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

HELAINE CRISTINA SILVA SOUSA

CONSTRUÇÃO AUTOMATIZADA DE CASOS DE TESTE USANDO ENGENHARIA DIRIGIDA POR MODELOS

São Luís
2009

HELAINE CRISTINA SILVA SOUSA

CONSTRUÇÃO AUTOMATIZADA DE CASOS DE TESTE USANDO ENGENHARIA DIRIGIDA POR MODELOS

Dissertação apresentada ao Curso de Pós-Graduação em Engenharia de Eletricidade da Universidade Federal do Maranhão para obtenção do título de Mestre em Engenharia de Eletricidade, área de Concentração Ciência da Computação.

Orientador: Ph.D. Zair Abdelouahab
Co-orientador: Dr. Denivaldo Cicero Pavão Lopes

São Luís
2009

Sousa, Helaine Cristina Silva

Construção Automatizada de Casos de Teste Usando Engenharia Dirigida por Modelos/ Helaine Cristina Silva Sousa. – São Luís, 2009.

142 f.

Orientador: PhD. Zair Abdelouahab

Co-orientado: Dr. Denivaldo Cicero Pavão Lopes

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

1. Software - Testes. 2. Abordagem Dirigida por Modelos. 3. Software - Desenvolvimento. I. Título.

CDU 004.415.53

CONSTRUÇÃO AUTOMATIZADA DE CASOS DE TESTE USANDO ENGENHARIA DIRIGIDA POR MODELOS

Helaine Cristina Silva Sousa

Dissertação aprovada em 14 de maio de 2009.

Prof. Zair Abdelouahab, Ph.D.
(Orientador)

Prof. Denivaldo Cicero Pavão Lopes, Dr.
(Co-orientador)

Prof. Pedro de Alcântara dos Santos Neto, Dr.
(Membro da Banca Examinadora)

Profa. María del Rosario Girardi, Ph.D.
(Membro da Banca Examinadora)

Prof. Sofiane Labidi, Dr.
(Membro da Banca Examinadora)

Aos meus pais Hely e Neide Sousa

AGRADECIMENTOS

Em primeiro lugar a Deus, pela vida concedida, pela força, paz, sabedoria e perseverança, que me permitiram concluir este trabalho e superar os desafios encontrados ao longo da minha vida.

Aos meus pais, Neide e Hely Sousa, que são tudo na minha vida, por estarem comigo nas horas boas e más, sempre dando todo o apoio que precisei. Sem eles minhas conquistas não seriam possíveis. Devo tudo a eles.

Aos meus avôs, Abdias e Orlandina da Silva, pelo exemplo de vida, pelo carinho e amor a mim dedicados. Início de tudo.

Aos meus irmãos, Heloneida Mouta, Markson Sousa e Helines Sousa, pelo companheirismo, apoio e incentivo dado em todos os momentos difíceis ao longo desta e de outras trajetórias.

Ao meu orientador, professor Ph.D. Zair Abdelouahab, pela confiança a mim depositada, pela oportunidade a mim concedida, pela orientação, dedicação e conhecimento passado. Os meus sinceros agradecimentos.

Ao meu co-orientador, professor Dr. Denivaldo Lopes, pela orientação, pela oportunidade concedida, pela paciência e apoio para a concretização deste trabalho, pela fé na minha evolução e pelo incentivo em todos os momentos. Meus sinceros agradecimentos.

Ao professor Dr. Sofiane Labidi, pelo estímulo que se iniciou na graduação até o término deste mestrado, pela boa indicação e confiança.

Ao meu amigo professor Dr. Luís Carlos Fonseca, pelo incentivo e força para o ingresso no Mestrado.

Aos meus colegas e amigos de laboratório, Aline Lopes, Falkner Moraes, Flávio Ramos, Márcio Cristiano, Johnneth Fonseca, Simone Bandeira e Ricardo Ataíde, por terem me acolhido como integrante do laboratório e por ter propiciado ótimos momentos tanto de descontração como de muita troca de conhecimentos.

Aos meus colegas e amigos de mestrado, Adriana Leite, Alex Barradas Filho, Geysa Dias, Pedriana Pavão, em especial ao meu amigo Irlandino Oliveira pelo apoio incondicional, amizade, pelas trocas de conhecimento e diversões.

Aos meus colegas e amigos de almoço, Geraldo Braz Junior, Gilberto Cunha Junior e Victor Hugo Barros, pelos momentos de descontração na hora do almoço e do sorvete, pelo incentivo e torcida para a finalização deste trabalho.

As minhas amigas/irmãs, Aleandra Vale, Alexandra Nascimento e Tereza Pereira, pela amizade verdadeira, pela paciência, pelo incentivo e pela compreensão diante da minha ausência.

Aos meus amigos de graduação, Lívia Ferreira, Marcelo Cardoso e Silvia Linhares e amigos de estágio Claudineth Picanço, Fabio Henrique Carneiro e Jequedima Caldas pela amizade sincera e por sempre estarem comigo.

Ao meu cunhado Adenilson Mouta pelas palavras de incentivo, pelos conselhos e carinho.

A todos os meus parentes e amigos pelo carinho, amor e pela torcida incondicional.

Ao Programa de Pós-Graduação de Engenharia de Eletricidade, representados por Prof. Dr. Sebastian Yuri Catunda e Profª. Ph.D. Maria da Guia e funcionário representado por Alcides Martins Neto.

Ao Prof. Dr. Pedro de Alcântara e Profª Dr. Maria del Rosário que disponibilizaram um pouco do seu tempo para participar da banca examinadora do meu trabalho. Muito obrigada.

A FAPEMA em parceria com a CAPES pelo financiamento de minha pesquisa.

A todos aqueles que direta ou indiretamente contribuíram para a realização deste trabalho.

“Para realizar grandes conquistas, devemos não apenas agir, mas também sonhar; não apenas planejar, mas também acreditar”

Anatole France

RESUMO

O surgimento das Abordagens Dirigidas por Modelos fornece uma nova alternativa para o gerenciamento da complexidade do desenvolvimento de *software*, para criação de testes de *software*, para automação dos processos de testes e para fornecimento da ampla reutilização de modelos desenvolvidos durante a fase de análise dos requisitos e projeto de *software*, reduzindo a possível injeção de erros e o tempo de desenvolvimento do *software*. No entanto, com a utilização das Abordagens Dirigidas por Modelos, possíveis erros podem ser injetados na criação das regras de transformação para implementar um determinado sistema de *software*. Propõe-se neste trabalho metamodelos de testes, uma metodologia e um *framework* ATCM (*Automatic Test Case based on Models*) com a finalidade de gerar casos de teste a fim de testar o código-fonte gerado por uma Abordagem Dirigida por Modelos. Um protótipo do *framework* ATCM foi desenvolvido, fornecendo ferramentas que minimizam a injeção de erros durante a geração dos casos de teste, tornando esta tarefa menos dependente de pessoas e menos propensa a erros reduzindo o tempo de desenvolvimento e provendo maior qualidade e eficiência nos casos de teste gerados.

Palavras-chave: Testes, Modelos, Metamodelagem, Abordagem Dirigida por Modelos, Engenharia Dirigida por Modelos, Arquitetura Dirigida por Modelos e Teste Dirigido por Modelos.

ABSTRACT

The emergence of model driven approaches provides a new alternative for managing the complexity involved in the creation of test cases, for enhancing the automation of software testing and for promoting the broad reuse of models developed during the analysis of requirements and design of software. In addition, it reduces the injection of errors and software development time. However, in the use of model driven approaches, possible errors can be injected during the manual creation of transformation rules applyied to develop a software system. In this dissertation, we propose metamodels for test, a methodology and a framework called Automatic Test Case based on Models (ATCM) in order to generate test cases to test the source code generated by an model driven approach. A prototype of the framework ATCM was developed, providing tools that minimize the injection of errors during the generation of test cases, making this task less dependent on people and less error-prone reducing the development time and providing high quality and efficiency of test cases.

Keywords: Testing, Models, Metamodels, Approaches Model Driven, Model Driven Engineering, Model Driven Architecture, Model Driven Testing.

LISTA DE FIGURAS

p.

LISTA DE TABELAS

p.

LISTA DE LISTAGEM

p.

LISTA DE SIGLAS

ATCM	*Automatic Test Case based on Models*
ATL	*Atlas Transfomation Language*
ATM	*Automatic Teller Machine*
DSL	*Domain-Specific Language*
EMF	*Eclipse Modeling Framework*
IBM	*International Business Machines*
IDE	*Integrated Development Environment*
MBT	*Model Based Testing*
MDA	*Model Driven Architecture*
MDE	*Model Driven Engineering*
MDT	*Model Driven Testing*
MDx	*Model Driven Approaches*
MOF	*Meta Object Facility*
MPIT	*Metamodel for Platform Independent Testing*
MPST	*Metamodel for Specific Independent Testing*
MT4MDE	*Mapping Tool for MDE*
OCL	*Object Constraint Language*
OMG	*Object Management Group*
PIM	*Platform Independent Model*
PSM	*Platform Specific Model*
QVT	*Query/View/Transformation*
SAMT4MDE	*Semi-Automatic Matching Tool for MDE*
SUT	*System Under Test*
T2MDT	*Testing for Model Driven Testing*
TBM	Teste Baseado por Modelo
TDD	*Testing Driven Developed*
U2TP	*UML 2.0 Testing Profile*
UML	*Unified Modeling Language*
V&V	Verificação e Validação
XMI	*XML Metadata Interchange*
XML	*eXtensible Markup Language*
XP	*eXtreme Programming*

SUMÁRIO

1 INTRODUÇÃO

Este capítulo descreve a contextualização, a problemática, a motivação para encontrar uma possível solução, os objetivos geral e específicos para alcançar a solução proposta, a metodologia utilizada e a apresentação dos capítulos desta dissertação.

1.1 Contextualização

O uso do *software* está cada vez mais presente nas áreas da atividade humana. Entretanto, *software* estão cada vez mais complexos devido a grande diversidade de tecnologias e plataformas envolvidas e a constante necessidade de evolução, de manutenção e de integração com novas tecnologias (OMG, 2003). Estes fatores dificultam a criação de *software* confiáveis e de qualidade, implicando muitas vezes em projetos de *software* mal sucedidos.

Muitas tecnologias, metodologias e ferramentas surgiram com o objetivo de auxiliar os desenvolvedores de *software* no gerenciamento da complexidade dos sistemas de *software*, por exemplo, padrões de projeto (GAMMA et al., 1995), XP (*eXtreme Programming*) (BECK, 1999) e modelo (OMG, 2003). Algumas dessas soluções desapareceram e outras evoluíram.

O uso de modelos durante o desenvolvimento de *software* é de extrema importância, pois elevam o nível de abstração. As Abordagens Dirigidas por Modelos (MDx, *Model Driven Approaches*) surgiram como um complemento aos métodos tradicionais de desenvolvimento de *software*, para enfrentar a crescente complexidade no processo de criação, evolução e manutenção de sistemas de *software* (SOUSA et al., 2008).

Abordagens tais como MDE, *Model Driven Engineering* (HADJ et al., 2008) (CANCILA, 2008), por exemplo, EMF, *Eclipse Modeling Framework* (STEINBERG, 2008) (LITANI, 2005), MDA, *Model Driven Architecture* (OMG, 2003) (STAHL et al., 2006) e MDT, *Model Driven Testing* (SUSS et al., 2008) (IBM, 2003) têm como objetivo o fornecimento de respostas rápidas para atender novos requisitos tanto funcionais e não-funcionais e a adaptação de novas tecnologias aos sistemas legados, favorecendo a integração entre diferentes tecnologias.

Apesar das vantagens fornecidas pelas Abordagens Dirigidas por Modelos, é irreal considerar que os sistemas desenvolvidos utilizando tais abordagens estão livres de possíveis problemas durante o processo do seu desenvolvimento. Por serem sistemas complexos, os

requisitos não são fielmente transcritos dos modelos aos ambientes de execução, sendo comum que erros sejam injetados no código-fonte em alguma fase do desenvolvimento.

Portanto, a integração de testes no início da fase do desenvolvimento do *software* se torna importante para a garantia de qualidade do produto final de *software* (IBM, 2003). Com a integração do processo de teste cada vez mais cedo nas fases de desenvolvimento do *software*, é possível que a descoberta de erros e falhas no projeto ajudem a diminuir o custo e o retrabalho.

1.2 Problemática

As Abordagens Dirigidas por Modelos vêm atraindo rapidamente a atenção de pesquisadores, resultando em métodos de construção de sistemas que ofereçam vantagens sobre abordagens tradicionais em termos de confiabilidade, consistência e sustentabilidade.

Entretanto, para que as Abordagens MDx forneçam uma alternativa adequada às abordagens existentes, elas devem oferecer uma sustentação comparável aos processos da tecnologia de programação existente.

Assim como as técnicas tradicionais de desenvolvimento de sistemas de *software*, uma Abordagem Dirigida por Modelos necessita de rigorosos processos de testes, durante o desenvolvimento de *software*, desde a criação dos modelos até a geração do código-fonte, pois se alguma etapa do processo de desenvolvimento de um sistema de *software* em uma visão MDx não estiver correta, poderá ocorrer a injeção de erros no sistema (JAVED et al., 2007).

Na fase de testes, concentram-se os maiores esforços, custos e tempo do desenvolvimento de *software* (FLEEGER, 2004). A construção manual de casos de teste implica em uma grande demanda de tempo e uma parcela significativa do custo total do desenvolvimento do *software* e está sujeita a erros humanos. Uma forma de reduzir tais problemas é a automação das atividades de teste, incluindo a criação de casos de teste. A automação da criação dos casos de teste é uma forma de reduzir o tempo gasto em testes, tornando mais viável e justificável a utilização de testes desde o início do desenvolvimento de *software*.

1.3 Motivação

O processo de teste é fundamental para garantia de qualidade dos *software*. A utilização do processo de teste cada vez mais cedo, paralelamente as outras fases de desenvolvimento do *software*, ajuda a diminuir o custo e o retrabalho, pois dessa forma, falhas de projeto e de implementação podem ser descobertas mais cedo. Quanto mais cedo um defeito for descoberto, os custos serão menores para corrigir os erros associados (SOMMERVILLE, 2007).

As técnicas para Validação e Verificação (V&V) são algumas das atividades relacionadas à garantia da qualidade (JAVED et al., 2007). Entretanto, essas atividades apesar de serem importantes, boa parte das organizações têm dúvidas sobre em executar ou não tais atividades, devido aos altos custos para sua execução (PRESSMAN, 2006).

Embora os testes tenham um papel fundamental no aprimoramento da qualidade dos produtos de *software*, eles são deixados de lado por algumas organizações (IBM, 2003). Freqüentemente, essas atividades de testes são reduzidas ou mesmo eliminadas devido aos atrasos durantes as atividades de desenvolvimento e o alto custo no desenvolvimento de *software*.

Metodologias de desenvolvimento, tais como MDT (*Model Driven Testing*) (SUSS et al., 2008), enfatizam a necessidade de um processo rigoroso de teste durante o processo de desenvolvimento de *software*, essa abordagem é a garantia de uma boa implementação de MDA. As atividades de teste permitem a realização de melhorias no processo utilizado, ao mesmo tempo em que auxilia a garantia de qualidade do produto desenvolvido.

Embora as atividades de teste ainda não sejam utilizadas por todas as empresas de desenvolvimento de *software*, seu uso vem crescendo nos últimos anos, assim como a utilização de modelos de desenvolvimento de *software* (IBM, 2003). Um modelo é uma simplificação da realidade, permitindo uma validação ou proposição de teorias, com riscos e custos reduzidos. Utilizando modelos, pode-se entender melhor um sistema a ser desenvolvido (OMG, 2003).

A maioria dos *software* de modelagens comerciais (por exemplo, Rational XDE e Telelogic Tau) são destinados ao projeto e desenvolvimento de *software* em vez da garantia de qualidade e testes (IBM, 2003).

O teste de um sistema necessita de modelos que descrevam os elementos envolvidos durante as atividades de testes, tais como componentes alvos, carga de trabalho,

resultados esperados e ferramentas usadas. A formalização desses modelos torna possível sua documentação e facilita sua análise, permitindo não só a integração com os modelos do restante do sistema, mas também o reúso de documentação e de ferramentas nas diversas fases do desenvolvimento.

1.4 Objetivos

Como forma de minimizar os problemas da construção de testes, este trabalho traça alguns objetivos descritos nas próximas subseções.

1.4.1 Objetivo geral

O objetivo deste trabalho é propor uma abordagem dirigida por modelos e um protótipo que permitam a geração automática de casos de teste a fim de verificar o código-fonte de um sistema gerado por transformações de modelos.

1.4.2 Objetivos específicos

Para atingir o objetivo geral, os seguintes objetivos específicos tiveram que ser contemplados:

a) Desenvolvimento de uma metodologia para criação dos modelos de casos de teste;
b) Abordagem para modelagem de teste baseado em metamodelos independentes de plataforma de teste;
c) Desenvolvimento de um *framework* para geração automática de casos de teste;
d) Implementação do *framework* de testes utilizando uma ferramenta de criação de modelos de correspondência dentro do contexto MDE (SAMT4MDE);
e) Avaliação do *framework* e extensão proposto através de coleta dos dados de testes e estudo de caso.

1.5 Metodologia

Primeiramente, iníciou-se com a pesquisa bibliográfica, com o intuito de coletar informações de livros, teses e dissertações, periódicos, anais de congressos, especificações e *Web Sites* para uma total contextualização da literatura especializada.

Tal revisão literária consistiu no estudo mais aprofundado sobre a Engenharia de *Software*, mais precisamente, Testes de *Software*, em particular as técnicas e fases de testes. Pesquisou-se padrões que compõem a Abordagem Dirigida por Modelos, tais como, MDE (*Model Driven Engineering*) (KENT, 2002), MDD (*Model Driven Development*) (SALVADOR, 2007) (SWITHINBANK et al., 2005), MDA (*Model Driven Architecture*) (OMG, 2003) (OMG, 2008) (STAHL et al., 2006) e MDT (*Model Driven Testing*) (SUSS et al., 2008) (IBM, 2003).

O estudo sobre tais assuntos foi necessário para identificar e modelar elementos de testes.

Posteriormente, um estudo e a utilização de possíveis ferramentas e metodologias a serem utilizadas para o desenvolvimento do trabalho proposto, tais como:

- ATL (*Atlas Transformation Language*) - Linguagem para definição de transformação de modelos (ATL, 2006);
- EMF (*Eclipe Modeling Framework*) - É uma ferramenta que facilita o desenvolvimento baseado em aplicações Java e se constitui em uma plataforma para a integração de ferramentas de desenvolvimento (STEIBERG, 2008);
- Ecore - É uma linguagem de metamodelagem que faz parte da EMF. É o modelo usado para representar modelos em EMF, ou seja, é um metametamodelo (BUDINSKY et al., 2003);
- JUnit - *Framework* que auxilia a criação e a execução de testes unitários sobre classes Java. Com esta plataforma de teste, pode-se verificar se cada método de uma classe funciona de forma esperada, exibindo possíveis erros ou falhas (MASSOL, 2004);
- NUnit - *Framework* de testes unitário para linguagem de programação CSharp ou C# e Microsoft .Net (HAMILL, 2004);
- UML 2.0 (*Unified Modeling Language*) - É uma linguagem de modelagem para especificação, visualização, construção e documentação de artefatos UML de Sistemas de *Software* (OMG, 2008);

- U2TP (*UML 2.0 Testing Profile*) - Trata-se de um perfil definido com base no metamodelo de UML, visando a integração e o reúso de conceitos já existentes. Herda de UML a organização em metacamadas de abstração, usando pacotes e a extensabilidade para adaptação a domínios e plataformas especificadas (OMG, 2003);
- xUnit - Família de estruturas de testes de unidade usadas para escrever e executar testes repetitivos em aplicações de *software* (MESZAROS, 2007).

No percurso do desenvolvimento deste trabalho, os passos a seguir foram realizados para o desenvolvimento deste trabalho:

- Levantamento de trabalhos relacionados;
- Avaliação de ferramentas;
- Estudo das plataformas de testes *unit* tais como, xUnit, JUnit e NUnit;
- Levantamento das características necessárias para o desenvolvimento do metamodelo de teste independente de plataforma e metamodelos de testes específicos de plataformas utilizando a plataforma *Omondo eclipse* 3.2 com os *plug-ins* de modelagem EMF, GEF e UML 2.0;
- Formulação e concretização da metodologia proposta para a execução do *framework* para a geração automatizada de casos de teste;
- Prototipação do *framework* de teste para geração automatizada de casos de teste;
- Estudo das ferramentas MT4MDE e SAMT4MDE para a geração semi-automática das especificações das correspondências entre o metamodelo de teste independente de plataforma e metamodelos de testes específicos de plataformas;
- Implementação do *framework* de testes integrando as ferramentas MT4MDE e SAMT4MDE para geração de correspondências entre metamodelos de testes e o motor de tranformação ATL;
- Geração de especificações de correspondências entre metamodelos de testes;
- Criação de um exemplo ilustrativo que consiste de um modelo de negócio conforme o metamodelo de teste independente de plataforma proposto neste trabalho;
- Execução das regras de transformação em ATL modelo-a-modelo para geração do modelo de teste específico de plataforma;
- Criação de regras de transformação em ATL modelo-a-texto para geração do código-fonte de teste.

1.6 Apresentação da Dissertação

A presente dissertação está estruturada em 6 (seis) capítulos.

No Capítulo 1, apresenta-se uma descrição geral do trabalho, os objetivos e os elementos motivadores deste trabalho.

No Capítulo 2, apresenta-se a fundamentação teórica e tecnologias envolvidas na construção deste trabalho.

No Capítulo 3, apresenta-se as abordagens de testes pesquisadas, a diferença entre as abordagens tais como Testes Dirigidos por Modelos, Testes Baseados em Modelos, Desenvolvimento Dirigido a Teste e Especificação de Perfis de Teste UML 2.0. Ainda neste capítulo, trabalhos que utilizam tais abordagens são apresentados.

No Capítulo 4, descreve-se a proposta do *framework* para construção de teste, a metodologia desenvolvida, os metamodelos propostos para construção dos casos de teste, descrição das especificações das correspondências entre os metamodelos propostos e a prototipagem.

No Capítulo 5, apresenta-se um estudo de caso com a criação do modelo de negócio de um sistema ATM, a utilização do protótipo proposto, as definições de transformações transcritas na linguagem ATL, a geração do código-fonte de teste e a avaliação dos testes.

No Capítulo 6, apresenta-se as considerações finais desta pesquisa, os resultados obtidos, limitações e trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo descreve os principais conceitos e tecnologias que foram utilizados no desenvolvimento do *framework*, metodologia e metamodelos propostos para suportar a abordagem para criação de casos de teste.

2.1 Teste de *Software*

Testes são atividades cruciais no processo de V&V (Verificação e Validação) de *Software*. Este processo envolve altos custos e é fortemente dependente de automação. A falta de confiabilidade nos testes pode invalidar todo um processo de V&V de *Software*. Os testes precisam ser continuamente mantidos e executados durante todo o ciclo de vida de um *software*.

Com a utilização do processo de testes é possível uma redução significativa da probabilidade de ocorrência de uma ou mais falhas durante o tempo de vida de um *software* em produção, minimizando os riscos para o negócio e garantindo que as necessidades do cliente sejam atendidas.

Alguns conceitos são de extrema importância para se entender o processo de teste de *software*, o glossário padrão do IEEE dá as seguintes definições (IEEE, 1990):

- Defeito - é um ato inconsistente cometido por um indivíduo ao tentar entender uma determinada informação;
- Erro - é a manifestação concreta de um defeito num artefato de *software*. É qualquer estado intermediário incorreto ou resultado inesperado na execução de um programa;
- Falha - é o comportamento operacional do *software* diferente do esperado pelo usuário. Uma falha pode ser causada por diversos erros e alguns erros podem nunca causar uma falha;
- Teste - é o processo de analisar o *software* para detectar diferenças entre condições existentes e requeridas para avaliar as características do *software*. Testar um *software* implica descobrir falhas;
- Depurar - é o procedimento de localizar e corrigir erros.

A Figura 2.1 ilustra a diferença entre esses conceitos. Defeitos fazem parte do universo físico (a aplicação propriamente dita) e são causados por pessoas, ou seja, através do mal uso de uma determinada tecnologia. Portanto, defeitos podem ocasionar a manifestação

de erros em um produto, ou seja, a construção de um *software* de forma diferente ao que foi especificado (universo de informação) (DIAS Neto, 2008).

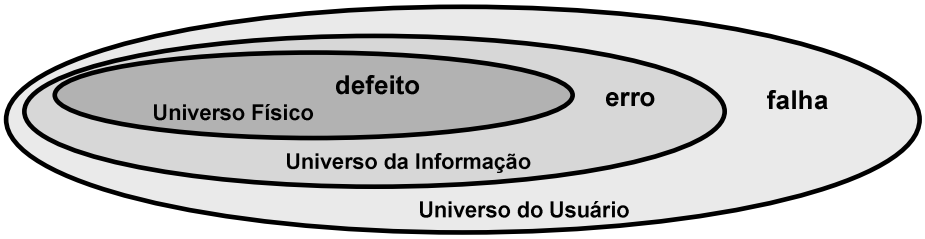


Figura 2.1 - Defeito x Erro x Falha (DIAS Neto, 2008)

Por fim, os erros geram falhas, que são comportamentos inesperados em um *software* que afetam diretamente o usuário final da aplicação (universo do usuário) e pode inviabilizar a utilização de um *software*. Dessa forma, exemplifica-se que teste de *software* revela somente falhas em um produto de *software*. Sendo necessária, após a execução dos testes, a execução de um processo de depuração para a identificação e correção dos defeitos que originaram essa falha. Portanto, testar não é depurar (DIAS Neto, 2008).

O processo de teste de *software* é composto por atividades, seu objetivo é executar um programa de teste com a finalidade de localizar falhas e avaliar sua qualidade. Este processo é muito importante para todas as fases do desenvolvimento de *software* e se bem aplicado, proporciona um produto final confiável, eficaz e com qualidade.

A Norma IEEE-829 define um conjunto de artefatos para as atividades de testes. Esta norma separa o processo de testes em três etapas: Preparação de Teste, Execução do Teste e Registro de Teste (IEEE, 1998).

A Figura 2.2 ilustra essas três etapas da atividade de testes conforme a Norma IEEE-829. Um conjunto de documentos para as atividades de teste de *software* são descritas a seguir:

- Plano de Teste - planejamento para execução do teste. Esta atividade de teste inclui a abrangência, abordagem, recursos e cronograma das atividades de teste;
- Especificação de Plano de Teste - refina a atividade anterior e identifica as funcionalidades e características a serem testadas pelo projeto;
- Especificação de Caso de Teste - define os casos de teste, incluindo dados de entrada, resultados esperados, ações e condições gerais para a execução do teste;
- Especificação de Procedimento de Teste - define os passos para executar um conjunto de casos de teste;

- Diário de Teste - nesta atividade localizam-se os registros cronológicos dos detalhes relevantes relacionados com a execução dos testes;
- Relatório de Incidente de Teste - nesta atividade, é documentado qualquer evento que ocorra durante a atividade de teste e que requeira análise posterior;
- Relatório Resumo de Teste - apresenta de forma resumida os resultados das atividades de teste associadas com uma ou mais especificações de projeto de teste, provendo avaliações baseadas nesses resultados;
- Relatório de Encaminhamento de Item de Teste - identifica os itens encaminhados para teste, essa atividade é necessário caso as equipes de desenvolvimento e de teste sejam distintas.

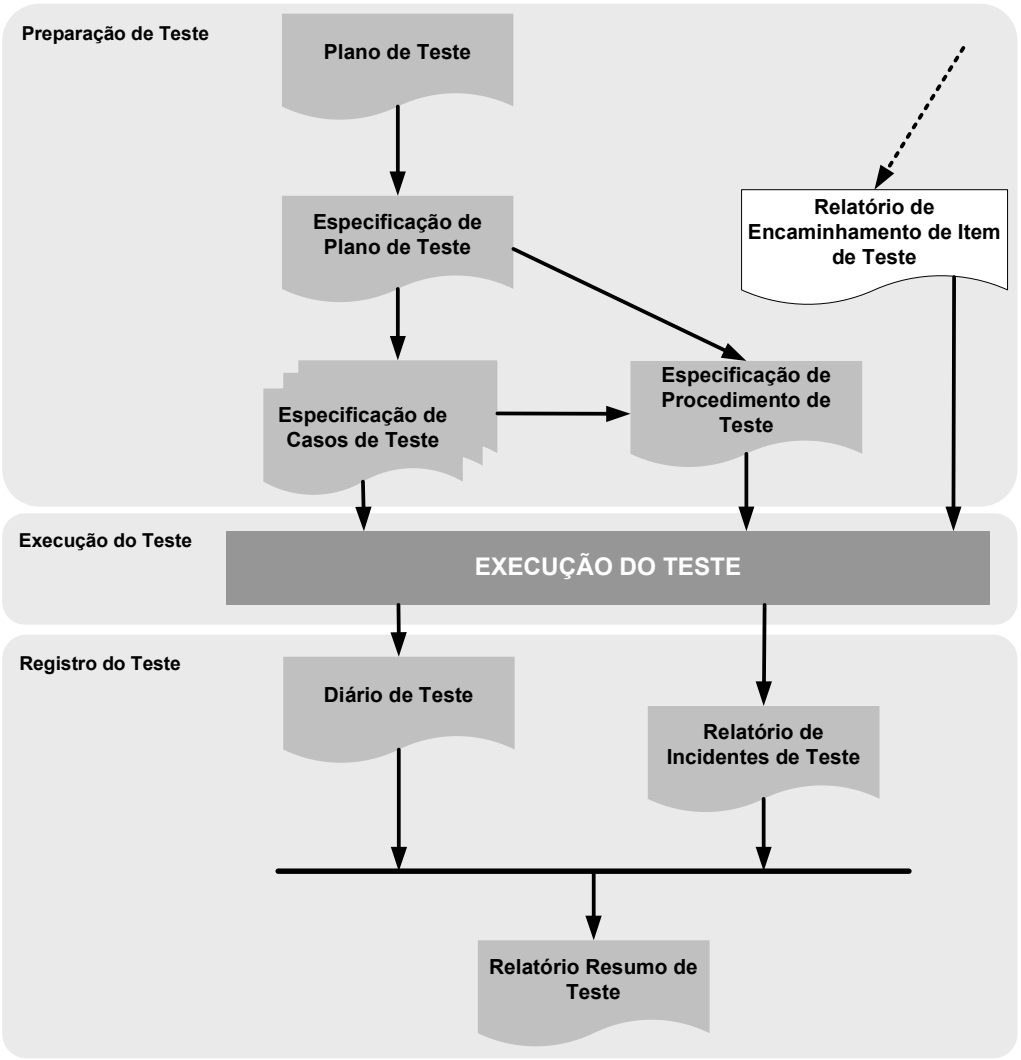


Figura 2.2 - Atividades de Teste conforme a Norma 829 IEEE (adaptado IEEE, 1998)

Durante a elaboração do planejamento de testes é que vêm as três perguntas básicas sobre qual escolha da estratégia de testes será utilizada para a preparação e execução dos testes: O que testar? (tipos de teste); Quando testar? (fases de teste); e Como testar? (técnicas de testes). A Figura 2.3 ilustra os relacionamentos entre os tipos, fases e técnicas de testes de *software* (BIASI, 2006).

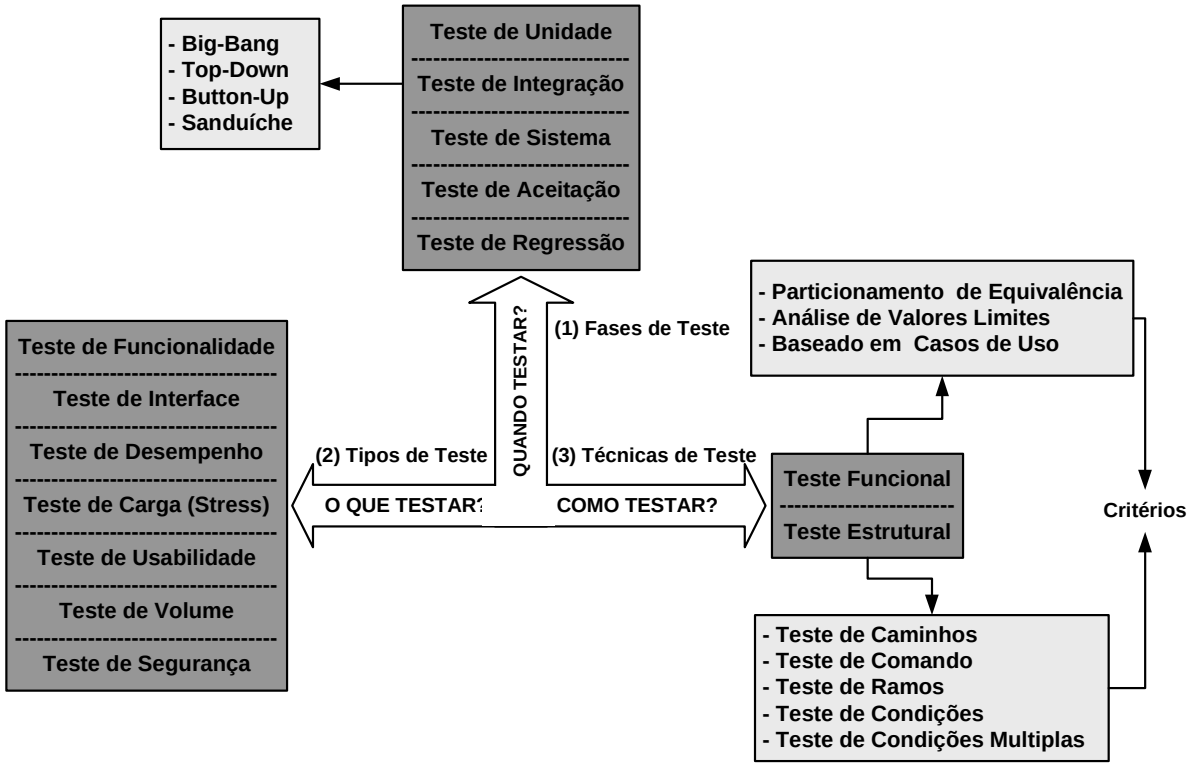


Figura 2.3 - Relação de nível, tipos e técnicas de teste (adaptado de BIASI, 2006)

2.2.1 Fases de teste de *software*

As fases de teste devem ocorrer em paralelo a fase do desenvolvimento de *software* como apresentado na Figura 2.4.

