requisitos pelos desenvolvedores e testadores. Minimizar estas diferenças irá evitar que alguns defeitos sequer cheguem à fase de execução do teste, ajudando a diminuir os custos de teste (CLARKE, 1998).

Há várias abordagens que podem ser usadas para desenvolver testes a partir de modelos de comportamento do sistema. Algo comum a todas é o conceito de caminho. Um caminho é uma sequência de eventos ou ações que passam pelo modelo definindo um cenário de uso real do sistema. Um gerador de código de teste pode automaticamente encontrar caminhos válidos pelo modelo. Cada caminho válido do modelo é convertido em um caso de teste através da substituição de cada transição do caminho por suas informações de teste. Então um caso de teste completo é a concatenação de toda a informação de teste para um caminho válido. Uma vez que um caminho através do modelo foi definido, um *script* de teste pode ser criado para este caminho. Quando este *script* é aplicado no sistema real, este seguirá a mesma sequência, ou caminho, como definido pelo caminho do modelo do qual foi extraído. Este processo pode ser repetido para outro caminho, o qual define outro cenário de uso e verifica outra sequência de ações. Vários modelos podem ser utilizados para selecionar caminhos, cada um com seus objetivos e vantagens distintos (APFELBAUM, 1997). Estes casos de teste podem ser produzidos em qualquer linguagem (CLARKE, 1998).

A abordagem MDT oferece considerável garantia de redução do custo da geração de testes, aumento da eficácia dos testes, e diminuição do ciclo de teste. A geração de testes pode ser especialmente efetiva para sistemas que mudam frequentemente, pois os testadores podem atualizar o modelo de dados e então rapidamente gerar novamente a suíte de testes, evitando a edição manual dos testes, que é sujeita a erros (DALAL, 1999).

A notação ideal para o modelo deve: ser fácil para testadores entenderem, descrever sistemas de diversas complexidades, e ainda ter um formato que possa ser tratado por uma ferramenta de geração de teste. É importante que o modelo represente apropriadamente o sistema sob teste. Especialistas do domínio devem participar da sua confecção, ou realizar verificações sobre os modelos. Uma abordagem iterativa para a construção do modelo também pode ser efetiva (DALAL, 1999).

Um conjunto de ferramentas MDT é composto por: um gerador, que entende os modelos e gera descrições de casos de teste que atendam os objetivos de teste, e tem seu comportamento definido pelo tipo de modelo que deve entender; um componente responsável por gerar os casos de teste concretos (que irão interagir com o sistema sob teste), e que extrai os dados de teste que fazem parte destes casos de teste, logo este deve ter conhecimento tanto do modelo quanto do sistema sob teste; e um *driver* de teste (KRISHNAN, 2009).

2.4.1 UML

UML é uma linguagem visual de modelagem de propósito geral que é usada para especificar, visualizar, construir e documentar os artefatos de um sistema de *software* (HARTMANN, 2000). Técnicas de modelagem com UML têm uma grande aceitação na comunidade de desenvolvimento de *software*. É possível envolver os desenvolvedores

no processo de definições de teste se a linguagem de modelagem permite a integração de informações relacionadas a teste (SCHIEFERDECKER, 2003).

A arquitetura dirigida por modelos (Model Driven Architecture, MDA) da OMG (Object Management Group) busca padronizar o uso de linguagens de descrição, como UML. O desenvolvimento da versão 2 de UML, alinhado com a estratégia de MDA, teve como um de seus objetivos possibilitar a "execução" de UML, permitindo não só a geração de código, simulação e validação dos modelos, mas também a geração de testes. Esse objetivo permitiria uma melhor implementação de um processo de teste baseado em modelos.

Uma questão importante é a testabilidade, ou seja, o quanto um modelo (neste caso um diagrama UML) tem informação suficiente para permitir a geração automática de casos de teste. É necessário então tratar dos requisitos de testabilidade que precisam ser incluídos nos artefatos UML (e consequentemente no método de desenvolvimento) para permitir o suporte eficiente a testes funcionais de sistema (BRIAND, 2001).

Casos de uso são uma boa fonte para derivar requisitos de teste de sistema, pois eles representam em alto nível as funcionalidades que o sistema provê ao usuário (BRIAND, 2001).

Um Diagrama de Estado UML pode ser usado para descrever o comportamento dinâmico de um sistema (ou partes dele) ao longo do tempo, sendo um guia para projetar testes funcionais (HOLT, 2014). Os estados e transições do diagrama definem todos os estados e mudanças de estados que pode acontecer durante o ciclo de vida do sistema. Mudanças de estados ocorrem como reações a eventos recebidos das interfaces do sistema (HARTMANN, 2000) (KANSOMKEAT, 2003). Transições em um diagrama de estados UML são parametrizadas por ações. Para fazer com que um diagrama destes se torne "executável", uma semântica operacional deve ser definida para a linguagem usada para descrever as ações (PICKIN, 2002).

2.4.2 O Perfil de Teste da UML (U2TP)

O perfil de teste da UML define uma linguagem para projetar, visualizar, especificar, analisar, construir e documentar os artefatos dos sistemas de teste. É uma linguagem de modelagem que pode ser utilizada com as principais tecnologias de componentes e objetos e aplicada para testar sistemas em vários domínios de aplicação (OMG, 2005). O perfil de teste da UML pode ser usado somente para a manipulação dos artefatos de teste ou de uma maneira integrada com UML para a manipulação conjunta de um sistema com seus respectivos artefatos de teste.

Este perfil de teste é baseado na especificação da UML 2.0, e foi criado seguindo estes princípios de projeto: integração com a UML; e reuso (dos conceitos da UML) e minimalismo (acrescentar apenas o necessário). Permite a especificação de testes para aspectos estruturais (estáticos) e comportamentais (dinâmicos) de modelos computacionais UML e é capaz de interoperar com tecnologias de teste existentes para testes caixa-preta.

U2TP estende a UML com conceitos específicos de teste que são agrupados em conceitos de arquitetura de teste, dados de teste, comportamento de teste, e tempo. Por ser um perfil da UML, ele se integra facilmente com esta, pois é baseado no seu metamodelo e reusa sua sintaxe.

A criação do U2TP foi baseada no metamodelo MOF (Meta-Object Facility), o padrão da OMG para a construção de metamodelos. É uma especificação que define uma linguagem abstrata para metamodelagem e um *framework* para especificação, construção e gerenciamento de metamodelos independentes de plataforma. Exemplos de sistemas que usam o MOF incluem ferramentas de modelagem e desenvolvimento, sistemas *data warehouse*, repositórios de metadados, entre outros (OMG, 2006).

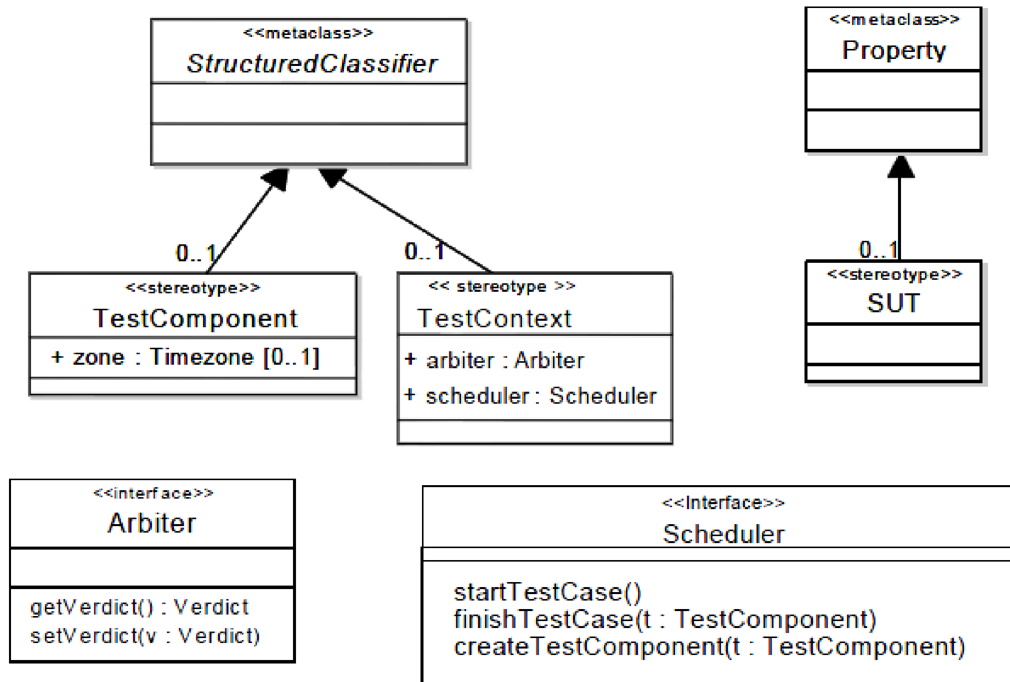
U2TP pode ser usado isoladamente para a manipulação de artefatos de teste ou de uma maneira integrada com o resto da aplicação, manipulando simultaneamente artefatos de sistema e de teste. Havendo um modelo de projeto UML, é possível especificar testes para um sistema estendendo este modelo com os conceitos U2TP (DAI, 2004). Primeiramente, criando um novo pacote UML de teste, importando os elementos necessários do modelo de projeto do sistema, e após, especificando os elementos relativos aos conceitos da U2TP.

2.4.2.1 Arquitetura de Teste

Arquitetura de teste é um conjunto de conceitos para especificar os aspectos estruturais de um contexto de teste cobrindo componentes de teste, o sistema sob teste, sua configuração, etc. A Figura 2.3 (OMG, 2005) descreve os seus principais elementos. São eles:

- **TestContext:** Um *TestContext* (contexto de teste) é uma classe que representa o agrupamento de vários casos de teste, ou seja, representa o conceito conhecido como suíte de teste. A notação para o elemento *TestContext* é uma classe com o estereótipo <<TestContext>>.
- **TestComponent:** Um *TestComponent* (componente de teste) é uma classe de um sistema em teste. *TestComponents* interagem com o SUT ou com outros *TestComponents* para realizar os casos de testes que são definidos dentro do *TestContext*. A notação para o elemento *TestComponent* é uma classe com o estereotipo <<TestComponent>>.
- **SUT:** O SUT é o sistema, subsistema ou componente sendo testado. O SUT é exercitado através de suas operações públicas pelos *TestComponents*. Nenhuma informação a mais pode ser obtida do SUT, pois este é uma caixa-preta. A notação para o SUT é nomear o que se deseja testar com o estereotipo <<Sut>>.
- **Arbiter:** Uma propriedade de um caso de teste ou um *TestContext* para avaliar os resultados do teste e designar o *Veredict* de um caso de teste ou *TestContext*, respectivamente. Há um algoritmo padrão de arbitragem baseado em testes funcionais e de conformidade, que gera "*Pass*", "*Fail*", "*Inconc*" e "*Error*" como *Veredict*. Este algoritmo pode ser definido pelo usuário.
- **Scheduler:** Uma propriedade de um *TestContext* utilizado para controlar a execução de diferentes *TestComponents*. O *Scheduler* vai ter a informação sobre quais *TestComponents* existem em qualquer ponto no tempo, e vai colaborar com o *Arbiter* para informá-lo quando o *Veredict* final deve ser dado. Controla a criação e destruição de *TestComponents* e sabe quais fazem parte de cada caso de teste.

Figura 2.3 - Arquitetura de Teste



2.4.2.2 Comportamento de Teste

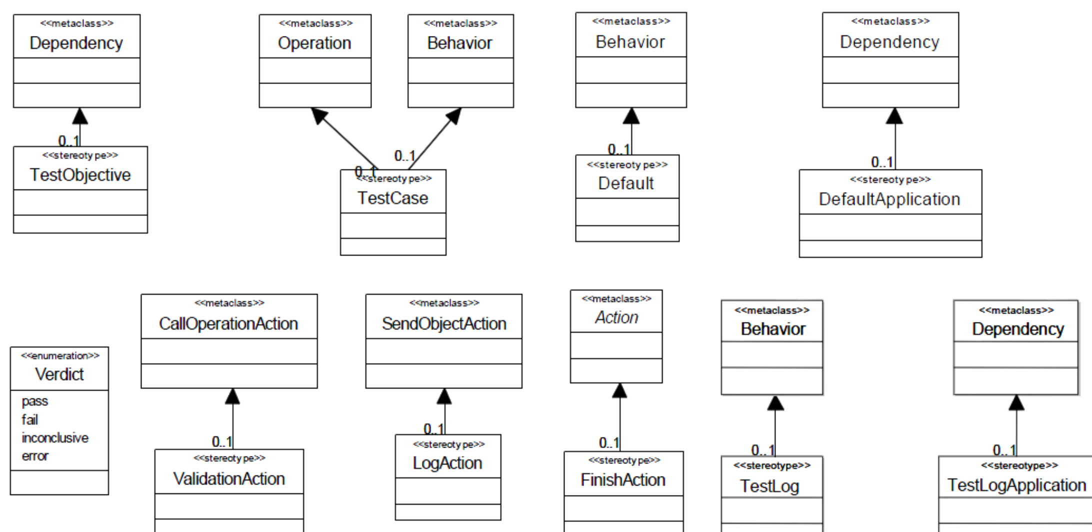
Comportamento de Teste é o conjunto de conceitos que especificam comportamentos de teste, seus objetivos e a avaliação de sistemas sob teste. Este grupo define os conceitos necessários para representar todos os elementos que fazem parte dos aspectos dinâmicos dos procedimentos de teste. A Figura 2.4 (OMG, 2005) ilustra os principais elementos. São eles:

- **TestControl:** Um *TestControl* é uma especificação para a invocação de casos de teste em um *TestContext*. É uma especificação técnica de como o SUT deve ser testado no dado *TestContext*. Permite especificar a ordem de execução dos casos de teste.

- **TestCase:** Um *TestCase* é uma especificação de um caso para testar o sistema incluindo o quê testar, qual a entrada, o resultado e sob quais condições. Ele implementa um objetivo de teste. O *TestCase* utiliza um *Arbiter* para avaliar o resultado do seu comportamento de teste. É uma operação que especifica como um conjunto de *TestComponents* cooperativos interagindo com um SUT realizam um objetivo de teste. Um *TestCase* pertence a um *TestContext*. O tipo de retorno de um caso de teste deve ser um *Verdict*. Se o estereótipo de um caso de teste for aplicado em uma operação, a classe desta operação tem que ter estereótipo *TestContext* aplicado. Mas, se o estereótipo de um caso de teste for aplicado a um comportamento, o comportamento desse caso de teste tem que ter o estereótipo *TestContext* aplicado. O estereótipo não pode ser aplicado a ambos. A notação para um caso de teste é uma operação com o estereótipo <<TestCase>>.

- **TestObjective:** Um *TestObjective* (objetivo de teste) é um elemento que descreve o que deve ser testado, e está associado a um *TestCase*.
- ***Default*:** É um comportamento disparado por uma observação de teste que não é tratado pelo comportamento do caso de uso em si. *Defaults* são executados por *TestComponents*.
- **Veredict:** *Veredict* é a avaliação da corretude do SUT, produzida pelos casos de uso. *Veredicts* podem também ser utilizados para reportar defeitos no sistema de teste. *Verdict* é um tipo de dados *enumeration* pré-definido que contém os valores "*pass*" (o sistema está correto para este caso de uso), "*fail*" (o propósito do teste foi violado), "*inconclusive*" (resultado inconclusivo) e "*error*" (erro no teste ou na sua execução). *Veredicts* podem ser definidos pelo usuário e são calculados pelo *Arbiter*.
- **ValidationAction:** Uma ação para avaliar o estado da execução de um *TestCase* através da avaliação das observações do SUT e/ou características ou parâmetros adicionais do SUT. É realizada por um *TestComponent* e define o *Veredict* local para aquele *TestComponent*.
- **LogAction:** Uma ação para registrar uma informação no *TestLog*.
- **TestLog:** Um *log* é uma interação resultante da execução de um caso de teste. Ele representa as mensagens trocadas entre os *TestComponents* e o SUT e/ou os estados dos *TestComponents* envolvidos. É associado com um *Veredict* representando a aderência de um SUT ao objetivo de teste do caso de teste associado.

Figura 2.4 - Comportamento de Teste



2.4.2.3 Dados de Teste

O grupo de dados de teste define a sintaxe e semântica dos dados usados como entrada e saída dos procedimentos de teste. Dados explícitos e classes de equivalência

formam conjuntos de dados; essas classes de equivalência especificam regras para formação de dados de entrada ou saída. Valores coringas (*wildcards*), em adição aos já presentes em UML padrão, são oferecidos. Os elementos seletores de dados (*data selectors*) são usados para facilitar a criação de estratégias de teste. A Figura 2.5 (OMG, 2005) ilustra os elementos principais do grupo de dados de teste. São eles:

- **Wildcard:** Permite que o usuário especifique explicitamente se o valor está presente ou não, e/ou se é de algum valor. *Wildcards* são símbolos especiais que representam valores ou intervalos de valores. Seus valores possíveis são: "*Any*" (qualquer valor), ou "*AnyOrNull*" (qualquer ou nenhum valor - omitido).

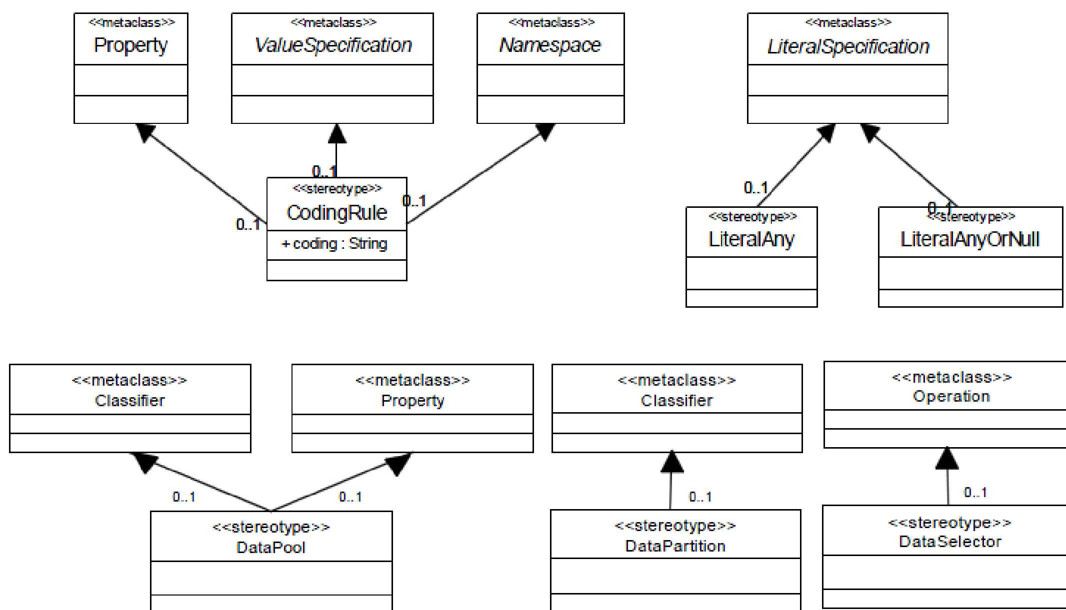
- **DataPool:** É uma coleção de *DataPartitions* ou valores explícitos que são utilizados em um *TestContext*, ou *TestComponent*, durante a avaliação dos contextos e casos de teste. Um *DataPool* contém um ou vários *DataPartition*, mas só pode estar associado a um *TestContext* ou um *TestComponent*. A notação para um *DataPool* é uma classe estereotipada com <<DataPool>>.