As principais fases de teste de *software* são (SOMMERVILLE, 2007):

- Teste de Unidade ou Teste de Módulos - tem o objetivo de encontrar defeitos nas unidades de projeto de *software* (componente ou módulo de *software*), explorando cada unidade do projeto, provocando erros ocasionados por defeitos de lógica e de implementação, verificando se cada unidade satisfaz as suas especificações estabelecidas no início do projeto;

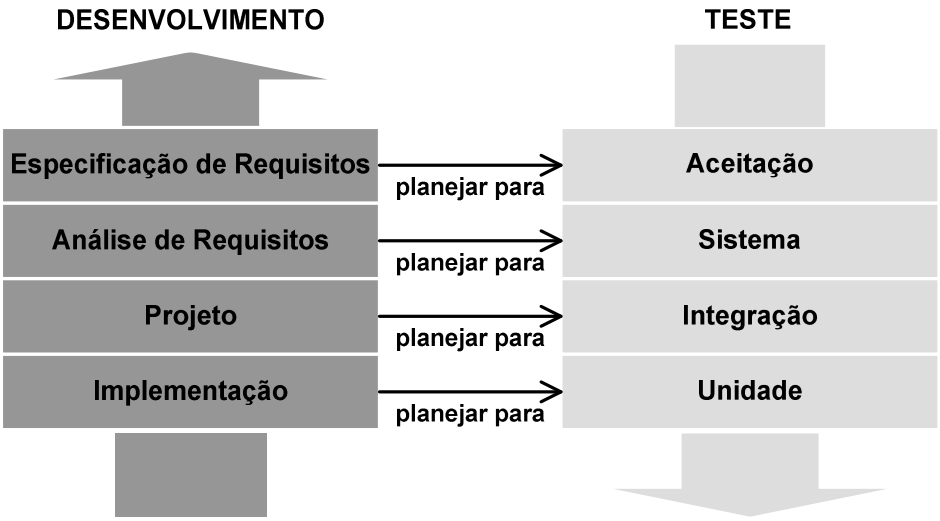


Figura 2.4 - Paralelismo entre as atividades de desenvolvimento e teste de *software*

- Teste de Integração - testa a integração de dois ou mais unidades que implementam funções ou características do sistema com a finalidade de provocar erros associadas às interfaces entre os módulos. Estes módulos são agrupados para compor os subsistemas, conforme a arquitetura definida no projeto inicial, ou seja, seu objetivo é a localização de defeitos de interface entre os módulos e os subsistemas. Tem-se 4 (quatro) abordagens para a execução dessa fase de teste, são elas (PFLEEFER, 2004):
  - Integração *Big-Bang* - nesta abordagem, após realizado os testes dos módulos isoladamente, o teste de integração é feito de forma não incremental, ou seja, testa-se todos os módulos do sistema ao mesmo tempo, porém usualmente isto resulta em um caos, pois um conjunto de erros são encontrados, mas o testador não sabe em qual junção de módulos originou-se o erro, impossibilitando, assim, a localização do defeito;
  - Integração *Top-Down* - é uma abordagem de teste que visa testar a integração dos módulos de forma incremental, onde o nível superior da hierarquia, geralmente o módulo de controle, é testado isoladamente, em seguida todos os componentes chamados por este módulo são combinados e testados como uma única unidade. Os módulos são integrados pela hierarquia de controle;

- Integração *Bottom-Up* - os módulos são integrados e testados de baixo para cima, ou seja, cada módulo no nível inferior da hierarquia é testado individualmente, incluindo no teste os módulos de forma ascendente;
- Integração "sanduíche" - esta abordagem é a junção das abordagens *Top-Down* e *Bottom-Up*. O sistema é visualizado em três camadas, assim como um sanduíche, a camada alvo no meio, os níveis acima do alvo onde é utilizada a abordagem *Top-Down* e os níveis abaixo do alvo que é utilizada a abordagem *Bottom-Up*. Esta abordagem de teste permite que o teste de integração comece cedo no processo de teste.

- Teste de Sistema - assegura que o sistema desempenhe as funções desejadas pelos clientes, ou seja, que o sistema satisfaça os requisitos especificados na fase de análise. Esta fase de teste na verdade é um conjunto de diferentes testes cuja finalidade principal é testar por completo o sistema baseado em computador, este teste verifica se os elementos do sistema foram adequadamente integrados e executam suas funções a eles determinadas (PRESSMAN, 2006);
- Teste de Aceitação - possibilita que os clientes e usuários determinem se o sistema desenvolvido satisfaz suas necessidades e expectativas. Este teste visa verificar o *software* em conjunto, incorporando todos os componentes, tanto o hardware como *software*, para determinar se o sistema completo satisfaz os requisitos informados pelos usuários (PRESSMAN, 2006);
- Teste de Regressão - este teste é aplicado na fase de manutenção dos sistemas, aplica-se este teste em versões atuais do *software*, para verificar que não surgiram novos defeitos em componentes testados anteriormente. Se surgirem novos defeitos no momento da junção dos novos componentes ou alterações, com os componentes anteriormente testados do sistema, então considera-se que o sistema regrediu (PRESSMAN, 2006).

2.2.2 Tipos de teste de *software*

Os tipos de teste referem-se às características do *software* que podem ser testadas, são eles (BIASI, 2006):

- Teste de Funcionalidade - visam verificar se as funcionalidades especificadas foram devidamente implementadas. Este teste determina se as funções descritas na especificação dos requisitos são realmente realizadas pelo *software*;

- Teste de Interface - ocorre quando módulos ou subsistemas são integrados para criar sistemas maiores, tem como objetivo a detecção de erros devido a defeitos ou suposições inválidas sobre interfaces;
- Teste de Desempenho - é idealizado para testar o desempenho de *run-time* do *software* dentro do contexto de um sistema integrado. O teste de desempenho ocorre ao longo de todos os passos do processo de teste. São freqüentemente acoplados a testes de carga e usualmente requerem instrumentação tanto de hardware quanto de *software*. Este teste compara o sistema com o restante dos requisitos de *software* e hardware;
- Teste de Carga - executa o sistema a ser testado de tal forma que demanda recursos em quantidade, freqüência ou volumes anormais, ou seja, testa o sistema além de sua carga máxima em um curto período de tempo, até que o sistema falhe;
- Teste de Usabilidade - é um processo onde é avaliado o grau que um produto se encontra em relação a critérios específicos de usabilidade, ou seja, estes testes verificam os requisitos que se relacionam com a interface com o usuário do sistema;
- Teste de Volume - consistem na transmissão de um grande volume de informações quando o sistema está com a carga normal. Geralmente, este tipo de testes é usado para sistemas "*batch*";
- Teste de Segurança - Este teste tenta verificar se todos os mecanismos de proteção embutidos num sistema o protegerão de acessos indevidos. Durante este teste, o analista desempenha papéis de pessoas que desejam invadir o sistema em teste, tentando desarmar o sistema e derrubar as defesas que tenham sido construídas.

2.2.3 Técnicas de teste de *software*

As técnicas de teste são classificadas de acordo com a origem das informações utilizadas para estabelecer os requisitos de teste. Elas também estão relacionadas com a fase de desenvolvimento de *software*. Uma técnica de teste direciona a escolha de critérios para geração de casos de teste que, ao serem executados vão exercitar os elementos requeridos pela abordagem do teste.

As principais técnicas de teste de *software* existentes são (PFLEEFER, 2004):

- Teste Estrutural ou Caixa-Branca - avalia o comportamento interno do componente de *software*. Analisa o código-fonte e elabora casos de teste que cubram todas as possibilidades do componente de *software*. Esta técnica de teste estabelece os requisitos de teste com base em uma dada implementação, requerendo a execução das partes ou de componentes elementares do programa. Essa técnica trabalha diretamente sobre o código-fonte do componente de *software* e avalia aspectos tais como (PRESSMAN, 2005):
  - Teste de Caminhos - os caminhos lógicos do *software* são testados, fornecendo-se casos de teste que põem à prova tanto conjuntos específicos de condições e/ou laços, bem como pares de definições e usos de variáveis;
  - Teste de Comandos - neste teste são modelados os casos de teste para executar sentenças. Pelo menos uma vez em algum teste todos os comandos dos componentes são executados;
  - Teste de Ramos - é executado pelo menos uma vez todo o caminho distinto ao longo do código-fonte;
  - Teste de Condições - este teste verifica as condições lógicas contidas num módulo de programa;
  - Teste de Condições Múltiplas - neste método, os casos de teste são modelados para executar combinações de resultados de condição única (dentro de uma sentença).
- Teste Funcional ou caixa-preta - é uma técnica utilizada para se projetarem casos de teste na qual o *software* é considerado uma caixa-preta. Para testar o *software*, as entradas são fornecidas e as saídas geradas são avaliadas para verificar se estão em conformidade com os resultados pretendidos. Nesta técnica, os detalhes da implementação do *software* não são considerados. Os componentes de *software* a ser testado pode ser um método, uma função interna, um programa, um componente, um conjunto de programas e/ou componentes ou mesmo uma funcionalidade. Um conjunto de critérios pode ser aplicado a esta técnica de teste, tais como (BIASI, 2006):
  - Particionamento de Equivalência - esta técnica tem como objetivo fazer uma divisão (particionamento) do domínio de entrada do programa em classes de dados a partir das quais os casos de teste podem ser derivados. Procura definir um caso de teste que descubra classes de erros. Portanto,

uma classe de equivalência representa um conjunto de estados válidos para condições de entrada;

- Análise de Valores Limites - é uma técnica de projeto de casos de teste que completa o particionamento de equivalência. Esta técnica leva à escolha de casos de teste que testam os valores limites. Assim, em vez de escolher qualquer elemento de uma classe de equivalência, esta técnica leva a escolha de casos de teste nas bordas da classe, ela deriva de casos de teste para o domínio de saída, em vez de focalizar somente as condições de entrada;
- Baseado em Casos de Uso - os casos de uso consistem em uma fonte valiosa para a atividade de teste, a partir de um caso de uso, é possível derivar casos de teste sem que nenhum código tenha sido gerado anteriormente.

2.3 Engenharia Dirigida por Modelos

A Engenharia Dirigida por Modelos (MDE, *Model Driven Engineering*) tem como enfoque utilizar modelos para prover benefícios tais como reduzir os custos e tempo no desenvolvimento do *software* e melhorar a qualidade final dos produtos de *software*. A relevância dos modelos em MDE não consiste apenas em uma documentação do *software* desenvolvido, eles podem ser compreendidos por computadores, ou seja, esses modelos contêm informações que são manipuladas por computadores (KENT, 2002).

Na Engenharia Dirigida por Modelos, os modelos têm função principal, pois não apenas são utilizados para documentação, eles são fundamentais para o desenvolvimento do sistema, pois podem ser manipulados, refinados e evoluem para uma nova versão, e a partir deles são gerados o código-fonte (KLEPPE, 2003). Portanto, os modelos podem ser manipulados e transformados entre si, constituindo artefatos para garantir a longevidade, qualidade e baixo custo das aplicações de *software*.

MDE não pretende substituir os processos tradicionais de desenvolvimento de *software*, e sim contribuir para seu aprimoramento, porém, de uma maneira racional para mover informações de uma fase para outra fase do desenvolvimento, trazendo respostas rápidas e eficientes para atender as necessidades inesperadas de novos requisitos tanto funcionais quanto não-funcionais, apresentando uma melhor e rápida adaptação de novas tecnologias e integração de *software* desenvolvidos em diversas plataformas (LOPES, 2007).

Os modelos são visões do *software* em um determinado nível de abstração. Segundo MELLOR, modelo “é uma abstração de um sistema físico que distingue o que é pertinente do que não é com o intuito de simplificar a realidade. Um modelo contém todos os elementos necessários à representação de um sistema real” (MELLOR et al., 2004).

Modelo é “uma descrição de um (ou de uma parte de) sistema expresso em uma linguagem bem definida, isto é, respeitando um formato preciso (uma sintaxe) e um significado (uma semântica). Esta descrição deve ser conveniente para uma interpretação automatizada por computador” (KLEPPE, 2003).

A criação de modelos é feita utilizando uma linguagem de modelagem bem definida que apresente uma sintaxe e uma semântica para regulamentar a criação dos elementos e suas relações. Uma linguagem de modelagem é uma especificação formal bem definida que contém os elementos de base para construir modelos. Além disto, uma linguagem de modelagem é concebida dentro de um domínio limitado e com objetivos específicos (LOPES, 2007).

A linguagem concebida para criar modelos é freqüentemente descrita por um metamodelo, isto é, o que precede o modelo. Um metamodelo é “um modelo que define a linguagem para exprimir um modelo” (OMG, 2008). “É simplesmente um modelo da linguagem de modelagem. Ela define a semântica e as restrições para uma família de modelos” (MELLOR et al., 2004).

Um metamodelo é definido por um metametamodelo. Um metametamodelo é “um modelo que define a linguagem para expressar metamodelos. A relação entre um metametamodelo e um metamodelo é similar à relação entre um metamodelo e um modelo” (OMG, 2008). Um metamodelo descreve determinadas plataformas, ou seja, a plataforma de um sistema pode ser descrita por um ou mais metamodelos.

Plataforma é “um conjunto de subsistemas e tecnologias que fornecem um conjunto coerente de funcionalidades através de interfaces e padrões de uso especificado” (OMG, 2003). “Especificação de um ambiente de execução para um conjunto de modelos, que deve ter uma implementação da especificação que a plataforma representa” (MELLOR et al., 2004).

A Figura 2.5 ilustra o relacionamento entre sistema, modelo, linguagem de modelagem, metamodelo, metametamodelo e plataforma segundo (MELLOR et al., 2004).

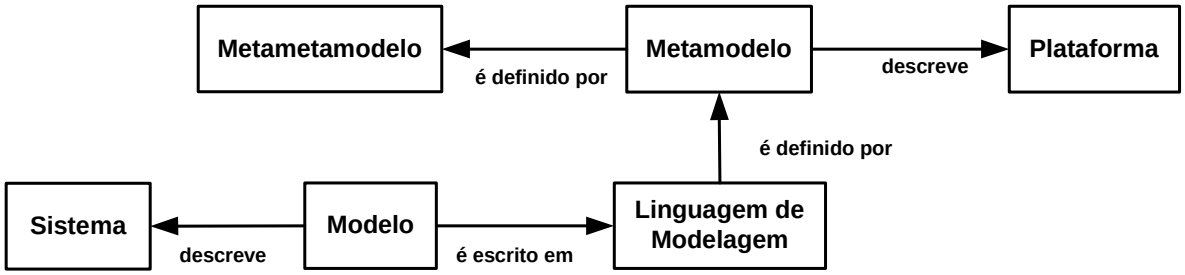


Figura 2.5 - Modelos, Metamodelos e Plataformas (adaptado de MELLOR et al., 2004)

A OMG propôs a relação entre um metametamodelo, um metamodelo, um modelo e uma informação através de uma arquitetura de metamodelagem. A Figura 2.6 ilustra a descrição dos 4 (quatro) níveis de camadas de modelagem (KLEPPE, 2003).

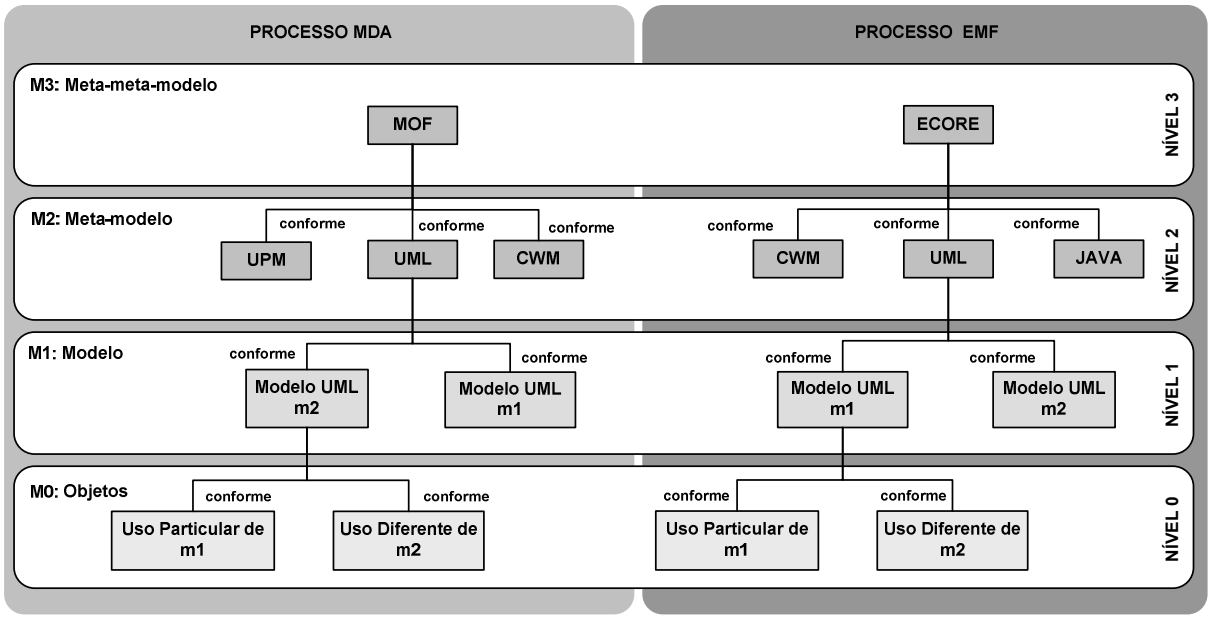


Figura 2.6 - Arquitetura de quatro camadas (adaptado KLEPPE, 2003)

Os 4 (quatro) níveis são descrito a seguir:

- M3 (metametamodelo) - é a camada de mais alto nível, constitui a base da arquitetura de metamodelagem, define uma linguagem abstrata e um *framework* para representar metamodelos. Um metametamodelo define um modelo de mais alto nível de abstração que o metamodelo. Dentre as linguagens de metamodelagem, destacam-se MOF (*Meta Object Facility*) da OMG (OMG, 2008) e Ecore do Projeto Eclipse (BUDINSKY et al., 2003);
- M2 (metamodelo) - é a camada que se encontram os metamodelos. A função principal da camada de metamodelo é definir uma linguagem para especificar

modelos. Os metamodelos são tipicamente mais elaborados que os metametamodelos;

- M1 (modelo) - é a camada onde se encontram todos os modelos do mundo real, representados pelos conceitos definidos no metamodelo correspondente na camada M2. A função principal da camada de modelo é definir uma linguagem para descrever um domínio da informação;
- M0 (informação) - é a camada onde estão as representações de instâncias de conceitos do mundo real. A principal responsabilidade dos objetos de usuários é descrever em um domínio computacional uma plataforma final.

Nesta arquitetura, deve-se notar a existência de poucos metametamodelos (por exemplo, MOF e ECORE), vários metamodelos (por exemplo, UML, UPM, Java e CWM), um grande número de modelos e uma infinidade de informações.

Nos últimos anos, algumas abordagens que visam complementar a iniciativa MDE foram lançadas. *Model Driven Architecture* (MDA) do *Object Management Group* (OMG) (OMG, 2003) e *Model Driven Testing* (MDT) da IBM (IBM, 2003) são abordagens representativas de MDE.

2.3.1 Arquitetura dirigida por modelo

MDA (*Model Driven Architecture*) é uma iniciativa desenvolvida pela OMG (*Object Management Group*) com intuito de promover o uso de modelos no desenvolvimento de *software*, para fornecer uma solução ao gerenciamento da complexidade do desenvolvimento, manutenção e evolução de sistemas de *software* e favorecer a interoperabilidade e portabilidade desses sistemas. O padrão MDA é “um caso particular da abordagem MDE” (FAVRE, 2004).

MDA é uma iniciativa que abrange a geração de código a partir de um modelo, ou seja, é uma abordagem na qual a especificação do sistema é feita de forma independente de plataforma e, para cada uma das plataformas específicas, tal especificação pode ser automaticamente transformada em uma implementação correspondente (OMG, 2003).

MDA foi desenvolvida sobre as normas da OMG, incluindo: UML (*Unified Modeling Language*), XMI (*XML Metadata Interchange*) e CORBA (OMG, 2003). Esta iniciativa apresenta alguns benefícios, tais como (KLEPPE, 2003):

- Produtividade: a transformação do PIM (*Platform Independent Model*) para o PSM (*Platform Specific Model*) precisa ser definida uma única vez e pode ser

aplicada no desenvolvimento de diversos sistemas. Devido a este fato, tem-se uma redução no tempo de desenvolvimento;

- Portabilidade: um mesmo PIM pode ser automaticamente transformado em vários PSMs de diferentes plataformas, através de mapeamentos;
- Interoperabilidade: diferentes PSMs gerados a partir de um mesmo PIM podem ter relacionamentos entre si.

O *framework* básico de MDA é ilustrado na Figura 2.7.

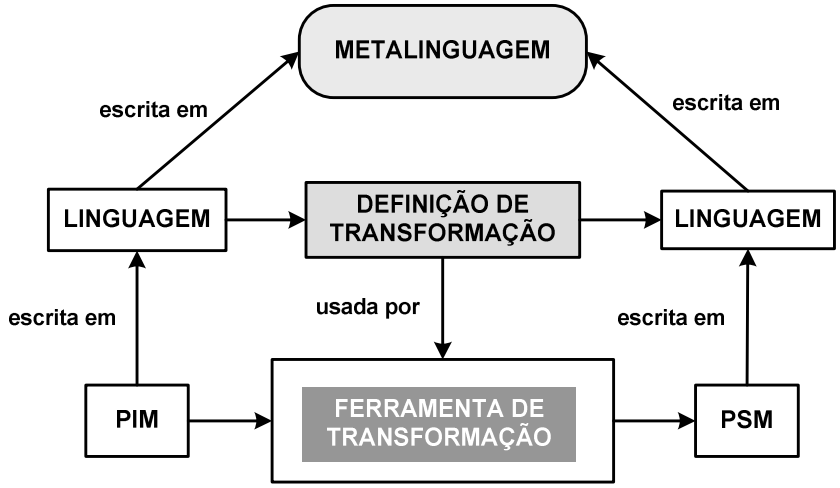


Figura 2.7 - *Framework* básico de MDA (KLEPPE, 2003)

Um Modelo Independente de Plataforma (PIM, *Platform Independent Model*) é um modelo de alto nível de abstração, este é independente de qualquer tecnologia de implementação. Neste modelo, todas as funcionalidades do sistema e restrições do negócio são modeladas.

Um Modelo Específico de Plataforma (PSM, *Platform Specific Model*) é um modelo feito para uma plataforma específica, ou seja, descreve um sistema com pleno conhecimento sobre uma determinada plataforma. Este modelo descreve detalhes da implementação em uma plataforma e as informações do PIM. Ele é gerado a partir de um PIM através da transformação de modelos.

Uma definição de transformação descreve como um modelo fonte, em uma linguagem fonte (metamodelo fonte) poderá se transformar em um modelo alvo em uma linguagem alvo (metamodelo alvo). Para executar uma definição de transformaçao é necessário um motor de transformação (KLEPPE, 2003).

Uma transformação de modelos é “um processo de conversão de um modelo em outro modelo” (KLEPPE, 2003). Neste processo de conversão, os modelos são refinados e

novas informações são adicionados a eles. O processo de transformação é à base da abordagem MDA, pois após sucessivas transformações é que o resultado do processo de desenvolvimento MDA são atingidas. Ao final desses processos é obtido o código-fonte do sistema. As transformações de modelos podem ser agrupadas em (OMG, 2003):

- PIM a PIM: este tipo de transformação é necessário para o acréscimo ou subtração de informações nos modelos ou mudança da linguagem de modelagem;
- PIM a PSM: este tipo de transformação é necessário quando um PIM está suficientemente enriquecido e deseja-se obter este modelo com detalhes de plataforma;
- PSM a PIM: é útil para separar a lógica de negócio da sua plataforma. Particularmente, este tipo de transformação é mais utilizado no processo de engenharia reversa;
- PSM a PSM: é utilizada nas fases de refinamento da plataforma, de implementação, de otimização, de configuração ou mudança de plataforma.

O processo de desenvolvimento de *software* utilizando abordagem MDA consiste basicamente em quatro passos. Figura 2.8 ilustra tais passos e estes são descritos a seguir (ARLOW & NEUSTADT, 2003):

1) Criação do PIM, este modelo deve captar conceitos do domínio sem especificar detalhes de uma plataforma;
2) Transformação de um PIM em um ou mais PSMs, neste passo são adicionados regras transformação para a criação de uma ou mais plataformas especificas;
3) O PSM é transformado em código-fonte;
4) Por fim, ocorre a implantação do sistema em um ambiente específico.

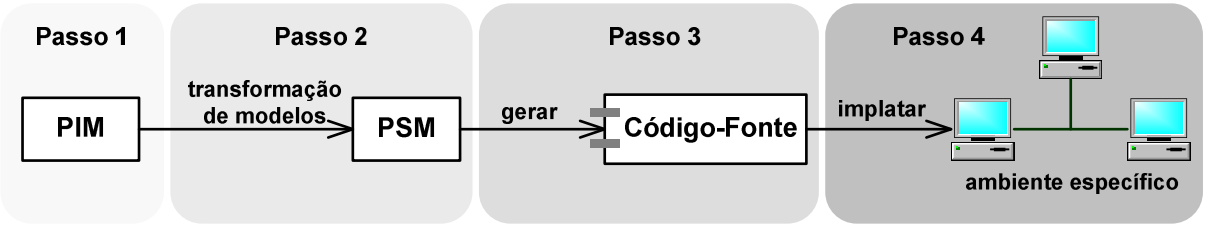


Figura 2.8 - Processo do desenvolvimento MDA (ARLOW & NEUSTADT, 2003)

A relação entre PIM, PSM, metamodelos e plataforma está descrita no metamodelo ilustrado na Figura 2.9 (OMG, 2003). Analisando esta figura, observa-se que MDA fornece a estrutura pelo qual um modelo é transformado em outro modelo.

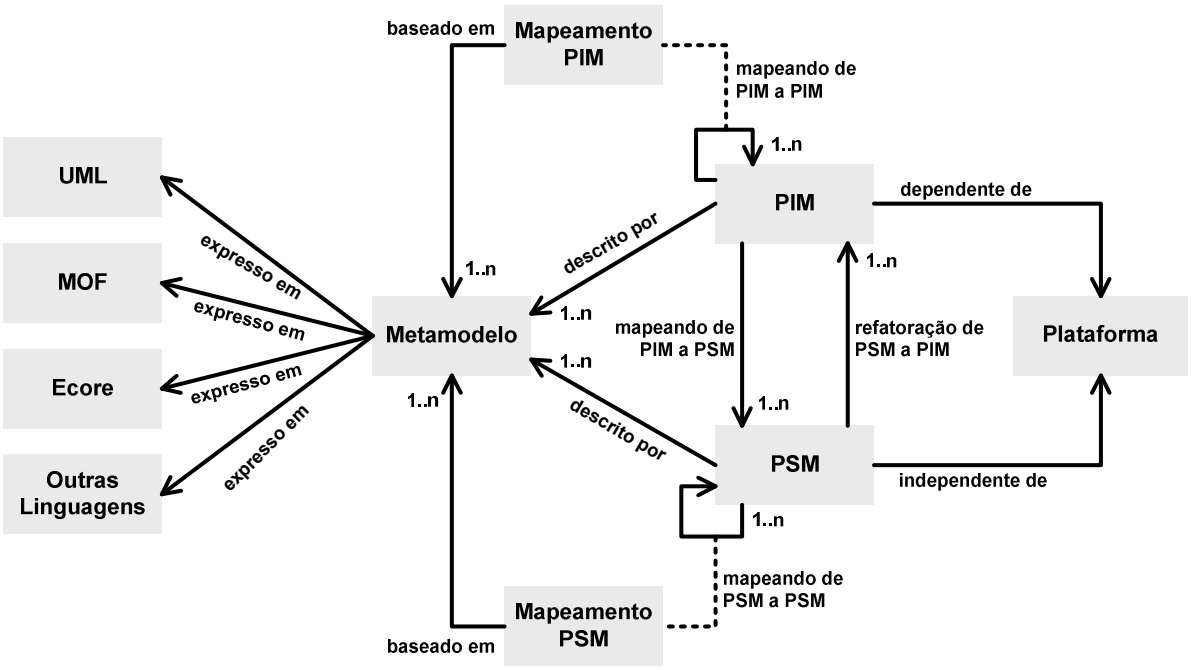


Figura 2.9 - Descrição do metamodelo MDA (OMG, 2003)

MDA é constituída de diversas tecnologias e especificações padronizadas pela OMG, as principais estudadas para o desenvolvimento deste trabalho foram:

- UML 2.0 (*Unified Modeling Language*) - segundo a OMG "é uma linguagem para especificação, construção, visualização e documentação de artefatos de software" (OMG, 2003). Esta linguagem permite a modelagem de diferentes aspectos ou pontos de vista de um sistema (OMG, 2003);
- MOF (*Meta-Object Facility*) - define uma linguagem abstrata e um *framework* para especificação, construção e gerenciamento de metamodelos independentes de plataforma. Esta especificação contém um conjunto de construção que são utilizados para a definição de metamodelos (OMG, 2008);
- MOF QVT - define uma linguagem de transformação de modelos padronizados, é uma especificação hibrida padronizada para transformação de modelos no contexto da metamodelagem MOF. Esta linguagem aceita construções declarativas e imperativas (OMG, 2008).

2.3.2 Teste dirigido por modelos

A noção de Teste Dirigido por Modelos (MDT) tem surgido recentemente como característica necessária para uma implantação bem-sucedida de MDA (IBM, 2003) (DAI, 2004) (JAVED et al., 2007). Assim como o teste é fundamental no desenvolvimento

tradicional de *software*, o teste dentro do contexto de MDE é necessário para que esta abordagem possa atingir seus objetivos (SUSS et al., 2008).

Em MDE, os modelos são o centro do desenvolvimento, tais informações podem ser incorporadas, pois são sempre atuais ao sistema em teste (ALVES et al., 2008). Ferramentas baseadas nesta abordagem mecanizam as atividades de testes de *software* por especificações e execução das regras de transformação, no qual reduz o tempo de desenvolvimento, custo e facilidade na manutenção (JAVED et al., 2007).

2.4 Conclusão

Neste capítulo, apresentou-se uma revisão dos principais conceitos das tecnologias utilizadas para o desenvolvimento deste trabalho. Conceitos sobre testes de *software*, quais possíveis técnicas, fases e tipos de testes utilizados em um processo de teste foram apresentados. Também a abordagem MDE foi descrita incluindo suas principais características, seus casos particulares, ou seja, sua representação tais como MDA e MDT, as tecnologias e benefícios.

3 ESTADO DA ARTE

Este capítulo visa descrever as principais abordagens de automação de testes de *software* utilizados no meio acadêmico e industrial. Neste capítulo, as vantagens de algumas abordagens propostas na literatura são relatadas. A diferença entre especificação de correspondência e definição de transformação é descrita. Uma conceitualização de linguagens de transformação pesquisadas é apresentada. Ainda neste capítulo, os trabalhos relacionados, que serviram de base para o desenvolvimento desta dissertação, são descritos.

3.1 Abordagens de Testes

Para construção automatizada de casos de teste é necessário a escolha de uma abordagem de teste. Abordagens de teste dizem respeito à profundidade da análise a ser realizada em um determinado *software*.

As principais abordagens estudadas para o desenvolvimento deste trabalho foram o TBM (Teste Baseado em Modelos) (UTTING et al., 2006) (REZA et al., 2008), Desenvolvimento Dirigido por Teste (GRENNING, 2007), U2TP (*UML 2.0 Testing Profile*) (OMG, 2005) e Teste Dirigido por Modelo (HECKEL & LOHMANN, 2003) (DAI, 2004) (SUSS et al., 2008).

Portanto, todas as abordagens têm como objetivo a descoberta de falhas em um determinado sistema de *software*.

3.1.1 Testes baseados em modelos

TBM (Teste Baseado em Modelos) consiste na geração automática de casos de teste, utilizando os modelos desenvolvidos no início do projeto, ou seja, usa modelos estruturais e comportamentais descrito, por exemplo, em diagramas UML (WIECZOREK et al., 2008)

TBM é uma variante de teste em que modelos explícitos são usados para capturar o comportamento e o ambiente de um sistema de *software* (LEAL, 2008). Esta abordagem de teste é a automação da técnica de teste funcional ou caixa-preta (REZA et al., 2008).

TBM “é o processo de derivação automática de casos de teste concretos a partir de modelo formais abstratos e execução dos casos de teste” (UTTING et al., 2006).

A Figura 3.1 descreve o processo de teste baseado em modelo (UTTING et al., 2006).

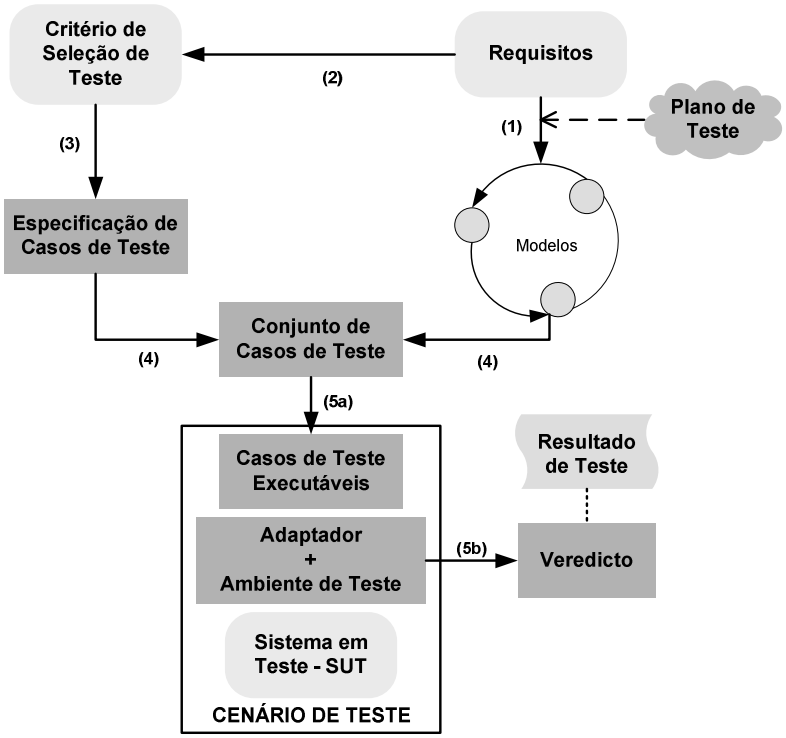


Figura 3.1 - Processo de Teste Baseado em Modelo (Adaptação: UTTING et al., 2006)

O processo TBM, ilustrado na Figura 3.1 é constituído por cinco fases descritas a seguir (UTTING et al., 2006):

1. Construção de um modelo abstrato do sistema em teste com base na especificação de requisitos - este modelo retrata o comportamento desejado para o sistema, adota características funcionais e/ou não-funcionais, que são avaliadas ou descartadas no teste;
2. Definição do critério de seleção de teste - este critério define que comportamento descrito pelo modelo será avaliado;
3. Transformação do critério de seleção de teste em especificação de caso de teste - este critério formaliza o critério e o torna operacional;
4. Geração do conjunto de teste - neste passo um grupo de teste é gerado a fim de satisfazer uma determinada especificação de caso de teste;
5. Execução dos casos de teste - corresponde à execução e avaliação dos cenários de teste no processo de teste, este passo é dividido em duas etapas:
    a. Conciliação dos diferentes níveis de abstração entre o modelo e o sistema em teste - é necessário aplicar uma entrada de dados concreta ao sistema em

teste e registrar a saída do mesmo. Um adaptador é a ponte para concretizar a entrada de dados e o resultado esperado, comparar resultados concretos e gerar a análise;

b. Geração do veredicto - nesta etapa, tem-se o resultado da comparação da saída do sistema com a saída definida no caso de teste. Tem-se como veredicto os seguintes valores “passou”, “falhou”, “inconclusivo” e “erro”.

TBM ajuda o teste, pois o torna mais rápido e menos suscetível a erros humanos, automatizando atividades rotineiras e propensas a erros (SANTOS-Neto et al., 2008). Entretanto, os modelos se tornam desatualizado, caso haja mudanças nos requisitos ou mesmo surgirem novos requisitos, pois nem sempre é costume dos desenvolvedores atualizarem os modelos modelados no início do projeto de *software*.

3.1.2 Desenvolvimento dirigido por teste

Nesta abordagem de teste, o desenvolvimento do código-fonte é direcionado pelo desenvolvimento de casos de teste para enfim desenvolver o código-fonte. TDD (*Test Driven Development*) é uma das principais práticas de *eXtreme Programming* (XP) uma das abordagens mais fáceis de entender e ao mesmo tempo uma das mais difíceis de executar (SLYNGSTAD et al., 2008).

Esta técnica é inversa à da programação convencional em que se desenvolve primeiramente o código-fonte para em fim ser construído os casos de teste, sejam estes executados manualmente ou automaticamente.

TDD é uma programação a prova de riscos, que investe tempo no início para evitar falhas no final (GRENNING, 2007). Esta abordagem foi desenvolvida de forma diferente das abordagens existentes para garantir que o código-fonte está sendo testado o tempo todo (BECK, 2003).

O código-fonte desenvolvido utilizando abordagens TDD é escrito de forma que os módulos sejam testáveis isoladamente. Esta abordagem aprimora o projeto de *software*, pois escrevendo um caso de teste completo, força o desenvolvedor a criar um código-fonte desacoplado, ou seja, não vinculado estritamente a outro código-fonte (RENDELL, 2008).

Os casos de teste também agem como documentação, proporcionando uma especificação de trabalho, fornecendo exemplos concretos de como utilizar o módulo a ser testado e informando de forma clara a finalidade do código-fonte (AMBLER, 2003).

Os TDD também agem como uma rede de segurança, pois fornece *feedbacks* contínuos quando ocorre mudanças no código, assim o programador é informado quando um defeito é introduzido no sistema (AMBLER, 2003). Quando isso acontece, os TDD têm uma recuperação rápida da falha, pois o teste de unidade isola os problemas rapidamente, conseqüentemente as atividades de depuração são reduzidas, permitindo assim que o sistema se recupere de falhas com agilidade (AMBLER, 2003).

Seu código-fonte é escrito de forma nítida, ou seja, é expresso com clareza, podendo ser alterado para receber novos recursos e sem duplicação. A refatoração mantém a semântica comportamental do código-fonte, nunca adiciona ou remove funcionalidades (SLYNGSTAD et al., 2008).

3.1.3 Perfil UML 2.0 de testes

A fim de realizar a descrição de modelos de teste, a OMG, grupo responsável pela manutenção da UML, desenvolveu o Perfil UML 2.0 de Testes (U2TP, *UML 2.0 Testing Profile*). Esta abordagem de teste é uma extensão da UML para realizar a descrição dos componentes e atividades de testes (OMG, 2005).

A Arquitetura Dirigida por Modelos (MDA) busca padronizar o uso de linguagens de descrição tal como UML. A UML 2.0 juntamente com a abordagem MDA tem como objetivo tornar possivel a “execução” de UML, permitindo a geração de teste, não se delimitando em apenas gerar o código-fonte mas também em simular e validar os modelos (GERCHMAN, 2008).

U2TP define uma linguagem para projetar, visualizar, especificar, construir e documentar os artefatos, que abrange tanto aspectos estáticos como dinâmicos de testes de sistemas. Esta abordagem pode ser aplicada para testes de sistemas em vários domínios de aplicações (OMG, 2005).

Esta linguagem é padronizada, suportando todos os demais componentes e elementos presentes em uma atividade de teste. U2TP herda de UML a organização em meta-camadas de abstração, uso de pacotes e a extensibilidade para adaptação a domínios e plataformas específicas (BIASI, 2006).

U2TP foi projetada com os seguintes princípios (OMG, 2005):

- Integração com UML - U2TP é definido na base do metamodelo UML, usando os mesmo princípios e elementos da linguagem dos perfis UML;

- Reúso - U2TP usa, quando possível, os conceitos e elementos presentes na linguagem UML, podendo assim adicionar e estender novos conceitos e elementos quando necessário.

A arquitetura do Perfil UML 2.0 de Testes é dividida em quatro grupos de conceitos. São eles (OMG, 2005):

- Arquitetura de Teste - define conceitos da estrutura e configuração de um teste e os relacionamentos entre o teste e o Sistema em Teste (SUT, *System Under Test*);
- Comportamento de Teste - define conceitos relacionados aos aspectos dinâmicos dos procedimentos de teste. É dirigido às observações e atividades executadas durante uma campanha de teste;
- Dados de Teste - define conceitos para dados de testes usados em procedimentos de testes, define sintaxe e semântica dos valores de entrada e saída;
- Temporização de Teste - define conceitos quantificados por tempo para procedimentos de teste. Neste grupo, têm-se elementos para especificar restrições de tempo durante a execução.

3.1.4 Teste dirigido por modelo

MDT (*Model Driven Testing*) é uma abordagem promissora para a automação de teste de *software*. Esta abordagem é uma melhoria da abordagem de TBM, pois esta não considera a separação entre os modelos independente de plataforma e modelos específicos de plataforma (HECKEL & LOHMANN, 2003).

MDT usa a abordagem de transformação de modelos, metamodelos e um grupo de regras (que são definidos por meio de especificação de correspondências entre os elementos dos metamodelos). A principal vantagem da utilização de MDT em relação a MBT ocorre pois, na primeira abordagem, os modelos são sempre atualizado, assim no final do projeto, os modelos estão conforme o código-fonte gerado.

Esta iniciativa permite aos desenvolvedores e testadores tornar o *software* mais produtivo, reduzindo o tempo de desenvolvimento e comercialização do *software* deixando-o com elevados padrões de qualidade (IBM, 2003).

Em MDT, o modelo de teste independente de plataforma é reutilizado para garantir a interoperabilidade dos mesmos por várias plataformas e caberia ao modelo

específico especificar qual linguagem especifica é necessária para execução dos casos de teste.

MDT é uma importante característica para uma implantação bem-sucedida de MDA, pois tem o intuito de se beneficiar da separação de modelos na geração e execução de testes. A Figura 3.2 apresenta à estratégia de teste (DAI, 2004).

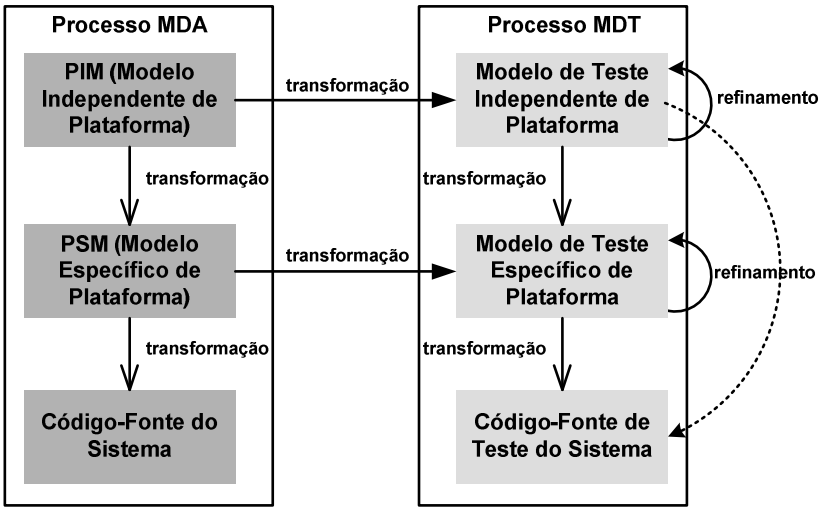


Figura 3.2 - Estratégia de Teste de MDT (DAI, 2004)

A mesma concepção de modelos utilizados em MDA é empregada em MDT. Além disso, os PIM podem ser aplicados diretamente, permitindo a integração rápida do teste para o processo de desenvolvimento de *software*. Assim, o modelo de teste pode ser transformado diretamente para o código-fonte de teste ou para uma plataforma específica de teste (DAI, 2004).

MDT separa a análise lógica da aplicação de teste. Portanto, permite aos programadores concentrar-se na criação de bons testes específicos de plataforma (LI et al,. 2006). Os pacotes de Testes Dirigidos por Modelos são um conjunto de ferramentas que ajudam os desenvolvedores a executarem as seguintes tarefas (IBM, 2003):

- Criar e editar modelos de componentes de *software*;
- Simular modelos;
- Gerar testes suítes para componentes de *software*;
- Executar teste suítes juntos com os componentes de *software*;
- Criar *templates* para testar diretrizes de execução e testar *proxies*;
- Traduzir testes suítes para criar scripts para testar *driver's* legados;
- Analisar tanto os testes suítes e a execução guardando a traçabilidade.

MDT favorece a redução do tempo na construção dos testes, aumenta a qualidade e gerencia a complexidade dos testes, oferece uma abordagem sistemática para geração do teste *suite* e reduz despesas na manutenção dos testes (IBM, 2003).

Apesar de muitos benefícios, poucas pesquisas foram realizadas a fim de uma melhor definição da abordagem automatizada que vise à validação das Transformações de Modelos (SUSS et al., 2008) (JAVED et al., 2007).

MDT se baseia em definições de transformações tal como definido no conteúdo de MDA. Contudo, criar definições de transformações de modelos é, atualmente, uma tarefa manual e propensa a erros, acarretando a injeção de erros nos modelos alvos e, consequentemente, no código-fonte final. Por isso, o código-fonte que faz o teste de um sistema de software pode conter erros ou diferir em algum aspecto do que tenha sido especificado nos modelos.

3.2 Especificação de Correspondência e Definição de Transformação

Os termos especificação de correspondência e definição de transformação são assuntos centrais para MDx, eles formam a base para o processo de obtenção do código-fonte de um determinado sistema de *software*.

Em (LOPES, 2007), uma separação explicita entre especificação de correspondência e a definição de transformação é proposta conforme a Figura 3.3.

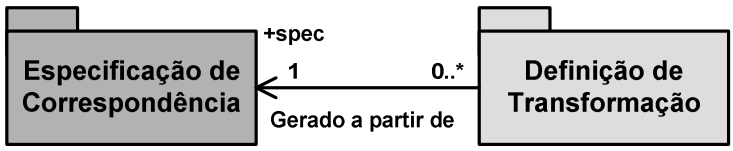


Figura 3.3 - Relação entre especificação de correspondência e definição de transformação (LOPES, 2007)

Especificação de correspondência é a primeira tendência, que consiste em estudar os problemas ligados a criação de correspondências entre metamodelos (ou modelos) (LOPES, 2007). Esta também é chamada de mapeamento (*mapping*) de equivalências entre os elementos pertencentes aos metamodelos fonte e alvo, ou seja, apresenta a lógica utilizada para estabelecer as relações entre dois metamodelos.

A definição de transformação consiste na concepção e realização de programas de transformação em uma linguagem de transformação segundo as recomendações da OMG, ou seja, contém regras precisas para transformar um modelo em outro modelo (LOPES, 2006a).

Um modelo de transformação é gerado a partir de um modelo de correspondência e um programa de transformação em execução é baseado neste modelo de transformação (SOUZA Jr et al., 2008). Na realidade, a especificação de correspondência pode ser considerada como um PIM e a definição de transformação como um PSM.

Portanto, entende-se que a especificação de correspondência apresenta a lógica utilizada para estabelecer as relações entre dois metamodelos, enquanto a definição de transformação contém regras precisas para transformar um modelo em um outro modelo. Podendo ser considerada a especificação de correspondência como um PIM e a definição de transformação como um PSM.

A Figura 3.4 apresenta a arquitetura para a tranformação de modelos e as relações de cada entidade envolvida.

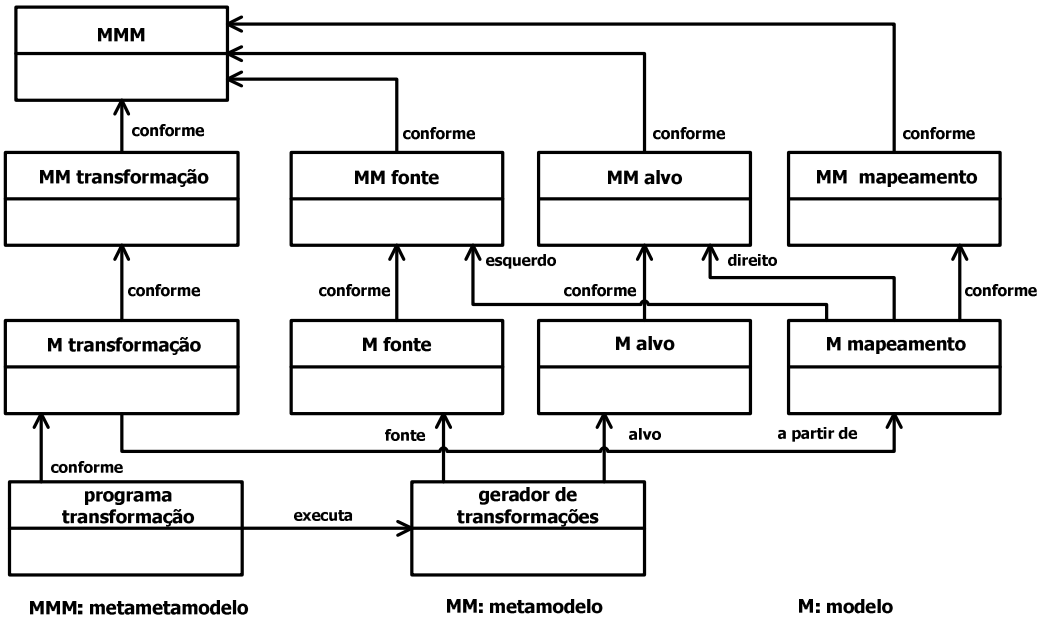


Figura 3.4 - Uma proposta de arquitetura para transformação de modelos (LOPES, 2006a)

Nesta arquitetura ilustrada na Figura 3.4 apresenta a sepação entre o modelo de correspondência (i.e. a especificação de correspondência - M mapeamento) é separado do modelo de transformação (i.e. a definição de transformação - M transformação). Esta proposta de arquitetura tipo contém as seguintes entidades (LOPES, 2006a):

- MMM (um metamodelo): por exemplo, MOF ou Ecore;
- MM fonte e MM Alvo (um metamodelo fonte e um alvo): por exemplo, o metamodelo UML ou EDOC;
- M fonte e M Alvo (um modelo fonte e um alvo): por exemplo, um modelo de uma agência de viagem em UML que deve ser transformado em um modelo Java;
- MM mapeamento (um metamodelo de correspondência): a linguagem para modelagem de correspondências entre os elementos de um metamodelo fonte e os elementos de um metamodelo alvo;
- M mapeamento (um modelo de correspondência): o modelo contendo as correspondências entre dois metamodelos;
- MM transformação (um metamodelo de transformação): o formalismo que permite a criação precisa de transformações de um modelo fonte em um modelo alvo;
- M transformação (um modelo de transformação): descreve a transformação de modelos;
- Programa transformação (um programa de transformação): um programa executável para realizar a transformação;

Portanto, a especificação de correspondência é uma atividade que antecede às definições de transformações, pois ela é a base de conhecimento sobre qual definição de transformação é gerada.

Já o termo transformação refere-se à atividade de transformar um elemento fonte em um elemento alvo em conformidade com as definições de transformações. Essa definição de transformação é executada em uma linguagem de computador que permite a construção das regras de transformação.

Várias linguagens de transformação de modelos foram propostas ao longo dos anos, entre elas se podem citar, QVT (*Query/View/Transformation*) da OMG (OMG, 2008), ATL (*Atlas Transfomation Language*) da Universidade de Nantes (ATL, 2006) e MOFScript do Eclipse Project (OLDEVIK, 2006).

Uma breve descrição destas linguagens de transformação é apresentada a seguir:

- Linguagem de transformação QVT - estimula a criação de uma linguagem de transformação de modelos padronizada. QVT não é uma implementação, mas sim uma especificação. A linguagem QVT tem uma especificação híbrida padronizada para transformação de modelos no contexto de metamodelagem

MOF 2.0. QVT aceita construções declarativas e imperativas, é formada por três sub-linguagens: núcleo (*core*), relações (*relations*) e mapeamentos operacionais (*operational mappings*). Sendo as duas primeiras da arquitetura declarativa e a última imperativa (OMG, 2008);

- Linguagem de transformação ATL - é uma linguagem híbrida, como a QVT, aceita tanto construções declarativas e como imperativas. As construções declarativas de ATL são baseadas na idéia de regras de definição (*matched rule*), consiste em estabelecer ligações entre padrões fonte e padrões alvo. A construção imperativa em ATL é criada de duas formas: regras invocadas (*called rules*) e bloco de ações (*actions blocks*). Uma regra invocada é chamada como função, através de um nome e um conjunto de argumentos. Um bloco de ações é uma seqüência de instruções imperativas que podem ser utilizadas pelas regras de combinação ou regras de invocação (ATL, 2006);
- Linguagem de transformação MOFScript - ferramenta de transformação de modelos para texto, por exemplo, para apoiar a geração de código ou documentação da implementação de modelos. MOFScript é baseada em EMF e Ecore (OLDEVIK, 2006).

Para o desenvolvimento desta dissertação, a linguagem de transformação ATL foi utilizada, pois é uma linguagem que realiza transformações modelo-a-modelo e modelo-a-texto. Além disso, há um ambiente de desenvolvimento para ATL como *plug-in* para Eclipse que é de fácil utilização.

3.3 Trabalhos Relacionados

Uma descrição dos trabalhos realizados sobre temas co-relacionados ao tema desta dissertação foi levantada a fim de identificar e selecionar métodos e técnicas a serem utilizados para a concretização dos objetivos propostos nesta dissertação.

Os trabalhos discutidos nesta seção foram de suma importância para o desenvolvimento de um *framework*, metodologia e metamodelos que permite a geração automática de casos de teste a fim de inspecionar o código-fonte de um sistema gerado por transformações de modelos.

A seção 3.3.1 apresenta os trabalhos relacionados para o desenvolvimento do *framework* para geração automatizada de casos de teste. A seção 3.3.2 apresenta os trabalhos

relacionados para o desenvolvimento dos metamodelos independentes e específicos de plataformas.

3.3.1 Trabalhos relacionados a geração automatizada de casos de teste

No trabalho (JAVED et al., 2007) uma Abordagem Dirigida a Modelos para testar aplicações de *software* é apresentado. Uma visão geral desta abordagem é ilustrada na Figura 3.5, onde a geração de casos de teste é descrita em 2 (dois) passos.

No primeiro passo, o diagrama de seqüência é criado e este modelo é transformado automaticamente, utilizando transformação modelo-a-modelo, em uma unidade de modelos de casos de teste, ou seja, em um modelo xUnit que é uma *framework* de teste (JAVED et al., 2007).

No segundo passo, o modelo de teste é convertido em um caso de teste concreto e executável. Uma transformação vertical de modelo-a-texto é aplicada no modelo xUnit para gerar plataformas específicas, tais como JUnit e SUnit (JAVED et al., 2007).

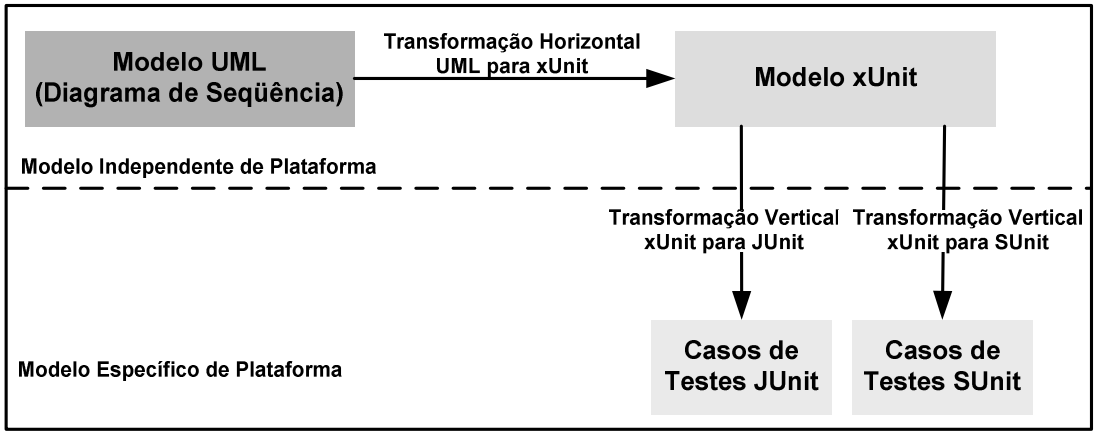


Figura 3.5 - Processo para geração de casos de teste proposto em (JAVED et al., 2007)

No trabalho (LI et al., 2006), uma metodologia de MDT para aplicações *Web* foi proposta. Os modelos de casos de teste são produzidos com base no modelo de aplicação *web*. Neste trabalho, dois tipos de modelos para teste são desenvolvidos: modelo de implantação para teste e modelo de controle para teste. Estes modelos são projetados para descrever o ambiente e o processo de execução de teste (LI et al., 2006).

Um *framework* chamado MDWATP (*Model-Driven Web Application Testing*) é projetado para dar suporte a esta metodologia. A Figura 3.6 apresenta esta arquitetura. Para

tornar esse *framework* mais extensível e flexível, este é dividido em testador e modelador (LI et al., 2006).

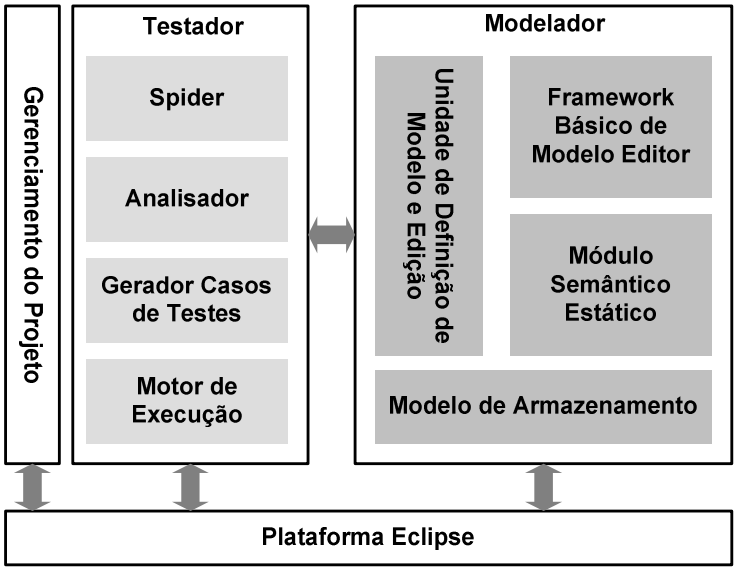


Figura 3.6 - *Framework* de MDWATP proposto em (LI et al., 2006)

O testador recupera semi-automaticamente o modelo de aplicação *web*, gerando casos de teste e executando-os automaticamente. Este é divido em *Spider*, Analisador (*Parser*), Gerador de Casos de teste (*Test Case Generator*) e Motor de Execução (*Execution Engine*).

O *Spider* e Analisador abstraem a informação da navegação e informações das páginas HTML. O Gerador de Casos de Teste gera casos de teste baseado no modelo de aplicação *web.* O Motor de Execução executa os casos de teste baseados nos modelos de teste de implementação e modelos de teste de controle (LI et al., 2006).

Após essas etapas, os resultados obtidos nos testes são coletados e anexados em um relatório de teste para testar os modelos. O modelador é responsável em criar, visualizar e salvar modelos. O modelador implementa dois mecanismos de extensão UML2: MOF e Perfil. Assim não só implementa metamodelos definido em (LI et al., 2006), mas qualquer modelo em UML 2.0 (LI et al., 2006).

Neste módulo, tem-se a Unidade de Definição de Modelo e Edição que consiste de uma biblioteca meta-gráfico, um arquivo de definição modelo editor e um arquivo descrição de interface. Um *framework* básico de modelo editor é o núcleo do Modelador, ele implementa funções comuns tais como a interpretação e processamento de entrada de usuário, configuração de interface e conexão com o módulo semântico estático.

O módulo semântico estático, por sua vez, contém a definição dos metamodelos. O modelo de armazenamento é responsável para salvar todos os modelos produzidos na base de dados (LI et al., 2006).

No trabalho de (ALVES et al., 2008), os autores discutem os objetivos e questões fundamentais sobre como integrar processos de MDD e MDT e apresentam uma proposta concreta de integração com base no uso de perfis de teste em UML 2.0.

Esta proposta utiliza transformações de modelos entre modelos implementadas na linguagem de transformação ATL. Ainda neste trabalho, um estudo de caso ilustra a aplicação desta proposta. A Figura 3.7 ilustra o *framework* proposto no trabalho (ALVES et al., 2008).

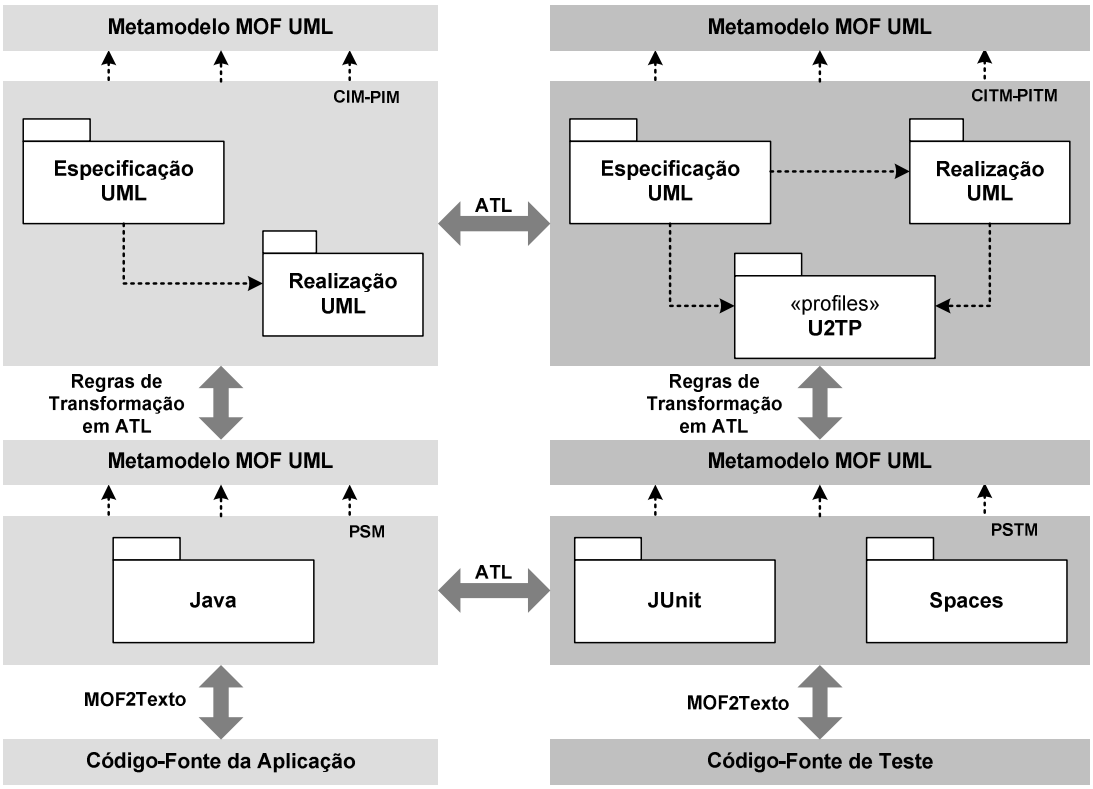


Figura 3.7 - *Framework* proposto em (ALVES et al., 2008)

A Figura 3.7 descreve o *framework* proposto em (ALVES et al., 2008), onde o processo para a geração do código-fonte corresponde ao mesmo processo de geração do código-fonte de teste, onde são permitidas a geração do modelo independente de plataforma para teste utilizando a linguagem U2TP diretamente do modelo de négocios utilizado para a geração do código-fonte. Igualmente poderá ser feito o processo para a geração do modelo específico de plataforma para teste, para enfim poder gerar o código-fonte de teste.

Este *framework* proposto em (ALVES et al., 2008) utiliza a linguagem U2TP que apresenta um metamodelo com quatro grupos de conceitos em uma visão de metamodelagem MOF.

3.3.2 Trabalhos relacionados a modelagem de metamodelos de testes

No trabalho (JAVED et al., 2007), além de uma Abordagem Dirigida por Modelos para testar aplicações de *software* usando diagramas de seqüências, também é apresentado metamodelo de testes específico de plataforma. A finalidade deste metamodelo é servir como base para a geração dos casos de teste (JAVED et al., 2007), assim como proposta nesta dissertação.

Neste trabalho, encontra-se um metamodelo para plataforma xUnit, apresentado na Figura 3.8, ficando a critério do programador decidir por meio das regras de transformação qual plataforma *unit* este deseja utilizar para realização dos testes (JAVED et al., 2007).