- **DataPartition:** Um valor lógico para um parâmetro utilizado em um estímulo ou em uma observação. Geralmente define uma classe de equivalência para um conjunto de valores ou tipo de dados. A notação para o elemento *DataPartition* é uma classe com o estereótipo <<DataPartition>> aplicado, e deve estar associado somente a um *DataPool* ou outro *DataPartition*.

- **DataSelector:** Uma operação que define como valores de dados ou classes de equivalência são selecionados de um *DataPool* ou *DataPartition*. A notação para o elemento *DataSelector* é uma operação estereotipada com <<DataSelector>>.

- **CodingRule:** As interfaces de um SUT utilizam certas codificações (CORBA, IDL ou XML, por exemplo) que têm que ser respeitadas pelos sistemas de teste. Então, *CodingRules* são parte de uma especificação de teste.

Figura 2.5 - Dados de Teste



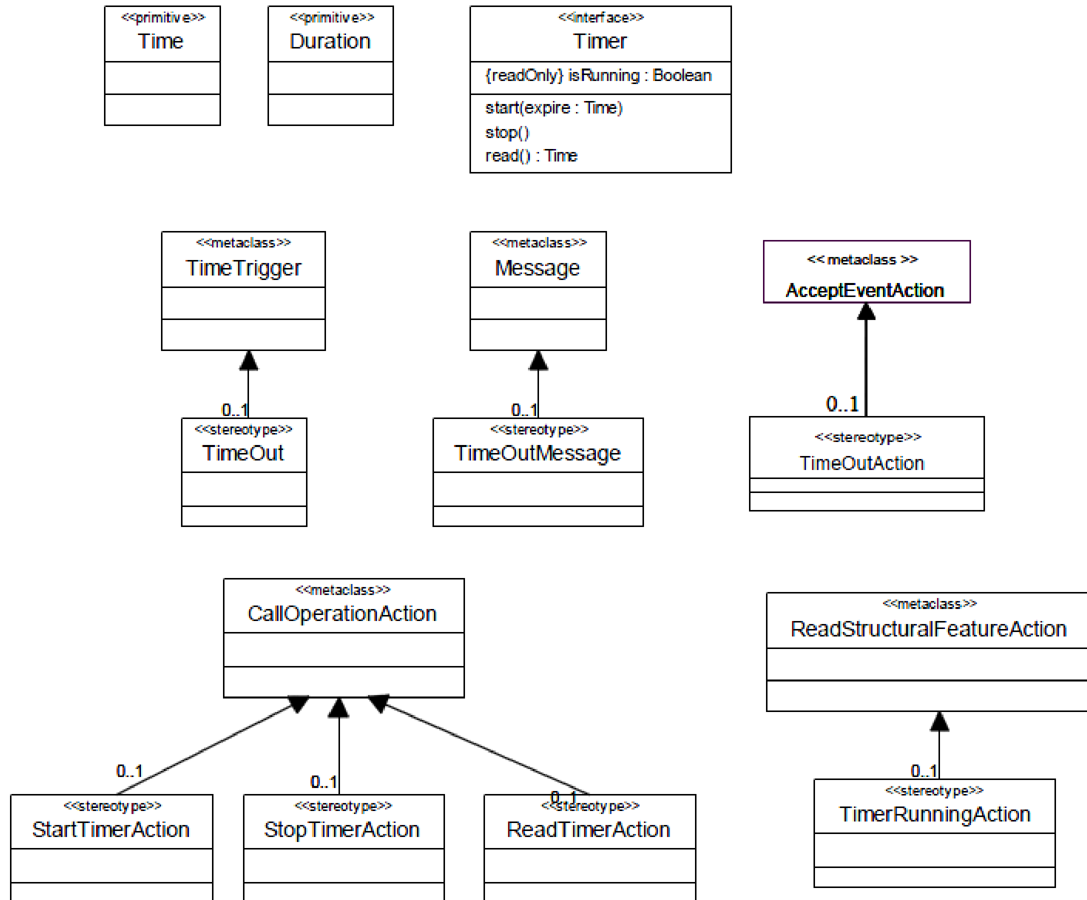
2.4.2.4 Conceitos de Tempo

Conceitos de tempo são um conjunto que especifica restrições de tempo, observações de tempo e/ou *timers* nas especificações de comportamento de teste de maneira a ter uma execução de teste com tempo quantificado e/ou a observação da execução temporizada de casos de testes. A Figura 2.6 (OMG, 2005) ilustra estes conceitos. São eles:

- **Timer**: *Timers* são mecanismos que podem gerar um evento de *timeout* quando um valor de tempo especificado ocorrer. Pode ser quando um intervalo de tempo pré-especificado expira relativo a um instante dado. *Timers* pertencem a componentes de teste e são definidos como propriedades destes. Um *timer* pode ser parado. O tempo para expiração e o estado de um *timer* podem ser verificados.

- **Timezone**: É um mecanismo de agrupamento para *TestComponents*. Cada *TestComponent* pertence a no máximo um *Timezone*. *TestComponents* que pertencem ao mesmo *Timezone* têm o mesmo tempo, ou seja, estão sincronizados no tempo.

Figura 2.6 - Conceitos de Tempo



2.4.2.5 Metamodelo MOF para Teste

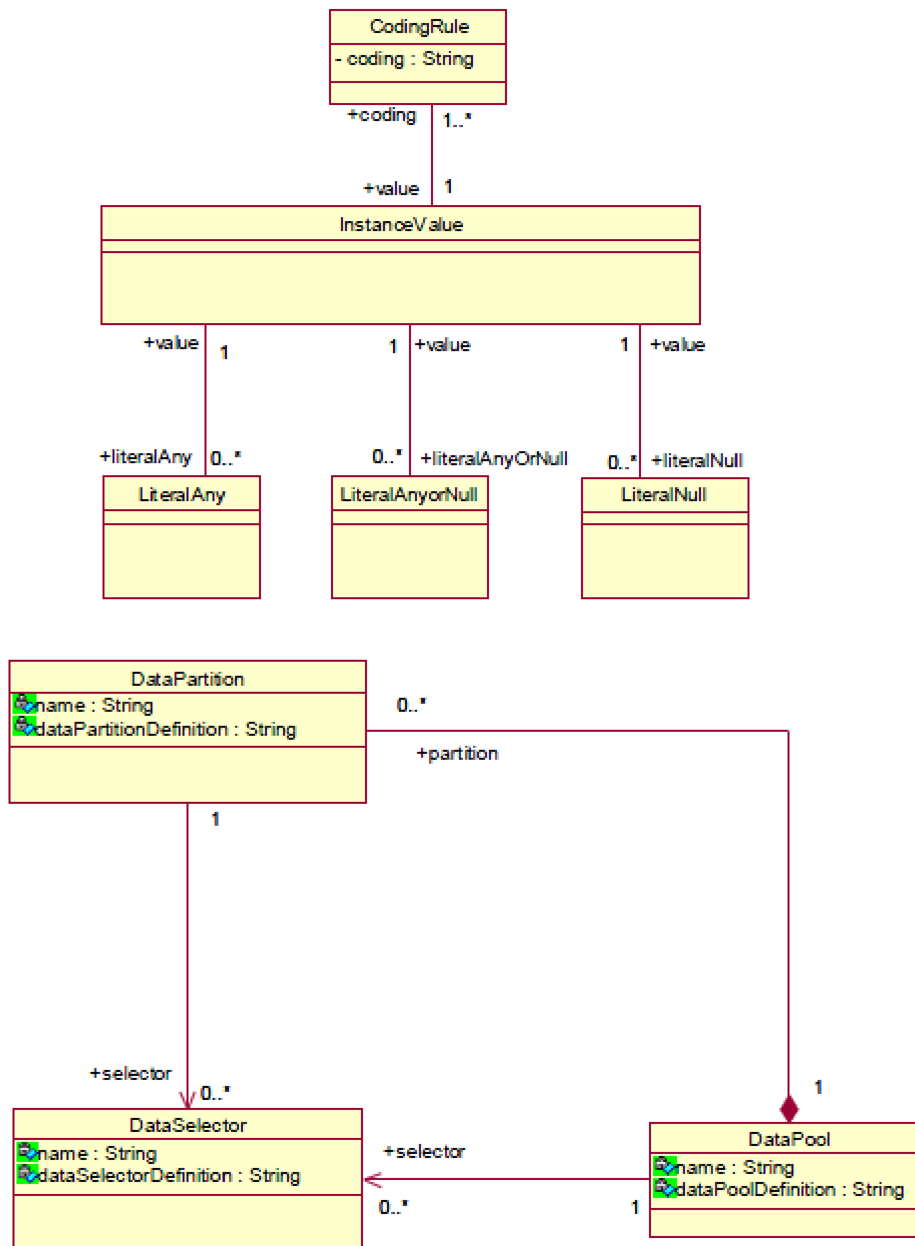
Esta seção apresenta um metamodelo padrão para o Perfil de teste da UML 2.0, o qual é uma instância do MOF. Este metamodelo é limitado aos elementos da arquitetura e comportamento de teste e é apresentado na Figura 2.7 (OMG, 2005). A maioria dos elementos do Perfil de Teste apresentados anteriormente (Seções 2.4.2.1 a 2.4.2.4) também estão presentes no metamodelo.

Figura 2.7 - Arquitetura e Comportamento de Teste do Metamodelo baseado no MOF

O elemento central dessa figura é o elemento *TestContext*, que tem associação com todos os demais elementos. Ele representa o passo inicial para especificação da parte de arquitetura de teste, e a partir dele, os demais elementos são instanciados e utilizados.

A Figura 2.8 (OMG, 2005) mostra os elementos principais do grupo de dados de teste. O elemento central desse grupo é o *DataPool*. O elemento *DataPartition* é usado obrigatoriamente sempre que o elemento *DataPool* for usado, pela associação de agregação entre esses dois elementos. O elemento *DataSelector*, deve estar associado a no máximo um *DataPool* e um *DataPartition*.

Figura 2.8 - Grupo Dados de teste do Metamodelo baseado no MOF



2.4.3 Modelagem Ágil

Se for desejado trabalhar com casos de uso de negócio para criar a visão do produto, então estórias de usuário podem ser derivadas de casos de uso, através da divisão de cada cenário do caso de uso em uma ou mais estórias. As estórias são então incluídas no Product Backlog (PICHLER, 2010).

Estórias de usuário e casos de uso são técnicas poderosas para capturar requisitos. Ambas as técnicas colocam o cliente ou usuário no centro do esforço de desenvolvimento. Casos de uso descrevem como um cliente interage com o produto utilizando um ou mais cenários. Um caso de uso pode consistir em uma curta narrativa ou de vários cenários estruturados incluindo o cenário principal de sucesso, cenários

alternativos e cenários de falha. Estórias de usuário descrevem como um cliente ou usuário aplica o produto. Estórias consistem de um nome, uma breve narrativa e um conjunto de critérios de aceitação. Estes formulam condições que devem ser satisfeitas, fazendo com que a estória se torne mais precisa. Casos de uso e estórias de usuário empregam papéis ao cliente conhecidos como atores primários e papéis de usuário respectivamente. Ambas as técnicas podem ser aplicadas em níveis diferentes de granularidade: casos de uso podem ser escritos no nível de sistema ou negócio, estórias de usuário podem ser formuladas como épicos ou estórias detalhadas (PICHLER, 2010).

Estórias de usuário são ótimas para capturar a funcionalidade de um produto a partir da perspectiva de um usuário ou cliente. Cada estória de usuário descreve uma parte da funcionalidade de um produto, por exemplo: "Como um fornecedor da aplicação, Eu quero me registrar ao centro da aplicação, Para que eu possa utilizar os seus serviços". Por focar em uma área distinta da funcionalidade, a estória permite à equipe entender, implementar e testar o requisito (PICHLER, 2011).

Diagramas não listam todos os épicos contidos no Product Backlog, e não especificam todos os papéis de usuário. Eles focam nos itens relevantes para uma conversação entre o Product Owner e a equipe, ou a equipe e os *stakeholders*. Isto resulta em um diagrama que é simples e fácil de entender, ao invés de algo complexo (PICHLER, 2011).

Modelagem de estórias de usuário é uma ferramenta útil para o Product Owner. Mas como qualquer ferramenta, ela precisa ser aplicada apropriadamente (PICHLER, 2011):

- Criar modelos colaborativamente: a modelagem de estórias de usuário serve para criar um entendimento compartilhado da funcionalidade desejada para o produto. Contexto de uso e diagramas de atividade capturam a essência de uma conversação, mas não a substituem. Os diagramas devem ser criados e atualizados colaborativamente envolvendo a equipe e, quando apropriado, os *stakeholders*;

- Foco: a modelagem deve ser aplicada seletivamente e somente descrever aspectos relevantes. O foco deve estar nos itens relevantes do Product Backlog. Muita informação e detalhes desnecessários não devem ser incluídos;

- Manter simples: Os diagramas devem ser mantidos simples e fáceis de compreender. É melhor utilizar diagramas adicionais para ilustrar outros aspectos do que ocupar um modelo com informação demais. Diagramas complexos geralmente não são úteis;

- Utilizar quadros e cartões mais do que ferramentas eletrônicas para iniciar a modelagem: Ferramentas físicas simples encorajam a participação e a colaboração. Elas evitam a impressão de completude e perfeição.

Uma das características importantes de alguns métodos ágeis é que testes automatizados são utilizados em todos os estágios. A experiência prática mostra que quando isto é feito apropriadamente, a taxa de defeitos é consideravelmente baixa (RUMPE, 2003).

3 TRABALHOS RELACIONADOS

Neste capítulo são descritas sucintamente algumas abordagens que geram, a partir de modelos e de forma automática, tanto testes executáveis automaticamente quanto cenários de teste para execução manual.

Primeiramente, são apresentadas abordagens de geração de cenários de teste a partir de modelos para execução manual. Em seguida, as abordagens de geração de cenários de teste a partir de modelos para execução automatizada, e finalmente abordagens para geração de testes de aceitação automatizados para aplicações *web*.

3.1 Geração de Cenários de Teste a Partir de Modelos para Execução Manual

Uma abordagem proposta por Naresh (2002) transforma requisitos funcionais em diagramas de fluxo e realiza uma cobertura de caminhos para gerar casos de teste. O resultado do processamento é uma tabela com todos os caminhos de execução possíveis extraídos do diagrama e expressos em linguagem natural. Porém este trabalho não define um formato padrão ou regras para a definição dos requisitos funcionais.

Outra abordagem gera os cenários de teste a partir de árvores de estado, com alguma automatização das etapas intermediárias (TSAI, 2003). Já Pickin (2007) utiliza diagramas de estado para modelar a aplicação e diagramas de sequência UML para a modelagem dos cenários, utilizando técnicas formais para a geração de casos de teste, também representados como diagramas de sequência.

UMLTGF (LINZHANG *et al.*, 2004) é uma ferramenta para geração automatizada de casos de teste a partir de diagramas de atividade. Ela percorre o diagrama de atividades realizando uma cobertura básica de caminhos, restringindo que laços sejam executados no máximo uma vez. Hartmann *et al.* (2005) também descreve uma abordagem para gerar testes em nível de sistema a partir de diagramas de atividade. Nesta, os diagramas de atividade são manualmente anotados antes da geração dos casos de teste. As anotações auxiliam a determinar diferentes variáveis e possíveis escolhas de dados para estas variáveis. Os casos de teste são então gerados considerando todos os caminhos do diagrama anotado. Estes casos de teste podem ser convertidos em *scripts* ou procedimentos executáveis de teste. Já a abordagem de Nayak (2009) identifica e extrai cenários de teste de diagramas de atividade exportando-os para um documento XML e depois os transformando num Modelo Intermediário de Teste, utilizando regras de classificação de estruturas de controle, que tratam também de possíveis agrupamentos no diagrama de atividade. Outros trabalhos ainda realizam a geração de planos de teste a partir de diagramas de atividade UML que modelam os processos de negócio, utilizando modelos de fluxo intermediários como auxílio à geração dos casos

de teste (HEINECKE, 2010), com possível otimização dos casos de teste gerados, utilizando algoritmos genéticos (JENA, 2014). A abordagem apresentada por Philips (2014) gera casos de teste a partir de diagramas UML de estado ou atividade, utilizando tabelas intermediárias para representação e processamento dos modelos.

GenTCase (IBRAHIM ET AL., 2007) é uma ferramenta para geração automática de casos de teste que suporta diagramas de caso de uso. Cada caso de uso deve ter como complemento uma descrição tabular em linguagem natural do seu fluxo de eventos, incluindo pré-condições, pós-condições, caminhos alternativos e um diagrama de sequência UML. Depois de analisar os diferentes caminhos, o resultado é um arquivo texto com os casos de teste gerados. A abordagem de Sarma (2007) combina diagramas de caso de uso e diagramas de sequência em um grafo de teste de sistema que, em conjunto com um diagrama de caso de uso estendido e um dicionário de dados em OCL, é utilizado para a geração de casos de teste de sistema. Utilizando diagramas UML de estado, o trabalho de Ali (2014) gera casos de teste com auxílio de diagramas UML de caso de uso e OCL.

STUSD (NAYAK, 2012) utiliza diagramas de sequência para sintetizar cenários de teste. Estes diagramas na UML 2.0 podem ser compostos por diversos fragmentos e agrupamentos, o que dificulta a extração de um fluxo de controle. Esta abordagem auxilia na representação dos fluxos de controle e seu uso na síntese de cenários de teste, simplificando os fluxos para primitivas de controle UML 2.0 e transformando-os em um grafo de cenários, que por sua vez é transformado em Modelos Intermediários de Teste, satisfazendo um conjunto de critérios de cobertura. Os casos de teste são gerados no formato de uma lista de nodos (ou agrupamentos de nodos) do grafo de cenários. Esta abordagem permite automações posteriores, porém estas não foram tratadas pelo trabalho.

3.2 Geração de Cenários de Teste a Partir de Modelos para Execução Automatizada

Outras abordagens tratam explicitamente da geração de cenários de teste que podem ser executados automaticamente. A metodologia de teste de sistema TOTEM é baseada nos artefatos produzidos durante a fase de análise. Estes artefatos incluem: Diagrama de casos de uso, descrições de casos de uso, diagramas de sequência ou colaboração para cada caso de uso, diagramas de classe compostos de classes da aplicação de domínio, um dicionário de dados que descreve cada classe, método e atributo. Além disso, assume que cada classe é caracterizada por um invariante de classe descrito em OCL e cada operação é descrita por um contrato em OCL, detalhando pré e pós-condições (BRIAND, 2001).