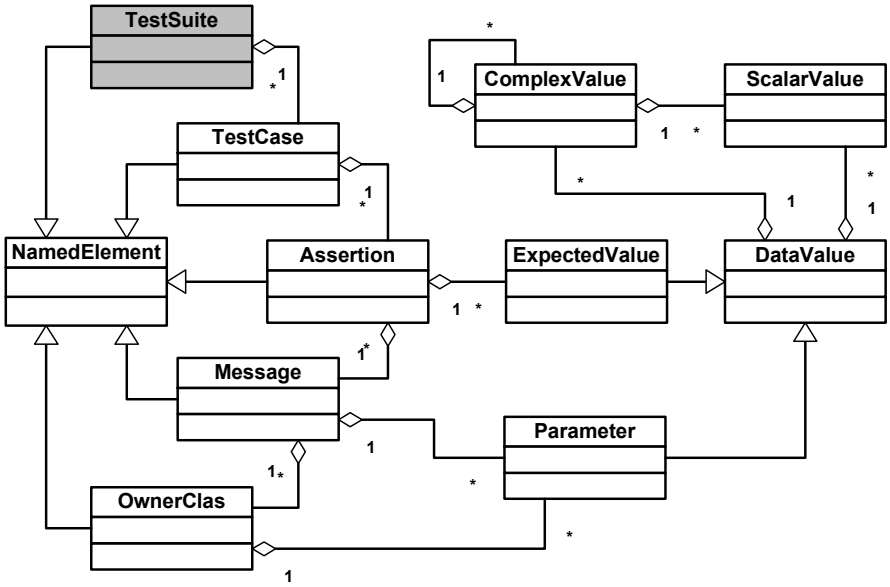


Figura 3.8 - Fragmento do Metamodelo xUnit proposto em (JAVED et al., 2007)

O trabalho (DUEÑAS et al., 2004) apresenta um metamodelo para teste, embora este siga os princípios gerais declarado em U2TP. Ele possibilita a modificação e ampliação dos perfis em vários aspectos (DUEÑAS et al., 2004). A Figura 3.9 apresenta os pacotes do metamodelo de teste e do metamodelo de especificação de teste.

Este metamodelo de teste é específico para testes de *software* em um contexto de Engenharia de Família de Produtos, para a identificação de mecanismo de derivação quem cubram a transformação de testes de produtos de família em teste específicos de produto (DUEÑAS et al., 2004).

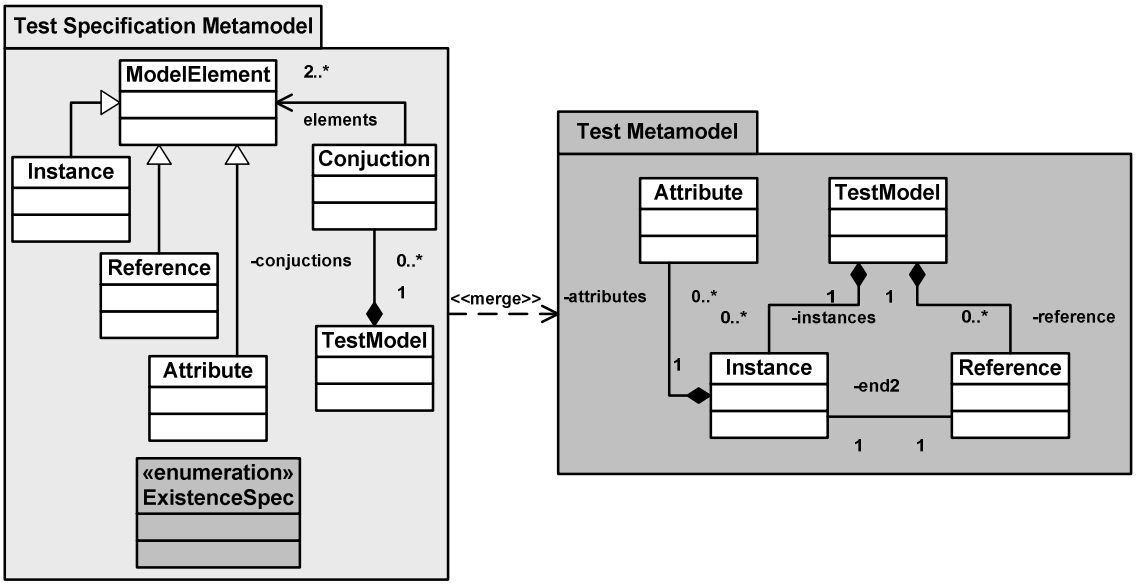


Figura 3.9 - Fragmento dos Metamodelos de Teste proposto em (DUEÑAS et al., 2004)

Em (SADILEK & WEIßLEDER, 2008), apresenta-se metamodelos em uma visão de metamodelagem MOF com a finalidade de testar metamodelos e modelos desenvolvidos em um processo segundo uma abordagem MDx (SADILEK & WEIßLEDER, 2008), apresentado na Figura 3.10.

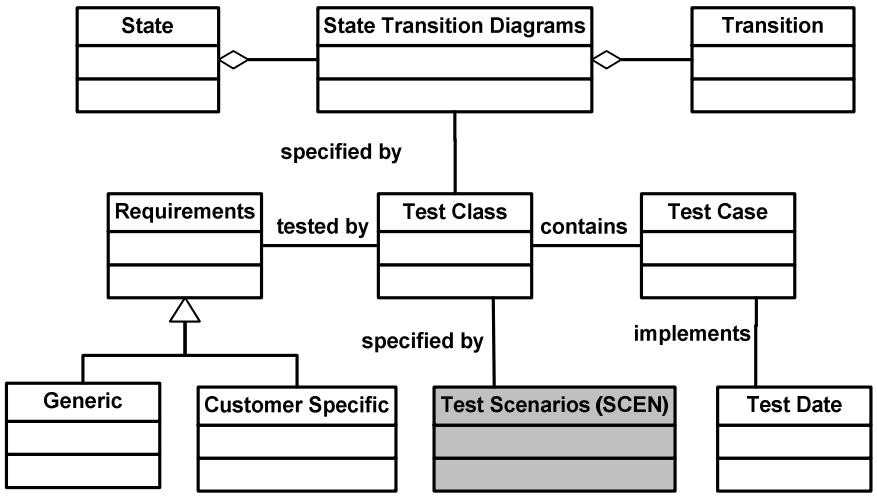


Figura 3.10 - Fragmento do Metamodelo para Teste proposto em (SADILEK & WEIßLEDER, 2008)

Este metamodelo contém classes e atributos que armazena erros, associações entre classes multiplicadas ou ausentes com a finalidade em testar um conjunto de especificações (SADILEK & WEIßLEDER, 2008).

3.3.3 Análise dos trabalhos relacionados

No trabalho de (JAVED et al., 2007), analisou-se a ausência de testes específicos para teste unitário e testes de integração. Este trabalho apresenta uma proposta para geração do código-fonte de teste, porém torna essa geração restrita para modelos de negócios na notação UML em diagramas de seqüências. Também se limita em utilizar a técnica de teste funcional não aclopando em seu desenvolvimento outras técnicas e fases de testes.

Outro ponto analisado nos trabalhos relacionados foi à ausência de um metamodelo de teste independente de plataforma que busque acoplar mais técnicas de testes além do especificado no metamodelo U2TP. Este metamodelo é utilizado em muitos trabalhos para geração de casos de teste, porém ele é um metamodelo que utiliza somente metamodelagem MOF e seu mapeamento para plataformas é limitado, isto é, somente para plataforma COTE, TTCN-3 e JUnit.

O trabalho de (LI et al., 2006) limitou-se em testar aplicações web, impossibilitando que o *framework* proposto tenha portabilidade e interoperabilidade nos testes. O trabalho de (ALVES et al., 2008) utiliza o metamodelo U2TP para a geração do código-fonte de teste.

Os metamodelos pesquisados e apresentados na subseção 3.3.2 são específicos para uma função particular. O metamodelo xUnit proposto em (JAVED et al., 2007) se adaptou a fim de testar modelos de negócios na notação UML em diagramas de seqüências.

O metamodelo de teste apresentado no trabalho (DUEÑAS et al., 2004) foi adaptado para gerar casos de teste em um contexto de Família de Produtos. Em (SADILEK & WEIßLEDER, 2008) apresenta um metamodelo que contém classe e atributos com informações sobre erros a fim de testar os modelos e metamodelos.

Portanto, neste trabalho de dissertação propõe-se um *framework* para geração de casos de teste que tenha uma diversidade de possibilidades de tipos, técnicas e fases de teste, que busque benefícios tais como, a redução do tempo para o desenvolvimento de testes, a interoperabilidade, portabilidade e produtividade na geração dos testes, a minimização da injeção de erros durante a geração dos casos de teste, qualidade e eficiência nos casos de teste

e automação da geração dos casos de teste diminuindo a dependência das intervenções humanas.

A Tabela 3.1 apresenta uma comparação do *framework* proposto nesta dissertação com os trabalhos estudados a cima.

Tabela 3.1 - Comparativos entre Trabalhos Relacionados

	JAVED et al, 2007	LI et al, 2006	ALVES et al, 2007
Técnica de Teste	Funcional	Funcional	Funcional
Fase de Teste	Unidade	Unidade	Unidade
Metamodelos Independente de Plataforma	Seqüência	U2TP	U2TP
Metamodelos de Teste Específicos de Plataforma	xUnit	Web	JUnit e Spaces
Metamodelagem	MOF	MOF	MOF
Modelagem PIM	Diagramas de Seqüência UML	Diagramas de Classe UML	Diagramas de Classe UML
Código-Fonte de Teste	Qualquer tipo de aplicação	Aplicações WEB	Java e Spaces

3.4 Conclusão

Neste Capítulo, apresentou-se o estudo do estado da arte sobre principais abordagens de testes automatizados. Portanto, abordagem de teste que se adaptou para o desenvolvimento desta dissertação é a abordagem MDT, pois é uma abordagem de teste que visa tornar a implementação das abordagens MDx bem sucedidas.

MDT reutiliza os modelos de negócios desenvolvidos dentro de uma visão MDx e utiliza-os para a geração dos casos de teste, além de permitir testes independentes de plataformas, tornando possível um aumento da produtividade, portabilidade e interoperabilidade dos testes.

Ainda neste Capítulo, a diferença entre especificação de correspondências e definição de transformação foi descrita. Estes são processos que resultam na obtenção do código-fonte de um determinado sistema de *software* utilizando uma abordagem MDx.

Por fim, os trabalhos relacionados que serviram de base para o desenvolvimento deste trabalho foram descritos e seus pontos fracos foram analisados.

4 PROPOSTA DE UM *FRAMEWORK* PARA TESTE

Neste capítulo, apresenta-se um *framework* chamado *Automatic Test Case based on Models* (ATCM) e metamodelos de teste com a função de gerar automaticamente o código-fonte de testes e seus respectivos casos de teste a fim de testar o código-fonte gerado por uma abordagem MDx.

Procurou-se desenvolver uma metodologia para descrever de forma minuciosa e detalhada todas as etapas para a geração automatizada do código-fonte de teste. Ainda neste capítulo, apresentam-se os metamodelos desenvolvidos para auxiliar no projeto do *framework* ATCM e a abordagem para o *matching* desses metamodelos de testes, desenvolvido por (LOPES, 2007).

A prototipação do *framework* ATCM é composta por 4 (quatro) ferramentas que auxiliam na geração automatizada de casos de teste conforme mencionado neste capítulo.

4.1 Abordagem

A necessidade de um processo de teste é extremamente importante para validar qualquer processo de desenvolvimento de *software*. Nas abordagens MDx, há uma carência de técnicas e processos de testes para verificar se o *software* possue falhas.

A fim de prover um processo de teste em uma abordagem MDx, um *framework* é proposto para dar suporte a geração automatizada do código-fonte de teste. Este *framework* chamado ATCM, apresenta dois processos distintos: processo de geração de *software* de teste e processo de geração de *software*.

O primeiro processo envolve a geração automatizada de casos de teste para testar um sistema de *software*. O segundo processo consiste da parte de um processo MDA, isto é, um Modelo Independente de Plataforma, PIM, através de uma transformação de modelo-a-modelo gera um Modelo Específico de Plataforma, PSM, e uma transformação de modelo-a-texto cria o código-fonte de um *software*.

Ambas as partes estão relacionadas através do Modelo de Teste que faz referência aos elementos do PIM, bem como o Modelo de Teste Específico de Plataforma que faz referência ao PSM, e o Código-Fonte de Teste que faz referências ao Código-Fonte do *software* a ser testado. Assim, para cada nível de modelo, há um modelo adequado para o teste.

A Figura 4.1 apresenta o processo do *framework* ATCM que é composto por dois processos distintos.

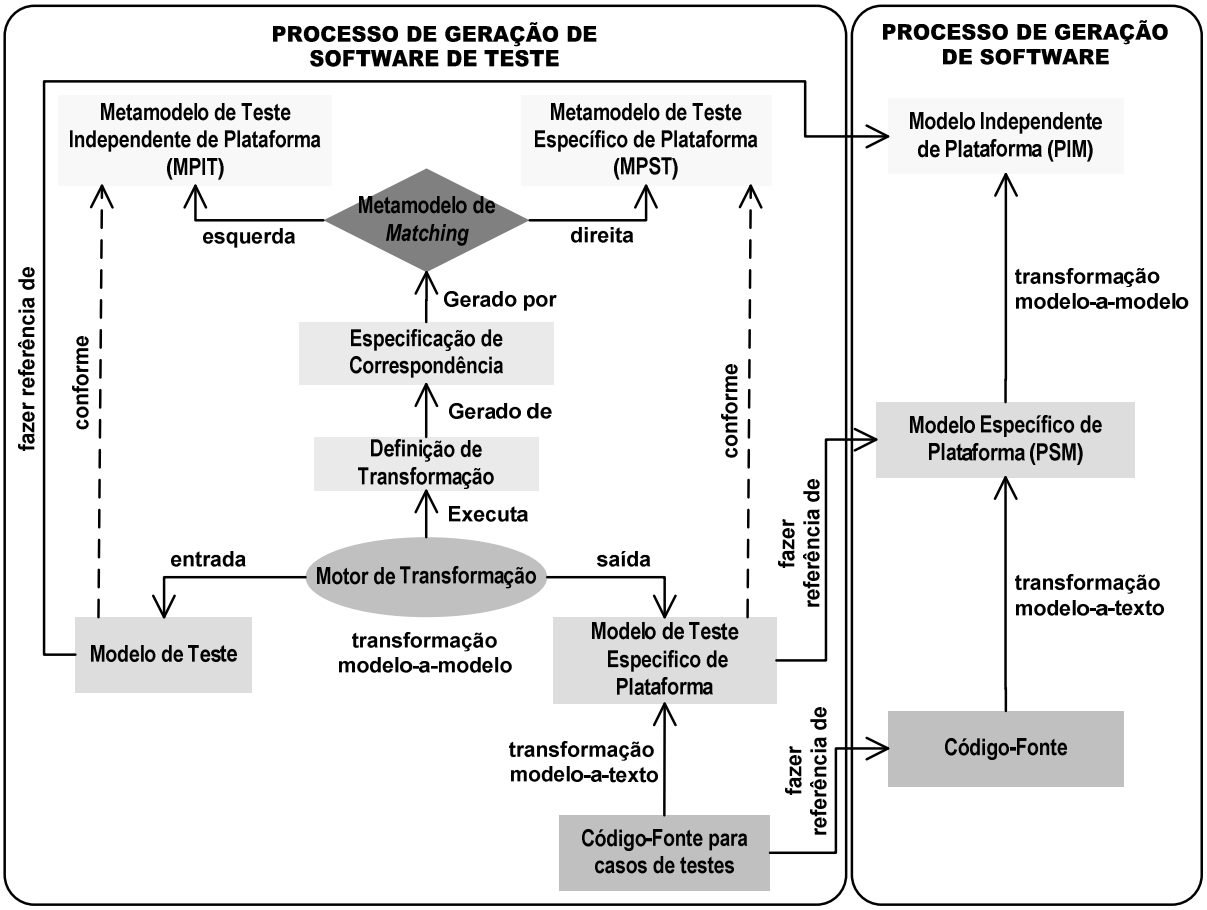


Figura 4.1 - Processo do *framework* ATCM

O objetivo do *framework* ATCM é transformar um Modelo de Teste em um Modelo de Teste Específico de Plataforma a fim de se obter o Código-Fonte de Teste. Esse processo é possível através da execução das definições de transformação por meio de um motor de transformação.

O Modelo de Teste é conforme ao Metamodelo de Teste Independente de Plataforma (MPIT, *Metamodel for Platform Independent Testing*) proposto nesta dissertação, e o Modelo de Teste Específico de Plataforma é conforme ao Metamodelo de Teste Específico de Plataforma (MPST, *Metamodel for Platform Specific Testing*).

Este *framework* ATCM é baseado na geração automática de definições de transformações através das especificações de correspondências. As especificações de correspondências são geradas semi-automaticamente através da correspondência entre os elementos dos metamodelos MPIT e MPST.

A tarefa de construir a especificação de correspondências é um risco, pois se delegada a pessoas e eventualmente pode ocorrer erros ou ser incompletas, por ser uma

atividade que requer muita atenção, por envolver metamodelos que são geralmente complexos.

Portanto, se as especificações de correspondências contêm erros, a definição de transformação conseqüentemente apresentará erros inviabilizando assim a geração automatizada dos casos de teste com perfeição.

Assim, utiliza-se um metamodelo de *mapping* para gerar automaticamente as definições de transformação através da geração semi-automatizada das especificações de correspondências. Esse metamodelo de *mapping* detecta os elementos dos metamodelos de testes que são semanticamente equivalentes ou similares. Gerando as definições de transformação automaticamente, torna-as menos propensa a erros, e consequentemente gera o código-fonte de teste mais confiável e eficaz.

Na seção 4.3, descreve-se o metamodelo de *matching*, metamodelo de teste independente de plataforma e os metamodelos de teste específicos de plataforma, xUnit, JUnit e NUnit. Para a implementação do *framework* ATCM, modelou-se um diagrama de classe que descreve as classes e relacionamentos do *framework* ATCM como apresentado na Figura 4.2.

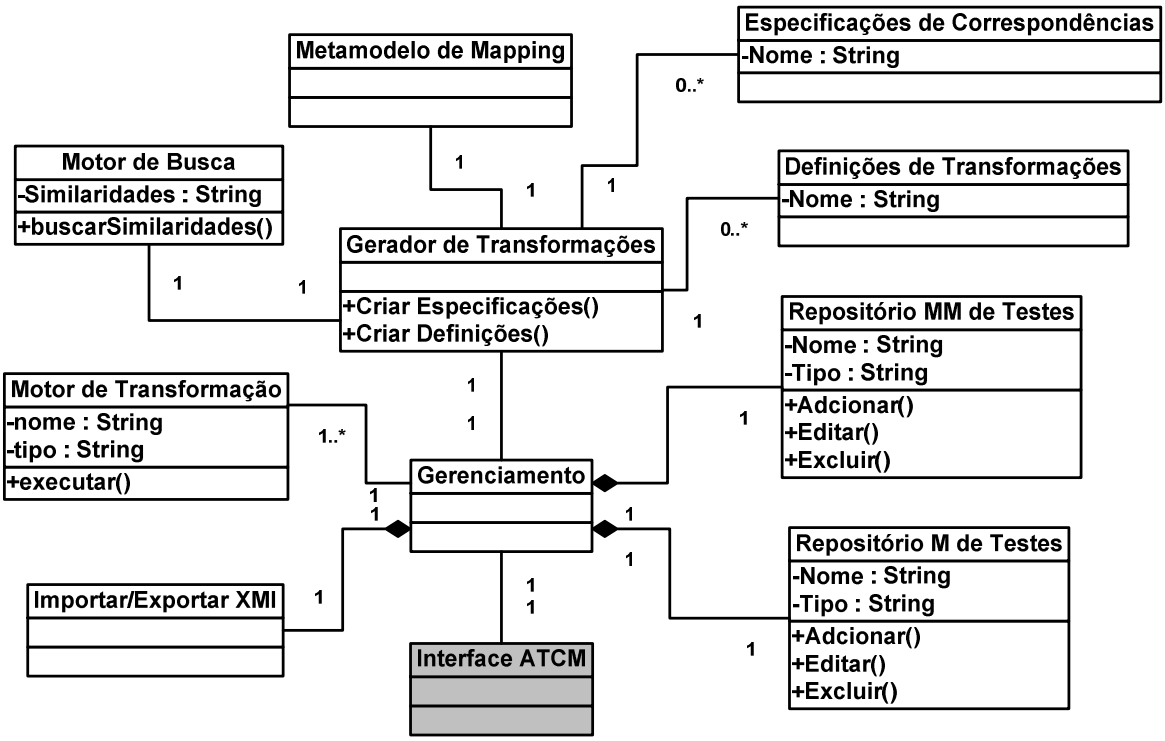


Figura 4.2 - Diagrama de classe do *framework* ATCM

O *framework* ATCM é composto pelas seguintes classes:

- Interface ATCM - interface gráfica para usuário manipular a entrada de metamodelos e criar e editar de modelos;
- Gerenciamento - gerencia a execução dos blocos funcionais que permite a criação e edição dos modelos. Este elemento gerencia o banco de dados de armazenagem dos modelos e metamodelos de teste e invoca a ferramenta motor de transformação;
- Motor de Transformação - executa as definições de transformações e armazena motores de transformações tais como, ATL (ATL, 2006), Epsilon (KOLOVOS et al., 2008), MOFScript (OLDEVIK, 2006), Tefkat (STEEL, 2004), MOFQVT (OMG, 2008);
- Importar/Exportar XMI - é o módulo que traduz um metamodelo do formato XMI para o formado ecore. Ele permite também traduzir um metamodelo conforme ao metamodelo ecore no formato XMI;
- Repositório de MM de Testes - é o banco de dados e armazenagem dos metamodelos de testes;
- Repositório de M de Testes - é o banco de dados de armazenagem dos modelos de testes;
- Gerador de Tranformações - executa a geração das Especificações de Correspondências e Definições de Tranformações, gerencia o Metamodelo de *Mapping* para criar modelos de correspondências e executa o Motor de Buscar que busca as similaridades entre os elementos dos metamodelos fonte e alvo para criar as Especificações de Correspondências;
- Metamodelo de Mapping - é um metamodelo usado para criar modelos de correspondência;
- Motor de Busca - realiza a busca de sinônimos, ou seja, busca por similaridades entre os elementos de metamodelos;
- Especificações de Correspondências - cria um modelo de correspondência a partir dos modelos validados pelo usuário e armazena as especificações de correspondências;
- Definições de Tranformações - cria e armazena as definições de transformações.

4.2 Metodologia

A Figura 4.3 apresenta a metodologia usada no *framework* ATCM em forma de diagrama de atividades, descrevendo cada passo a ser seguido para a utilização deste *framework.*

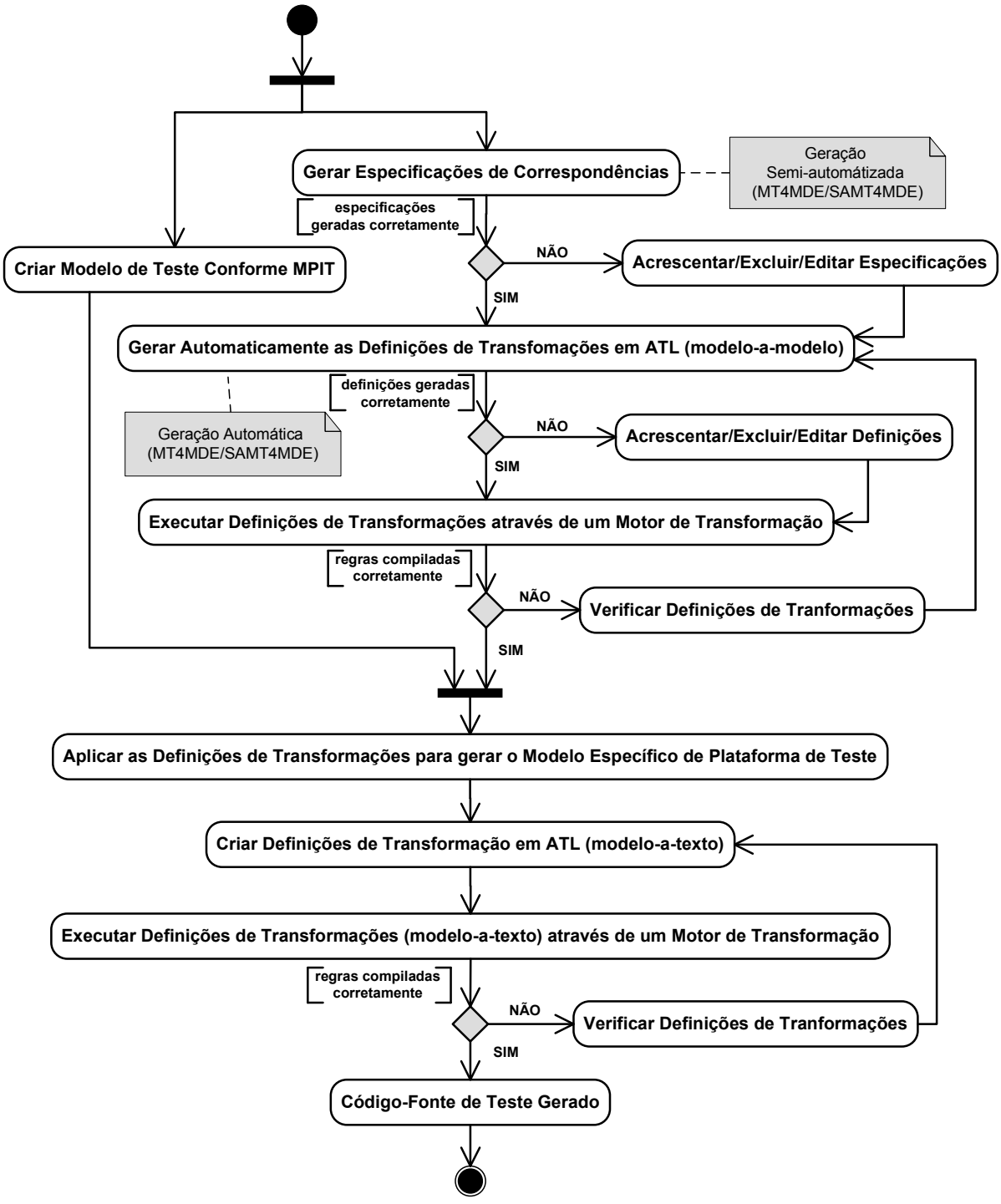


Figura 4.3 - Metodologia do *framework* proposto

Nesta metodologia, o passo inicial é a semi-automatização das correspondências entre os elementos do metamodelos MPIT e MPST. A ferramenta MT4MDE/SAMT4MDE possibilita essa semi-automatização através de um metamodelo de *matching* que descreve as correspondências entre os elementos do metamodelo fonte e metamodelo alvo.

Após a geração das especificações de correspondências, verifica-se se as especificações foram geradas corretamente. A ferramenta possibilita acrescentar, excluir e editar essas especificações. Com as especificações geradas, obtêm-se automaticamente as definições de transformação de modelo-a-modelo em ATL.

O desenvolvedor verifica as definições geradas, se estas tiverem incorretas ou incompletas, o desenvolvedor pode acrescentar, excluir e editar essas definições. Após a verificação das definições, estas são executadas em um motor de transformação para enfim gerar o modelo de teste específico de plataforma.

Uma vez que o modelo de teste específico de plataforma está gerado, segue para a etapa de geração do código-fonte de teste. Para geração do código de teste, a definição de transformação de modelo-a-texto é criada em ATL, para em seguida ser executadas em um motor de transformação para enfim obter-se o código-fonte de teste.

4.3 Metamodelos Desenvolvidos

Para concretização do processo de desenvolvimento de casos de teste proposto no *framework* ATCM, metamodelos foram desenvolvidos para assegurar a mais ampla qualidade desse processo. Os metamodelos MPIT e MPST são base para a geração automática de casos de teste para um determinado *software*.

4.3.1 Metamodelo de teste independente de plataforma

Para a criação do metamodelo de teste independente de plataforma (MPIT, *Metamodel for Platform Independent Testing*), utilizou-se a técnica de Teste Funcional (ou caixa-preta), como estudado na subseção 2.2.3. Nesta técnica, os dados de entrada são fornecidos, o teste é executado e o resultado obtido é comparado com o resultado esperado previamente conhecido (SOMMERVILLE, 2007).

Utilizou-se também os níveis Teste de Módulo (ou teste de unidade) e o Teste de Integração, como estudado na subseção 2.2.1. O nível de teste de Unidade explora cada unidade do projeto, provocando falhas ocasionadas por defeitos de lógica e de

implementação, a fim de encontrar defeitos nos componentes (DELAMARO et al., 2007). O nível de teste de Integração provoca falhas associadas às interfaces entre os módulos, esta fase testa a integração dos módulos (DELAMARO et al., 2007). Nesta fase de teste, quatro tipos de abordagens foram estudadas: *big-bang*, *topdown*, *bottom-up* e sanduíche.

Após estudadas cada abordagem, a abordagem *topdown* foi selecionada para a modelagem do metamodelo proposto, pois esta abordagem permite a verificação antecipada do comportamento de alto nível. Nesta abordagem, os testes se iniciam em um módulo principal e vai descendo para os módulos subordinados a este módulo principal. Assim, os módulos podem ser adicionados, um por vez e suporta as abordagens *breadth first* e *depth first* (PFLEEGER, 2004).

O MPIT se constitui em uma linguagem específica de domínio (DSL, *Domain-Specific Language*) para o domínio de teste, ou seja, é uma linguagem dedicada a resolver um problema específico. O MPIT é dedicado a resolver o problema de teste.

A Figura 4.4 apresenta um fragmento do metamodelo de teste independente de plataforma, MPIT.

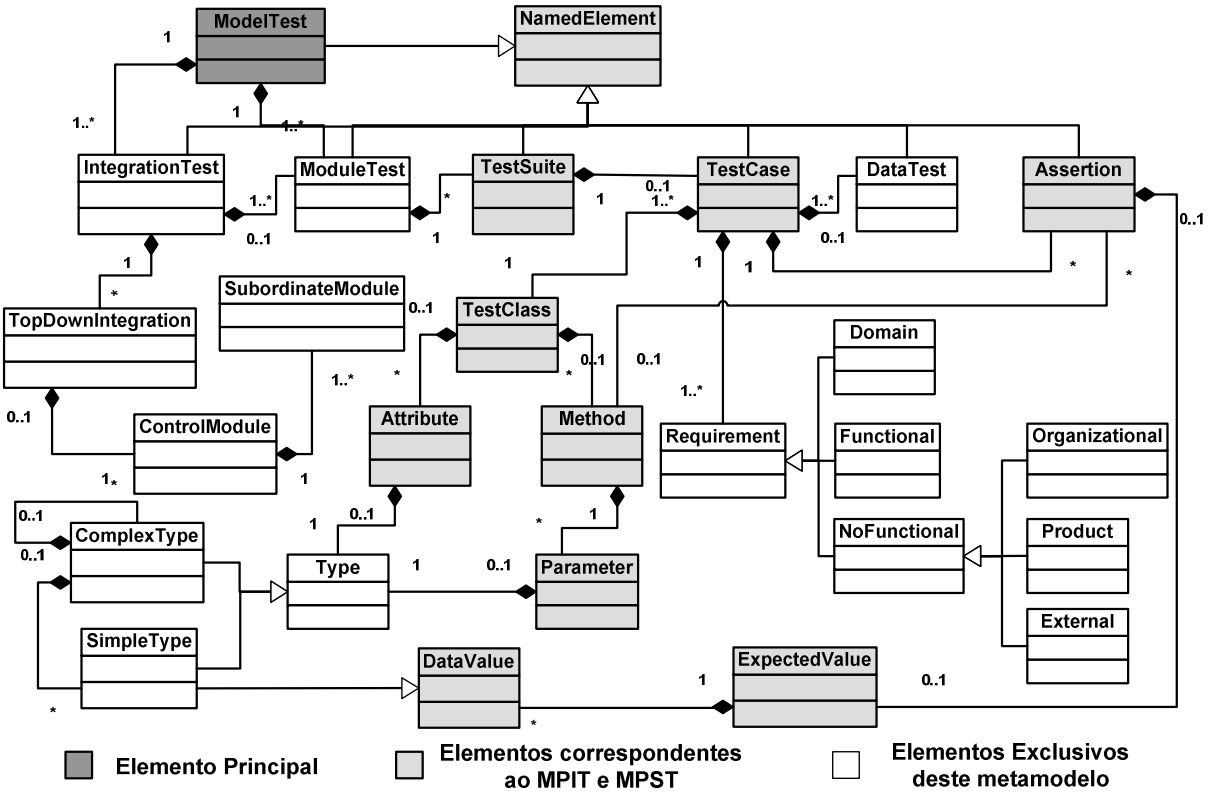


Figura 4.4 - Fragmento do Metamodelo MPIT

A Figura 4.4 apresenta o fragmento do MPIT que é composto pelos seguintes elementos de testes:

- NamedElement (Nome do Elemento) - este elemento é generalização dos componentes de testes. Os principais elementos de testes têm um nome que o identifica e uma nota que o descreve;
- ModelTest (Modelo de Teste) - é o elemento principal do metamodelo MPIT, este elemento contém as fases de teste de unidade e teste de integração;
- ModuleTest (Teste de Módulo) - este elemento apresenta características de testes para serem executados os testes da fase de testes de unidade. Este elemento é um repositório do TestSuite e TestCase;
    - TestSuite (Teste Suíte) - este elemento de teste tem a função de executar um conjunto de TestCase. Contém instruções detalhadas para cada conjunto de TestCase e informações sobre a configuração do sistema a ser utilizado durante os testes;
    - TestCase (Casos de Teste) - apresenta os casos de teste para a execução dos testes de *software*. Este elemento contém o conjunto de condições e variáveis que são determinadas pelo testador a fim de determinar se a aplicação apresenta alguma falha, ou seja, é uma especificação de uma situação particular expressa pelo testador para testar uma aplicação de *software*. Os dados de testes (DataTest) contribuem para a execução deste elemento de teste. Também apresenta um ou vários Assertions. Como atributos deste elemento tem-se as entradas de dados (input) os valores esperados (expectedValue) os processos (procedures) e pré-condições (pré-conditions);
    - DateTest (Dados de Testes) - representa um conjunto de dados para o teste que é usado para produzir o resultado de teste;
    - Assertion - é uma condição que deve ser *true* após executar um TestCase. Este elemento pode ser de vários tipos que são especificados pelo seu tipo de atributo. É neste elemento que se fazem as comparações entre os valores esperados com os valores colhidos no teste;
    - Requirements (Requisitos) - apresentam os requisitos colhidos do sistema em teste, este elemento podem ter requisitos funcionais (Functional), não-funcionais (NoFunctional) e Domínio (Domain) que são herdadas do

elemento Requiriments. Os Requisitos não-funcionais é a generalização dos componentes Organizational, Product e External;

- TestClass (Teste de Classe) - neste elemento, os grupos de objetos e comportamentos do sistema a ser testados são descritos;
- Attribute (Atributo) - está associada ao elemento TestClass, ele controla os valores de uma classe;
- Type (Tipo) - é um elemento abstrato do elemento Attribute, ele representa os tipos de atributos a serem testados e podem ser:
  - ComplexType (Tipo Complexo) - são valores que contém outros valores;
  - SimpleType (Tipo Simples) - são valores simples;
- Method (Método) - é um elemento abstrato do elemento Attribute, ele representa uma seqüência de declarações ou parâmetros (Parameter) para executar uma ação;
- Parameter (Parâmetro) - apresenta um conjunto de valores das características que determinam as funções de um determinado sistema;
- DataValue (Dados de Valor) - este elemento armazena os valores colhidos durante a execução dos testes de *software*, eles possuem valores determinados no elemento TDataType;
- ExpectedValue (Valores Esperados) - neste elemento os valores esperados que são comparado com os valores colhidos coma execução dos testes são armazenados. Isso acontece na classe Assertion. Este elemento possui valores determinados no elemento TDataType;
- TDataType - é um tipo de dados *enumeration* predefinido que contém valores: *integer*, *string*, *float*, *character*, *varchar*, *double*, *long*, *short*, *date*, *boolean*.

o IntegrationTest (Teste de Integração) - este elemento contém a descrição da execução dos testes de integração, onde primeiro se testa um módulo principal do sistema em teste para finalmente testar os módulos subordinados a ele;

- TopDownIntegration - elemento que representa o tipo de abordagem utilizada para a execução da técnica de teste de integração;
- ControlModule - este elemento indica qual o módulo principal do sistema em teste, aqui também são listados os módulos subordinados a este módulo principal;
- SubordinateModule - este elemento indica os módulos subordinados

No Anexo A, apresenta-se a descrição completa do MPIT com seus respectivos elementos, atributos e relacionamentos. MPIT é baseado na linguagem de metamodelagem *Ecore*, pertencente ao *framework* EMF. A Figura 4.5 apresenta o MPIT *Ecore* em forma de árvore.

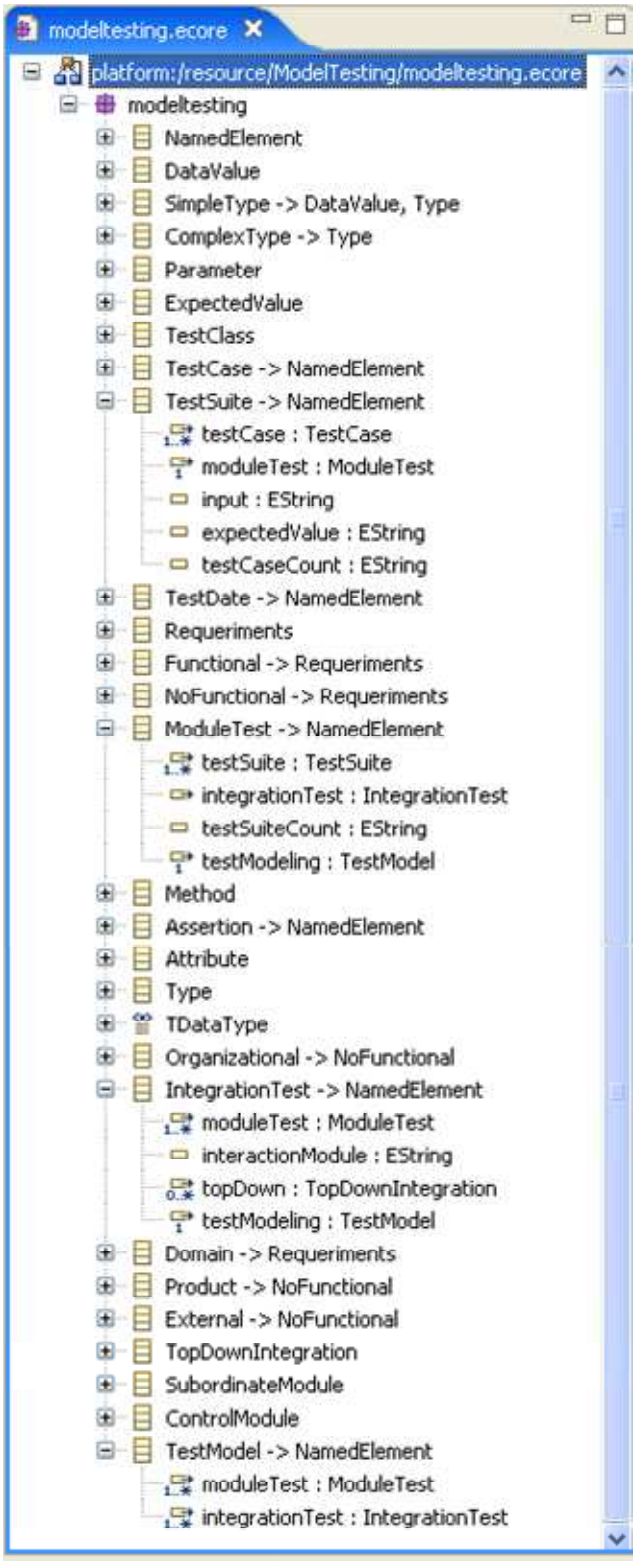


Figura 4.5 - MPIT.ecore em forma de árvore

O metamodelo de teste independente de plataforma (MPIT) proposto neste trabalho é capaz de lidar com a variabilidade de teste igualmente como o metamodelo proposto no U2TP. Porém, o MPIT apresenta algumas diferenças com relação ao metamodelo proposto no U2TP. A Tabela 4.1 apresenta um comparativo desses metamodelos.

O MPIT foi desenvolvido com base no metamodelo proposto no U2TP, porém com uma visão voltada para a metamodelagem *ecore* e aumentando o leque de contribuições para suportar a fase de teste de integração. Na Tabela 4.2, apresenta-se os elementos correspondentes entre o metamodelo U2TP e o MPIT.

Tabela 4.1 - Comparativo entre U2TP e MPIT

	U2TP	**MPIT**
Integração com UML	UML 2.0	UML 2.0
Metamodelagem utilizada	MOF	Ecore
Técnica de Teste	Testes Funcionais	Testes Funcionais
Fases de Testes	Teste de Unidade	Teste de Unidade e Integração
Mapeamento para Plataformas Específicas	COTE, TTCN-3 e JUnit	Família xUnit (JUnit, NUnit, etc)

Tabela 4.2 - Elementos correspondentes entre U2TP e MPIT

U2TP	**MPIT**
SUT	TestModel
TestContext	TestSuite
TestCase	TestCase
DataPool	DataType

4.3.2 Metamodelo específico de plataforma de teste xUnit

O metamodelo de teste específico de plataforma (MPST, *Model Platform Specific for Testing*) foi desenvolvido para a plataforma xUnit. Esta plataforma é da família unitária de testes, utilizada para escrever e executar testes repetitivos em aplicações de *software*

(MESZAROS, 2007). xUnit é um *Framework open source* que facilita a criação de código-fonte para a automação de testes unitários.

Este *framework* de teste unitário verifica se cada método de uma classe funciona da forma esperada, exibindo possíveis erros ou falhas (MESZAROS, 2007). Após algumas pesquisas, encontrou-se um metamodelo de teste específico de plataforma xUnit no trabalho de JAVED (JAVED et al., 2007).

No entanto, algumas adaptações neste metamodelo foram feitas, pois como no trabalho de (JAVED et al., 2007) este metamodelo xUnit é direcionado a testes em diagramas de seqüência, tendo métodos de chamadas. Assim, propõe-se um metamodelo de teste específico de plataforma xUnit a fim de testar modelos de testes para notação UML em diagramas de classes.

A Figura 4.6 apresenta o fragmento do metamodelo de teste específico de plataforma xUnit, denominado MPST-xUnit.

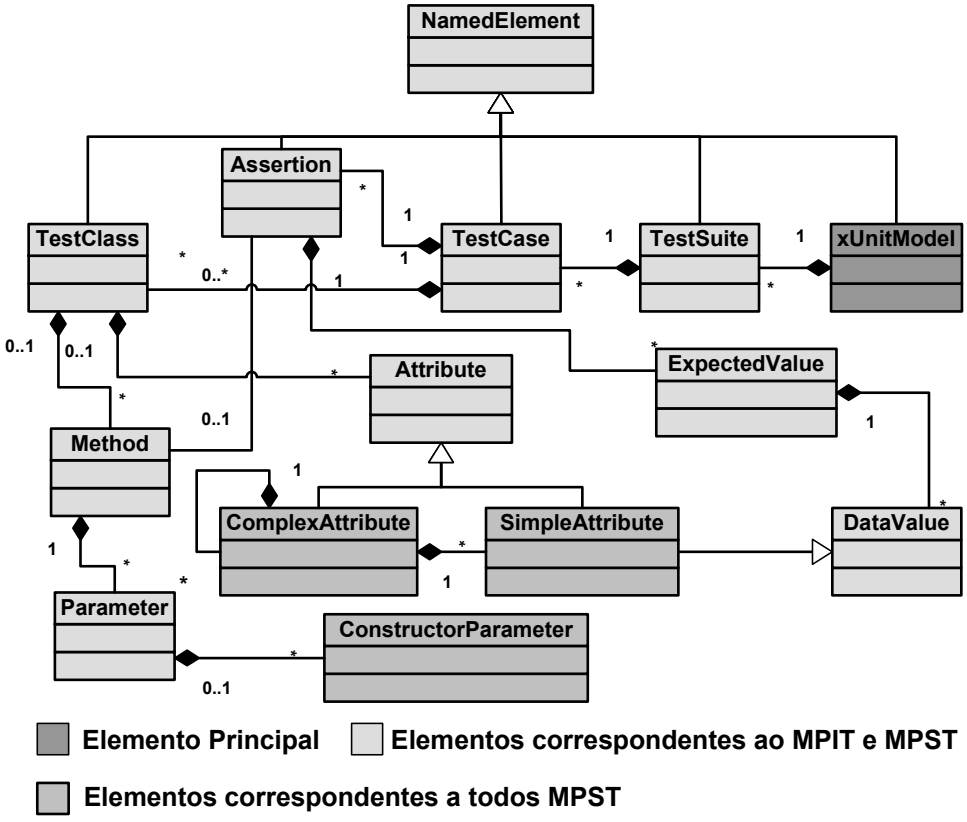


Figura 4.6 - Fragmento do Metamodelo MPST-xUnit

Alguns elementos apresentados na Figura 4.5 foram descritos na subseção 4.3.3. A nomenclatura nas figuras informa quais elementos são correspondentes a todos os

metamodelos, para todos os metamodelos MPST e quais são específicos para cada metamodelo. Assim os elementos de testes do MPST-xUnit são descritos a seguir:

- xUnitModel (Modelo xUnit) - é o elemento principal do metamodelo MPST-xUnit, este elemento contém vários TestSuite;
- ConstructorParameter - é um bloco declarações. É semelhante a uma instância de método.

No Anexo B, apresenta-se a descrição completa do MPST-xUnit com seus respectivos elementos, atributos e relacionamentos. Assim como o MPIT, MPST-xUnit é baseado na linguagem de metamodelagem *Ecore*, pertencente ao *framework* EMF. A Figura 4.7 apresenta o MPST-xUnit *Ecore* em forma de árvore.

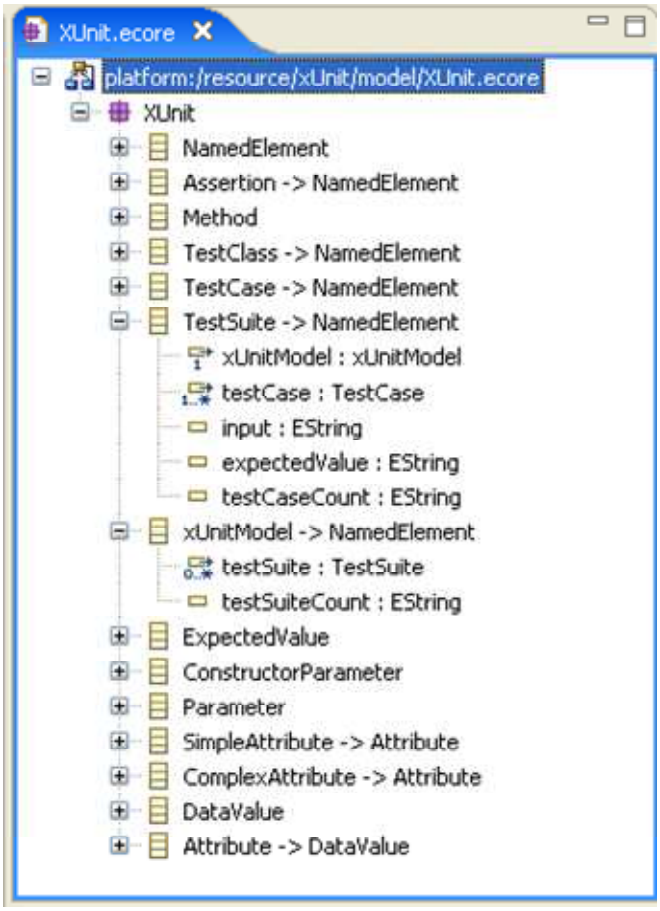


Figura 4.7 - MPST-xUnit.ecore em formato de árvore

4.3.3 Metamodelo específico de plataforma de teste JUnit

O Metamodelo de Teste Específico de Plataforma é baseado no *framework* JUnit que tem estruturas de teste usados para escrever e executar testes de *software* repetitivos e de

auto-controle em Java (RAINSBERGER, 2005). JUnit facilita a criação automatizada de código fonte de teste. Este *framework* de teste facilita a criação de código para a automação de testes com apresentação dos resultados.

Com este *framework* de teste, pode-se verificar se cada método de uma classe funciona de forma esperada, exibindo possíveis erros ou falhas podendo ser utilizado tanto para a execução de baterias de testes como para extensão (MESZAROS, 2007).

O MPST-JUnit é baseado no trabalho de JAVED (JAVED et al., 2007). Após algumas pesquisas literárias (MESZAROS, 2007) (RAINSBERGER, 2005) (MASSOL, 2004), assim acrescentou-se elementos ao MPST-xUnit apresentado na subseção 4.3.3. Estes elementos acrescentados ao MPST-xUnit descrevem o *framework* de teste JUnit.

A Figura 4.8 apresenta o fragmento do metamodelo de teste específico de plataforma JUnit, denominado MPST-JUnit.

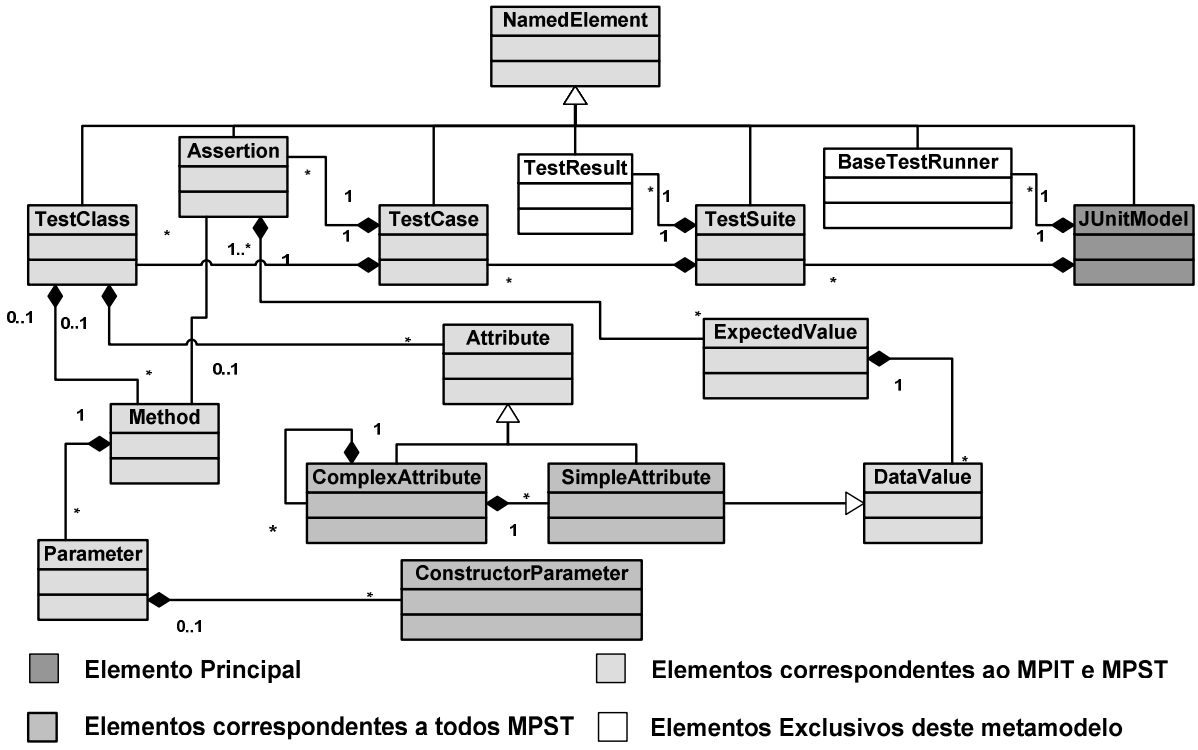


Figura 4.8 - Fragmento do Metamodelo MPST-JUnit

Esses elementos referentes ao MPST-JUnit são descritos a seguir:

- JUnitModel (Modelo JUnit) - é o elemento principal do metamodelo MPST-JUnit, este elemento contém vários TestSuite;
- BaseTestRunner - é um lançador do TestSuite, pois fornece vários *runners* que podem ser utilizados para executar seus testes;

- TestResult - coleta todos os resultados dos testes, tanto como os erros e falhas ocorridas durante a execução do teste.

No Anexo C, apresenta-se a descrição completa do MPST-JUnit com seus respectivos elementos, atributos e relacionamentos. Assim como o MPIT, MPST-xUnit, o metamodelo de teste específico de plataforma JUnit é baseado na linguagem de metamodelagem *Ecore*, pertencente ao *framework* EMF. A Figura 4.9 apresenta o MPST-JUnit *Ecore* em forma de árvore.

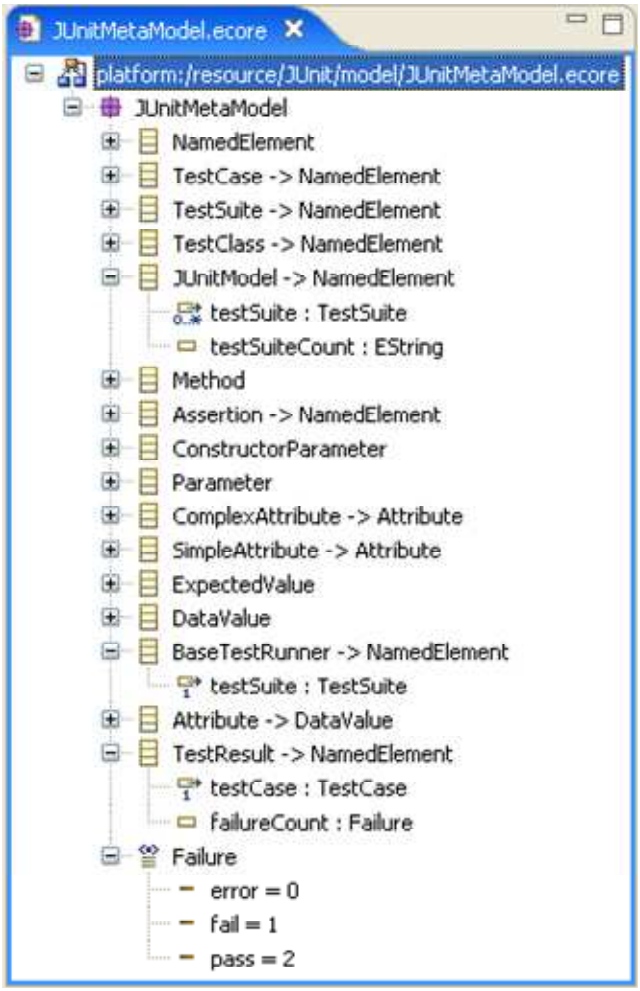


Figura 4.9 - MPST-JUnit em formato de árvore

4.3.4 Metamodelo específico de plataforma de teste NUnit

O Metamodelo de Teste Específico de Plataforma NUnit é baseado no *framework* que define testes para linguagem de programação CSharp ou C#. NUnit define testes usando

atributos C# ao invés de objeto, herança, assim os detalhes de seu *framework* diferem significativamente dos JUnit (HAMILL, 2004).

NUnit é um *framework* de teste unitário que segue a linha da família de teste unitários xUnit. É um *framework open-source* da Microsoft (JEFFRIES, 2004).

O MPST-NUnit também é baseado no trabalho de JAVED (JAVED et al., 2007). Após pesquisas literárias (JEFFRIES, 2004) (HAMILL, 2004) (NANTZ, 2004). Assim, acrescentou-se elementos que descrevem o *framework* de teste NUnit ao MPST-xUnit apresentado na subseção 4.3.3.

A Figura 4.10 apresenta o fragmento do metamodelo de teste específico de plataforma NUnit, denominado MPST-NUnit.

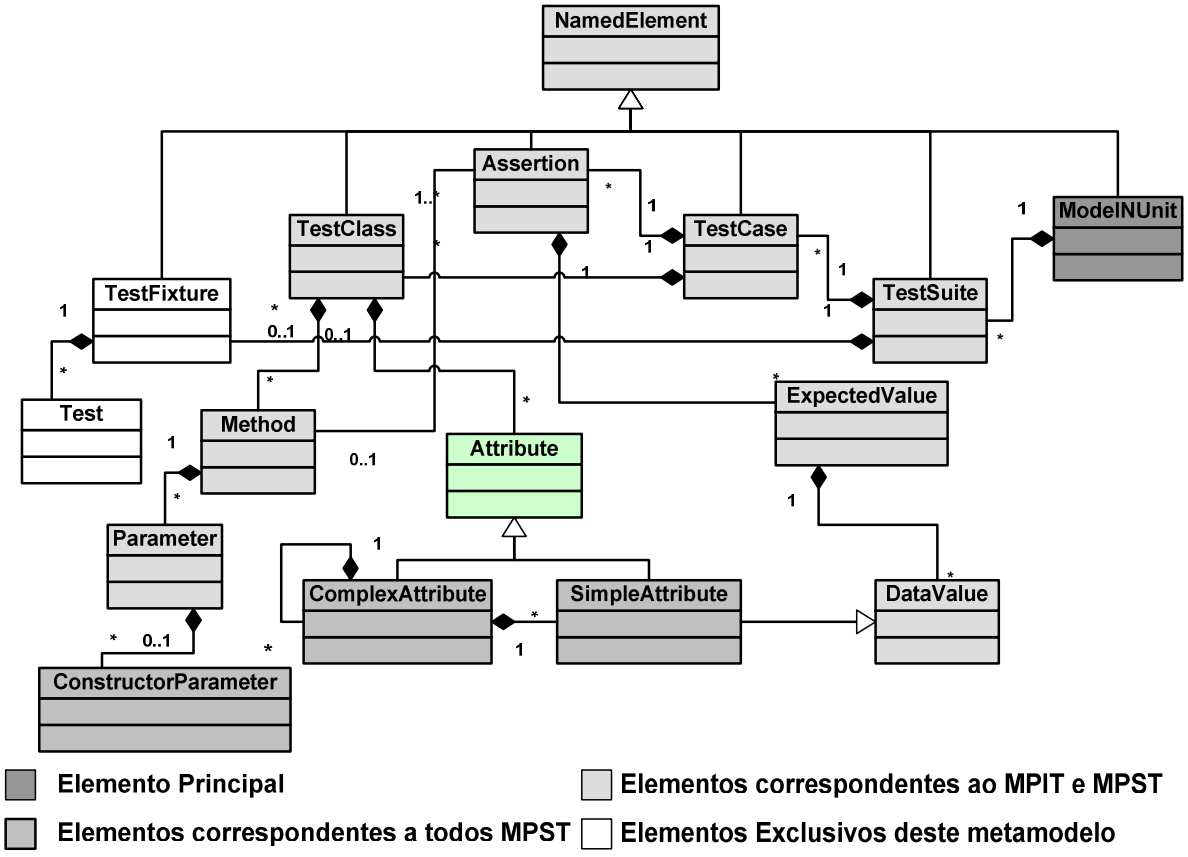


Figura 4.10 - Fragmento do Metamodelo MPST-NUnit

Esses elementos referentes ao MPST-NUnit são descritos a seguir:

- NUnitModel (Modelo NUnit) - é o elemento principal do metamodelo MPST-NUnit, este elemento contém vários TestSuite;

- TestFixture - refere-se ao estado fixo usado como uma base para a execução de testes de *software* testes. Utilizado para agrupar e executar múltiplos testes que um teste lógico recolha de funcionalidade. Ele é composto por um ou vários Test;
- Test - podem ter métodos *setup* e *teardown* de preparação e limpeza após um teste de *software*.

No Anexo D, apresenta-se a descrição completa do MPST-JUnit com seus respectivos elementos, atributos e relacionamentos. Assim como o MPIT, MPST-xUnit e MPST-JUnit, o metamodelo de teste específico de plataforma NUnit é baseado na linguagem de metamodelagem *Ecore*, pertencente ao *framework* EMF, a Figura 4.11 apresenta o MPST-NUnit *Ecore* em forma de árvore.

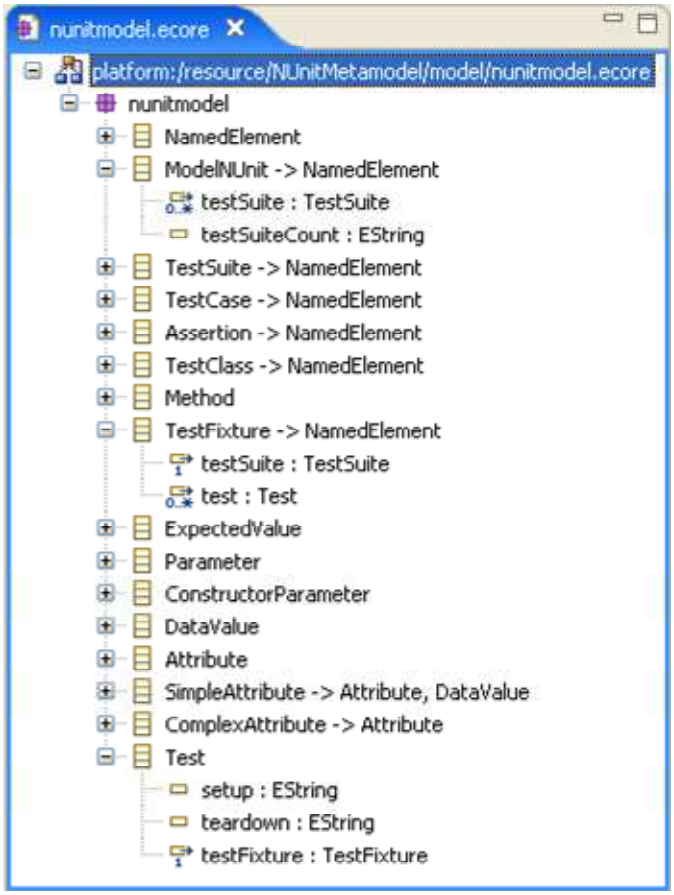


Figura 4.11 - MPST-NUnit em formato de árvore

4.4 Abordagem para *Matching* de Metamodelos

No trabalho de (LOPES, 2006a) (LOPES, 2006b) (LOPES, 2006c) (SOUZA Jr et al., 2009), um algoritmo e uma ferramenta para *matching* de metamodelo são apresentados,

possibilitando gerar as especificações de correspondência entre os metamodelos. Neste trabalho, este metamodelo é utilizado para assegurar a qualidade das especificações de correspondência e conseqüentemente a geração das definições de transformações a fim de se obter o código-fonte de teste.

Este metamodelo é utilizado para evitar injeção de erros no código-fonte inseridos durante a criação das especificações de correspondências. A Figura 4.12 apresenta o metamodelo de *matching* onde cada elemento demonstrado tem um papel bem definido na modelagem das correspondências entre os modelos. A descrição precisa destes elementos se encontra em (LOPES, 2006b).

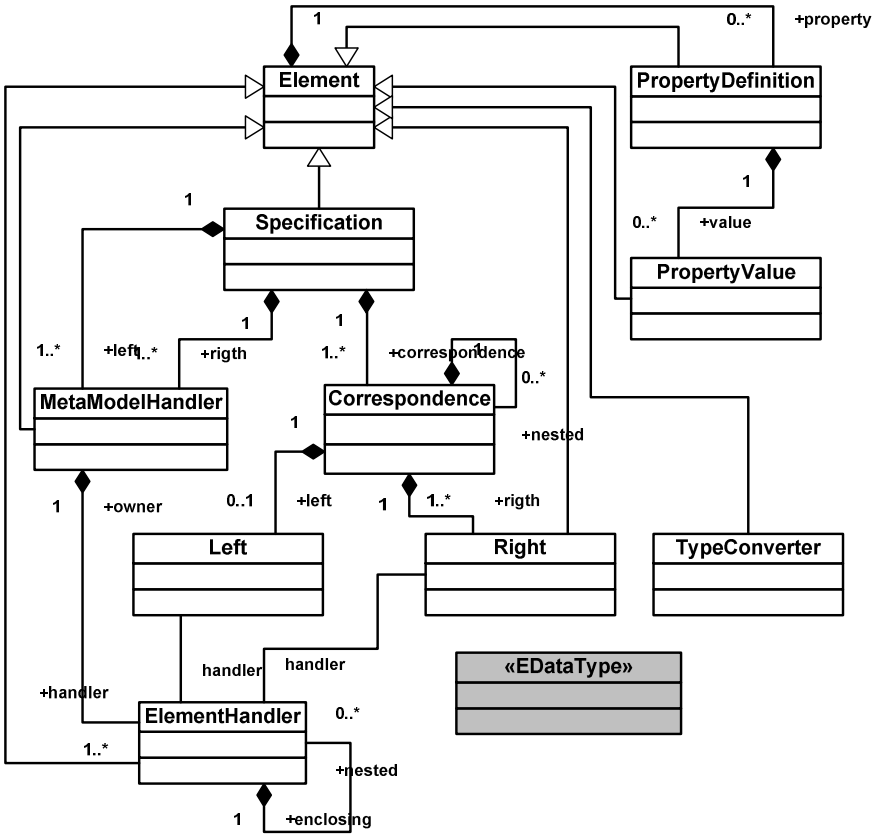


Figura 4.12 - Metamodelo de *Matching* (LOPES, 2006b)

Este metamodelo de *matching* utiliza as similaridades para medir quão separados em significados e estruturas estão dois metamodelos. Este é fundamental para determinar uma transformação de modelos. Se dois metamodelos M_a e M_b estão muito distantes semanticamente, não é possível transformar um modelo M_1, que é conforme a M_a, em um modelo M_2, que é conforme M_b.