O uso de cenários e casos de uso para a geração de casos de teste executáveis utilizando uma ferramenta própria é a abordagem desenvolvida por Riebish (2002). Nebut (2006) utiliza diagramas de caso de uso UML como ponto de partida para a geração de cenários de aceitação executáveis em JUnit, com auxílio de diagramas de sequência que especificam objetivos de teste. Outra abordagem utiliza, além de diagramas UML, auxílio de especificações formais para a geração de casos de teste, e o *framework* de testes de aceitação FIT para representação dos testes de aceitação (MAFRA, 2008). Já Holt (2014) gera, a partir de diagramas de estado UML, *scripts* executáveis de casos de teste em linguagem C++, utilizando para isto transformações de modelos que foram implementadas como extensões à ferramenta TRUST. Diagramas

de estado também foram utilizados para a geração de casos de teste exploratórios, com auxílio de OCL (SCHAEFER, 2014).

ISTA (Integration and System Test Automation) é uma ferramenta para geração e execução automatizada de código para testes funcionais utilizando redes de Petri como modelo de estados finitos (XU, 2011). Esta gera testes executáveis a partir de modelos de descrição da implementação, que incluem a rede de Petri e o respectivo mapeamento para as estruturas da implementação.

3.3 Geração de Testes de Aceitação Automatizados para Aplicações Web

Algumas abordagens aproveitam-se das estruturas das aplicações *web* para auxiliar na geração automática dos testes de aceitação. CrawlScripter (NEGARA, 2012), por exemplo, suporta três tipos de instruções: eventos, especificações de entrada e asserções. A maioria das ações realizadas pelos usuários de aplicações *web* pode ser representada como comandos que consistem de um verbo, que corresponde à ação executada, e um ou dois substantivos que são os argumentos sobre os quais a ação será executada. Instruções de eventos representam tipos de evento de clique da página e como argumento recebem o nome do elemento que será clicado. Especificação de entrada exige o nome na árvore DOM de um elemento de entrada e o valor que preencherá o campo. Asserções são usadas para verificar a presença ou não de um elemento da árvore DOM da página, seguido de uma mensagem de falha.

Já outra abordagem (MAJCHRZAK, 2012) combina MDSD (Model Driven Software Development) e TDD com a utilização do *framework* para desenvolvimento *web* Spring Roo para a geração de testes automatizados. Foi desenvolvida uma biblioteca dedicada, integrada com JUnit 4 que auxilia na abstração dos elementos DOM da página, focando em testes de aceitação orientados a negócio.

A abordagem proposta por Colombo (2014) se aplica quando os cenários de aceitação já existem (em linguagem Gherkin), mas a combinação da execução destes cenários é desejada. Para isto, o conjunto de cenários de teste é transformado em um diagrama de estados, que por sua vez é transformado em *scripts* de teste, pela ferramenta MBT QuickCheck, para uso com o *framework* Selenium Webdriver. Por outro lado, a abordagem de Madhavan (2014) gera código de teste (Selenium Webdriver) a partir de roteiros de teste escritos em inglês imperativo, utilizando técnicas de processamento de linguagem natural.

3.4 Síntese

A partir da descrição destas abordagens, nota-se que, com relação a testes de aceitação, inclusive para sistemas *web*, algumas abordagens geram, a partir de modelos e de forma automática, tanto testes executáveis automaticamente quanto cenários de teste para execução manual.

Em geral o processo de geração adotado por estas abordagens é similar ao aplicado neste trabalho, com algumas variações, principalmente quanto aos diagramas utilizados e ao processo intermediário para transformação dos modelos em cenários de aceitação ou casos de teste executáveis. Porém a maioria destes trabalhos foi elaborada focando em uma aplicação específica, um domínio restrito, ou o uso de técnicas e ferramentas próprias, tornando estas soluções restritas ao contexto em que foram construídas. São

necessárias, portanto, soluções que possibilitem tanto a aplicação em diferentes contextos, quanto o uso de ferramentas já existentes e bem aceitas na indústria de desenvolvimento de *software*. O uso de MDT, principalmente alguns diagramas UML e U2TP específicos, em conjunto com BDD permitiriam um nível de abstração adequado para evitar estas restrições. WATGUM é uma abordagem que pode ajudar a satisfazer estes critérios.

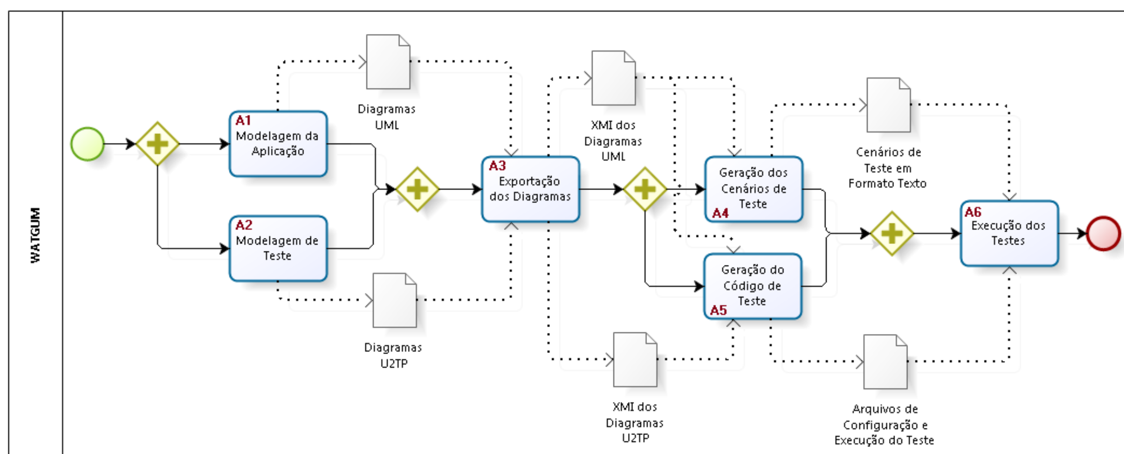
4 WATGUM: UMA ABORDAGEM DE GERAÇÃO DE TESTES DE ACEITAÇÃO A PARTIR DE MODELOS U2TP PARA APLICAÇÕES WEB

Neste capítulo é descrita WATGUM (Web Acceptance Test Generation from U2TP Models, em português, Geração de Testes de Aceitação Web a partir de Modelos U2TP), a abordagem proposta por este trabalho, apresentando as atividades que a compõem e os passos necessários para sua realização.

O teste orientado a modelos depende de três tecnologias principais: a notação utilizada para o modelo, o algoritmo de geração de testes e as ferramentas responsáveis pela geração da infraestrutura de apoio aos testes. Diferente da geração da infraestrutura de testes, a notação do modelo e o algoritmo de geração de teste são portáveis entre projetos (DALAL, 1999).

O processo básico para os diversos tipos de abordagens para a geração de testes a partir de modelos geralmente é: elaboração do(s) modelo(s) (projeto e/ou teste), exportação do(s) modelo(s) para um formato mais facilmente computável do que o gráfico, e a geração de código através de um gerador especializado. Uma visão geral da abordagem proposta por este trabalho pode ser vista na Figura 4.1.

Figura 4.1 - Visão Geral da Abordagem WATGUM



As premissas que nortearam esta abordagem são:

- Utilizar modelos criados em uma linguagem alto nível, que representem estrutura e comportamento de teste;
- Usar o perfil U2TP e UML para especificação do projeto e dos casos de testes;
- Ter o código necessário para execução dos casos de teste gerado automaticamente;
- Possibilitar execução automática dos testes;
- Possibilitar a verificação automática dos resultados dos testes.

Conforme a Figura 4.1, a abordagem WATGUM introduz algumas atividades, explicadas a seguir. Em primeiro lugar, deve ser feita a modelagem da aplicação e de suas funcionalidades, descrevendo suas estruturas e comportamentos (Atividade A1). Uma funcionalidade (ou estória de usuário) deve ser escolhida por vez para a realização do processo completo. Esta modelagem é algo a ser feito em conjunto com o Product Owner ou os *stakeholders* do projeto, gerando assim, diagramas UML que representam uma funcionalidade da aplicação de cada vez. Após, o testador cria o modelo de testes para a funcionalidade em questão (Atividade A2), descrevendo o contexto e as estruturas necessárias para confecção e execução do mesmo, segundo os padrões definidos pela U2TP, gerando, portanto, um diagrama deste perfil.

Estes diagramas são então exportados através da própria ferramenta de modelagem para o formato XMI, facilitando o seu processamento e, por fim, a geração dos casos de teste (Atividade A3).

Com isto é possível, então, a geração dos testes de fato. Utilizamos o arquivo XMI dos diagramas da aplicação para a geração dos cenários de teste, que serão disponibilizados em formato texto (Atividade A4). Além disso, este arquivo XMI é utilizado em conjunto com o XMI do diagrama de teste para gerar o código responsável pela execução dos testes (Atividade A5).

Após a geração de todos estes artefatos, é possível realizar a execução automatizada dos casos de teste para a estória em questão (Atividade A6). Esta execução tem seus resultados avaliados de forma também automatizada, possibilitando ter, ao final, um relatório sobre a execução dos testes sobre a aplicação.

Nas próximas seções, cada passo da abordagem será descrito em maiores detalhes, com auxílio de um exemplo para melhor compreensão da aplicação do processo.

4.1 Modelagem

Em testes de aceitação, é verificado se o sistema sob teste está em conformidade com os requisitos estabelecidos através de casos de teste que exploram o comportamento normal e excepcional do sistema. É recomendado também que, sempre que possível, os requisitos sejam representados como especificações UML (casos de uso, diagramas de classe, atividades, interações, estados, etc.), para poderem ser aproveitados durante esta fase de teste.

A arquitetura dirigida por modelos da OMG busca padronizar o uso de linguagens de descrição, como UML. O desenvolvimento da versão 2.0 de UML, alinhado com a estratégia de MDA (Model Driven Architecture), teve como um de seus objetivos possibilitar a "execução" de UML, permitindo não só a geração de código, simulação e

validação dos modelos, mas também a geração de testes. Esse objetivo permitiria uma melhor implementação de um processo de MDT.

A definição de uma semântica formal e executável para UML ainda está sob investigação. Portanto, não há uma abordagem universal para a geração automática de teste a partir de modelos UML. No entanto, abordagens específicas com base em diferentes modelos semânticos existem. Estes são baseados em abordagens como a utilização de sequências características de estados e transições de estado para orientar o desenvolvimento do teste para os elementos do modelo, tais como diagramas UML de estado e atividades.

A base da aplicação de UML em testes de nível de aceitação é utilizar especificações UML de requisitos de usuário. Os diagramas que podem ser utilizados incluem: casos de uso, que representam as funcionalidades principais do sistema; atividade, que descrevem processos de negócio; interação, que descrevem comunicação; e diagramas de estado, que tratam do comportamento (BAKER, 2007).

Portanto, para melhor compreensão da aplicação, tanto com relação ao que e como deve ser feito, como com relação ao entendimento do que é desejado com o *software*, uma prática imprescindível é a modelagem do sistema. Esta modelagem começa juntamente com a elicitação de requisitos do sistema com os *stakeholders* e Product Owners e prossegue durante todo o ciclo de desenvolvimento, sendo aprofundado para cada funcionalidade escolhida para ser desenvolvida.

Primeiramente é necessário definir o escopo do sistema e suas funcionalidades básicas, em alto nível, além dos atores que irão interagir com o sistema. Para isto, utilizamos o diagrama UML de Casos de Uso. Cada caso de uso pode gerar uma ou mais estórias em um processo ágil (PICHLER, 2010).

Após, para cada estória de um caso de uso, são criados dois modelos: comportamental e estrutural de teste. O modelo comportamental é representado através de um Diagrama UML de Estados que descreve o comportamento desejado pelos *stakeholders* para a funcionalidade descrita pela estória. A partir deste diagrama, os cenários de aceitação são extraídos.

Como o foco deste trabalho é o comportamento do sistema, pois é isto que interessa ao aplicar os testes de aceitação, foi escolhido o diagrama de estados para representar o comportamento das funcionalidades do sistema. É esperado que tanto os comportamentos básicos quanto os excepcionais e alternativos estejam descritos no modelo, para que testes realmente eficazes possam ser gerados a partir deste, pois o sistema deve se comportar de maneira adequada tanto nos casos de sucesso quanto na ocorrência de defeitos. Além disso, o escopo de cada diagrama deve se resumir ao comportamento de apenas uma funcionalidade (caso de uso ou estória de usuário) por vez, pois tratar com modelos referentes a sistemas complexos inteiros seria impraticável.

A partir disto, a estrutura necessária para criação e execução do teste de aceitação é modelada através de um Diagrama de Classes U2TP estrutural de teste, onde os elementos que compõem esta estrutura de teste são representados por classes, instâncias e seus atributos, métodos e relacionamentos. A partir deste modelo, em conjunto com o modelo de comportamento, o código de teste é gerado.

Para possibilitar a geração automatizada e satisfatória de cenários e código de teste a partir dos modelos propostos, estes devem possuir algumas propriedades: ser tão

completos quanto possível com relação aos critérios de aceitação, prevendo cenários de sucesso e de falha; descrever ações possíveis e fluxo da funcionalidade; e observar alguns detalhes que permitem a manipulação das estruturas contidas neles (não ter acentos, nomes iguais para itens correspondentes, etc.). Estes itens serão explicados com mais detalhe nas próximas seções, utilizando um exemplo de aplicação.

Diversas ferramentas *open source* e proprietárias disponíveis para modelagem UML e exportação para XMI foram avaliadas. Destas foi escolhida a ferramenta Enterprise Architect versão 10.0 para uso neste trabalho, pois ela atende necessidades de modelagem UML 2.0 e exportação para XMI 2.0, possui versão disponível para *download*, está participando de uma tentativa de padronização do formato XMI (OMG, 2010) e finalmente por familiaridade com a ferramenta. Esta é uma ferramenta comercial de autoria da Sparx Systems (SPARX, 2013).

4.1.1 Atividade A1: Modelagem de Aplicação

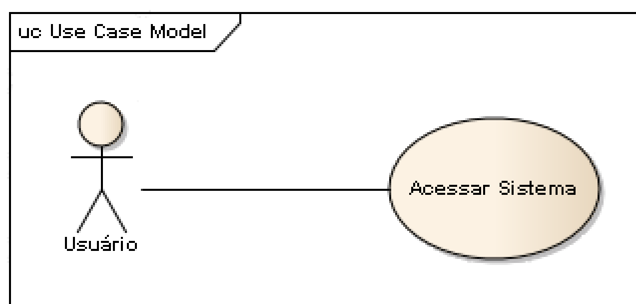
Começamos a modelagem a partir da definição do escopo do sistema e suas funcionalidades básicas, em alto nível, além dos atores que irão interagir com o sistema, utilizando, então, o Diagrama de Casos de Uso.

Optou-se nesta abordagem pela utilização apenas do diagrama de casos de uso, pois utilizar casos de uso estendidos (textuais) tornaria a descrição da funcionalidade redundante (pois já estará coberta em outros modelos), além de tornar a modelagem em si menos ágil, e tornaria muito mais complexa a extração dos cenários de aceitação (pois seria feita a partir de texto livre).

Após a definição do escopo geral da aplicação, a cada iteração de desenvolvimento, as prioridades são definidas, e as estórias criadas a partir dos casos de uso são analisadas individualmente para serem de fato implementadas. Cada caso de uso pode gerar uma ou mais estórias de usuário.

No exemplo da Figura 4.2, utilizamos uma seção do diagrama de casos de uso de uma aplicação *web* genérica, focando no caso de uso "Acessar Sistema", onde o usuário, através de um conjunto de parâmetros, poderá acessar a aplicação. Este caso de uso gera apenas uma estória de usuário: "Como um usuário, Eu quero acessar o sistema *web*, Para que eu possa ter acesso às funcionalidades disponíveis".

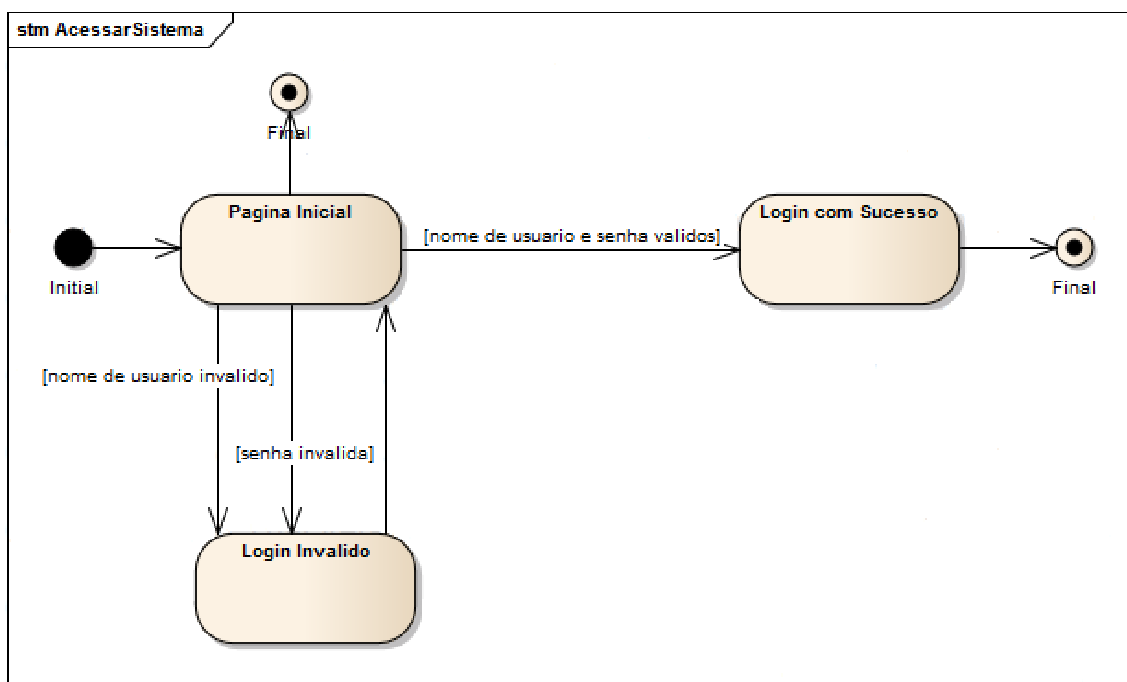
Figura 4.2 - Seção do Diagrama de Casos de Uso da Aplicação



Para cada estória de usuário, obtemos com os *stakeholders* ou o Product Owner informações sobre o comportamento desejado para a funcionalidade que esta estória

compreende. Este comportamento é descrito através de Diagramas UML de Estados. Cada caminho deste diagrama de estados define um critério de aceitação e um cenário de uso da estória. Os cenários sem sucesso também são descritos, para avaliar se o sistema se comporta de maneira adequada diante de situações de falha. Na Figura 4.3 temos um exemplo de diagrama de estados para a estória "Acessar Sistema".

Figura 4.3 - Diagrama de Estados para estória Acessar Sistema



De acordo com os preceitos da modelagem de sistemas *web*, neste diagrama temos representadas as páginas *web* (ou variações dentro das mesmas) como estados e as ações possíveis sobre elas (ou *links* que podem ser acessados) como transições. Porém, por limitações das ferramentas que manipulam o XMI extraído deste diagrama, tanto os nomes dos estados quanto das transições não podem possuir acentos nem caracteres especiais.