A função de transformação pode ser definida da seguinte forma:

(Equação 4.1) $Transf\left(M_1(s)/M_a, C_{M_a \to M_b}/M_c\right) \to M_2(s)/M_b$

Onde:

- *Transf* é um operador que tem dois parâmetros (um modelo de um sistema *s* para ser transformado e a especificação de correspondências) e produz um outro modelo para o mesmo sistema *s*;
- M_1 é um modelo de um sistema *s* criado conforme ao metamodelo M_a;
- M_2 é um modelo do mesmo sistemas *s* criado conforme ao metamodelo M_b;
- $C_{M_a \to M_b}$ é o mapeamento entre M_a e M_b criado usando o metamodelo M_c.

As especificações de correspondências são criadas através do operador *Match*, como seguinte:

(Equação 4.2) $Match(M_a, M_b, M_c) \to C_{M_a \to M_b}/M_c$

Sendo que:

- *Match* é um operador que tem três parâmetros (um metamodelo M_a, um metamodelo M_b e um metamodelo M_c) e produz um modelo de correspondência $C_{M_a \to M_b}$ que contém relacionamento entre M_a e M_b. Este modelo de correspondência é conforme o metamodelo M_c;
- M_a é um metamodelo para criação de uma família de modelos;
- M_b é outro metamodelo para a criação de uma família de modelos;
- M_c é um metamodelo para criação de uma família de especificações de correspondências.

De acordo com *framework* ATCM, MPIT desempenha o papel de M_a, o MPST desempenha o papel de M_b. O metamodelo M_c e o operador *Match* são fornecidos em (LOPES, 2006b) (SOUZA Jr et al., 2009). MPIT pode ser apenas um metamodelo para uma família de modelos de teste. MPST é um conjunto de metamodelos de acordo com uma plataforma específica para o teste.

4.5 Implementação do Protótipo

O *framework* ATCM foi implementado como um *plug-in* para o IDE Eclipse. Este *framework* proposto é implementado por ferramentas que o auxiliam na geração automatizada de casos de teste com a finalidade de testar o código-fonte de um determinado sistema de *software*.

A Figura 4.13 apresenta o *plug-in* do *framework* ATCM. Este *plug-in* é composto pelas ferramentas T2MDT, SAMT4MDE, MT4MDE e Motor de Transformação.

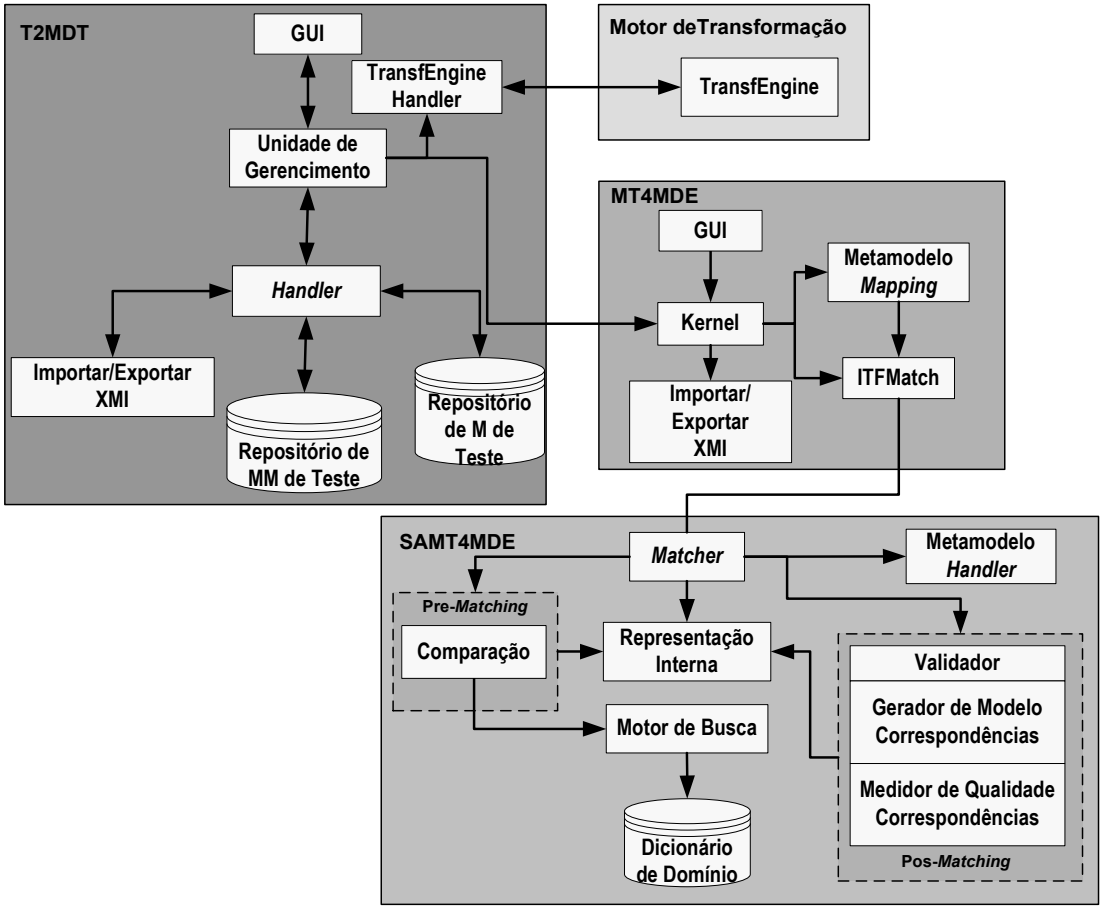


Figura 4.13 - *Plug-in* de uma Abordagem para Teste Dirigido por Modelos: Arquitetura

Estas ferramentas têm funções específicas para auxiliarem o processo de geração de casos de teste do *framework* ATCM. A ferramenta T2MDT contém os metamodelos de testes que auxiliam a geração do código-fonte, podendo criar e editar modelos de testes, também suporta a transformação de modelos.

A MT4MDE e SAMT4MDE suportam a geração semi-automática das especificações de correspondências e automatiza a geração de definições de transformações.

O Motor de Transformação suporta a execução das definições de transformação. Estas ferramentas são descritas detalhadamente com seus respectivos componentes nas sub-seções seguintes.

4.4.1 Ferramenta para teste dirigido a modelos (T2MDT)

A Ferramenta para Teste Dirigido a Modelos (T2MDT, *Tool for Model Driven Testing*) é uma contribuição deste trabalho para auxiliar a geração de casos de teste e código-fonte de testes.

Esta ferramenta é responsável pelo gerenciamento do repositório dos metamodelos de testes. Ela permite que o desenvolvedor crie e edite os modelos de testes conforme os metamodelos armazenados neste repositório. T2MDT é composta pelos seguintes elementos:

- GUI - é a interface gráfica do T2MDT;
- Unidade de Gerenciamento - gerencia a execução de outros blocos funcionais que permite a criação e edição dos modelos. Este elemento invoca a ferramenta MT4MDE e a ferramenta Motor de Transformação;
- *Handler* - permite a manipulação e navegação através dos metamodelos de testes e modelos de testes;
- Repositório de MM de Teste - é o banco de dados e armazenagem dos metamodelos de testes;
- Repositório de M de Teste - é o banco de dados e armazenagem dos modelos de testes;
- TransfEngineHandler - permite chamar um motor de transformação para executar as definições de transformações;
- Importar/Exportar XMI - permite a troca dos metamodelos ou modelos a partir de/para repositórios de/para ambientes externos.

4.4.2 Ferramenta de correspondências para MDE (MT4MDE)

MT4MDE (*Mapping Tool for Model Driven Engineering*) é uma ferramenta que permite a criação das especificações de correspondências, ou seja, determina uma possível

especificação de correspondência e a geração de definições de transformação em ATL a partir de um modelo de correspondência (LOPES, 200b) (LOPES, 200c).

A ferramenta MT4MDE foi desenvolvida por (LOPES, 2006b) (LOPES, 200c) (SOUZA Jr et al., 2009). Ela é composta pelos elementos conforme apresentado na Figura 4.13 e descritos a seguir (LOPES, 2007):

- GUI - é a interface gráfica do MT4MDE;
- Kernel - executa todas as funcionalidades básicas de MT4MDE. É o núcleo do *plug-in*;
- Importar/Exportar XMI - é o módulo que traduz um metamodelo do formato XMI para o formado *ecore*. Ele permite também traduzir um metamodelo conforme o metamodelo *ecore* no formato XMI;
- Metamodelo de *Mapping* - é um metamodelo usado para criar modelos de correspondência;
- ITFMatch - é a interface para manipular *Matcher*.

O MT4MDE foi implementado com a intenção de ser facilmente extensível, dessa forma as ferramentas que precisam dos mecanismos fornecidos por geração e correspondências ou para definição de transformação podem reutilizá-lo (LOPES, 2007).

4.4.3 Ferramenta de geração semi-automática de correspondências para MDE (SAMT4MDE)

O SAMT4MDE (*Semi-Automatic Matching Tool for MDE*) teve como base a implementação do algoritmo de busca por similaridades estruturais e em *cross-relationship* (POTTINGER & BERINSTEIN, 2003). Esta ferramenta foi implementada como um *plug-in* para Eclipse usando o *framework* EMF. Ela é uma extensão da ferramenta MT4MDE.

Esta ferramenta foi desenvolvida por (LOPES, 2006b) (LOPES, 2006c). A arquitetura de SAMT4MDE apresentada na Figura 4.8 é composta de:

- *Matcher* - é o módulo que implementa a interface ITFMatch para executar as funcionalidades do MT4MDE e coordena a criação de correspondências entre dois metamodelos;
- Representação Interna - é uma representação mais adaptada para criação de correspondências como sendo um conjunto de elementos inter-relacionados;
- Metamodelo *Handler* - é módulo que permite a navegação entre os metamodelos;
- Motor de Busca - realiza a busca de nomes sinônimos, ou seja, busca por similaridades entre os elementos de metamodelos;

- Dicionário de Domínio - é uma base de dados que armazena dicionários de domínios;
- Validador - recebe os elementos correspondentes e interage com o usuário a fim de validá-los;
- Modelo Gerador Correspondências - cria um modelo de correspondência a partir dos modelos validados pelo usuário;
- Medidor de Qualidade de Correspondências - avalia os resultados das medidas de qualidade das correspondências e fornece medidas de qualidade de correspondência.

SAMT4MDE é uma ferramenta que completa a ferramenta MT4MDE, esta junção tem a funcionalidade de determinar as correspondências entre dois metamodelos de forma semi-automática. A SAMT4MDE permite a criação de um modelo de correspondência baseado nas correspondências (LOPES, 2006b) (SOUZA Jr et al., 2009).

4.4.4 Motor de transformação

O Motor de Transformação (*Transformation Engine*) executa as definições de transformação geradas anteriormente. As ferramentas utilizadas para geração automática das definições de transformação e semi-automatização das especificações de correspondências são independentes de um determinado motor de transformação.

Esta ferramenta pode ser qualquer motor de transformação, como ATL (ATL, 2006), QVT (OMG, 2008), MOFScript (OLDEVIK, 2006) para executar as transformações. O *framework* ATCM utiliza a linguagem de transformação ATL (ATL, 2006) para criar a definição de transformação e executá-lo, respectivamente.

4.6 Conclusão

Neste capítulo, apresentou-se um *framework* para a geração automatizada de casos de teste de *software*, com a finalidade de testar o código-fonte gerado por uma abordagem MDx. Um *framework* chamado ATCM foi desenvolvido a fim de obter o código-fonte de teste. Para a implementação deste *framework*, desenvolveu-se metamodelos de testes.

Um *plug-in* composto por quatro ferramentas foi implementado para o Eclipse. Estas ferramentas auxiliam na geração do código-fonte de teste de forma automatizada, tornando essa geração de forma mais segura e menos propensa a erros.

A Tabela 4.3 apresenta um comparativo entre o *framework* ATCM e os *framework* dos (JAVED et al, 2007), (LI et al, 2006) e (ALVES et al, 2007).

Tabela 4.3 - Comparativo entre *framework* ATCM e outros

	ATCM	JAVED et al	LI et al	ALVES et al
Técnica de Teste	Funcional	Funcional	Funcional	Funcional
Fase de Teste	Unidade e Integração	Unidade	Unidade	Unidade
Metamodelos Independente de Plataforma	MPIT	Seqüência	U2TP	U2TP
Metamodelos de Teste Específicos de Plataforma	MPSTxUnit, MPSTJUnit e MPSTNUnit	xUnit	Web	JUnit e Spaces
Metamodelagem	EMF	MOF	MOF	MOF
Modelagem PIM	Diagramas de Classe UML	Diagramas de Seqüência UML	Diagramas de Classe UML	Diagramas de Classe UML
Motor de Transformação	ATL	Tefkat e MOFScript	-	ATL
Especificação de Correspondências	Semi-Automática (MT4MDE e SAMT4MDE)	Manual	Manual	Manual
Definição de Transformação	Automática (MT4MDE e SAMT4MDE)	Manual	Manual	Manual
Código-Fonte de Teste	Qualquer tipo de aplicação	Qualquer tipo de aplicação	Aplicações WEB	Java e Spaces

A Tabela 4.3 apresenta as contribuições desta dissertação em relação a outros trabalhos relacionados pesquisados.

O *framework* ATCM apresenta Fase de Teste de Integração e é totalmente extendido a utilização da Técnica de Teste Estrutural. Apresenta um metamodelo de teste independente de plataforma e metamodelos de teste específicos de plataformas xUnit, JUnit e NUnit. Utiliza a metamodelagem em EMF.

Também possibilita a geração semi-automaticamente das especificações de correspondências e geração automatizada das definições de transformações. O motor de transformação embora tenha sido feito com ATL, ATCM pode ser adaptável a qualquer motor de transformação.

5 ESTUDO DE CASO

Neste capítulo, apresenta-se um estudo de caso a fim de avaliar a funcionalidade do *framework* ATCM e os metamodelos de testes descritos no Capítulo 4. Para isto, desenvolve-se um modelo de negócio de uma aplicação de um caixa eletrônico.

A meta é avaliar a geração automatizada de casos de teste e testar o código-fonte de uma aplicação de simulação de um caixa eletrônico. Após os testes concluídos, os resultados do caso de teste são descritos e analisados.

5.1 Apresentação do Estudo de Caso

A fim de ilustrar as funcionalidades do *framework* ATCM, propõe-se um estudo de caso que consiste de desenvolvimento de *software* de um caixa eletrônico. Este *software* ATM (*Automated Teller Machine*) permite que os clientes realizem operações bancárias básicas tais como: retirada de dinheiro, transferência, depósito e consulta do saldo (BJORK, 2004).

A Figura 5.1 ilustra as operações básicas de uma aplicação que simula um caixa eletrônico em forma de diagrama de caso de uso em UML.

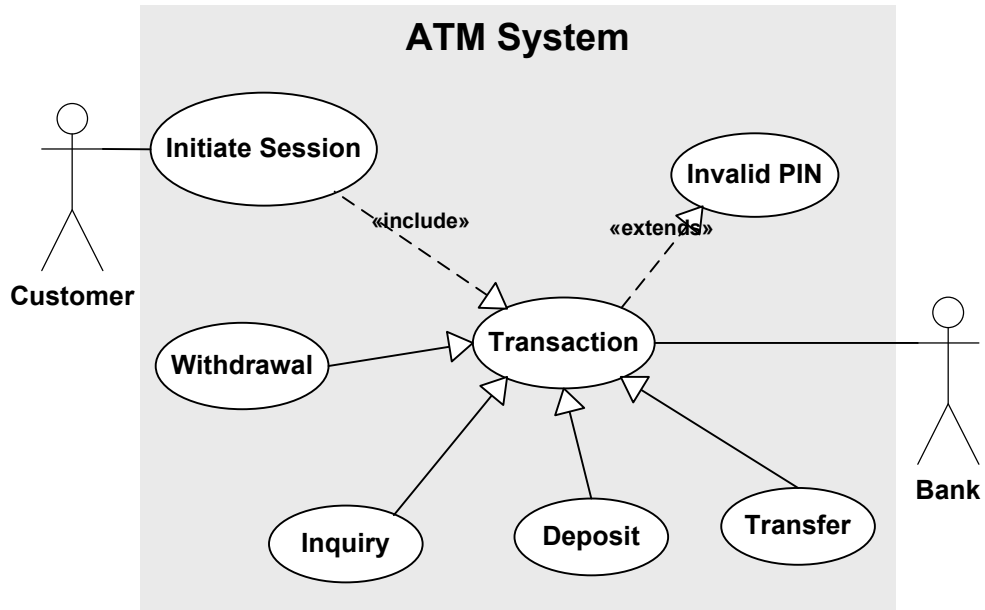


Figura 5.1 - Simulação de um Caixa Eletrônico: Diagrama de Caso de Uso

Um modelo independente de plataforma (PIM) de uma aplicação de um caixa eletrônico é desenvolvido conforme apresentado na Figura 5.2. Este PIM para ATM (*business model*) é apresentado em um diagrama de classe notação UML.

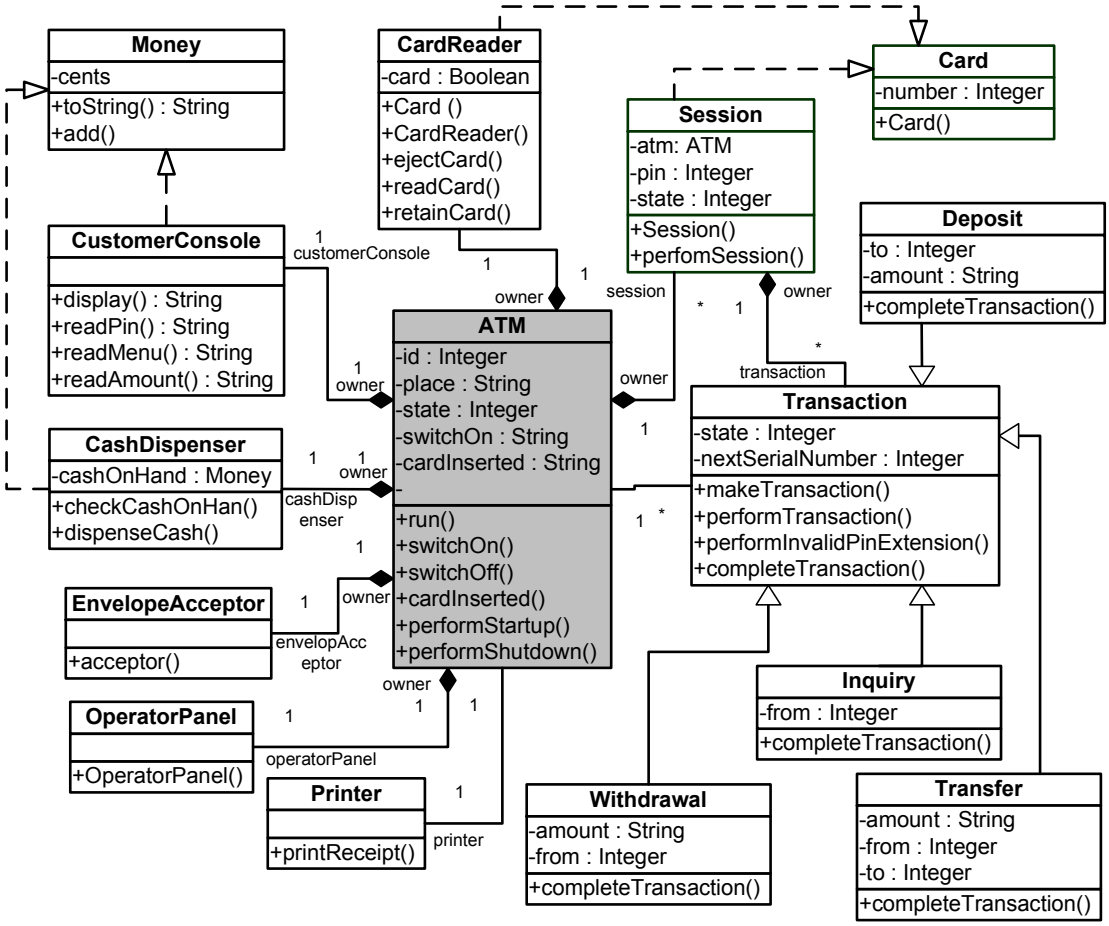


Figura 5.2 - PIM para ATM em notação UML: Diagrama de Classe

Este PIM para um sistema ATM consiste das seguintes classes:

- ATM - representa um terminal particular de um caixa eletrônico;
- Session - representa uma sessão particular, isto é, as operações que o usuário executa antes de ejetar o cartão;
- CustomerConsole - entradas de dados (*pin*, *card*, *read menu* e *amount*);
- Money - armazena a quantidade de dinheiro;
- CardReader - lê, valida e liberta cartão bancário;
- Card - armazena a lógica da numeração dos cartões bancários;
- CashDispenser - verifica a quantidade de dinheiro e libera caso tenha suficiente;
- EnvelopeAcceptor - gerencia a entrada de envelopes para a operação de depósito;
- Print - gerencia a impressão dos recibos de confirmação de depósito, transferência e extrato;

- Transaction - é uma classe base que instancia todos os tipos das transações;
  - Withdrawal - representa uma operação de retirada e é instanciada como o usuário opta para retirar uma quantidade;
  - Deposit - representa uma operação de depósito, onde o usuário irá especificar que tipo de depósito deseja realizar: em dinheiro ou em cheque;
  - Transfer - representa uma operação de transferência entre contas;
  - Inquiry - é uma operação de visualização de saldo da conta bancária.

Os casos de teste a serem construídos são conforme especificados nos diagramas de seqüência a seguir. A Figura 5.3 apresenta a seqüência de casos de teste para iniciar uma sessão de um sistema ATM (*Session Initiate*).

A Figura 5.3 demonstra a inicialização de uma sessão (Session) de um caixa-eletrônico. Através de um terminal ATM é inserido o cartão bancário, este é verificado pelo objeto da classe CardReader, após verificado-se se o cartão é valido ou não, então o usuário insere sua senha (pin) na classe CustomerConsole senão uma mensagem de cartão inválido é mostrada. Em seguida, se a senha estiver correta será iniciada uma sessão podendo realizar uma ou todas as transações bancárias (depósito, saque, consulta e transferência).

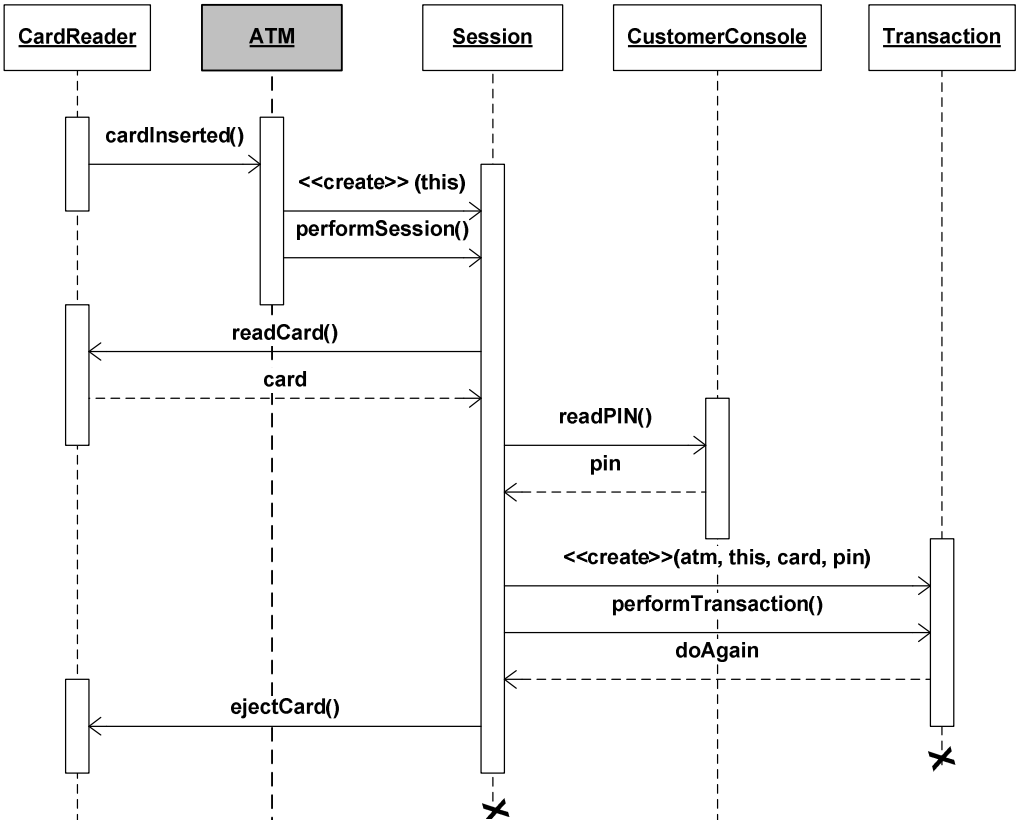


Figura 5.3 - Casos de teste para iniciar uma Sessão do ATM: Diagrama de Seqüência

Após a sessão do caixa-eletrônico ter sido iniciada, é possível realizar as transações disponíveis neste *software*. A Figura 5.4 monstra a transação consultar saldo (Inquiry) na qual se verifica a quantidade de saldo que o cliente tem disponível em sua conta e é possível imprimir este saldo através da classe Printer.

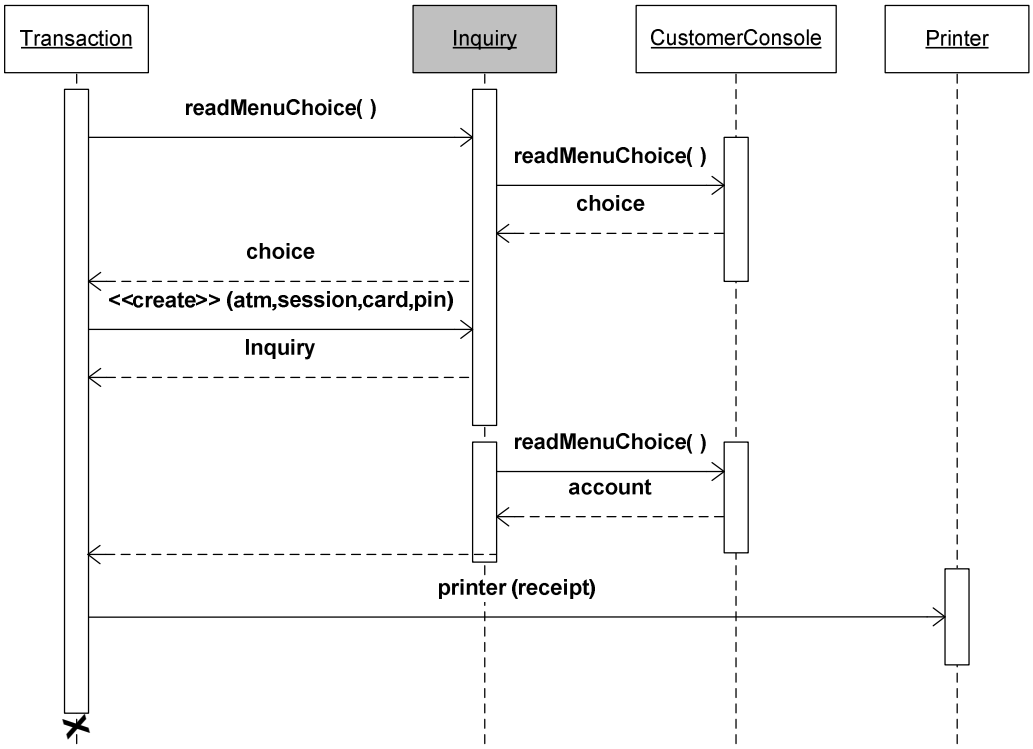


Figura 5.4 - Diagrama de Seqüência Transação Saldo

A Figura 5.5 apresenta a transação de transferência (Transfer), após escolhido a opção no menu através da classe CustomerConsole é informado qual conta será transferido um determinado valor e o valor que será transferido. É verificado se o cliente tem esse valor disponível em sua conta-bancária na classe CashDispenser. Após verificado o valor a ser transferido é informado na classe CustomerConsole para quem será transferido e manda para classe Printer para finalmente imprimir a comprovação da transferência.

A transação de retirada (Withdrawal) é apresentado na Figura 5.6, a conta-bancária que será efetuada a retirada é informado. Após que a conta do cliente é identificada, informa-se a quantia da retirada. Ambas as operações são analisadas através da classe CustomerConsole e através da classe CashDispenser. Esta última, verifica se a quantia requerida está disponível para saque da conta-bancária informada no início. Portanto, se esta quantia estiver disponível para saque, a classe CashDispenser libera a quantia requerida.

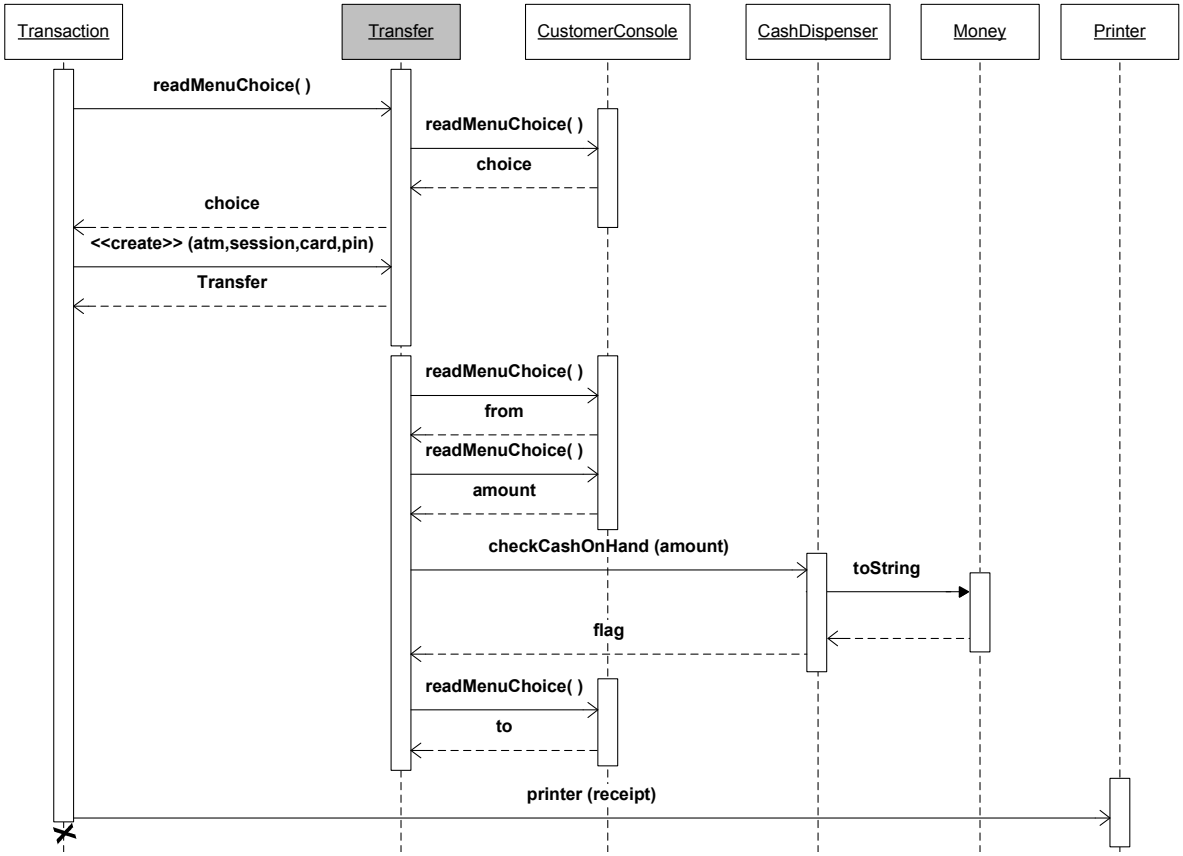


Figura 5.5 - Diagrama de Seqüência Transação Transferência

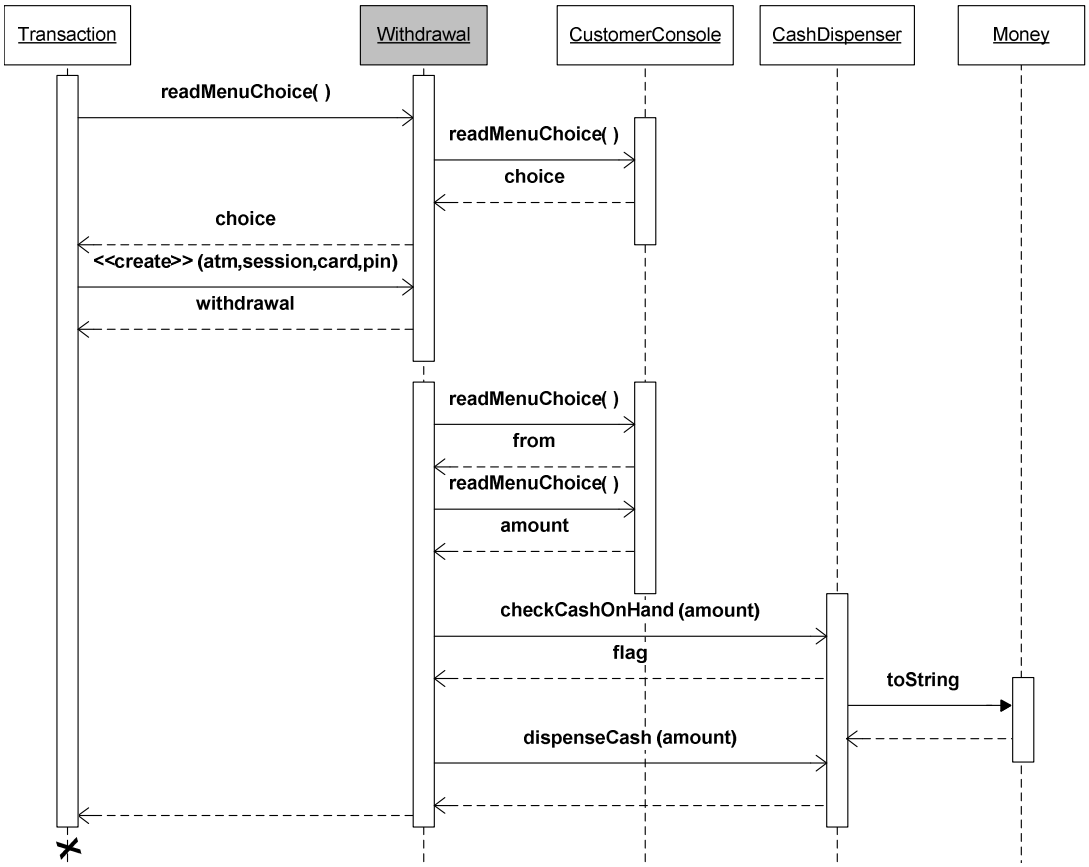


Figura 5.6 - Diagrama de Seqüência Transação Retirada

A Figura 5.7 apresenta a transação Depósito (Deposit) na qual o cliente informa para qual conta-bancária será feito o depósito e a quantidade a ser depositada. Após informado estes itens terem sido informados, será solicitado que o cliente insira o envelope no local indicado. A classe EnvelopeAccept analisa se o envelope é válido. Após que o envelope é verificdo e validado, um recibo da transação é impresso pela classe Printer.

Essas figuras representam o caso de teste do sistema ATM utilizado para a criação do Modelo de Teste descrito na subseção a seguir.

A Figura 5.4, Figura 5.5, Figura 5.6 e Figura 5.7 apresentam os diagramas de seqüência relativos aos casos de teste para Transações (*Transaction*) bancárias. Essas transações correspondem a Saldo (*Inquiry Transaction*), Transferência (*Transfer*), Retirada (*Withdrawal*) e Depósito (*Deposit*), respectivamente.

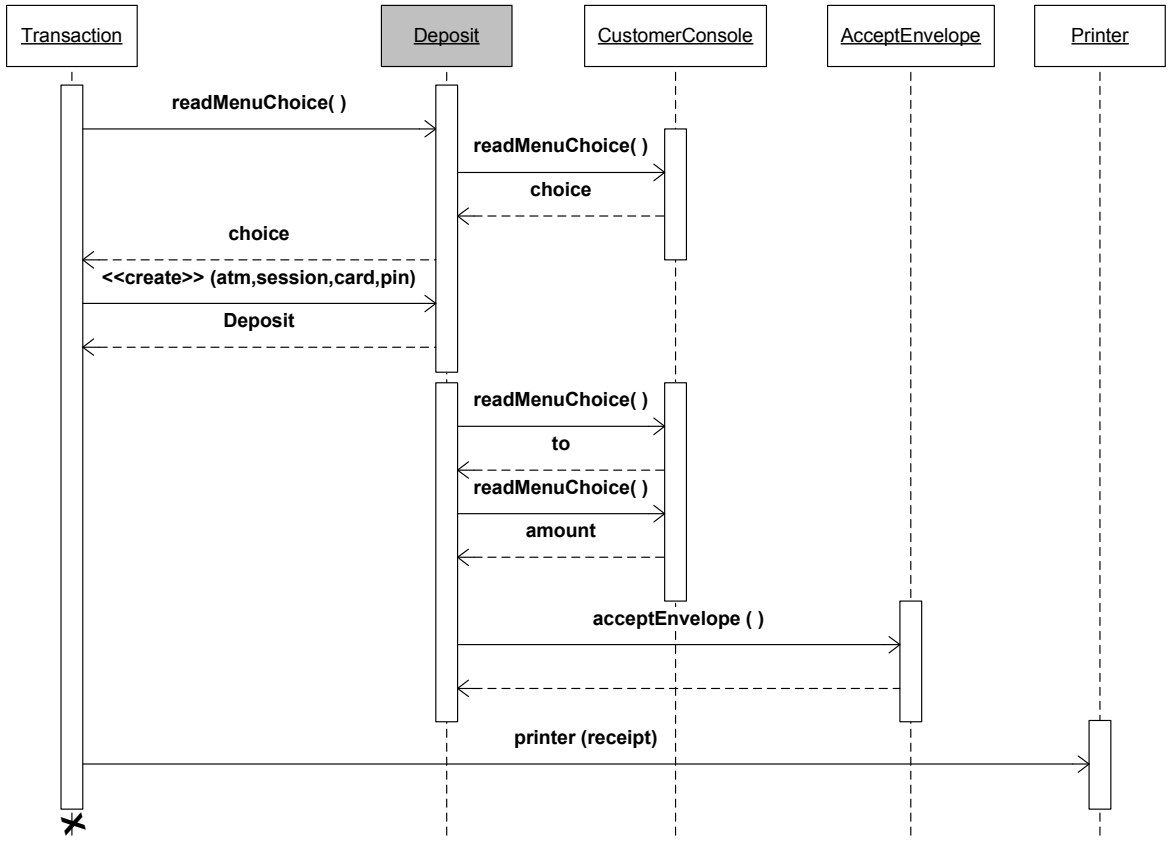


Figura 5.7 - Diagrama de Seqüência Transação Depósito

5.2 Modelagem do Modelo de Negócio: PIM

Um Modelo de Teste para testar o modelo de negócio para um *software* ATM deve ser desenvolvido conforme a Figura 4.1. O modelo de negócio contém a lógica do negócio, e o Modelo de Teste é específico de modelagem de testes. Contudo, ambos modelos são PIMs.

Este Modelo de Teste é conforme o MPIT proposto na subseção 4.3.1 a fim de testar o *software* ATM. De fato MPIT é uma linguagem específica de domínio (DSL, *Domain-Specific Language*) para o domínio de teste.

Para melhor visualização do Modelo de Teste, do sistema ATM este é apresentado conforme à notação UML e perfis extraídos do metamodelo MPIT. A Figura 5.8 apresenta um fragmento deste Modelo de Teste que representa a inicialização de uma sessão. Neste fragmento, tem-se o <<ModuleTest>> ATM e o <<TestSuite>> Session, este <<TestSuite>> contêm os <<TestCase>> CustomerConsule, CardReader e Transaction.

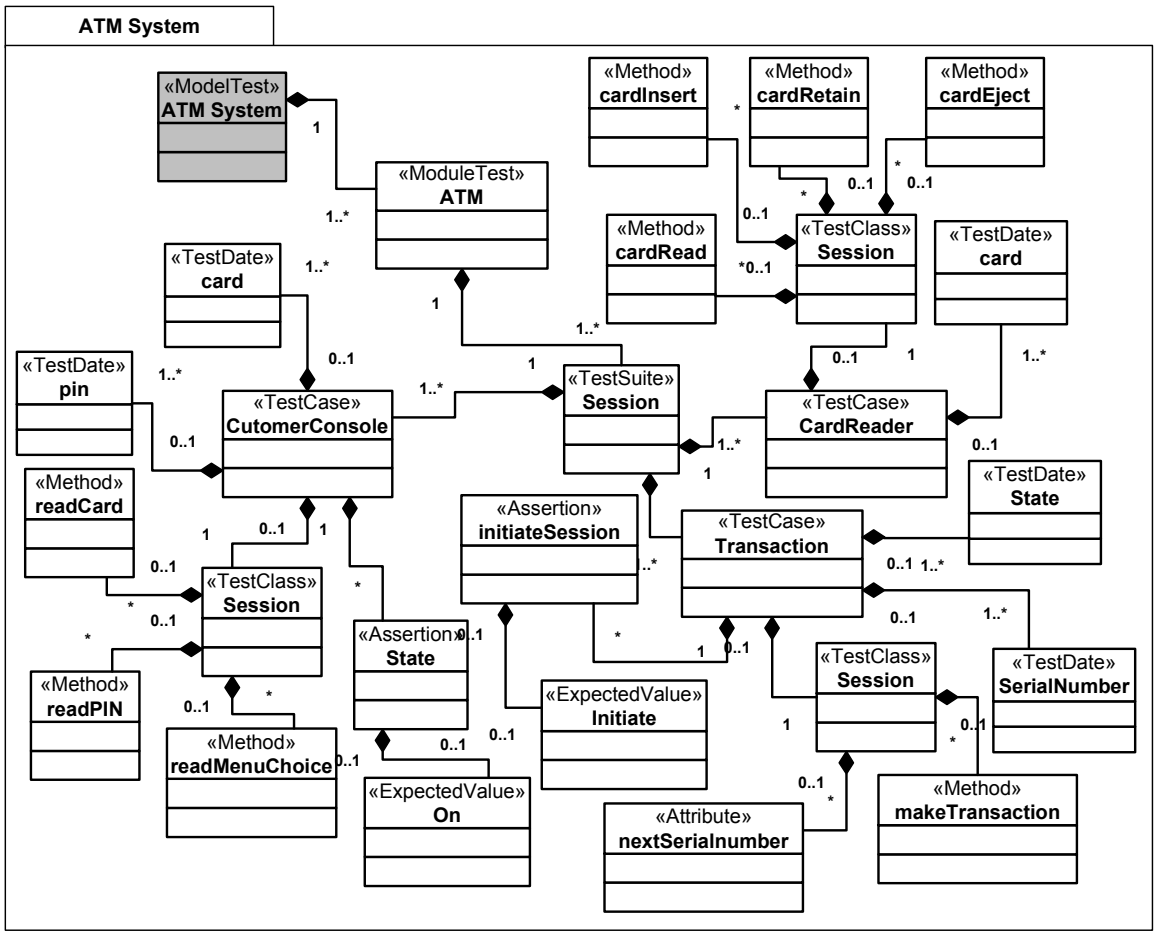


Figura 5.8 - Notação UML: ModuleTest ATM

A Figura 5.9, Figura 5.10, Figura 5.11 e Figura 5.12 ilustram fragmentos do <<ModuleTest>> Transaction para execução das transações disponíveis de um caixa-eletrônico, este módulo contém os <<TestSuite>> Inquiry, Deposit, Withdrawal e Transfer.

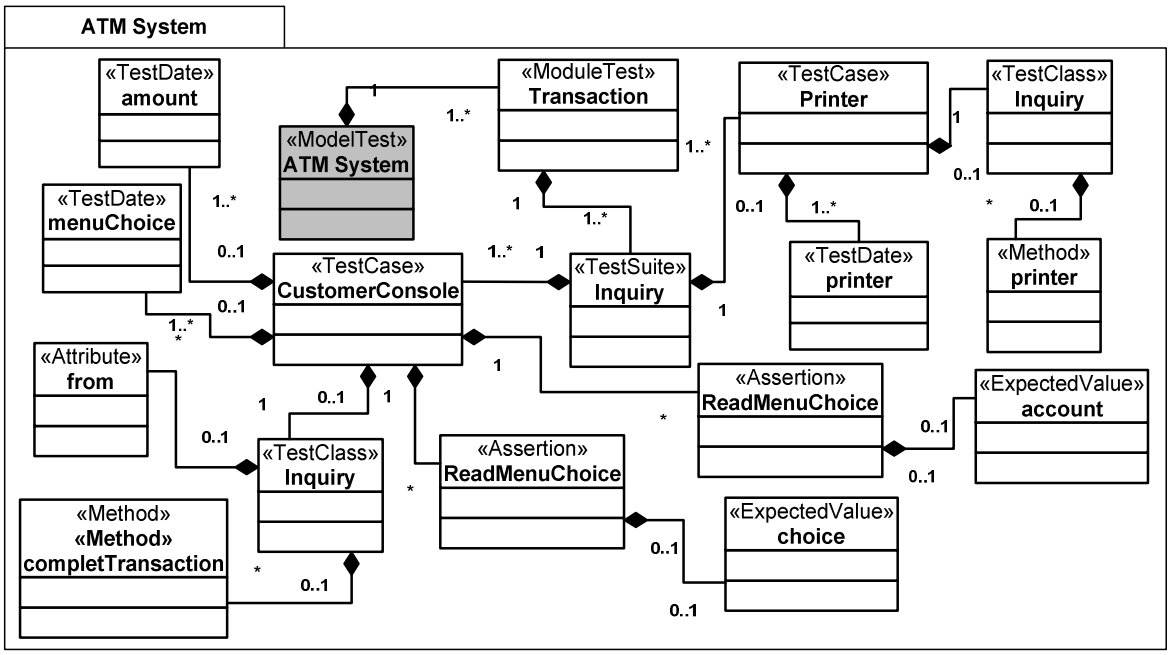


Figura 5.9 - Notação UML: TestSuite Inquiry

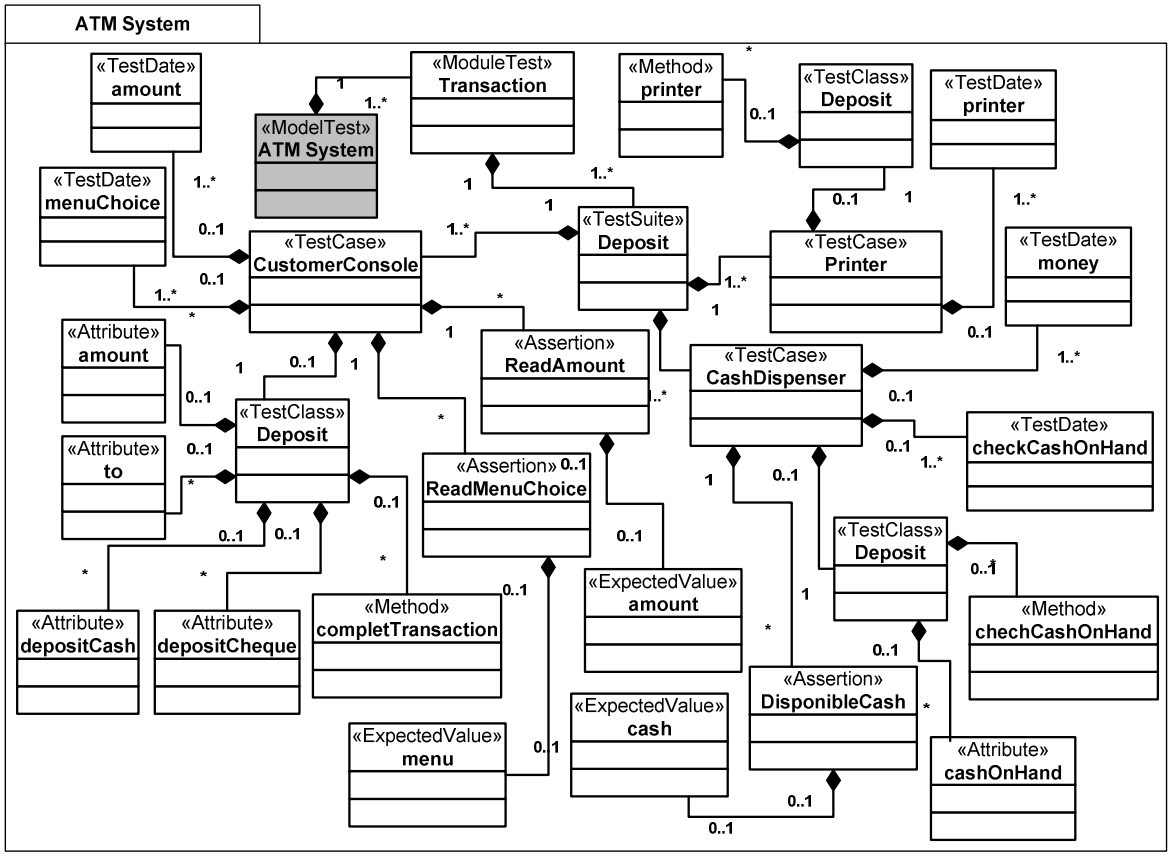


Figura 5.10 - Notação UML: TestSuite Deposit

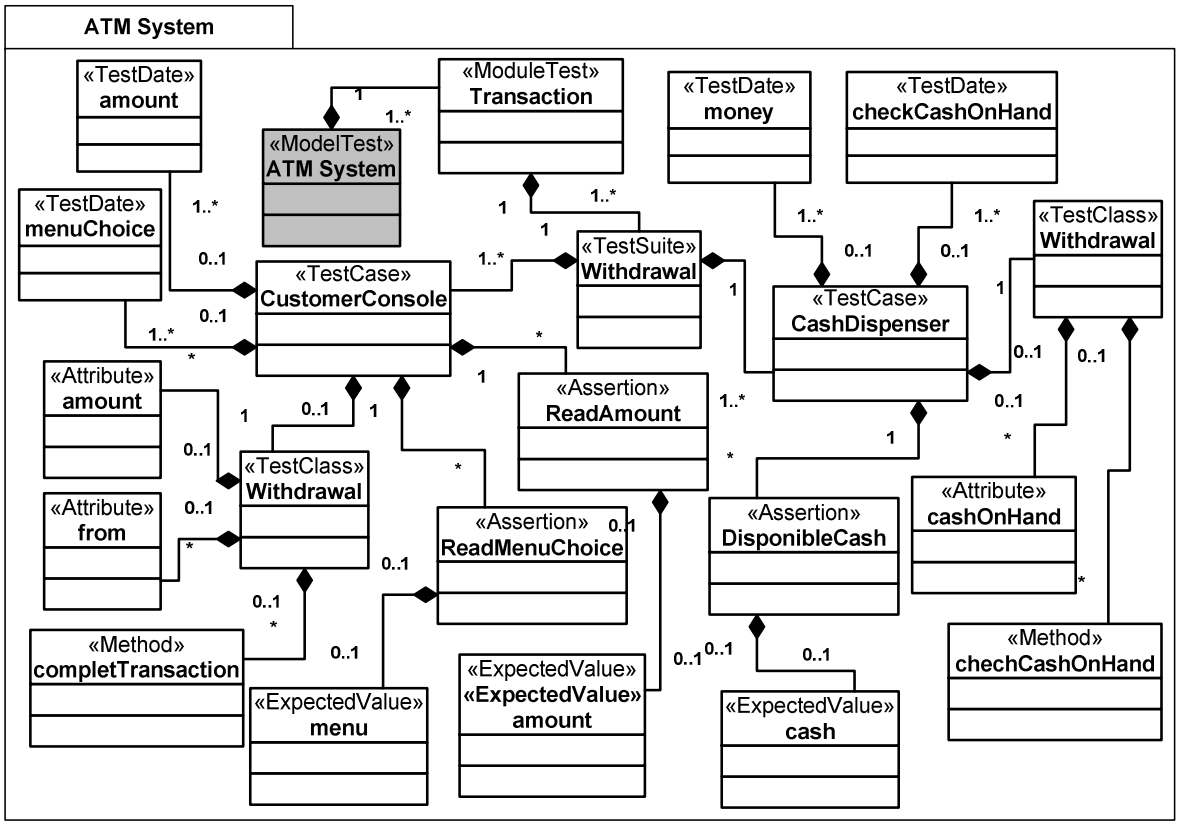


Figura 5.11- Notação UML: TestSuite Withdrawal

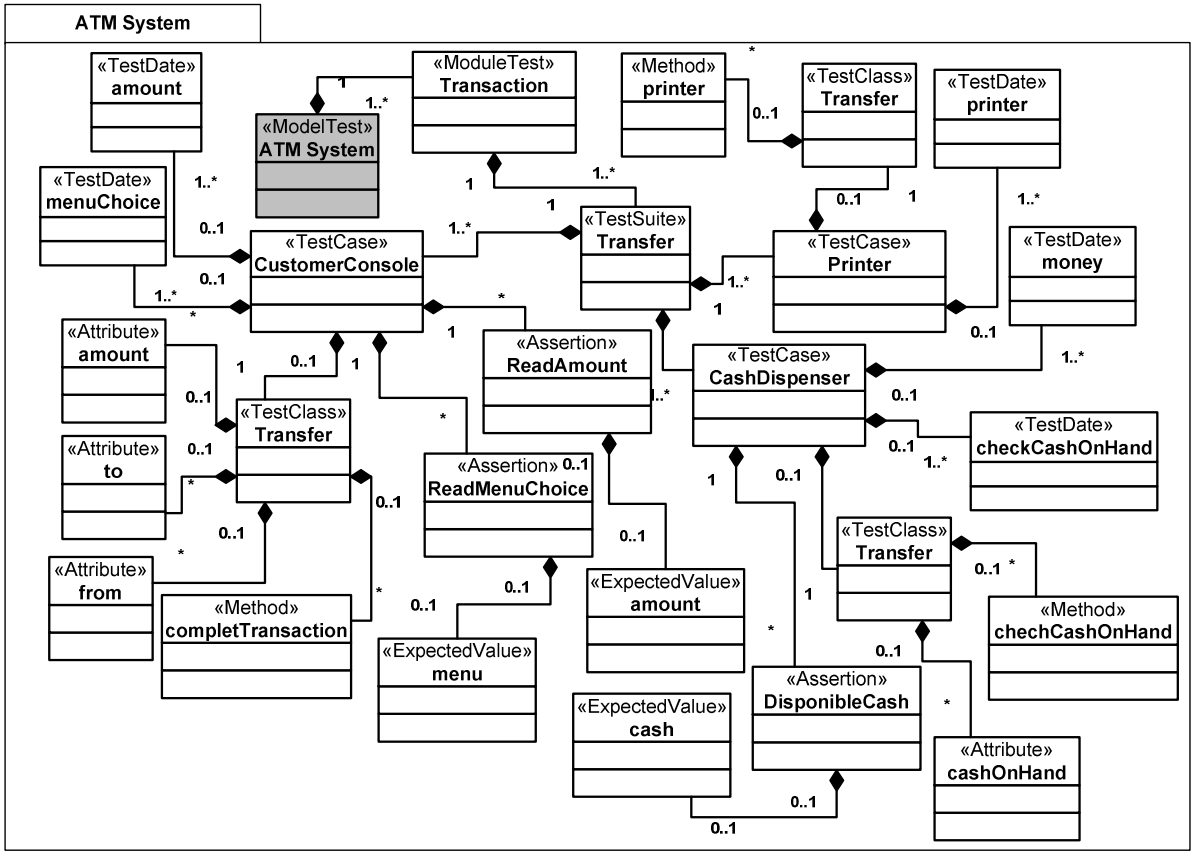


Figura 5.12 - Notação UML: TestSuite Transfer

Estas figuras são fragmentos do <<ModuleTest>> Transaction e possui <<TestSuite>> Inquiry (ver Figura 5.9), Deposit (ver Figura 5.10), Withdrawal (ver Figura 5.11) e Transfer (ver Figura 5.12).

A Figura 5.13 apresenta o fragmento do Modelo de Teste ATM da integração dos módulos, <<IntegrationTest>> Integration.

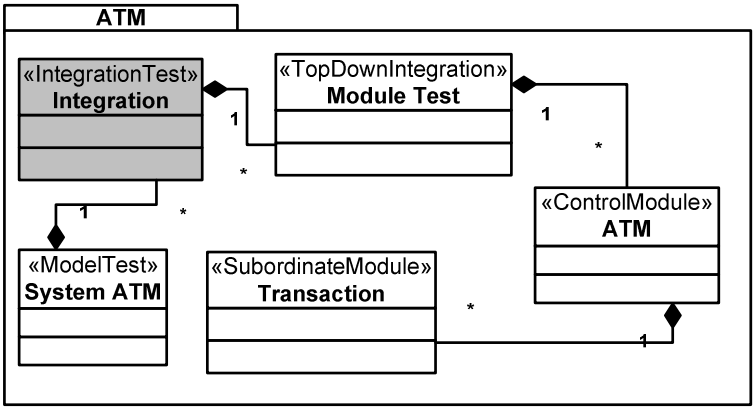


Figura 5.13 - Notação UML: IntegrationTest Integration

Portanto, este Modelo de Teste para o sistema ATM (System ATM) é composto por:

- <<ModuleTest>> ATM - este teste de módulo verifica e valida o cartão bancário e a senha (PIN) para inicia uma sessão do *software* ATM (ver Figura 5.8);
- <<ModuleTest>> Transaction - verifica todas as possíveis transações realizadas por um determinado cliente no *software* ATM, tais como, Saldo (Ver Figura 5.9), Depósito (Ver Figura 5.10), Saque (Ver Figura 5.11), e Transferência (ver Figura 5.12);
- <<IntegrationTest>> Integration - testa a integração dos módulos. O ModuleTest Transaction é um módulo subordinado ao módulo <<ModuleTest>> System ATM (ver Figura 5.13).

A Figura 5.14 apresenta o Modelo de Teste em forma de árvore criado usando o protótipo do T2MDT que implementa o metamodelo MPIT. Nesta figura é descrito todos os dados e requisitos a serem testados do modelo de teste para um sistema ATM são apresentados e descritos na subseção 5.1.

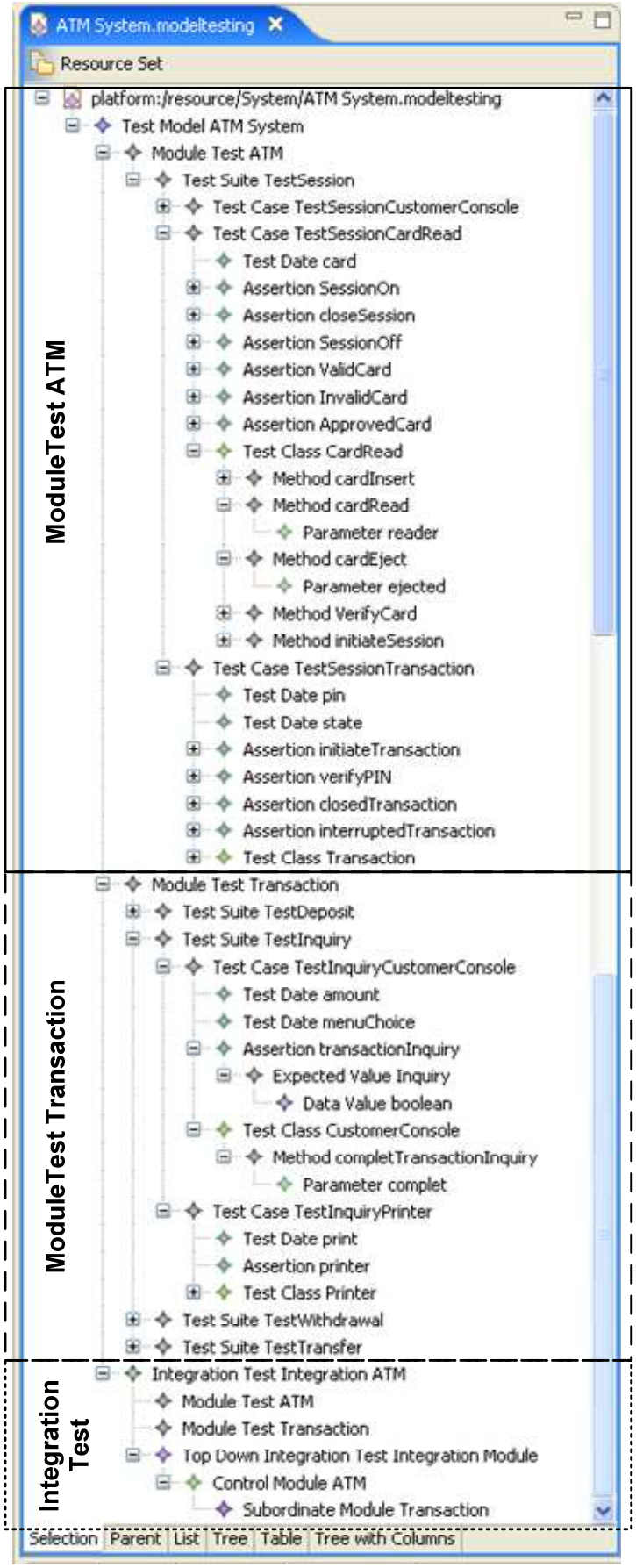


Figura 5.14 - PIM para teste conforme ao MPIT

5.3 Aplicando o *framework* ATCM

Após criado o modelo de negócios e o Modelo de Teste para um *software* ATM, o *plug-in* do *framework* ATCM (ver Figura 4.12) é utilizado para gerar os casos de teste para testar o código-fonte de um sistema ATM. Este Modelo de Teste consiste no PIM que é transformado em um Modelo de Teste Específico de Plataforma que consiste no PSM (Ver Figura 4.1).

O estudo de caso proposto neste capítulo tem a funcionalidade de testar o código-fonte do sistema ATM. A plataforma de testes unitários JUnit foi utilizada nestes testes. Portanto, conforme os passos mencionados na metodologia proposta na subseção 4.3 (Ver Figura 4.2) são gerados as especificações de correspondências e definições de transformações utilizando as ferramentas MT4MDE e SAMT4MDE.

A Figura 5.15 apresenta as especificações de correspondências, isto é, o modelo de correspondências (ao centro), o metamodelo MPIT (ao lado esquerdo) e o MPST-JUnit (ao lado direito). A janela "*Match*" ao centro da Figura 5.15 apresenta a semi-automatização das especificações de crorrespondências entres os metamodelos fonte e alvo, MPIT e MPST-JUnit, respectivamente.

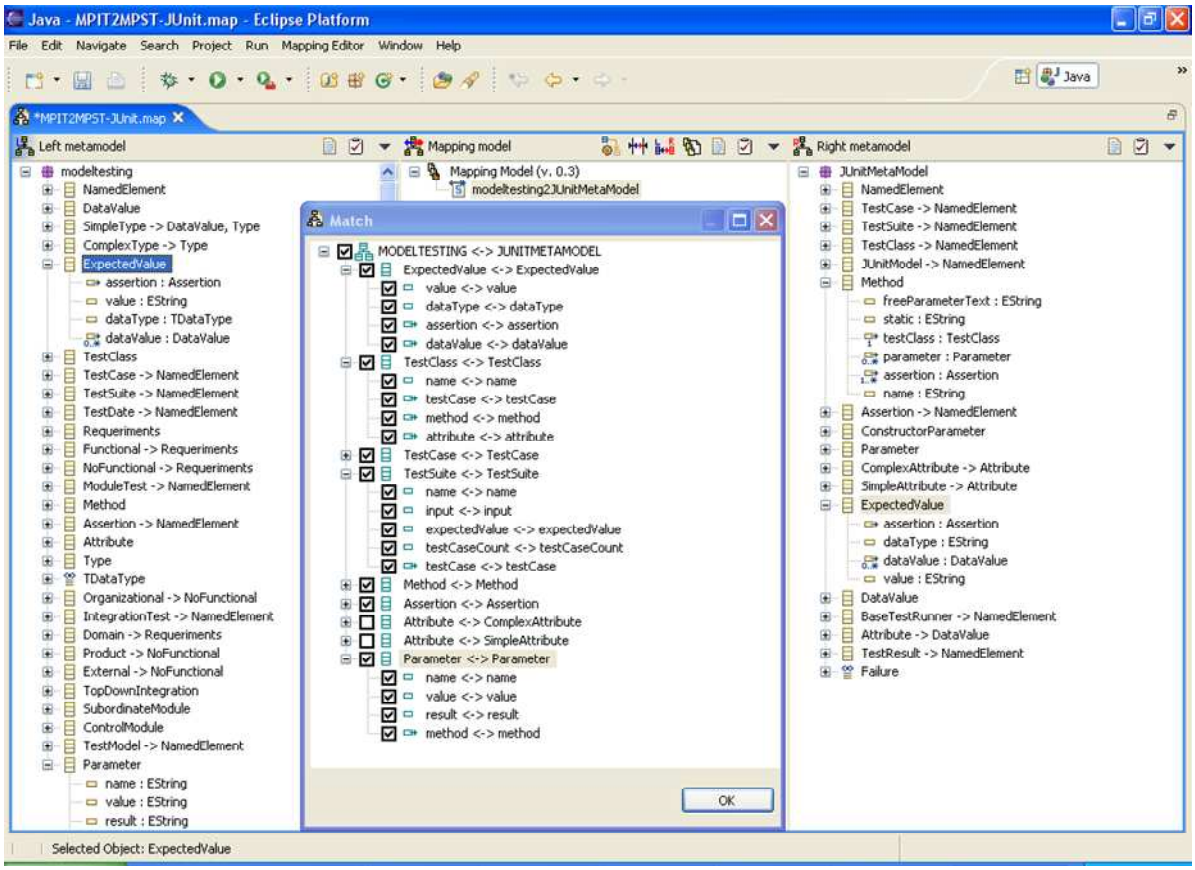


Figura 5.15 - Especificação de Correspondências entre MPIT e MPST-JUnit

Essas especificações de correspondências são geradas de forma semi-automatizada através das ferramentas MT4MDE e SAMT4MDE (Ver Figura 4.12). Estas ferramentas disponibilizam aos usuários a opção de escolher quais especificações são válidas ou se é necessário acrescentar mais especificações de correspondências.

Uma vez que as especificações de correspondências são geradas semi-automaticamente, as definições de transformações entre o MPIT e MPST-JUnit são geradas automaticamente.