Ações que o usuário pode realizar sobre a aplicação são descritas pelos nomes das transições. Já as ações automáticas do sistema também podem ser descritas por transições no modelo, porém estas devem ter seus nomes omitidos, pois não fazem parte do escopo de ações cobertas pelos testes de aceitação, que tratam apenas de ações do usuário sobre o sistema.

Com estes modelos definidos, é possível então extrair o XMI correspondente aos diagramas, e a partir deste, gerar os cenários de aceitação para o teste. Detalhes sobre este processo serão descritos nas seções 4.2 e 4.3.1.

4.1.2 Atividade A2: Modelagem de Teste

Após a concepção dos modelos UML de projeto da estória, é necessário definir a estrutura de teste necessária para geração e execução dos testes de aceitação. Esta é

representada através de um modelo estrutural U2TP, mais especificamente um diagrama de classes.

Para auxiliar a especificação de um metamodelo de teste U2TP foi feito um mapeamento entre as estruturas de projeto e teste. Este mapeamento pode ser visto na Tabela 4.1.

Tabela 4.1 - Mapeamento entre itens dos modelos de projeto e teste

Modelos de Projeto	Modelo de Teste
Caso de Uso (estória)	*TestContext*
Cenário de Aceitação (caminho no Diagrama de Estados)	*TestCase*
Sistema *web*	*SUT*
Página(s) que faz(em) parte do teste	*TestComponent*
Estado ao final do teste	*TestVeredict*
Ator do caso de uso	métodos no *TestComponent*
Conjunto de possíveis dados de teste	*DataPool*
Dados de teste definidos pelas transições do Diagrama de Estados	*DataPartition*

A estória descrita pelo caso de uso é mapeada como uma classe com estereótipo <<TestContext>> no diagrama de classes U2TP. Esta classe deve existir e ser única no modelo e deve conter pelo menos uma operação com o estereótipo <<TestCase>>. Cada caminho no diagrama de estados, que define um cenário de aceitação, é representado por um método estereotipado <<TestCase>> da classe *TestContext*. Os atributos do sistema *web* relevantes para o teste são encapsulados em uma classe com estereótipo <<SUT>>, representando o sistema sob teste.

Para a realização de todos os passos do cenário de teste, será necessário passar por uma ou mais páginas do sistema. Estas páginas devem estar representadas como classes no modelo de teste, utilizando o padrão Page Object (SELENIUM, 2013a). Este tipo de classe possui todos os atributos e métodos necessários para manipular os itens da página, e neste modelo deve possuir o estereótipo <<TestComponent>>.

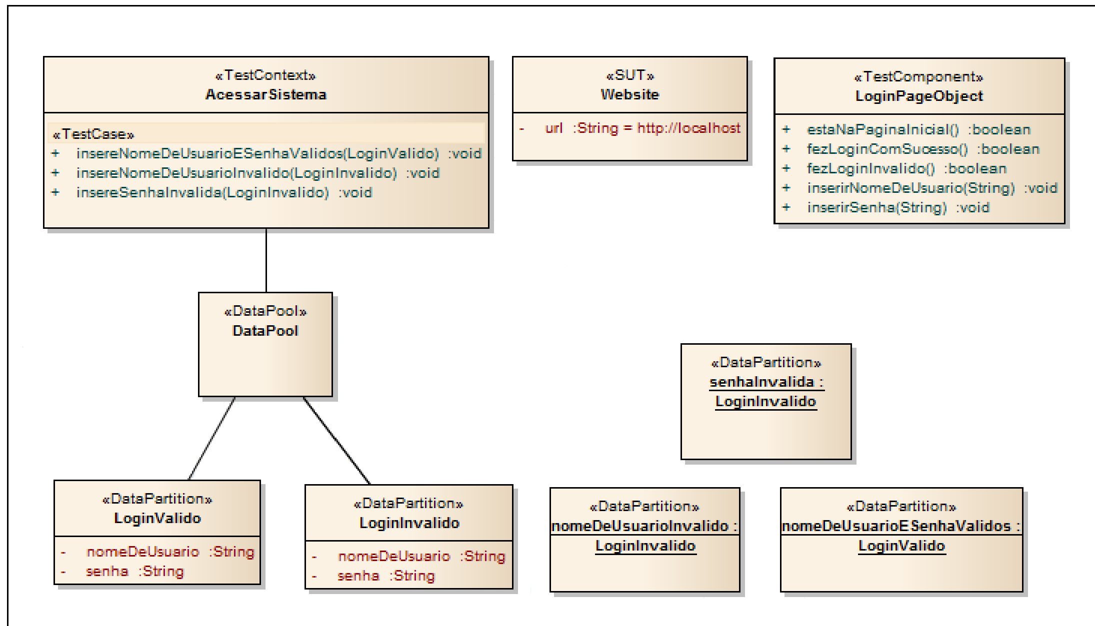
Para cada cenário de teste, o resultado da execução é definido através da comparação do estado em que se está ao final da execução com o estado desejado. Com isto temos o *TestVeredict* para cada cenário de aceitação, cujo valor *default* é "*pass*", pois considera-se que todos os passos do cenário foram realizados.

As ações que o ator do caso de uso pode realizar sobre o sistema são representadas através dos métodos definidos nas classes *TestComponent* (Page Objects). Já os possíveis dados de teste são encapsulados em classes com estereótipos <<DataPool>> e <<DataPartition>>. Estes dados de teste são utilizados de acordo com a transição do diagrama de estados relacionada. Por exemplo, a transição que trata de dados de nome de usuário inválido utilizará dados da partição correspondente. O elemento *DataPool* especifica um contêiner para valores explícitos ou partições de dados que são utilizados por casos de teste. Pode ser apenas referenciado por um *TestContext* ou um *TestComponent*, mas nunca ambos. Zero ou mais *DataPools* podem ser associados a

TestComponents ou *TestContexts*. Os elementos *DataPartition* devem estar associados a uma classe *DataPool*. Uma *DataPartition* especifica um contêiner para um conjunto de valores. Uma partição de dados pode estar associada apenas a um *DataPool* ou outra *DataPartition*. Zero ou mais *DataPartitions* podem ser definidas para um *DataPool.*

O exemplo de modelo de testes para a estória Acessar Sistema está representado na Figura 4.4.

Figura 4.4 - Modelo de Teste para a estória Acessar Sistema



Os dados do teste são definidos a partir das suas partições de equivalência. Cada conjunto de dados de teste é uma instância de uma *DataPartition*. Os valores dos itens testados devem ser definidos como valor *default* de cada uma destas instâncias. Desta maneira, o projetista de testes já pode, no momento da especificação do teste, e de maneira mais simples e em alto nível de abstração, definir os parâmetros que serão passados para cada caso de teste, pois estes também serão gerados automaticamente pelo gerador de código.

Os *TestCases* são gerados a partir de cada transição do diagrama de estados, e devem receber como parâmetro a instância de dados de teste relativa ao caso de teste em questão.

Para possibilitar a aplicação do algoritmo de geração de código de teste (que será apresentado na seção 4.3.2), algumas propriedades neste modelo devem ser satisfeitas:

- os nomes das instâncias de *DataPartition* devem ser iguais aos nomes das transições do diagrama de estados;
- os nomes dos métodos na classe Page Object devem conter (como *substring*):
  - o nome exato de um estado do diagrama de estados; ou
  - o nome exato de um atributo de uma classe *DataPartition*.

Tendo então todos os modelos elaborados, estes podem ser então exportados para um formato computável e utilizados para a geração dos casos de teste.

4.2 Atividade A3: Exportação dos Diagramas para XMI

Muitas abordagens utilizam a geração de um XMI intermediário a partir do modelo especificado, porém ainda não há um padrão definido para o formato do mesmo, o que cria uma dependência com o ferramental utilizado. O gerador de casos de teste, na maioria das vezes, é dependente da aplicação e do contexto em que é desenvolvido.

Apesar de existir uma especificação de mapeamento da U2TP para XMI desenvolvido pela OMG (OMG, 2005), esta especificação não é suportada por muitas ferramentas de modelagem UML, e não foi utilizada na realização deste trabalho. O mapeamento aqui utilizado é o especificado em (OMG, 2007).

Nesta abordagem, os modelos de projeto são exportados para um arquivo XMI e o modelo de teste para outro arquivo XMI, utilizando a própria ferramenta de modelagem. O primeiro arquivo é utilizado para geração dos cenários de teste e, em conjunto com o segundo arquivo, também é utilizado para a geração do código responsável pela execução do teste.

Os elementos UML/U2TP do modelo podem ser representados por mais de um elemento em representação XML, no documento XMI. Esses elementos no XMI representam uma visão do elemento U2TP que é representado. A Tabela 4.2 demonstra como os elementos dos modelos são representados nos documentos XMI.

Tabela 4.2 - Mapeamento de U2TP para XMI

U2TP	XMI do Modelo de Teste	XMI de Modelos de Projeto	Observação
TestContext	thecustomprofile:TestContext em uml:Class		Estereótipo TestContext
	uml:Class		Classe que representa o TestContext
		uml:StateMachine	Diagrama de estados da estória
SUT	thecustomprofile:Sut em uml:Class		Estereótipo Sut
	uml:Class		Classe que representa o e Sut
TestComponent	thecustomprofile:TestComponent em uml:Class		Estereótipo TestComponent
	uml:Class		Classes TestComponent
TestCase	thecustomprofile:TestCase em uml:Class		Estereótipo TestCase

	Operation em uml:Class		Métodos TestCase do TestContext
		uml:State e uml:Transition	Caminho que representa o caso de teste
DataPool	thecustomprofile:DataPool em uml:Class		Estereótipo DataPool
	uml:Class		Classes DataPool
DataPartition	thecustomprofile:DataPartition em uml:Class		Estereótipo DataPartition
	uml:Class		Classes DataPartition
	uml:InstanceSpecification		Instâncias das classes DataPartition

Um exemplo de XMI para os modelos de projeto gerado pela ferramenta para a estória Acessar Sistema pode ser visto na Figura 4.5. E na Figura 4.6, um exemplo de um arquivo XMI gerado a partir do modelo de teste para esta mesma estória.

Figura 4.5 - Arquivo XMI dos modelos de projeto


```xml
<?xml version="1.0" encoding="windows-1252"?>
<xmi:XMI xmi:version="2.1" xmlns:uml="http://schema.omg.org/spec/UML/2.1" xmlns:xmi="http://schema.omg.org/spec/XMI/2.1">
    <xmi:Documentation exporter="Enterprise Architect" exporterVersion="6.5"/>
    <uml:Model xmi:type="uml:Model" name="EA_Model" visibility="public">
        <packagedElement xmi:type="uml:Package" xmi:id="EAPK_ED60B8F4_36C1_45fe_8D58_70A6B9AD2CDE" name="Use Case Model" visibility="pub
            <packagedElement xmi:type="uml:Actor" xmi:id="EAID_C889DC61_E34A_4459_B8E7_370ECB8F74FD" name="Usuário" visibility="public"/
            <packagedElement xmi:type="uml:Association" xmi:id="EAID_27026DD3_D69C_45ab_8971_643286942FF7" visibility="public">
            <packagedElement xmi:type="uml:UseCase" xmi:id="EAID_ADC0CEB4_333D_4ae8_A0A4_1179683377B6" name="Fazer Login" visibility="pu
            <packagedElement xmi:type="uml:StateMachine" xmi:id="EAID_SM000001_36C1_45fe_8D58_70A6B9AD2CDE" name="EA_StateMachine1" visi
                <region xmi:type="uml:Region" xmi:id="EAID_SR000001_36C1_45fe_8D58_70A6B9AD2CDE" name="EA_Region1" visibility="public">
                    <subvertex xmi:type="uml:State" xmi:id="EAID_024F18BD_171F_4c43_A65F_8F2AB1BD3F96" name="Login Invalido" visibility=
                        <incoming xmi:idref="EAID_54BC6FC3_3309_4b01_A49E_BECA0BDF362F"/>
                        <incoming xmi:idref="EAID_24696123_9D56_4d2e_B726_378AD0DC7A0C"/>
                        <outgoing xmi:idref="EAID_836D404C_BD47_4303_8B8F_BD2F0EBC630B"/>
                    </subvertex>
                    <transition xmi:type="uml:Transition" xmi:id="EAID_836D404C_BD47_4303_8B8F_BD2F0EBC630B" visibility="public" kind="l
                    <subvertex xmi:type="uml:State" xmi:id="EAID_A9D8DB26_675A_4016_9B77_EE81472D7BA3" name="Login com Sucesso" visibili
                    <transition xmi:type="uml:Transition" xmi:id="EAID_2EE09487_DBCB_4c54_8691_D8068D90349F" visibility="public" kind="l
                    <subvertex xmi:type="uml:State" xmi:id="EAID_2F712283_AAB0_4b70_94E8_C24A4170FE80" name="Pagina Inicial" visibility=
                    <transition xmi:type="uml:Transition" xmi:id="EAID_00F20E58_60E2_4a1d_9D36_2A55C998E6B8" visibility="public" kind="l
                    <transition xmi:type="uml:Transition" xmi:id="EAID_24696123_9D56_4d2e_B726_378AD0DC7A0C" visibility="public" kind="l
                    <transition xmi:type="uml:Transition" xmi:id="EAID_36DDF20E_7189_498b_AB83_9E7DDFB5705E" visibility="public" kind="l
                    <transition xmi:type="uml:Transition" xmi:id="EAID_54BC6FC3_3309_4b01_A49E_BECA0BDF362F" visibility="public" kind="l
                    <subvertex xmi:type="uml:FinalState" xmi:id="EAID_21B46F27_FB66_4bd7_8C7A_3CF4B69FA153" name="Final" visibility="pub
                    <subvertex xmi:type="uml:FinalState" xmi:id="EAID_E6928267_AC5C_4d49_8FBF_C4F792317FA5" name="Final" visibility="pub
                    <subvertex xmi:type="uml:Pseudostate" xmi:id="EAID_4A5EAB6F_CF7E_439d_8E76_F8933AF9CB5F" name="Initial" visibility="
                    <transition xmi:type="uml:Transition" xmi:id="EAID_E7943627_98F1_40c6_950A_67FBCE27E569" visibility="public" kind="l
                </region>
            </packagedElement>
        </packagedElement>
    </uml:Model>
```

Figura 4.6 - Arquivo XMI do modelo de teste


```xml
<?xml version="1.0" encoding="windows-1252"?>
<xmi:XMI xmi:version="2.1" xmlns:uml="http://schema.omg.org/spec/UML/2.1" xmlns:xmi="http://schema.omg.org/spec/XMI/2.1" xmlns:thecustomprofile="http://w
    <xmi:Documentation exporter="Enterprise Architect" exporterVersion="6.5"/>
    <uml:Model xmi:type="uml:Model" name="EA_Model" visibility="public">
        <packagedElement xmi:type="uml:Package" xmi:id="EAPK_B8F0934C_BAEC_4978_B282_F09E7A2FEBE1" name="Class Model" visibility="public">
            <packagedElement xmi:type="uml:Class" xmi:id="EAID_D0FBFF27_4305_4ccc_9A48_1D6B33AB9BA9" name="AcessarSistema" visibility="public">
            <packagedElement xmi:type="uml:Association" xmi:id="EAID_297ED882_411C_46fc_B7A8_D91052C537DB" visibility="public">
            <packagedElement xmi:type="uml:Class" xmi:id="EAID_BA0C80EF_1F67_4c33_A498_D48EC3E2DFD6" name="DataPool" visibility="public"/>
            <packagedElement xmi:type="uml:Association" xmi:id="EAID_176C6B24_3581_4143_A9DB_90040ABA203C" visibility="public">
            <packagedElement xmi:type="uml:Association" xmi:id="EAID_45DAEAC6_D08C_4e89_8542_A09F6206F921" visibility="public">
            <packagedElement xmi:type="uml:Association" xmi:id="EAID_6E396ACF_4845_4665_9045_248BA2530480" visibility="public">
            <packagedElement xmi:type="uml:Association" xmi:id="EAID_C4A1B04A_7FD7_4c5e_97CD_0117F66DDA76" visibility="public">
            <packagedElement xmi:type="uml:Class" xmi:id="EAID_E0A7D952_9311_4940_9225_E0C63192607D" name="LoginInvalido" visibility="public"/>
            <packagedElement xmi:type="uml:Class" xmi:id="EAID_7216D819_C93D_4990_AC66_5698C68EDA00" name="LoginInvalido" visibility="public">
            <packagedElement xmi:type="uml:Class" xmi:id="EAID_D3B34469_63AD_486c_91F2_C973DBB73827" name="LoginInvalido" visibility="public">
            <packagedElement xmi:type="uml:Class" xmi:id="EAID_62E32D72_13C5_4c80_B5BD_AEB686239C05" name="LoginInvalido" visibility="public">
            <packagedElement xmi:type="uml:Class" xmi:id="EAID_56051886_CE91_415c_9358_E01108E57FB7" name="LoginPageObject" visibility="public">
            <packagedElement xmi:type="uml:Class" xmi:id="EAID_33CF0069_BB95_42b9_9496_711E8FB4EA31" name="LoginValido" visibility="public">
            <packagedElement xmi:type="uml:Class" xmi:id="EAID_C8A2C22D_0000_4ca5_83C0_AC7F49B8E515" name="LoginValido" visibility="public">
                <ownedAttribute xmi:type="uml:Property" xmi:id="EAID_EBE9F976_0C03_448f_8637_1647E097D6F4" name="nomeDeUsuario" visibility="private" isSt
                <ownedAttribute xmi:type="uml:Property" xmi:id="EAID_97ACC7C2_17FE_4952_9478_D0E9AEE1E94F" name="senha" visibility="private" isStatic="fa
            </packagedElement>
            <packagedElement xmi:type="uml:Class" xmi:id="EAID_51E9012D_1AD8_462f_84DE_25C0683C73ED" name="Website" visibility="public">
            <packagedElement xmi:type="uml:InstanceSpecification" xmi:id="EAID_18099532_4102_45cb_A9D4_0599B505315A" name="" visibility="public" classifi
            <packagedElement xmi:type="uml:InstanceSpecification" xmi:id="EAID_3A492A22_75C7_4ba4_9B32_08C7BBF8624C" name="nomeDeUsuarioESenhaInvalidos"
            <packagedElement xmi:type="uml:InstanceSpecification" xmi:id="EAID_1A2AFD3A_A2B5_4875_984C_8FC76CC40056" name="nomeDeUsuarioESenhaValidos" vi
            <packagedElement xmi:type="uml:InstanceSpecification" xmi:id="EAID_C6E39C33_ED99_4b49_A2AB_F0FD49CC1AB3" name="nomeDeUsuarioInvalido" visibil
            <packagedElement xmi:type="uml:InstanceSpecification" xmi:id="EAID_FD50FA14_7F1A_4db6_BCC7_5F2612D9F9B3" name="senhaInvalida" visibility="pub
            <packagedElement xmi:type="uml:Package" xmi:id="EAPK_1921D891_AF5D_4090_B852_48447BC9C7EB" name="System" visibility="public">
            <packagedElement xmi:type="uml:Package" xmi:id="EAPK_D489E6EE_DCBB_4a20_AAE0_0900F5D73070" name="Frameworks" visibility="public">
        </packagedElement>
```


4.3 Geração do Teste de Aceitação

A geração automática de testes exige uma linguagem de especificação que tenha uma semântica bem definida, como uma semântica baseada em Máquinas de Estados Finitos, por exemplo. O princípio de uma geração automática de teste é: a semântica da linguagem descreve como uma especificação escrita nesta linguagem pode ser traduzida automaticamente para algo (geralmente outra linguagem) que descreva o comportamento do sistema especificado. A geração automática do teste assume que uma aplicação, que implementa esta especificação, tem as mesmas propriedades do modelo (BAKER, 2007). No MDT, os artefatos de teste são gerados automaticamente de acordo com regras de transformação pré-definidas que derivam dos modelos de desenvolvimento (LIMA, 2007).

4.3.1 Atividade A4: Geração dos Cenários de Teste

Um caso de teste é uma implementação de um objetivo de teste para uma configuração particular, que é definida pelo comportamento de teste. Um objetivo de teste é uma descrição geral do que deve ser testado (SCHIEFEDECKER, 2003).

A geração dos casos de teste ocorre da seguinte maneira: a partir do XMI gerado pela ferramenta de modelagem, é feito um caminhamento na árvore XMI para encontrar caminhos válidos através do modelo. Estes caminhos consistem em estados e transições que representam o comportamento da aplicação que está sendo modelada. Cada caminho válido é convertido em um cenário de teste utilizando as transições do modelo como informação de teste. Assim, um cenário de teste completo é a concatenação de todas as informações de teste para um caminho válido. Este cenário de teste pode ser produzido em qualquer linguagem desejada (CLARKE, 1998).

É necessário definir um critério de cobertura para a geração dos cenários de teste a partir do diagrama de estados. Neste trabalho foi escolhida a cobertura de transições. Para atingir este nível de cobertura, a suíte de teste deve ser construída iterativamente, fazendo um caminhamento pelos elementos do XMI. Inicialmente, todas as transições são assinaladas como “não cobertas”. Então, um primeiro cenário de teste é gerado. No próximo ciclo, uma nova transição ainda não coberta é escolhida e um novo cenário de teste é gerado. Este procedimento é repetido até que todas as transições sejam cobertas por pelo menos um cenário de teste (FRÖHLICH, 2000). Cada cenário de aceitação será relacionado então a uma transição do diagrama.

Como cada transição é escolhida por vez, e cada elemento do caminho que forma o cenário é percorrido apenas uma vez, ciclos são avaliados com apenas uma execução, sem gerar *loops* infinitos.

Os casos de teste nesta abordagem são representados na forma de cenários de aceitação, descritos utilizando a linguagem textual Gherkin (NORTH, 2006). Um cenário de aceitação na abordagem proposta por este trabalho possui o formato descrito na Figura 4.7.

Figura 4.7 - Formato do cenário de aceitação proposto

Cenário: [ação da transição em questão]
Dado que [caminho até a transição em questão]
Quando [ação da transição em questão]
Então [condições do próximo estado]

A partir então do arquivo XMI relativo aos modelos de projeto, um arquivo texto (com extensão".story", para estar de acordo com o *framework* de teste utilizado, JBehave) é gerado com os cenários de aceitação, no formato descrito acima.

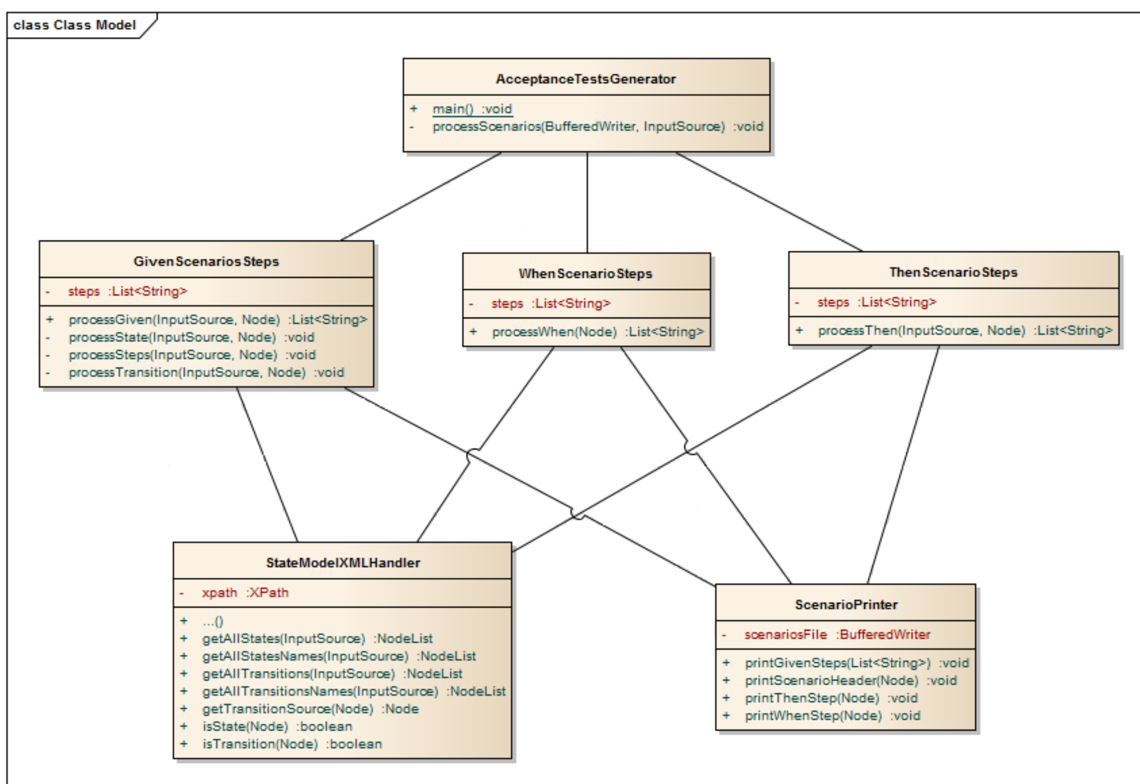
O algoritmo para gerar os cenários de teste é como segue:

- Para cada transição (T_n) do diagrama de estados que possui nome (ação do usuário):
    - O item "Cenário" é formado pelo nome da transição tratada T_n;
    - A partir do estado inicial (E_0) até a transição tratada (T_n), armazena todos os estados e transições que possuem nome. E_0 será impresso como "Dado que [nome do estado]", e os demais itens (estados e transições), se existentes, são impressos como "E que [nome do item]";
    - O nome da transição T_n é armazenado para ser impresso no item "Quando";
    - O nome do estado seguinte à T_n é armazenado para ser impresso no item "Então";
    - Cada item é então impresso na ordem da Figura 4.7.

O texto impresso relativo a cada estado e transição é igual ao nome definido no diagrama de estado, portanto estes devem ser escritos no modelo de maneira que faça tanto sentido quanto possível no texto final.

Para possibilitar a aplicação deste algoritmo no XMI existente, um aplicativo gerador de cenários e código de teste foi desenvolvido em linguagem Java, utilizando bibliotecas nativas para manipulação de XML através de comandos XPath (W3C, 2010). A organização deste gerador está descrita na Figura 4.8. A execução inicia na classe "AcceptanceTestsGenerator", que gerencia a geração dos passos dos cenários de aceitação. As classes "GivenScenarioSteps", "WhenScenarioSteps" e "ThenScenarioSteps" são responsáveis por gerar cada tipo de passo dos cenários de aceitação ("Dado que", "Quando" e "Então", respectivamente). Estas classes têm acesso a determinados nodos e estruturas da árvore XMI através de métodos da classe "StateModelXMLHandler" (para não poluir visualmente o diagrama, apenas alguns métodos desta classe estão representados). A impressão de cada passo de teste no arquivo texto é tratada pela classe "ScenarioPrinter".

Figura 4.8 - Estrutura do aplicativo gerador de cenários de teste



A aplicação como um todo ainda inclui as classes responsáveis pela geração do código de teste, que foram omitidas do diagrama neste momento para maior clareza. Elas serão tratadas em detalhe na seção 4.3.2.

Para o exemplo da estória "Acessar Sistema", os cenários de aceitação gerados a partir do XMI dos diagramas de projeto podem ser vistos na Figura 4.9.

Figura 4.9 - Cenários de aceitação gerados para a estória Acessar Sistema

```
Scenario: senha invalida
Given Pagina Inicial
When senha invalida
Then Login Invalido

Scenario: nome de usuario e senha validos
Given Pagina Inicial
When nome de usuario e senha validos
Then Login com Sucesso

Scenario: nome de usuario invalido
Given Pagina Inicial
When nome de usuario invalido
Then Login Invalido
```

4.3.2 Atividade A5: Geração do Código de Teste

Técnicas de geração de código e uso de alto nível de abstração já são utilizados para aumentar a produtividade do desenvolvimento de *software*. O grande número de ferramentas para modelagem UML que também geram código para diversas linguagens são uma demonstração disso. A geração de código é uma técnica já bem difundida no desenvolvimento de *software* visando aumentar a produtividade e diminuir a quantidade de erros de codificação, possibilitando que tarefas que são repetitivas e custosas em termos de tempo de desenvolvimento sejam realizadas muito mais rapidamente e com baixo risco de erro humano. O uso do alto nível de abstração permite que seja mais fácil criar, entender e manipular os artefatos de *software*, ao ocultar detalhes que não são interessantes para o nível em que se trabalha.

Em teste de *software*, a geração de código de teste é ainda mais difundida, para fazer com que quem esteja testando não use sua produtividade em tarefas simples e repetitivas, quando novo código ou testes mais complexos poderiam estar sendo feitos.

Para permitir a execução automatizada dos cenários de aceitação apresentados, é necessário ter o apoio de um *framework* BDD que forneça a infraestrutura necessária para isto. Neste trabalho, foi adotado o *framework* JBehave (JBEHAVE, 2013) que, através de um arquivo de configuração, um arquivo de código que será executado para cada passo de teste, e o arquivo de texto com os cenários de aceitação, permite a execução automatizada dos testes. Como este *framework* tem como base a estrutura de execução do JUnit (JUNIT, 2013), ao final da execução um relatório com o resultado dos testes é fornecido.

Este *framework* foi escolhido pela sua compatibilidade com a plataforma Java, por ter suas estruturas concebidas de maneira que torna mais simples a automatização da geração de código de teste, por permitir testes mais robustos e completos (já que utiliza linguagem Java como base), além de utilizar JUnit (um *framework* de testes bem aceito pela comunidade) como base para a execução dos testes.

Para a classe de configuração JBehave, seu nome vem do caso de uso ou estória. Esta classe deve referenciar a classe de passos de teste (descrita adiante) e o arquivo de cenários de teste, que por sua vez deve ter o mesmo nome da classe de configuração e estar num diretório pré-definido.

A etapa de geração do código de teste pode ser realizada após ou em paralelo com a geração dos cenários de teste. É apenas necessário que as classes Page Object já tenham sido desenvolvidas, pois serão elas que encapsularão as ações que podem ser executadas sobre cada item das páginas *web* envolvidas no teste. Estas ações devem ser programadas utilizando um *framework* de manipulação de elementos de páginas *web*, neste caso foi escolhido o WebDriver (SELENIUM, 2013). Além disso, o arquivo de configuração JBehave também deve ser programado de acordo com a infraestrutura do ambiente de teste da aplicação que será testada.

A partir dos arquivos XMI dos modelos de projeto e teste, e possuindo as classes citadas no parágrafo anterior, é possível então gerar a classe que define quais métodos serão executados para cada passo dos cenários de teste. O algoritmo que realiza esta etapa é como segue:

- Gera código padrão da classe, como: nome do pacote; importações de bibliotecas e outras classes relacionadas; a assinatura da classe, utilizando como nome desta classe o nome da classe *TestContext* do modelo de teste; e declaração dos objetos das classes relacionadas;
- Gera o método responsável pelas pré-condições de toda a suíte de testes que será executada. Este método instancia os Page Objects e cria o *driver* de execução do WebDriver;
- Gera o método responsável pelas pós-condições de toda a suíte de teste. Neste caso, esta classe apenas encerra o *driver* de execução do WebDriver;
- Após, gera o método responsável pelas pré-condições de cada cenário de teste. Neste caso, este método atribui valores aos atributos do *SUT*;
- Para cada passo dos cenários de teste, cria um método com os comandos necessários para sua execução:
  - Cada passo de teste é guardado em uma lista de acordo com o seu tipo ("Dado que", "Quando", "Então"), e esta lista é percorrida para criar a assinatura de cada método, que descreve o tipo do passo, seu nome e o nome do método (formado pela concatenação das palavras que compõem o nome do passo de teste), de acordo com as regras do JBehave;
  - Para os comandos que serão executados pelo método, é pesquisado no diagrama de estados se este passo é representado por um estado ou uma transição:
    - Se for um estado, cria uma asserção que avalia o resultado de um método do Page Object sobre o estado em questão;
    - Se for uma transição, busca uma instância de *DataPartition* cujo nome esteja contido no nome da transição e, para cada atributo desta instância, busca o método no Page Object que contenha o nome deste atributo. É gerada então a chamada deste método, passando como parâmetros os valores destes atributos da instância de *DataPartition* relacionada. Por exemplo, tomando como base o exemplo da estória Acessar Sistema, para um passo de teste "login inválido", a instância de dados seria "loginInvalido", cuja classe "LoginInvalido" possui atributos "nomeDeUsuario" e "senha". Os métodos do Page Object

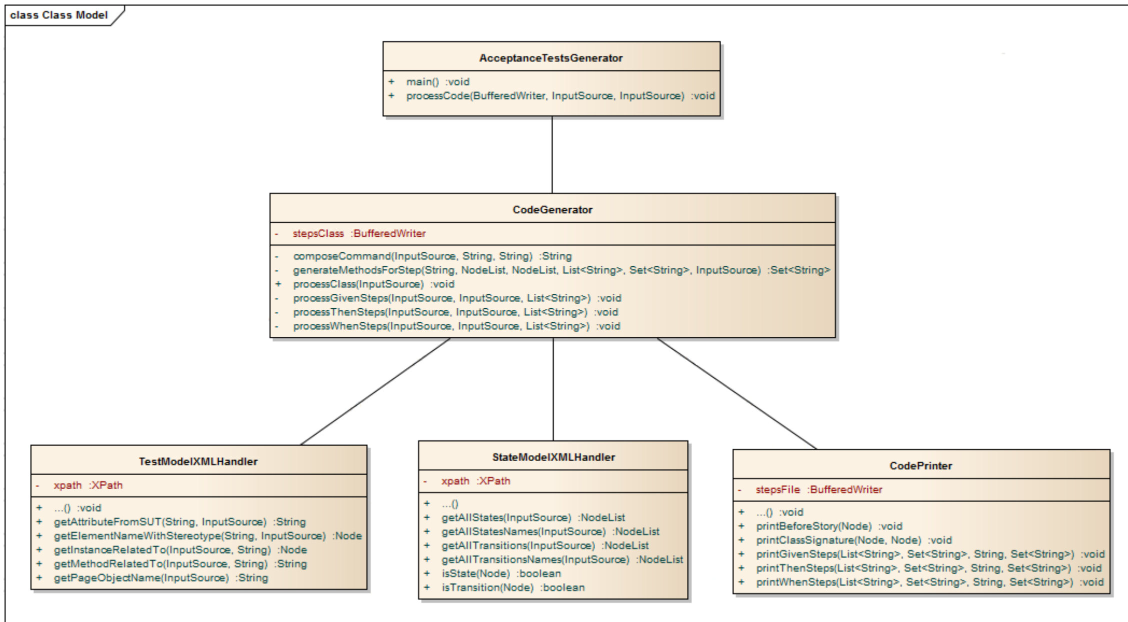
"inserirNomeDeUsuario" e "inserirSenha" seriam então chamados como comandos neste método de teste, tendo como parâmetros os valores *default* da instância "njfeller" e "teste123", respectivamente.

Este algoritmo gera um código de teste como o da Figura 4.10 para a estória Acessar Sistema. Para isto, o aplicativo gerador de cenários de teste em linguagem Java descrito na seção 4.3.1 foi estendido para tratar também o XMI do modelo de teste, entre outras responsabilidades. Sua organização está descrita na Figura 4.11.

Figura 4.10 - Código de teste gerado automaticamente

```
package testConfiguration;

import java.util.List;

public class AcessarSistemaSteps extends JUnitStories {
    private WebDriver driver;
    private LoginPageObject loginPageObject;

    @BeforeStory
    public void beforeStory() {
        driver = new FirefoxDriver();
        loginPageObject = new LoginPageObject(driver);
    }

    @AfterStory
    public void afterStory() {
        driver.quit();
    }

    @BeforeScenario
    public void beforeScenario() {
        driver.get("http://teste.elsa.ufrgs.br");
    }

    @Given("Pagina Inicial")
    public void givenPaginaInicial() throws InterruptedException {

        Assert.assertTrue(loginPageObject.estaNaPaginaInicial());

    }

    @When("senha invalida")
    public void whensenhaInvalida() throws InterruptedException {
        loginPageObject.inserirNomeDeUsuario("nadjia");
        loginPageObject.inserirSenha("123teste");

        loginPageObject.submit();
    }

    @When("nome de usuario e senha validos")
    public void whennomeDeUsuarioESenhaValidos() throws InterruptedException {
        loginPageObject.inserirNomeDeUsuario("nadjia");
        loginPageObject.inserirSenha("teste123");

        loginPageObject.submit();
    }

    @When("nome de usuario invalido")
    public void whennomeDeUsuarioInvalido() throws InterruptedException {
        loginPageObject.inserirNomeDeUsuario("njfeller");
        loginPageObject.inserirSenha("teste123");

        loginPageObject.submit();
    }

    @Then("Login Invalido")
    public void thenLoginInvalido() throws InterruptedException {

        Assert.assertTrue(loginPageObject.fezLoginInvalido());

    }

    @Then("Login com Sucesso")
    public void thenLoginComSucesso() throws InterruptedException {

        Assert.assertTrue(loginPageObject.fezLoginComSucesso());

    }
```

Figura 4.11 - Estrutura do aplicativo gerador de código de teste



A classe "AcceptanceTestsGenerator" é responsável por disparar a execução da geração do código de teste, que acontece na classe "CodeGenerator". Nesta, o código de teste é gerado de acordo com o tipo de passo de teste, além de avaliar e criar os tipos de comandos Java para execução do teste. A classe "CodeGenerator" tem auxílio das classes "TestModelXMLHandler" e "StateModelXMLHandler", que encapsulam os métodos que tratam os elementos dos arquivos XMI de teste e projeto, respectivamente, utilizando comandos XPath. A impressão de cada parte da classe de teste é responsabilidade da classe "CodePrinter". Para não poluir visualmente o diagrama da Figura 4.11, apenas alguns métodos destas últimas três classes foram representados.

Isto feito, temos toda a estrutura necessária para executar automaticamente os testes de aceitação, utilizando o *framework* JBehave. Detalhes sobre a execução dos testes serão tratados a seguir.

4.4 Atividade A6: Execução dos Cenários de Teste

Na abordagem de teste automatizado, a execução dos casos de testes é feita através de um programa ou *script*. Por este motivo, ela tem características como: a rápida execução dos casos de teste, a facilidade de repetição e a verificação automática dos resultados. Por permitir a execução de testes em lote, é a opção no caso de projetos de maior porte.

Por outro lado, testes automatizados apresentam um sobrecusto associado à implementação dos casos de teste. Para permitir a execução automática dos testes, o comportamento destes deve ser definido em algum formato padronizado que possa ser interpretado em um *software*, isto é, os testes precisam ser programados de alguma forma. Muitas vezes esse sobrecusto é muito elevado e pode não compensar os seus benefícios, o que nos leva a buscar soluções para o aumento de produtividade nessa atividade dos testes para que possamos utilizar esses benefícios num custo aceitável.

Os testes de aceitação gerados nas etapas anteriores podem ser executados diretamente, utilizando o *framework* JBehave. Para que um teste possa ser executado com este *framework*, são necessários, então:

- Cenários de aceitação em formato texto (seção 4.3.1);
- Classe de configuração (que pode ser gerada automaticamente, em conjunto com a classe de passos de teste, ou programada de acordo com a infraestrutura de desenvolvimento);
- Classe de Passos de Teste, responsável pela descrição da execução dos passos dos cenários de aceitação (seção 4.3.2).

O *framework* JBehave porém não possui sozinho as ferramentas necessárias para a execução de fato dos passos de teste, necessitando para isto utilizar as funcionalidades do *framework* JUnit. Além disso, no caso de sistemas *web*, é necessário um *framework* de interação com a aplicação, neste caso o WebDriver. Para isto, uma classe auxiliar também foi gerada, utilizando o padrão Page Object, e esta classe encapsula as ações possíveis em uma página *web*. Isto é feito para que o algoritmo de geração da classe de passos de teste tenha apenas a responsabilidade de tratar as ações que vêm diretamente dos cenários de teste, deixando assim os comandos do WebDriver encapsulados no Page Object auxiliar. Cada ação dos diagramas é mapeada para um ou mais métodos do Page Object.

Isto feito, é possível executar toda a suíte de testes gerada a partir dos diagramas da seção 4.1 com apenas um comando de execução ao JBehave. É necessário ter o JBehave, o JUnit e o WebDriver configurados e prontos pra uso. É possível visualizar então o resultado da execução do teste (passou ou falhou) tanto com o relatório gerado pelo JUnit, quanto no relatório gerado pelo JBehave, onde é descrito o resultado do teste para cada cenário de aceitação executado. Na Figura 4.12 temos um exemplo de relatório gerado pelo JBehave quando todos os cenários da estória Acessar Sistema são executados com sucesso.

Figura 4.12 - Relatório dos resultados da execução dos testes com JBehave

```
<terminated> AcessarSistemaTest [JUnit] C:\Program Files\Java\jre7\bin\javaw.exe
Using executor service com.google.common.util.concurrent.MoreExecutors$Same
Processing system properties {}
Using controls EmbedderControls[batch=false,skip=false,generateViewAfterSto

(BeforeStories)

Running story testConfiguration/acessar_sistema_test.story

(testConfiguration/acessar_sistema_test.story)
Scenario: senha invalida
Given Pagina Inicial
When senha invalida
Then Login Invalido

Scenario: nome de usuario e senha validos
Given Pagina Inicial
When nome de usuario e senha validos
Then Login com Sucesso

Scenario: nome de usuario invalido
Given Pagina Inicial
When nome de usuario invalido
Then Login Invalido



(AfterStories)

Generating reports view to 'C:\Users\Nadjia\workspace\ModelToAcceptanceTest
```

5 AVALIAÇÃO EMPÍRICA

Para mostrar um exemplo de aplicação da abordagem proposta, é apresentado neste capítulo seu uso em um processo de desenvolvimento de uma aplicação *web* real. Devido às suas características, esta abordagem pode ser inserida em diversos tipos de processo, inclusive no ciclo de desenvolvimento de uma equipe que utiliza métodos ágeis para o desenvolvimento de *software*.

5.1 Projeto do Experimento

Experimentos controlados permitem a avaliação de passos específicos do processo de *software*, através do controle das variáveis de entrada (WOHLIN *et al.*, 2012). A avaliação da aplicação de WATGUM em um processo de desenvolvimento real foi realizada através deste tipo de experimento, que compara três abordagens de criação e execução de testes de aceitação: teste manual, WATGUM e a criação manual de testes automatizados.

5.1.1 Escopo, Contexto e Participantes

O contexto de aplicação escolhido para isto foi o grupo de desenvolvedores do Centro de Coordenação de Estudos Multicêntricos (doravante chamado Dev-CCEM), envolvido nos projetos ELSA Brasil (ELSA, 2013) e LINDA Brasil (LINDA, 2013). Os sistemas desenvolvidos pelo Dev-CCEM são sistemas de coleta de dados para estudos epidemiológicos multicêntricos de saúde, onde são armazenados dados pessoais e resultados de exames dos participantes. Esta equipe utiliza adaptações de Scrum (SCHWABER, 2004), XP (BECK, 2000) e Kanban (OHNO, 1997) no seu processo de desenvolvimento de sistemas, e já trabalha com automação de testes de aceitação. Os testes de aceitação são desenvolvidos utilizando o *framework* JBehave, porém os cenários de teste e código de teste ainda são criados manualmente, causando em algumas situações um grande *overhead* de trabalho, aumentando o tempo despendido e o custo dos testes de aceitação. Em geral, para esta equipe, o desenvolvimento dos testes de aceitação custa pelo menos 50% do tempo de desenvolvimento total da funcionalidade.

Para avaliar a viabilidade e os possíveis benefícios da abordagem proposta por este trabalho neste ambiente de desenvolvimento, WATGUM foi aplicada no processo da equipe Dev-CCEM em paralelo com o desenvolvimento de testes já utilizado pela equipe. Como o escopo das aplicações já havia sido definido anteriormente, a participação ocorreu a partir das reuniões de priorização do *Product Backlog* e de planejamento de cada iteração.

Os integrantes da equipe que participaram dos experimentos possuem pelo menos conhecimento básico sobre UML, testes de aceitação e automação de teste (entre um e seis anos de experiência) e participaram por vontade própria enquanto não estavam executando nenhuma tarefa da iteração. Até a execução dos experimentos, nenhum dos participantes possuía alguma conexão com este trabalho. A equipe de desenvolvimento Dev-CCEM é formada por nove pessoas, sendo que oito possuem os pré-requisitos necessários, e cinco aceitaram participar dos experimentos propostos. Os participantes executaram as tarefas com auxílio da ferramenta de modelagem indicada (Enterprise Architect) e não foi limitado o tempo de execução das tarefas.

Todos os participantes foram instruídos sobre o uso da abordagem WATGUM, e o exemplo "Acesso ao Sistema", apresentado durante o capítulo 4, foi utilizado como exemplo e guia de aplicação da abordagem. O material utilizado durante a instrução estava disponível para consulta durante os experimentos. Além disso, os participantes puderam consultar materiais disponíveis na Internet, se achassem necessário. A ordem dos experimentos era alternada para diferentes participantes, evitando que o aprendizado da abordagem influenciasse nos resultados obtidos por uma ou outra tarefa do experimento.

5.1.2 Objetivos, Hipóteses, Parâmetros e Variáveis

A metodologia GQM (Goal-Question-Metric) provê um modelo para a definição de experimentos através da definição de objetivos mensuráveis (WOHLIN *et al.*, 2012). O objetivo geral do experimento é responder à pergunta: "A abordagem WATGUM oferece vantagens em comparação a métodos de teste manual ou criação manual de testes automatizados?".

O objeto de avaliação é a aplicação de toda a abordagem WATGUM (atividades A1 a A6) em um processo de desenvolvimento, observando o tempo despendido para a realização das tarefas, a manutenibilidade dos testes desenvolvidos e a melhor compreensão dos critérios de aceitação especificados. Algumas restrições inerentes ao processo de trabalho da equipe devem ser consideradas:

- Apesar da abordagem poder ser aplicada por pessoas com diferentes níveis de conhecimento em teste automatizado, as classes auxiliares (Page Objects, por exemplo) devem ser programadas por alguém com conhecimento do sistema e de padrões de desenvolvimento de testes automatizados;
- Sem validação experimental, inclusive pela própria equipe Dev-CCEM, é difícil requisitar que a equipe utilize a abordagem proposta em seu tempo de trabalho. Por isto, esta foi aplicada em paralelo ao ciclo de desenvolvimento da equipe, o que faz com que os dados coletados com relação às abordagens diferentes de WATGUM sejam dependentes dos que a própria equipe registra (que estão incluídos em seu processo de trabalho).

As perguntas que surgem a partir do objetivo proposto são: "Qual método é mais eficiente?", "Qual método possui melhor manutenibilidade?" e "Qual método ofereceu melhor compreensão dos critérios de aceitação para os usuários?". As principais métricas candidatas são: tempo de criação e execução dos testes; quantidade, tempo de realização e percepção de dificuldade de alterações realizadas; e percepção da compreensão dos critérios de aceitação especificados. A Tabela 5.1 apresenta as relações entre os itens da metodologia GQM apresentados.

Tabela 5.1 - Objetivo, questões e métricas para os experimentos propostos

Objetivo	Questões	Métricas
A abordagem WATGUM oferece vantagens em comparação à métodos de teste manual ou criação manual de testes automatizados.	Q1: Qual método é mais eficiente?	M1: Tempo de criação dos testes
		M2: Tempo de execução dos testes
	Q2: Qual método possui melhor manutenibilidade?	M3: Quantidade de alterações realizadas
		M4: Tempo para realização das alterações
		M5: Percepção de dificuldade das alterações realizadas
	Q3: Qual método ofereceu melhor compreensão dos critérios de aceitação para os usuários?	M6: Percepção da compreensão dos critérios de aceitação especificados

Considerando os itens descritos na Tabela 5.1, as seguintes hipóteses foram elaboradas:

- Hipótese Nula (H_0): Não há diferença entre os métodos analisados com relação a tempo, manutenibilidade e compreensão dos critérios de aceitação.
- Hipótese Alternativa 1 (H_1): WATGUM apresenta vantagens com relação à maioria dos itens analisados (tempo, manutenibilidade ou compreensão dos critérios de aceitação).
- Hipótese Alternativa 2 (H_2): WATGUM apresenta desvantagens com relação à maioria dos itens analisados (tempo, manutenibilidade ou compreensão dos critérios de aceitação).

Todos os experimentos assumem resultados estatisticamente relevantes em $\alpha = 0,05$. A abordagem de teste é o fator principal, ou variável independente do experimento. Esta variável independente possui três níveis, sendo A o uso da abordagem de teste manual, B para o uso da abordagem de teste automatizado Dev-CCEM e C para o uso de WATGUM. Para avaliar cada abordagem, as métricas descritas na Tabela 5.1 são utilizadas como critérios, tendo sua pontuação avaliada conforme as medidas realizadas durante o experimento e as respostas ao questionário aplicado. Cada um destes critérios é então uma variável dependente.

5.1.3 Tarefas e Instrumentos

As tarefas executadas durante os experimentos tratam da criação e execução de testes de aceitação para estórias que a equipe Dev-CCEM implementou em seus sistemas de coleta de dados. Um ou mais membros da equipe (independentes do experimento) implementou(aram) a tarefa da maneira já utilizada pela equipe, enquanto

os participantes realizaram individualmente e separadamente os experimentos utilizando WATGUM. Isto permite que a aplicação da abordagem proposta seja tão próxima quanto possível do processo real de desenvolvimento. As seções de experimento ocorreram para cada participante individualmente, em ambiente separado da equipe de desenvolvimento (para evitar qualquer interferência), onde um computador com as ferramentas que seriam utilizadas e *software* de apoio estavam disponíveis para uso do participante: Enterprise Architect para confecção dos modelos e exportação para XMI, o protótipo descrito na seção 4.3 para geração dos testes automatizados, e a IDE Eclipse (ECLIPSE FOUNDATION, 2014) com todas as bibliotecas necessárias para execução dos testes. Cada seção era acompanhada por um mediador, que apresentava o experimento, a documentação de treinamento de WATGUM, propunha as tarefas que seriam realizadas e respondia às perguntas dos participantes.

Uma das tarefas (Tarefa T1) consistiu em aplicar a abordagem WATGUM no desenvolvimento de uma tela de cadastro de participantes do estudo no sistema de coleta de dados do projeto LINDA Brasil. Já as outras tarefas compreenderam na aplicação de WATGUM no desenvolvimento (Tarefa T2) e na posterior alteração (Tarefa T3) de uma tela de busca de participantes do estudo neste mesmo sistema. As métricas de tempo (M1, M2, M4) foram obtidas com a utilização de cronômetros pelos mediadores do experimento durante a execução das tarefas (KITCHENHAM, 2002). A quantidade de alterações realizadas (M3) consiste nos itens alterados pelo participante (e contabilizados por ele): no caso de WATGUM, a quantidade de arestas e estados alterados no diagrama de estados, e a quantidade de métodos, atributos e instâncias alterados no diagrama de classes U2TP; e no caso da abordagem usual da equipe Dev-CCEM, os cenários, métodos e atributos alterados, independentemente do número de linhas de código afetadas.

Após a realização das tarefas, um questionário sobre a percepção dos participantes durante os experimentos foi respondido pelos mesmos. A partir deste questionário foi possível atribuir valores às métricas M5 e M6. O questionário continha as seguintes perguntas:

1. Qual das abordagens você acredita que tenha sido mais fácil de realizar as alterações nos testes da tela de busca de participante?
    a. Teste automatizado com criação manual dos cenários de aceitação e código de teste.
    b. Teste automatizado com modelagem e criação automatizada dos cenários de aceitação e código de teste.
    c. Não acredito que haja diferença entre o nível de dificuldade das abordagens.
2. Utilizando qual abordagem você acredita que seja melhor compreender os critérios de aceitação de uma estória?
    a. Teste manual.
    b. Teste automatizado com criação manual dos cenários de aceitação e código de teste.
    c. Teste automatizado com modelagem e criação automatizada dos cenários de aceitação e código de teste.

3. Você teria alguma observação ou sugestão a fazer sobre as abordagens propostas ou as tarefas realizadas?

5.1.4 Unidades Experimentais

Depois de definidas as funcionalidades que seriam desenvolvidas nas iterações acompanhadas, com os requisitos já elicitados e as regras de negócio já definidas, duas funcionalidades foram escolhidas como unidades para este experimento. Por serem funcionalidades comuns em muitos sistemas *web*, as estórias escolhidas foram: "Cadastro da Participante" e "Busca por participante", do sistema de coleta de dados LINDA. Nestas estórias, as participantes candidatas a entrar no estudo são cadastradas na base de dados e podem ser buscadas através da utilização de filtros. Em um momento posterior, viu-se a necessidade de adicionar um novo filtro à tela de busca de participante, então a estória "Adicionar filtro por data da Recepção na busca por participantes" foi criada. A estória "Cadastro da Participante" é a unidade utilizada durante a tarefa T1, enquanto as estórias "Busca por participante" e "Adicionar filtro por data da Recepção na busca por participantes" são utilizadas durante as tarefas T2 e T3, respectivamente. Todas as regras de negócio e critérios de aceitação das estórias relacionadas foram apresentadas e disponibilizadas aos participantes do experimento no início da execução de cada tarefa.

Com relação à estória "Cadastro da Participante", os campos de identificação seguem um conjunto de regras de validações, que devem ser satisfeitas para que os dados da participante sejam salvos com sucesso. Esta estória tem origem no caso de uso "Cadastro". Uma seção do diagrama de casos de uso do sistema que inclui esta estória pode ser vista na Figura 5.1. A Figura 5.2 apresenta um protótipo de interface da tela resultante da implementação desta estória.

Figura 5.1 - Seção do Diagrama de Casos de Uso que inclui "Cadastro"

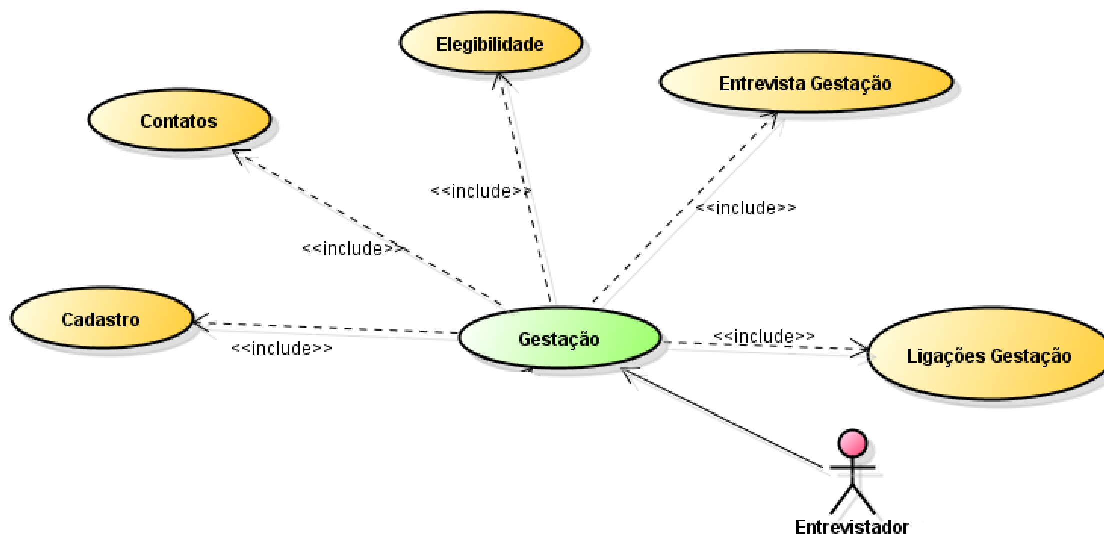


Figura 5.2 - Protótipo da Tela do Formulário de Cadastro de Participantes

Cadastro

Identificação e revisão inicial de prontuário/ficha:

Número de Registro *

Data da Entrevista *

Entrevistador *

Nome da Gestante *

Data de Nascimento *

Nome da Mãe da Gestante *

Prontuário *

Cadastrar Cancelar

Já para as estórias relacionadas às tarefas T2 e T3, a busca de participante é realizada utilizando filtros específicos, que também validam os dados inseridos e retornam as participantes que satisfazem os critérios de busca. Posteriormente, foi inserido o filtro de data de recepção, que retorna participantes que tiveram o formulário de recepção presencial no centro de investigação preenchido. Ambas as estórias "Busca por participante" e "Adicionar filtro por data da Recepção na busca por participantes" têm origem no caso de uso "Busca por participante", cuja seção do diagrama de casos de uso do sistema pode ser vista na Figura 5.3**Erro! Fonte de referência não encontrada.**. Os protótipos de interface da tela de sistema resultante da implementação destas estórias são apresentados na Figura 5.4 e na Figura 5.5, respectivamente.

Figura 5.3 - Seção do Diagrama de Casos de Uso que inclui "Busca por Participante"

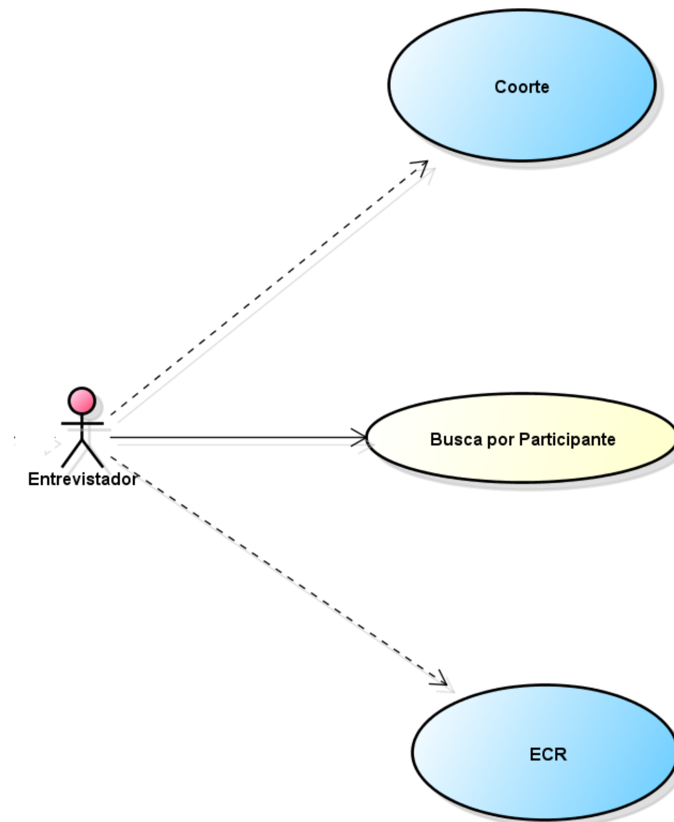


Figura 5.4 - Protótipo de Tela de Busca de Participantes com os filtros originais

Número de Registro:

Nome da Participante:

Centro: Selecione...

Buscar Limpar

Figura 5.5 - Protótipo de Tela de Busca de Participantes com novo filtro de data de recepção

Número de Registro:

Nome da Participante:

Centro: Selecione...

Data da Recepção da Presencial: até

Buscar Limpar

5.2 Resultados e Análise

Durante a execução do experimento nas condições propostas nas seções anteriores, as métricas foram coletadas e analisadas de acordo com cada questão apresentada na Tabela 5.1. Cada seção de experimento com cada participante durou aproximadamente duas horas. Nas próximas seções são apresentados os resultados obtidos e as possíveis ameaças à validade deste experimento.

5.2.1 Eficiência das abordagens

Para responder à questão Q1 (“Qual método é mais eficiente?”), as métricas de tempo de criação (M1) e tempo de execução dos testes (M2) foram coletadas para as três abordagens de maneira distinta. Para a abordagem WATGUM, as métricas M1 e M2 foram obtidas através da contagem do tempo de execução de partes das tarefas T1, T2 e T3, utilizando um cronômetro. Em particular, a métrica M1 foi composta pelos tempos para: criação dos diagramas de estados e classe (atividades A1 e A2); geração dos cenários de aceitação e código de teste (atividades A4 e A5); e refatoração das classes JBehave de teste, para adaptação ao ambiente específico da aplicação utilizada no experimento.

Já para as abordagens de teste manual e Dev-CCEM, estas métricas foram obtidas através de registros realizados pela própria equipe Dev-CCEM que foram disponibilizados para esta análise.

Os resultados obtidos para cada método em cada tarefa podem ser vistos na Tabela 5.2, com as médias de tempo representadas em minutos. Para a abordagem WATGUM foram calculados, além da média de tempo, o desvio padrão para cada situação.

Tabela 5.2 - Resultados obtidos por cada método para as métricas M1 e M2 para as tarefas T1, T2 e T3

		M1		M2	
Tarefa	Método	Média de tempo	Desvio Padrão	Média de tempo	Desvio Padrão
T1	Manual	120	N/A	95	N/A
	Dev-CCEM	915	N/A	2	N/A
	WATGUM	103	39,73	3,8	0,84
T2	Manual	15	N/A	45	N/A
	Dev-CCEM	325	N/A	1	N/A
	WATGUM	120,8	62,58	3	0,71
T3	Manual	10	N/A	30	N/A
	Dev-CCEM	300	N/A	1	N/A
	WATGUM	31,6	19,48	3,8	1,30

Como a distribuição de tempo não é normal, utilizamos o teste não-paramétrico Mann-Whitney para avaliar a diferença dos tempos entre as abordagens. Os valores de U_A e U_B (soma da ordenação dos itens de cada amostra), sendo “A” WATGUM e “B” Manual ou Dev-CCEM, para cada abordagem nas três tarefas são apresentados na Tabela 5.3.

Tabela 5.3 - Resultados do teste Mann-Whitney para cada tarefa e métricas M1 e M2

		Tarefa	U_A	U_B
M1	WATGUM vs. Manual	T1	3	2
		T2	0	5
		T3	0	5
	WATGUM vs. Dev-CCEM	T1	5	0
		T2	5	0
		T3	5	0
M2	WATGUM vs. Manual	T1	5	0
		T2	5	0
		T3	5	0
	WATGUM vs. Dev-CCEM	T1	0	5
		T2	0	5
		T3	0	5

Considerando um nível de significância de 0,05 para um teste não direcional, o intervalo crítico para U_A nas circunstâncias deste experimento é $-1 <= U_A <= 6$. Pode ser visto a partir dos dados da tabela que os resultados da comparação dos tempos em M1 entre a abordagem manual e WATGUM não são conclusivos, pois há situações em que se encontram no meio do intervalo, e em outras o teste manual obteve melhores resultados. Em M2 entre a abordagem Dev-CCEM e WATGUM, Dev-CCEM possui melhores resultados. Em compensação para as comparações em M1 entre WATGUM e Dev-CCEM, e em M2 entre WATGUM e a abordagem manual, é possível ver que WATGUM obteve resultados melhores.

5.2.2 Manutenibilidade das abordagens

Com relação à questão Q2 ("Qual método possui melhor manutenibilidade?"), foram consideradas apenas as abordagens Dev-CCEM e WATGUM na tarefa T3. Para a métrica M3, foram contabilizadas as mudanças efetuadas em código (cenários, métodos e atributos) para a abordagem Dev-CCEM, e as mudanças nos diagramas (transições, estados, métodos, atributos e instâncias) para a abordagem WATGUM. Já para a métrica M4, as medidas de tempo foram obtidas da mesma maneira que as métricas M1 e M2, apresentadas na seção anterior. A métrica M5 foi obtida através da frequência das respostas que os participantes forneceram no questionário para a pergunta "1".

Os valores obtidos para as métricas M3 e M4 podem ser observados na Tabela 5.4, onde as médias de tempo também estão representadas em minutos. Da mesma maneira, para a abordagem WATGUM foram calculados, além da média de tempo, o desvio padrão para cada situação.

Tabela 5.4 - Resultados obtidos para as métricas M3 e M4

	M3	M4

Tarefa	Método	Média	Desvio Padrão	Média	Desvio Padrão
T3	Dev-CCEM	11	N/A	300	N/A
	WATGUM	40,2	12,62	31,6	19,48

Utilizando novamente o teste Mann-Whitney para avaliar a diferença entre os valores de tempo obtidos para cada abordagem, os resultados para U_A e U_B, sendo "A" WATGUM e "B" Dev-CCEM, estão apresentados na Tabela 5.5.

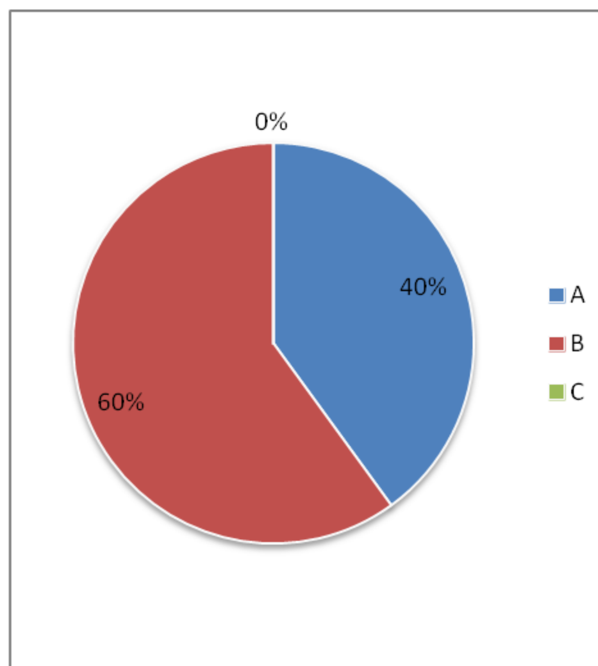
Tabela 5.5 - Resultado do teste Mann-Whitney para as métricas M3 e M4

WATGUM vs. Dev-CCEM	U_A	U_B
M3	0	5
M4	5	0

Considerando novamente um nível de significância de 0,05 e o intervalo crítico para U_A sendo $-1 <= U_A <= 6$, é possível ver que sobre M3 Dev-CCEM possui resultados melhores, porém com relação a M4, WATGUM obteve melhores resultados.

Sobre a impressão dos participantes a respeito da diferença das abordagens, a alternativa mais escolhida foi "b": teste automatizado com modelagem e criação automatizada dos cenários de aceitação e código de teste. No gráfico apresentado na Figura 5.6, é possível ver a distribuição da frequência de cada resposta.

Figura 5.6 - Frequência das respostas à questão 1 do questionário



5.2.3 Compreensão dos critérios de aceitação

Sobre a questão Q3 ("Qual método ofereceu melhor compreensão dos critérios de aceitação para os usuários?"), a métrica M6 relacionada foi obtida através das respostas dos participantes à questão "2" do questionário aplicado após o experimento. Houve unanimidade com relação à alternativa escolhida: "c" (teste automatizado com modelagem e criação automatizada dos cenários de aceitação e código de teste).

5.2.4 Análise

A partir dos resultados obtidos para cada métrica analisada, podemos traçar melhor o perfil da abordagem WATGUM para a situação estudada. Os resultados para M1 mostram que a utilização de modelos como ponto de partida para a criação de testes é mais vantajosa que a sua criação diretamente em texto e código (mais verbosos e repetitivos), apesar de mais trabalhosa do que o planejamento para execução apenas manual, considerando que a equipe Dev-CCEM realiza um planejamento simples apenas para elencar os casos de tese. Com relação à execução dos testes, WATGUM apresenta vantagens com relação à execução manual, o que, a longo prazo, pode ser significativo com relação à produtividade da equipe. A abordagem Dev-CCEM apresentou medidas de tempo de execução menores provavelmente devido a melhorias que a equipe já desenvolveu em seus códigos de teste.

Já para M3 e M4 é possível ver que, apesar de ser necessário realizar mais alterações nos modelos propostos do que diretamente no código de teste, o tempo utilizado para isto é menor com o uso da abordagem WATGUM.

Devido à subjetividade das métricas M5 e M6, acreditamos que os resultados obtidos estão relacionados com o perfil dos participantes, que preferem utilizar modelos para compreensão do que precisa ser testado e para realizar as alterações necessárias.

Levando estes aspectos em consideração, é possível então rejeitar a hipótese H_0 e aceitar a hipótese alternativa H_1, pois WATGUM apresenta vantagens com relação a pelo menos um dos itens analisados em comparação com as abordagens de teste manual e Dev-CCEM.

5.2.5 Ameaças à Validade

Nesta seção serão discutidas as ameaças que influenciam a validade do experimento realizado. A discussão é baseada nas quatro categorias de ameaças aos experimentos em engenharia de *software* detalhados por Wohlin (2012).

Com relação à validade interna, que trata de fatores que podem influenciar os resultados, a instrumentalização pode ser considerada uma ameaça, pois a documentação, tanto das regras de negócio quanto às relacionadas à abordagem WATGUM, foi escrita de maneira informal. Particularmente, a interpretação de cada participante sobre as regras de negócio apresentadas causa diferenças com relação ao número de casos de teste elaborados, o que pode afetar os tempos de criação dos diagramas e o número de alterações necessárias. Isto pode ser resolvido com reuniões de planejamento com todos os membros da equipe (uma prática dos métodos ágeis), onde apenas um conjunto de casos de teste, definido por toda a equipe, seria elaborado. Outra questão seria a subjetividade das respostas ao questionário (métricas M5 e M6), onde a resposta é influenciada pela preferência e facilidade que o participante tem em elaborar diagramas ou desenvolver código de teste. Isto também seria mitigado com reuniões e conversas entre os membros da equipe, onde todos chegariam em um acordo

sobre qual método é o mais adequado para o processo da equipe. A maturidade também é uma questão a ser considerada, pois conforme o participante utiliza a abordagem, ele aprende a realizar mais rapidamente as atividades e cria os diagramas com maior facilidade. Isto foi mitigado com a inversão na ordem de execução das tarefas T1 e T2 entre cada participante.

Uma ameaça à validade de conclusão (relacionamento entre tratamento dos dados e resultados) é o baixo poder estatístico, pois a amostra utilizada no estudo é muito pequena. Isto aconteceu pois era necessário o acesso a alguns trechos de código do sistema LINDA (que não possui código aberto) para implementação e acesso aos métodos da classe Page Object, portanto havia a limitação do número de membros da própria equipe.

Quanto à validade de construção, onde o relacionamento entre os conceitos da abordagem e a prática do experimento é avaliado, a necessidade de que cada participante realizasse o experimento individualmente está em desacordo com as práticas de métodos ágeis onde toda a equipe define em conjunto com o Product Owner os critérios de aceitação de cada estória. Além disso, os diagramas que representam os critérios de aceitação foram elaborados durante ou após a implementação das funcionalidades, o que não concorda com as práticas de ATDD. Isto também pode ser resolvido com a aplicação da abordagem WATGUM por toda a equipe em conjunto, dentro do processo da mesma. Outras ameaças são consideradas pequenas, pois os participantes não conheciam as hipóteses de estudo, os detalhes da abordagem WATGUM, e nem o funcionamento do protótipo para conversão de XMI em texto e código.

Ameaças à validade externa, ou seja, a limitação para a generalização dos resultados do experimento para a prática da indústria, são relacionadas principalmente às limitações das ferramentas utilizadas, além do contexto de aplicação do sistema de *software* utilizado. Isto foi mitigado: utilizando participantes que representam em parte o contexto da indústria; através do uso de práticas e artefatos bem reconhecidos pela indústria (por exemplo, uso do padrão Page Object); e aplicando a abordagem WATGUM em funcionalidades do sistema LINDA que podem ser facilmente mapeadas para outros sistemas *web* como telas de cadastro e busca com filtros.

6 CONCLUSÃO

Este trabalho apresentou uma abordagem para geração automática de testes de aceitação automatizados a partir de modelos U2TP. A utilização deste processo no ciclo de desenvolvimento de uma aplicação traz algumas vantagens, sendo uma das principais ser necessário gerar manualmente apenas o modelo de comportamento de cada funcionalidade da aplicação, pois os demais artefatos utilizados na especificação e execução dos testes de aceitação (cenários de aceitação, códigos de teste) são gerados automaticamente, consumindo menos tempo e estando menos sujeitos a erros.

Os modelos, se criados durante a fase de definição de requisitos do ciclo de desenvolvimento, podem prevenir diferentes interpretações dos requisitos pelos *stakeholders*, desenvolvedores e testadores. Minimizar estas diferenças irá prevenir que alguns defeitos cheguem inclusive à fase de execução de testes, auxiliando na redução de custo dos testes (CLARKE, 1998).

A abordagem de testes baseados em modelos oferece um aumento na eficácia dos testes, além de diminuir o próprio ciclo de teste. A geração de testes pode ser especialmente efetiva para sistemas que mudam frequentemente, pois os testadores podem atualizar o modelo e rapidamente gerar novamente a suite de testes, evitando a edição manual de suites de teste, sujeita a erros (DALAL, 1999).

O critério de cobertura proposto faz com que os testes sejam tão completos quanto o modelo elaborado, ou seja, caso um erro seja descoberto posteriormente na aplicação, significa que algum comportamento da mesma não foi mapeado previamente, e o modelo deve ser revisado em conjunto com os *stakeholders*.

Utilizar um diagrama de estados para modelar o comportamento do sistema e, a partir de seus caminhos possíveis, gerar os cenários de teste pode acarretar em uma grande quantidade de casos de teste. Mas isto possivelmente indicaria um problema no próprio modelo, pois modelos muito grandes ou muito minuciosos (que acarretariam em um número impraticável de casos de teste) são também poluídos visualmente e de difícil compreensão.

A geração dos cenários de teste em formato textual, além de ser um requisito do *framework* BDD, permite que a abordagem seja adaptada para utilização em outras plataformas. A utilização de UML e XMI também contribui para esta característica.

O uso de um *framework* bem aceito pela comunidade facilita a utilização da abordagem por equipes de desenvolvimento, pois o mesmo é *open source* e tem documentação e manutenção disponível na *web* para os usuários. Além de não ser necessário desenvolver e utilizar uma linguagem própria para a execução dos testes.

Alguns dos principais pontos fortes desta abordagem para geração automatizada de testes de aceitação são: o forte acoplamento entre os testes e os requisitos de usuário, a facilidade que UML oferece para a criação dos modelos e a habilidade de gerar novamente os testes em resposta a mudanças.

O tempo despendido na especificação dos modelos de projeto e teste é compensado pelo tempo economizado com a geração dos cenários e do código de testes, como pôde ser visto no experimento realizado no processo de desenvolvimento da equipe Dev-CCEM. Além disso, tem-se como vantagem modelos bem documentados do comportamento do sistema, das estruturas necessárias para o teste e dos dados de teste utilizados para cada caso de uso e estória de usuário, e a qualidade maior que se alcança trabalhando em um nível mais alto de abstração.

Como contribuições, então, deste trabalho podemos citar: aumento da produtividade no desenvolvimento de aplicações *web*, diminuindo o esforço para desenvolvimento e execução dos testes; por causa da utilização de modelos, os erros na especificação do projeto são descobertos mais cedo, diminuindo o custo da sua correção; auxílio na definição de critérios de aceitação que devem ser satisfeitos pelo sistema; manutenção dos testes do sistema facilitada, pois alterações podem ser feitas diretamente nos modelos, e os artefatos são gerados de forma automática novamente; além de ter demonstrado que é possível unir os conceitos e vantagens do uso de U2TP, MDT, BDD e automação de testes para sistemas *web* de maneira factível e satisfatória.

6.1 Limitações do Trabalho

Por outro lado, é improvável que o mesmo conjunto de modelos, mapeamentos e processo satisfaça todos os domínios, todas as organizações ou todos os projetos. Há critérios dos sistemas *web* que ainda não são tratados, como restrições de tempo ou domínios de dados complexos. Estes conceitos possuem representação em U2TP, apesar de não estar muito clara a sua aplicação na modelagem de alguns destes critérios.

Outra limitação está relacionada ao tratamento dos diagramas UML e U2TP, que não são utilizados em todo o seu potencial, com o auxílio de estruturas mais avançadas, além da otimização do uso das próprias estruturas já representadas.

Na abordagem proposta, ciclos no diagrama de estados são tratados com apenas uma execução, para evitar laços infinitos. A possibilidade de executar o laço um número determinado de vezes seria de interesse.

A representação dos dados de teste no modelo de teste, apesar de auxiliar na documentação e entendimento dos mesmos, faz com que a tarefa de definição dos dados de teste se torne sujeita a erros do testador, que pode esquecer ou modelar de maneira errada algum dos valores de teste. Melhorias neste aspecto foram sugeridas por participantes do experimento, trazendo também a possibilidade de automação inclusive de algumas etapas da modelagem, ou o uso de *templates* para facilitar a elaboração dos modelos.

Além disso, os arquivos XMI utilizados neste trabalho são dependentes da ferramenta Enterprise Architect, pois, ao contrário do que idealmente deveria, indefinições do padrão XMI causam diferença sensível entre os produtos existentes.

Há também uma limitação da própria área de Testes de Aceitação com relação à representação e mapeamento de critérios de aceitação relativos a usabilidade, desempenho, segurança e outros atributos de qualidade (PUGH, 2010).

6.2 Trabalhos Futuros

Com base nas limitações deste trabalho e possíveis melhorias, algumas possibilidades de trabalhos futuros podem ser citadas:

- Possibilitar seleção de cenários de teste, realizando uma cobertura parcial do diagrama, quando não é desejado testar todos os comportamentos possíveis da funcionalidade;
- Definir uma representação e possíveis mapeamentos para requisitos não funcionais, como usabilidade, segurança, desempenho, entre outros;
- Possibilitar, nos diagramas de estado que possuem ciclos, a avaliação de execução de laços um número determinado de vezes, ao mesmo tempo evitando a geração de um *loop* infinito;
- Oferecer a opção de geração automática dos dados de teste, sem necessariamente representá-los em detalhe no modelo. Algumas abordagens incluem: instanciar os dados no início de cada cenário; utilizar uma base de dados preparada especialmente para os testes; utilizar ferramentas para geração automatizada dos dados de teste;
- Utilizar estruturas mais avançadas dos diagramas de casos de uso, de estado e estrutural U2TP, além de desenvolver uma abordagem que permita o uso de outros conceitos do U2TP, como conceitos de tempo, muitas vezes utilizados em sistemas *web*;
- Estender esta abordagem para outros diagramas UML que também podem representar o comportamento de um sistema, porém com foco em outros tipos de fluxo, como diagramas de atividade e interação;
- Criar *templates* para facilitar a elaboração e reuso dos modelos, principalmente do modelo estrutural U2TP.

REFERÊNCIAS

ALI, M. A., SHAIK, K., KUMAR, S., Test Case Generation using UML State Diagram and OCL Expression. **International Journal of Computer Applications**, [S.l.]: v. 95, n. 12, p. 7-11, 2014.

APFELBAUM, L., DOYLE, J. Model based testing. In: SOFTWARE QUALITY WEEK CONFERENCE., 1997. [S.l.: s.n.], 1997.

BAKER, P. et al., **Model-Driven Testing:** Using the UML Testing Profile. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

BECK, K., **Extreme programming explained:** embrace change. Addison-Wesley Professional, 2000.

BIASI, L. D. **Geração Automatizada de *Drivers* e *Stubs* de Teste para JUnit a partir de Especificações U2TP.** 2006. 153 p. Dissertação (Mestrado em Ciência da Computação) – Faculdade de Informática, PUCRS, Porto Alegre.

BRIAND, L. C., LABICHE, Y., A UML-Based Approach to System Testing. In: INTERNATIONAL CONFERENCE ON THE UNIFIED MODELING LANGUAGE, MODELING LANGUAGES, CONCEPTS, AND TOOLS, 4., 2001. **Proceedings of the 4th International Conference on the Unified Modeling Language, Modeling Languages, Concepts, and Tools.** London, UK: Springer-Verlag 2001. p. 194-208.

CLARKE, J. M., Automated test generation from a behavioral model, In: PACIFIC NORTHWEST SOFTWARE QUALITY CONFERENCE, 1998. **Proceedings of the Pacific Northwest Software Quality Conference.** Portland, OR: [s.n.], 1998.

COLOMBO, C., MICALLEF, M., SCERRI, M., Verifying Web Applications: From Business Level Specifications to Automated Model-Based Testing. In: WORKSHOP ON MODEL-BASED TESTING, 9., 2014. Grenoble, France: [s.n.], 2014.

DAI Z. R. et al. From Design to Test with UML Applied to a Roaming Algorithm for Bluetooth Devices. In: TESTCOM, 2004, [S.l.: s.n.], 2004.

DALAL, S. R., et al., Model-based testing in practice, In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 21., 1999. **Proceedings of the 21st international conference on Software engineering.** New York, NY, USA: ACM, 1999. p. 285-294.

DI LUCCA, G. et al., Testing Web Applications, In: IEEE INTERNATIONAL CONFERENCE ON SOFTWARE MAINTENANCE, 18., 2002. **Proceedings of 18th IEEE International Conference on Software Maintenance.** [S.l.: s.n.], 2002.

ECLIPSE FOUNDATION, **Eclipse - The Eclipse Foundation open source community website**. Disponível em: < https://eclipse.org/home/index.php>. Acesso em: Dezembro de 2014.

ELSA, **ELSA Brasil**, Disponível em: <http://www.elsa.org.br/>. Acesso em: Dezembro de 2013.

FRÖHLICH, P., LINK, J., Automated Test Case Generation from Dynamic Models, In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 14., 2000. **Proceedings of the 14th European Conference on Object-Oriented Programming**. London, UK: Springer-Verlag, 2000. p. 472-492.

HARTMANN, J., IMOBERDORF, C., MEISINGER, M. UML-based integration testing. **ACM SIGSOFT Software Engineering Notes**. [S.l.]: v. 25, No. 5, p. 60-70, August, 2000.

HARTMANN, J. et al. A UML-based approach to system testing. **Innovations in Systems and Software Engineering**, [S.l.]: v. 1, n. 1, p. 12-24, 2005.

HECKEL, R., LOHMANN, M., Towards model-driven testing. **Electronic Notes in Theoretical Computer Science**, [S.l.]: v. 82, n. 6, p. 33-43, 2003.

HEINECKE, A. et al., Generating Test Plans for Acceptance Tests from UML Activity Diagrams, In: IEEE INTERNATIONAL CONFERENCE AND WORKSHOPS ON THE ENGINEERING OF COMPUTER-BASED SYSTEMS, 17., 2010. **Proceedings of the 2010 17th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems**, Washington, DC, USA: IEEE Computer Society, 2010. p. 57-66.

HELLESOY, A., **Cucumber**, Disponível em: <http://cukes.info/>. Acesso em: Dezembro de 2013.

HOLT, N. E., BRIAND, L. C., TORKAR, R., Empirical evaluations on the cost-effectiveness of state-based testing: An industrial case study. **Information and Software Technology**, [S.l.]: v. 56, n. 8, p. 890-910, 2014.

IBRAHIM, R. et al., An automatic tool for generating test cases from the system’s requirements. In: IEEE 7TH INTERNATIONAL CONFERENCE ON COMPUTER AND INFORMATION TECHNOLOGY, 7, 2007. Fukushima, Japan: [s.n.], 2007.

JBEHAVE, **What Is JBehave?**, Disponível em: <http://jbehave.org>. Acesso em: Dezembro de 2013.

JEEVARATHINAM, R., THANAMANI, A., Test Case Generation using Mutation Operators and Fault Classification. **International Journal of Computer Science and Information Security,** USA, v. 7, n. 1, p. 190-195, January 2010.

JENA, A. K., SWAIN, S. K., MOHAPATRA, D. P., A novel approach for test case generation from UML activity diagram. In: ISSUES AND CHALLENGES IN INTELLIGENT COMPUTING TECHNIQUES INTERNATIONAL CONFERENCE, 2014. [S.l.]: IEEE, 2014. p. 621-629.

JUNIT, **Resources for Test Driven Development**, Disponível em: <http://www.junit.org>. Acesso em: Dezembro de 2013.

KANSOMKEAT, S., RIVEPIBOON, W. Automated-generating test case using UML statechart diagrams. In: CONFERENCE OF THE SOUTH AFRICAN INSTITUTE OF COMPUTER SCIENTISTS AND INFORMATION TECHNOLOGISTS ON ENABLEMENT THROUGH TECHNOLOGY, 2003. **Proceedings of the 2003 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology**. Republic of South Africa: South African Institute for Computer Scientists and Information Technologists, 2003. p. 296-300.

KITCHENHAM, B. A. et al., Preliminary guidelines for empirical research in software engineering. **IEEE Transactions on Software Engineering,** [S.l.]: v. 28, n. 8, p. 721-734, 2002.

KRISHNAN, P., PARI-SALAS, P. Model-based testing and the UML testing profile. **Semantics and algebraic specification**. [S.l.]: v. 5700,. p. 315-328, 2009.

KOUDELIA, N. **Acceptance Test-Driven Development**, 2011 Master's Thesis in Information Technology (Software engineering) - Department of Mathematical Information Technology, University of Jyväskylä, Jyväskylä, Finland, December 2011.

LIMA, H. et al., Automatic Generation of Platform Independent Built-in Contract Testers, In: SIMPÓSIO BRASILEIRO DE COMPONENTES, ARQUITETURAS E REUTILIZAÇÃO DE SOFTWARE, 2007. [S.l.: s.n.], 2007 .

LINDA, **Projeto LINDA Brasil**. Disponível em: <http://www.geeafab.com/#!linda-brasil/c1r4x>. Acesso em: Dezembro de 2013.

LINZHANG, W. et al., Generating test cases from UML Activity diagram based on gray-box method. In: ASIA-PACIFIC SOFTWARE ENGINEERING CONFERENCE, 11., 2004. **Proceedings of the 11th Asia-Pacific Software Engineering Conference**. [S.l.]: IEEE, 2004. pp. 284–291.

LUMERTZ, P., BACELO, A. P. T., OLIVEIRA, T. Quantools: A MDA transformation approach**, Revista de Informática Teórica e Aplicada.** [S.l.]: v. 16, n. 2, p. 53-56, 2009.

MADHAVAN, A., **Semi Automated User Acceptance Testing using Natural Language Techniques.** 2014. Thesis (Master of Science Computer Science) - Iowa State University, USA, 2014.

MAFRA J. N. D., **Engenharia Direta e Reversa entre Requisitos e Testes**. 2008. Trabalho apresentado ao Curso de Mestrado em Ciência da Computação - Universidade Federal de Pernambuco, Centro de Informática, Recife, Pernambuco, Brasil, 2008.

MAJCHRZAK, T., SIMON, A. Using spring Roo for the test-driven development of Web applications. In: ANNUAL ACM SYMPOSIUM ON APPLIED COMPUTING, 27., 2012. **Proceedings of the 27th Annual ACM Symposium on Applied Computing.** New York, NY, USA: ACM, 2012. p. 664-671.

MARUCCI, R. et al. Oorts/prodes: Definição de técnicas de leitura para um processo de software orientado a objetos. In: SIMPÓSIO BRASILEIRO DE QUALIDADE DE SOFTWARE, 2002. [S.l.: s.n.], 2002.

MLYNARSKI, M., et al. From design models to test models by means of test ideas. In: INTERNATIONAL WORKSHOP ON MODEL-DRIVEN ENGINEERING, VERIFICATION AND VALIDATION, 6., 2009. **Proceedings of the 6th**

International Workshop on Model-Driven Engineering, Verification and Validation. New York, NY, USA: ACM, 2009.

NARESH, A. Testing from use cases using path analysis technique. In: INTERNATIONAL CONFERENCE ON SOFTWARE TESTING ANALYSIS & REVIEW, 2002. [S.l.: s.n.], 2002.

NAYAK, A., SAMANTA, D., Model-based test cases synthesis using UML interaction diagrams, **SIGSOFT Softw. Eng. Notes**. [S.l.]: v. 34, n. 2, p. 1-10, February 2009.

NAYAK, A., SAMANTA, D. Synthesis of Test Scenarios Using UML Sequence Diagrams, **ISRN Software Engineering**. [S.l.]: v.. 2012, 22 pages, 2012.

NEBUT, C. et al., Automatic Test Generation: A Use Case Driven Approach, **IEEE Trans. Softw. Eng.** [S.l.]: v. 32, n. 3, p. 140-155, March 2006.

NEGARA, N., STROULIA, E. Automated Acceptance Testing of JavaScript Web Applications. In: WORKING CONFERENCE ON REVERSE ENGINEERING, 19., 2012. [S.l.]: IEEE, 2012. p. 318-322.

NORTH, D. **Introducing BDD**, 2006. Disponível em: <http://dannorth.net/introducing-bdd/>. Acesso em: Dezembro de 2013.

OHNO, T., **O Sistema Toyota de Produção Além Da Produção**. Bookman, 1997.

OMG. **UML Testing Profile Version 1.0.** Technical Report PTC/05-07-07, OMG, 2005. 113 p. Disponível em: <http://www.omg.org/cgi-bin/doc?formal/05-07-07>. Acesso em: Dezembro de 2013.

OMG. **Meta Object Facility (MOF) Specification**. Technical Report PTC/06-06-01, OMG, 2006. 90p. Disponível em: <http://www.omg.org/cgi-bin/doc?formal/2006-01-01>. Acesso em: Dezembro de 2013.

OMG. **MOF 2.0/XMI Mapping, Version 2.1.1.** Technical Report Formal/07-12-01, OMG, 2007. 120 p. Disponível em: < http://www.omg.org/spec/XMI/2.1.1/PDF/index.htm>. Acesso em: Dezembro de 2013.

OMG. **Model Interchange Wiki**. Disponível em: <http://www.omgwiki.org/model-interchange/doku.php>. Acesso em: Dezembro de 2013.

PEZZÈ, M., **Software testing and analysis:** process, principles, and techniques, 487 p. Hoboken, N.J.: John Wiley & Sons, 2008.

PHILIPS, M. et al., Automated Test Case Generation Using Multiple Modelling Techniques. **International Journal of Science and Research**. [S.l.]: v. 3, n. 3, 2014.

PICHLER, R. **User Stories or Use Cases?**, 2010. Disponível em: <http://www.romanpichler.com/blog/user-stories/user-stories-or-use-cases/>. Acesso em: Dezembro de 2013.

PICHLER, R. **User Story Modeling**, 2011. Disponível em: <http://www.romanpichler.com/blog/user-stories/user-story-modelling/>. Acesso em: Dezembro de 2013.

PICKIN, S. et al. System test synthesis from UML models of distributed software. In: FORMAL TECHNIQUES FOR NETWORKED AND DISTRIBUTED SYSTEMS, 2002. [S.l.]: Springer Berlin Heidelberg, 2002. p. 97-113.

PICKIN, S. et al, Test Synthesis from UML Models of Distributed Software, **IEEE Trans. Softw. Eng.** . [S.l.]: v. 33, n. 4, p. 252-269, April 2007.

PUGH, K. **Lean-agile acceptance test-driven development**. [S.l.]: Pearson Education, 2010.

RIEBISCH, M. et al., UML-Based Statistical Test Case Generation, In: INTERNATIONAL CONFERENCE NETOBJECTDAYS ON OBJECTS, COMPONENTS, ARCHITECTURES, SERVICES, AND APPLICATIONS FOR A NETWORKED WORLD, 2002. **Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World**. London, UK: Springer-Verlag, 2002, p. 394-411.

RSPEC, **RSpec**, Disponível em: http://rspec.info/. Acesso em: Dezembro de 2013.

RUMPE, B. Model-based testing of object-oriented systems. **Formal Methods for Components and Objects**. [S.l.: s.n.], 2003.

SARMA, M., MALL, R., Automatic Test Case Generation from UML models, In: IEEE 10TH INTERNATIONAL CONFERENCE ON INFORMATION TECHNOLOGY, 2007, [S.l.: s.n.], 2007.

SCHAEFER, C. J., DO, H., Model-based exploratory testing: A controlled experiment. In: SOFTWARE TESTING, VERIFICATION AND VALIDATION WORKSHOPS, 7., 2014. **IEEE Seventh International Conference on software testing, verification and validation workshops**. [S.l.]: IEEE, 2014. p. 284-293.

SCHIEFERDECKER, I. et al. The UML 2.0 testing profile and its relation to TTCN-3. **Testing of Communicating Systems**. [S.l.: s.n.], 2003.

SCHIEFERDECKER, I., GRABOWSKI, J. The Graphical Format of TTCN-3 in the context of MSC and UML. **Telecommunications and beyond: The Broader Applicability of SDL and MSC**. [S.l.: s.n.] 2003.

SCHWABER, K., **Agile project management with Scrum**. Redmond: Microsoft press, 2004.

SELENIUM, **Browser automation framework.** Disponível em: <http://code.google.com/p/selenium/>. Acesso em: Dezembro de 2013.

SELENIUM, **Page Objects**. Disponível em: <https://code.google.com/p/selenium/wiki/PageObjects>. Acesso em: Dezembro de 2013.

SPARX, Sparx Systems Pty Ltd. **Enterprise Architect UML modeling tool**. Disponível em: <http://www.sparxsystems.com/>. Acesso em: Dezembro de 2013.

SOLÍS, C., WANG, X. A study of the characteristics of behavior driven development. In: EUROMICRO CONFERENCE ON SOFTWARE ENGINEERING AND ADVANCED APPLICATIONS, 37., 2011. **37th EUROMICRO Conference on software engineering and advanced applications**. [S.l.]: IEEE, 2011. p. 383-387.

SOMMERVILLE, I., **Engenharia de software**, 8. ed. São Paulo: Pearson Prentice Hall, c2007. xiv, 552 p. : il, 2007.

TRETMANS, J., BRINKSMA, E. TorX: Automated model based testing. In: EUROPEAN CONFERENCE ON MODEL DRIVEN SOFTWARE ENGINEERING, 1., 2003. Nuremberg, Germany: [s.n.], 2003.

TSAI, W. et al., Scenario-Based Test Case Generation for State-Based Embedded Systems, In: PERFORMANCE, COMPUTING, AND COMMUNICATIONS CONFERENCE, 2003. **Conference Proceedings of the 2003 IEEE International**. [S.l.]: IEEE, 2003. p. 335 – 342.

WAZLAWICK, R. S., **Engenharia de software: conceitos e práticas.** Rio de Janeiro: Elsevier, 2013.

WINTER, M. Managing object-oriented integration and regression testing. In: EUROSTAR, 1998. Munich, Germany: [s.n.], 1998.

WOHLIN, C. et al., **Experimentation in software engineering**. Springer, 2012.

W3C, **W3C Recommendation, XML Path Language (XPath) 2.0**. 2010. Disponível em: <http://www.w3.org/TR/xpath20/>. Acesso em: Dezembro de 2013.

XU, D. A tool for automated test code generation from high-level petri nets. In: INTERNATIONAL CONFERENCE ON APPLICATIONS AND THEORY OF PETRI NETS, 32., 2011. **Proceedings of the 32nd international conference on Applications and theory of Petri Nets**.. Springer-Verlag, Berlin, Heidelberg, 2011. p. 308-317,