Essa definição de transformações é criada usando a linguagem de transformação ATL (Ver Figura 4.12). A Figura 5.16 apresenta a geração automatizada das definições de transformações em ATL. As demais regras das definições de transformações (modelo-a-modelo) são apresentadas no Anexo E.

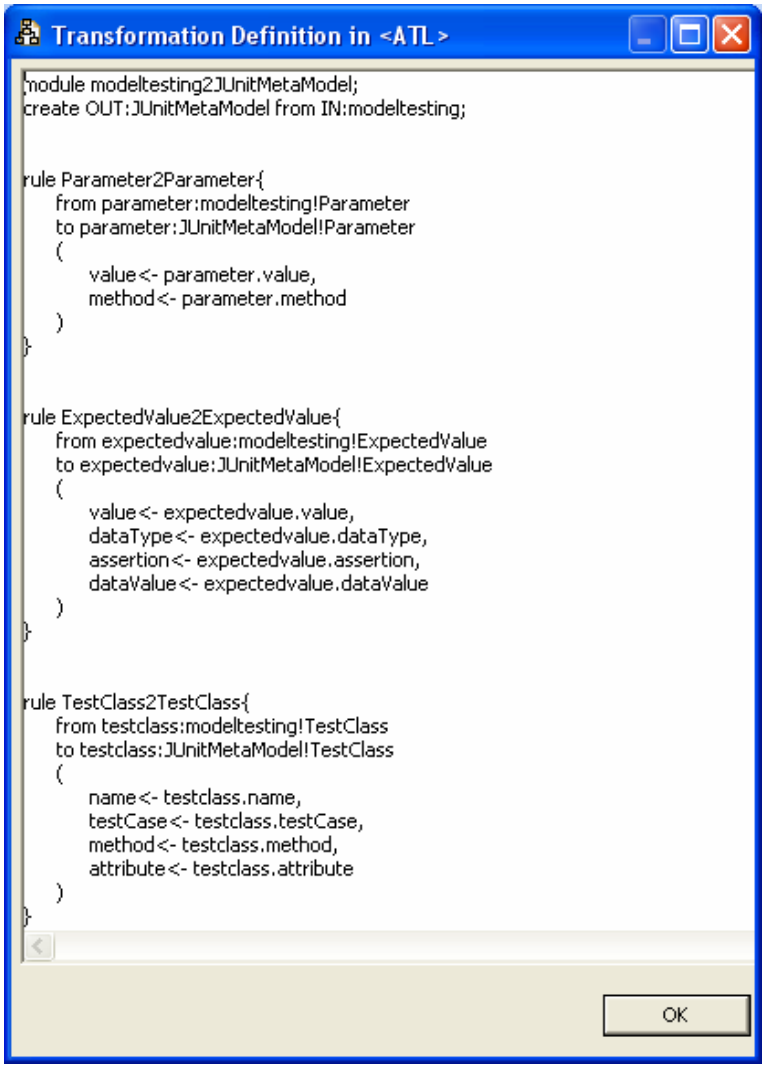


Figura 5.16 - Definições de Transformações de MPIT para MPST-JUnit

Assim, essas regras de definições de transformações são executadas por um motor de transformação para se obter o Modelo de Teste Específico de Plataforma, isto é, conforme o metamodelo MPST-JUnit (Ver Figura 4.1). Este Modelo de Teste Específico de Plataforma JUnit é apresentado na Listagem 5.1 no formato XMI. A listagem completa deste modelo no formato XMI é apresentado no Anexo F.

Listagem 5.1 - Fragmento do Modelo de Teste Específico de Plataforma JUnit

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <modeltesting:TestModel xmi:version="2.0"
   xmlns:xmi="http://www.omg.org/XMI"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:modeltesting="http://modeltesting" name="ATM System">
3    <moduleTest name="ATM">
4      <testSuite name="TestSession" input="" testCaseCount="">
5        <testCase name="TestSessionCustomerConsole">
6          <testDate name="pin" inputDates="123" dates="123"/>
7          <testDate name="card" inputDates="123456" dates="123456"/>
8          <assertion name="StateOn" assertionType="True"
   method="//@moduleTest.0/@testSuite.0/@testCase.0/@testClass/@method.0">
9            <expectedValue value="true" dataType="boolean">
10             <dataValue dataType="boolean" name="boolean"/>
11           </expectedValue>
12         </assertion>
13         <assertion name="StateOff" assertionType="False"
   method="//@moduleTest.0/@testSuite.0/@testCase.0/@testClass/@method.0">
14           <expectedValue value="false" dataType="boolean">
15             <dataValue dataType="boolean" name="boolean"/>
16           </expectedValue>
17         </assertion>
18         <assertion name="StateInterrupted" assertionType="False"
   method="//@moduleTest.0/@testSuite.0/@testCase.0/@testClass/@method.0">
19           <expectedValue value="false" dataType="boolean">
20             <dataValue dataType="boolean" name="boolean"/>
21           </expectedValue>
22         </assertion>
23         <testClass name="Session">
24           <method
   assertion="//@moduleTest.0/@testSuite.0/@testCase.0/@assertion.0
   //@moduleTest.0/@testSuite.0/@testCase.0/@assertion.1
   //@moduleTest.0/@testSuite.0/@testCase.0/@assertion.2"
   name="performSession">
25           <parameter name="Session" value="true" result="iniciate"/>
26         </method>
27         <attribute name="pin" value="123"/>
28         <attribute name="state" value="true"/>
29       </testClass>
```

```
30        </testCase>
31  ***
32    <integrationTest name="Integration ATM">
33      <moduleTest name="ATM"/>
34      <moduleTest name="Transaction"/>
35      <topDown name="Test Integration Module">
36        <controlModule name="ATM">
37          <subordinateModule depthFirst="true" name="Transaction"/>
38        </controlModule>
39      </topDown>
40    </integrationTest>
41  </modeltesting:TestModel>
```

Assim, após que o MODELO DE TESTE ESPECÍFICO DE PLATAFORMA é gerado, este é executado em um motor de transformação de modelos ATL modelo-a-texto para enfim gerar o código-fonte de teste baseado no *framework* de teste JUnit. A Listagem 5.2 apresenta um trecho das definições de transformações de modelo-a-texto para gerar o código-fonte de teste. A listagem completa destas definições está no Anexo G.

Listagem 5.2 - Fragmento das regras de transformação modelo-a-texto JUnit

```
1   library modelJUnit2SourceCode; -- Library Template
    ***
2   helper context JUnitMetaModel!TestCase  def: toString() : String =
3         'public class ' +
4          self.name +
5         ' extends junit.framework.TestCase{\n\n' +
6        self.assertion->iterate(i; acc:String=''| acc +
7        i.toString()+'\n') +
8         '\n}\n';
9
10  helper context JUnitMetaModel!TestSuite  def: toString() : String =
11        'public class ' +
12         self.name + ' ' +
13         'extende' + ' ' +
14         'Test Case' + ' ' +
15        '{\n\n'+
16         '\tTestSuite suite = new TestSuite("'+ self.name +'");\n'+
17          self.testCase->iterate(i; acc:String=''| acc +
18          '\tsuite.addTestCase('+ i.name+'.class' + ');\n') +
19         '\treturn suite;\n' +
20         '\n}\n\n';
```

Após executadas as definições de transformações (modelo-a-texto) em um motor de transformação ATL, obtêm-se o código-fonte de teste para o *framework* JUnit. A Listagem

5.3 apresenta o fragmento do código-fonte de teste para JUnit para o TestSuite TestSession, como apresentado no estudo de caso na subseção 5.2. No TestSuite contém todos os TestCase para enfim fazer o teste de integração dos mesmos. A listagem completa do código-fonte de teste está no Anexo H.

Listagem 5.3 - Fragmento do código-fonte de teste gerado para TestSuite

```
1  public class TestSession extende Test Case
2  {
3       TestSuite suite = new TestSuite("TestSession");
4          suite.addTestCase(TestSessionCustomerConsole.class);
5          suite.addTestCase(TestSessionCardRead.class);
6          suite.addTestCase(TestSessionTransaction.class);
7          return suite;
8  }
```

A Listagem 5.4 é o fragmento do código-fonte de teste do Teste *Case*, este faz parte do conjunto de Teste Suite apresentado na Listagem 5.3 apresentado na linha 5.

Listagem 5.4 - Fragmento do código-fonte de teste gerado para TestCase

```
1  public class TestSessionCardReader extends
   junit.framework.TestCase
2   {
3         public void testCard()
4     {
5           assertEquals(1, card.getNumber());
6     }
7         public void testPin()
8         {
9            assertEquals(42, msg.getPIN());
10        }
11  }
```

A geração dos testes para o *framework* NUnit está em anexo. O Anexo I apresenta o mapeamento entre o MPIT e o MPST-NUnit para geração semi-automática das especificações de correspondências a fim de gerar automaticamente as definições de transformação (modelo-a-modelo) apresentadas no Anexo J.

A partir desta definição de transformação é gerado o modelo específico de teste para NUnit apresentado no Anexo L. O Anexo M apresenta as regras de transformação

(modelo-a-texto) usadas para enfim gerar o código-fonte de teste para NUnit apresentado no Anexo N.

Após gerado os casos de testes, executamos este caso de teste no *framework* JUnit do Eclipse, como demostrado na Figura 5.17.

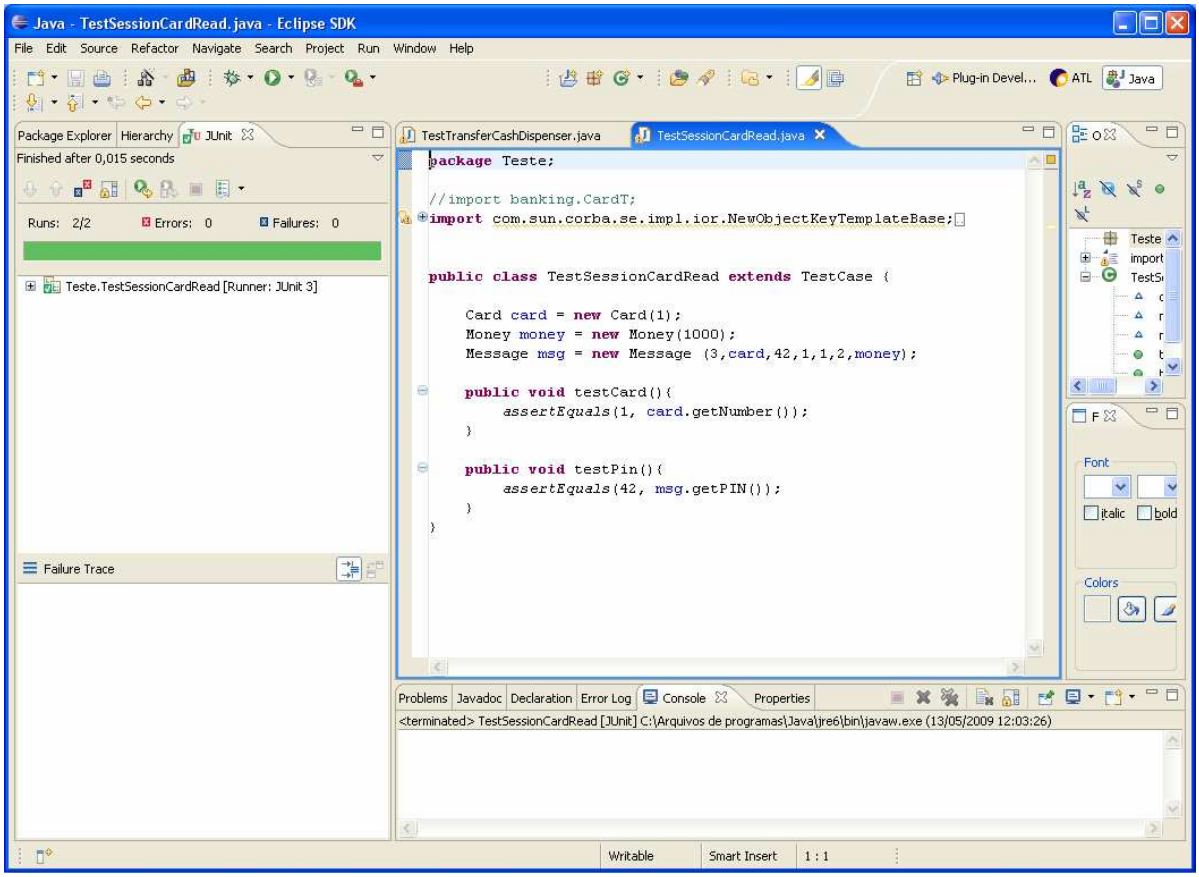


Figura 5.17 - Executando Caso de Teste gerado pela *framework* ATCM no JUnit Eclipse

5.4 Avaliação dos Resultados dos Testes

Durante o desenvolvimento dos testes utilizando o protótipo T2MDT, MT4MDE e SATM4MDE, demonstrou a geração automazada dos casos de teste do *framework* ATCM proposto nesta dissertação e a veracidade dos metamodelos (MPIT e MPST) e metodologia propostos.

Neste estudo de caso, supõem-se que com a utilização do *framework* ATCM para a geração do código-fonte de teste, obtêm-se uma redução significativa nos esforços dos desenvolvedores para a geração dos casos de teste de um determinado sistema a ser avaliado (neste estudo de caso se aplicou para um sistema de caixa-eletrônico - ATM), testando seus módulos ou unidades separadamente para enfim testar a integração desses módulos utilizando

a técnica de teste funcional, ou seja, informando a entrada dos dados, executando-as e comparando seu resultado com o valor esperado.

Observou-se neste estudo de caso uma possivel redução de possíveis injeções de erros durante o mapeamento entre os metamodelos fonte e alvo (MPIT e MPST respectivamente), pois o *framework* através dos protótipos MT4MDE e SAMT4MDE, possibilitou a semi-automatização deste processo e a geração automatizada das definições de transformação (modelo-a-modelo) a fim de gerar o modelo de teste conforme o PIM de um determinado sistema a ser testado (ver Figura 5.2).

Portando, supõe-se que terá uma redução significativa do tempo para a criação dos casos de teste, provendo qualidade e confiança aos casos de teste, possibilitando uma maior redução dos custos no desenvolvimento e manutenção do *software* testado.

5.5 Conclusão

No presente capítulo, apresentou-se um estudo de caso para comprovar a veracidade do *framework* ATCM e de sua implementação através das ferramentas T2MDT, MT4MDE e SAMT4MDE, metamodelos de teste e metodologia proposto nesta dissertação. Neste estudo de caso mostrou-se o desenvolvimento de um sistema ATM (caixa-eletrônico) que provê as funções de iniciar uma sessão de um caixa-eletrônico e transações bancárias tais como, retirada, depósito, saldo e transferência. Apresentou-se todas as etapas para a geração do código-fonte para o *framework* de testes JUnit.

6 CONCLUSÕES E SUGESTÕES PARA TRABALHOS FUTUROS

Neste capítulo, discute-se sobre as contribuições deste trabalho, resultados alcançados, as limitações do trabalho e sugestões para trabalhos futuros.

6.1 Conclusões do Trabalho

Este trabalho apresentou a proposta de um *framework* de testes dirigido a modelos. Como complemento deste *framenwork* chamado de ATCM, um metamodelo de teste independente de plataforma nomeado de MPIT e vários metamodelos de testes específicos de plataforma, tais como, MPST-JUnit, MPST-NUnit e MPST-xUnit foram propostos. Uma metodologia foi proposta para uma melhor compreensão das etapas do processo existente do *framework* ATCM. Um protótipo foi construído conforme o *framework* proposto, a fim de prover criação e edição dos modelos de testes.

Este protótipo contém ferramentas que auxiliam a construção semi-automática das especificações de correspondências entre os metamodelos alvo e fonte e a geração automática das definições de transformação com a finalidade de criar o modelo específico de plataforma. Uma vez que as deifnições de transformações estejam geradas, estas podem ser utilizadas para gerar o modelo específico de plataforma de teste. Este último modelo é utilizado para, enfim, gerar o código-fonte de teste.

O protótipo contém 4 (quatro) ferramentas, isto é, T2MDT, MT4MDE, SAMT4MDE e Motor de Transformação. A ferramenta T2MDT auxilia na geração de casos de teste e código-fonte de testes. A ferramenta MT4MDE permite a criação e edição manual de especificações de correspondências e geração automática de definição de transformação em uma determinada linguagem, como ATL. A ferramenta SAMT4MDE tem como base a implementação do algoritmo de busca por similaridades estruturais e em *cross-relationship* (SOUZA Jr et al., 2009), permite a criação de especificação de correspondências de forma semi-automatica. E o motor de transformação executa as definições de transformação geradas anteriormente.

Este *framework* ATCM junto com o protótipo visa gerar o código-fonte de teste para testar uma aplicação que seja gerada por uma Abordagem Dirigida por Modelos.

6.2 Resultados Alcançados

Com a utilização conjunta do *framwork* ATCM, seus respectivos metamodelos de testes e o protótipo implementado com a junção das ferramentas T2MDT, MT4MDE, SAMT4MDE e Motor de Transformação, supõem-se que os seguintes resultados foram alcançados:

- *Error-prone factor*: como o processo é automático, a intervenção humana é mínima e fator erros tais como, fadiga e negligência é evitado;
- Minimizar a injeção de erros no código fonte: como as definições de transformação são geradas automaticamente a partir das especificações de correspondências, sendo que o último é gerado de forma semi-automática pelo algoritmo *macthing*, possibilitando a minimização da injeção de erros. Vale ressaltar que a qualidade das definições de transformação é diretamente dependente da precisão do algoritmo de *macthing* para encontrar correspondências que são verdadeiros positivos e para eliminar correspondências que são falsos negativos;
- Reduzir o tempo para testes e, conseqüentemente, reduzir o tempo para entregar o sistema funcional: dado o modelo de negócio e as plataformas finais de testes, o *framework* proposto permite criar o modelo de teste e gerar as especificações de correspondências entre os elementos MPIT e MPST, gerar as definições de transformação e aplicar esta definição de transformação para gerar o código-fonte de teste. Neste processo, a intervenção humana aparece apenas na criação do modelo de teste que se refere ao modelo de negócio. As demais atividades são feitas pelo protótipo proposto;
- Aprimorar a qualidade de *software*: como o código-fonte de teste é gerado automaticamente, então o teste pode ser mais amplo e abranger mais possibilidades de casos de teste sem representar um custo adicional muito significativo. Assim, quanto mais o *software* é testado menos erros conterá o *software* final e sua qualidade é assegurada;
- Diminui a dependência de intervenções humanas no código-fonte: como os testes podem abranger mais possibilidades e casos de teste, então a qualidade de *software* é assegurada, e a intervenção humana é menos necessário para descobrir *bugs* e eliminar defeitos. Após que o sistema de *software* esteja em produção

6.3 Limitações

Apesar da obtenção de pontos positivos neste trabalho, alguns pontos ainda apresentam limitações, tais como:

- Ausência de uma ferramenta que auxilie na criação de uma notação gráfica mais amigável do que apresentada pelo *framework* ATCM em forma de árvore para criar o modelo de teste independente de plataforma conforme ao MPIT;
- Falta da utilização da Técnica de Teste Estrutural que teste a lógica interna do código-fonte de um software desenvolvido por abordagens MDx;
- O *framework* ATCM aceita somente diagramas de classes para modelo de negócio, não aceitando diagramas como de sequência e atividades;
- Criação das definições de transformação modelo-à-texto automaticamente, a construção dessas transformações ainda são realizadas de forma manual;
- As definições de tranformação são somente executadas em um motor de transformação ATL, falta a integração com outros motores de transformação tais como, Epsilon (KOLOVOS et al., 2008), MOFScript (OLDEVIK, 2006), Tefkat (STEEL, 2004), MOFQVT (OMG, 2008);
- Falta de semântica no Metamodelo de *Matching* para a geração das especificações de correspondências;
- Não foi verificada a aplicabilidade das ferramentas com modelos de negócios grandes e mais complexos;
- A criação de especificações de correspondências é limitada quanto a criação de expressões OCL mais complexas, isto é, que incluam a composição de operações tipo s*elect* e *collect*;
- Um teste experimental com usuários não foi feito. Por exemplo, uma equipe poderia realizar testes experimentais, ou seja, um grupo faria a contrução manual de casos de testes e aplica-los em um estudo de caso e outra equipe faria a contrução automatizada de casos de teste com o *framework* ATCM e também aplica-los em um mesmo estudo de caso usado pela outra equipe. Estes testes são baseados na Engenharia Experimental;

6.4 Trabalhos Futuros

Como propostas para trabalhos futuros, podem-se citar:

- Testes Experimentais com o *framework* ATCM, ou seja, uma equipe de testes seria respondável em criar casos de teste manuais e testar o código-fonte a ser testado e outra equipe de teste seria responsável em criar casos de teste utilizando o *framework* ATCM;
- A automatização das definições de transformações de modelo-a-texto, diminuindo ainda mais a intervenção humana para a criação do mesmo, tornando essas definições menos propensas a erros;
- A criação de novos metamodelos de testes específicos de plataforma para outras plataformas de testes tais como, CppUnit, XMLUnit, e SUnit;
- Adaptação do MPIT para Técnica de Teste Estrutural para o *framework* ATCM realize testes da lógica computacional do sistema em teste;
- Adaptação do MPIT para testar os metamodelos e modelos criados para uma transformação de modelos de qualquer aplicação;
- Possibilidade de o framework ATCM aceitar como modelos de negócios diagramas de sequência e diagramas de atividades;
- Adaptação do metamodelo de *matching* para utilizadar semântica para a criação das especificações de correspondências;
- Integração de outras linguagens e motores de transformação no *framework* ATCM, tais como Epsilon (KOLOVOS et al., 2008), MOFScript (OLDEVIK, 2006), Tefkat (STEEL, 2004), MOFQVT (OMG, 2008).

REFERÊNCIAS

ALVES, Everton L. G., MACHADO, Patrícia D. L., RAMALHO, Franklin. **Uma Abordagem Integrada para Desenvolvimento e Teste Dirigido por Modelos**. 2nd Brazilian Workshop on Systematic and Automated *Software* Testing, 2008.

AMBLER, Scott. **Agile Database Techniques Effective Strategies for the Agile Software Developer.** Wiley Application Development, 2003.

ARLOW, Jim, NEUSTADT, Ila. **Enterprise Patterns and MDA: Building Better Software with Archetype Patterns and UML**. Editora Addison Wesley, 2003.

ATL - ATLAS group LINA & INRIA Nantes. **ATL - ATLAS Transformation Language**. ATL User Manual 0.7, 2006.

BECK, Kent. **Extreme Programming Explained: Embrace Change**. Editora Addison Wesley, 2003.

BECK, Kent. **Test Driven Development: By Example**. Editora Addison-Wesley Professional; US ed edition, 1999.

BIASI, Luciano Bathaglini. **Geração Automatizada de Drivers e Stubs de Teste para JUnit a partir de Especificações U2TP**. 2006, 153 f. Dissertação (Programa de Pós-Graduação em Ciência da Computação) Pontifícia Universidade Católica do Rio Grande do Sul - Faculdade de Informática, Rio Grande do Sul, 2006.

BJORK, Russell C. **ATM System**. 2004. Disponível em: http://www.math-cs.gordon.edu/courses/cs211/ATMExample. Acessado em 04 de julho de 2008.

BUDINSKY, Frank; STEINBERG David, MERKS Ed, ELLERSICK Raymond, GROSE Timothy J., **Eclipse Modeling Framework: A Developer's Guide**, Addison-Wesley Pub Co, 2003.

CANCILA, Daniela E PASSERONE, Roberto, **Functional and Structural Properties in the Model-Driven Engineering Approach**, IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2008), p 809-816, 2008.

DAI, Z. R. **Model-Driven Testing with UML 2.0**. In: Proceedings of the Second European Workshop on Model Driven Architecture (MDA) with an emphasis on Methodologies and Transformations (EWMDA-2), Canterbury, England, 7-8 set. 2004.

DELAMARO, Márcio Eduardo; MALDONADO, José Carlos; JINO, Mário. **Introdução ao Teste de Software**. Editora Campus, 2007.

DIAS Neto, Arilo Cláudio. **Introdução a Teste de Software**. Engenharia de Software Magazine nº 1, 2008.

DUEÑAS, Juan C., MELLADO, Julio, CERÓN, Rodrigo, Arciniegas, José L., RUIZ, José L., CAPILLA, Rafael. **Model driven testing in Product Family Context.** First European Workshop on Model Driven Architecture with Emphasis on Industrial Application. 2004.

FAVRE, Jean-marie, **Towards a Basic Theory to Model Model Driven Engineering**, In Workshop on Software Model Engineering (WISME 2004), 2004.

GAMMA, Erich, HELM, Richard, JOHNSON, Ralph e VLISSIDES, John. **Design Patterns: Elements of Reusable Object-Oriented Software**. Editora Addison-Wesley Professional, 1995.

GRENNING, James. **Applying Test Driven Development to Embedded Software**. IEEE Instrumentation & Measurement Magazine, 2007.

HADJ KACEM Yessine, MAHFOUDHI Adel, KARAMTI Walid e ABID Mohamed. **A Model Driven Engineering Based Method For Scheduling Analysis**. 3rd International Design and Test Workshop, p. 326-330, 2008.

HAMILL, Paul. **Unit Test Frameworks**. O'Reilly, 2004.

HECKEL, R.; LOHMANN, M. **Towards Model-Driven Testing**. In: International Workshop on Test and Analysis of Component-Based Systems (Satellite Event of ETAPS 2003), Electronic Notes in Theoretical Computer Science, p.33-43, 2003.

IBM, **Overview Model-driven Testing Tools, Haifa Research Laboratory**, 2003. Available in < http://www.haifa.ibm.com/projects/verification/mdt/public.html> Accessed on 10/03/08.

IEEE - Institute of Electrical and Electronics Engineers. **IEEE Standard Glossary of Software Engineering Terminology**, ANSI/IEEE Std. 610.12-1990, 1990.

IEEE 829. **Standard for Software Test Documentation - Description**, ANSI/IEEE 829-1983, 1998.

JAVED A. Z., STROOPER, P. A., WATSON, G. N. **Automated Generation of Test Cases Using Model-Driven Architecture**, 2007. Proceedings of the Second International Workshop on Automation of Software Test, p. 3-3, 2007.

JEFFRIES, Ron. **Extreme Programming Adventures in C#**. Editora Microsoft Press, 2004.

KENT, Stuart. **Model Driven Engineering, Integrated Formal Methods.** Integrated Formal Methods - IFM, pages 286–298, May 2002.

KLEPPE, Anneke; WARMER, Jos; BAST. Wim. **MDA Explained: The Model Driven Architecture: Practice and Promise**, Editora Addison- Wesley, 1º Edição, 2003.

KOLOVOS, Dimitrios S., PAIGE, Richard F. and POLACK, Fiona A. C., **The Epsilon Transformation Language, In Theory and Practice of Model Transformations**, Publisher Springer, p. 46-60, 2008.

LEAL, Ricardo A. B. M. **Teste Funcional Baseado em Modelos Gramaticais**. 2008, 132f. Dissertação (Programa de Pós-Graduação em Informática) Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janiero, 2008

LI, Nuo, MA, Qin-qin, WU, Ji, JIN, Mao-zhong e LIU, Chao. **A Framework of Model-Driven Web Application Testing**. 30th Annual International Computer *Software* and Applications Conference, COMPSAC '06, p. 157-162, 2006.

LITANI, Elena; PATERNOSTRO, Marcelo. **Introduction to the Eclipse Modeling *Framework***. OMG Workshop on MDA, SOA and Web Services, p. 21-24, 2005.

LOPES, Denivaldo. **Introdução a Engenharia Dirigida por Modelos**. I Escola Regional de Computação Ceará Maranhão Piauí (ERCEMAPI), 2007.

LOPES, Denivaldo; HAMMOUDI, Slimane; BÉZIVIN, Jean; JOUAULT, Frédéric. **Mapping Specification in MDA: From Theory to Practice**. In: Interoperability of Enterprise *Software* and Applications - INTEROP-ESA. Springer, p. 253-264, 2006a.

LOPES, Denivaldo; HAMMOUDI, Slimane; ABDELOUAHAB, Zair. **Schema Matching in the context of Model Driven Engineering: From Theory to Practice**. In: Advances in Systems, Computing Sciences and *Software* Engineering, Springer, p. 253-264, 2006b.

LOPES, Denivaldo; HAMMOUDI, Slimane; SOUZA, José de and BONTEMPO, Alan. **Metamodel Matching: Experiments and Comparison**, 2006. Conference International on *Software* Engineering Advances on Volume, p. 2-2, 2006c.

MASSOL, Vincente. **JUnit in Action**. Editora Manning Publications, 2004.

MELLOR, Stephen; SCOOTT, Kendall; UHL, Axel and WEISE, Dirk. **MDA Destilada. Principios de Arquitetura Orientada por Modelos**. Editora Ciência Moderna, 2005.

MESZAROS, Gerard. **xUnit Test Patterns. Refactoring Test Code**. Edtora Pearson Education, 2007.

NANTZ, Brian. **Open Source .NET Development**, Editora Prentice Hall PTR, 2004.

OLDEVIK, Jon. **MOFScript User Guide**. Version 0.6 (MOFScript v 1.1.11), 2006.

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

OMG. **UML Testing Profile.** Version 1.0 Document Number: formal/05-07-07, 2005.

OMG. **Meta Object Facility (MOF) 2.0 Query/View/Transformation**, Version 1.0 Document Number: formal/2008-04-03, 2008.

PFLEEFER, Shari L. **Engenharia de *Software*: Teoria e Prática**. Editora São Paulo: Prentice Hall, 2004.

POTTINGER, R. A., e BERNSTEIN, P. A. **Merging Models Based on Given Correspondences**. Proceedings of the 29th VLDB Conference, p. 826-873, 2003.

RAINSBERGER, J. B. **JUnit Recipes: Practical Methods for Programmer Testing**. Editora Manning Publications, 2005.

RENDELL, Andrew. **Effective and Pragmatic Test Driven Development**. Conference Agile, p. 298-303, 2008

REZA, Hassan, OGAARD, Kirk, MALGE, Amarnath. **A Model Based Testing Technique to Test Web Applications Using Statecharts**. Fifth International Conference on Information Technology: New Generations, 2008.

SADILEK, Daniel A., WEIßLEDER, Stephan. **Testing Metamodels. Book Model Driven Architecture – Foundations and Applications**. Editora Springer Berlin / Heidelberg, 2008.

Sandhu, Singh, Singh, Pal, VERMA Kumar. **Evaluating Quality of Software Systems by Design Patterns Detection**. International Conference on Advanced Computer Theory and Engineering, 2008. ICACTE '08. p. 3-7, 2008.

SALVADOR, Trujillo, DON, Batory, OSCAR, Diaz, **Feature Oriented Model Driven Development: A Case Study for Portlets**, 29th International Conference on *Software* Engineering, p. 44-53, 2007.

SANTOS-Neto, Pedro, RESENDE, Rodolfo F., PÁDUA, Clarindo. **An Evaluation of a Model-Based Testing Method for Information Systems**. Symposium on Applied Computing Proceedings of the 2008 ACM symposium on Applied computing, p. 770-776, 2008.

SLYNGSTAD, Odd Petter N., LI, Jingyue, CONRADI, Reidar, RØNNEBERG, Harald, LANDRE, Einar e WESENBERG, Harald. **The Impact of Test Driven Development on the Evolution of a Reusable *Framework* of Components – An Industrial Case Study**. The Third International Conference on *Software* Engineering Advances, p. 214-223, 2008.

SOMMERVILLE, I. **Engenharia de Software**. 7ª edição, Addison Wesley, 2007.

SOUSA Helaine, LOPES, Denivaldo e ABDELOUAHAB, Zair. **Um Framework para Geração Automatizada de Casos de Testes Dirigido por Modelo**. 2ª Escola Regional de Computação do Ceará, Maranhão e Piauí – ERCEMAPI, 2008.

SOUZA Jr, José de, LOPES, Denivaldo, CLARO, Daniela e ABDELOUAHAB, Zair. A Set **Forward in Semi-Automatic Metamodel Mating: Algorithms and Tool**. 11th International Conference on Enterprise Information Systems LNBIP, 2009.

STAHL, Thomas, BETTIN, Jorn, VÖLTER, Markus, **Model-Driven Software Development Technology, Engineering, Management**, 1rst Edition, John Wiley Professional, p. 444, 2006.

STEEL, Jim, LAWLEY, Michael. **Model-Based Test Driven Development of the Tefkat Model-Transformation Engine**. 15th International Symposium on Software Reliability Engineering - ISSRE 2004, p. 151 - 160, 2004.

STEINBERG, Dave, BUDINSKY, Frank, PATERNOSTRO, Marcelo e MERKS, Ed. **EMF: Eclipse Modeling Framework**, 2nd Edition. Addison-Wesley Professional. 2008.

SUSS, J.G.; POP, A.; FRITZSON, P.; WILDMAN, L. **Towards Integrated Model-Driven Testing of SCADA Systems Using the Eclipse Modeling Framework and Modelica**. 19th Australian Conference on *Software* Engineering - ASWEC. Volume, Issue, 26-28 March 2008 p. 149 - 159, 2008.

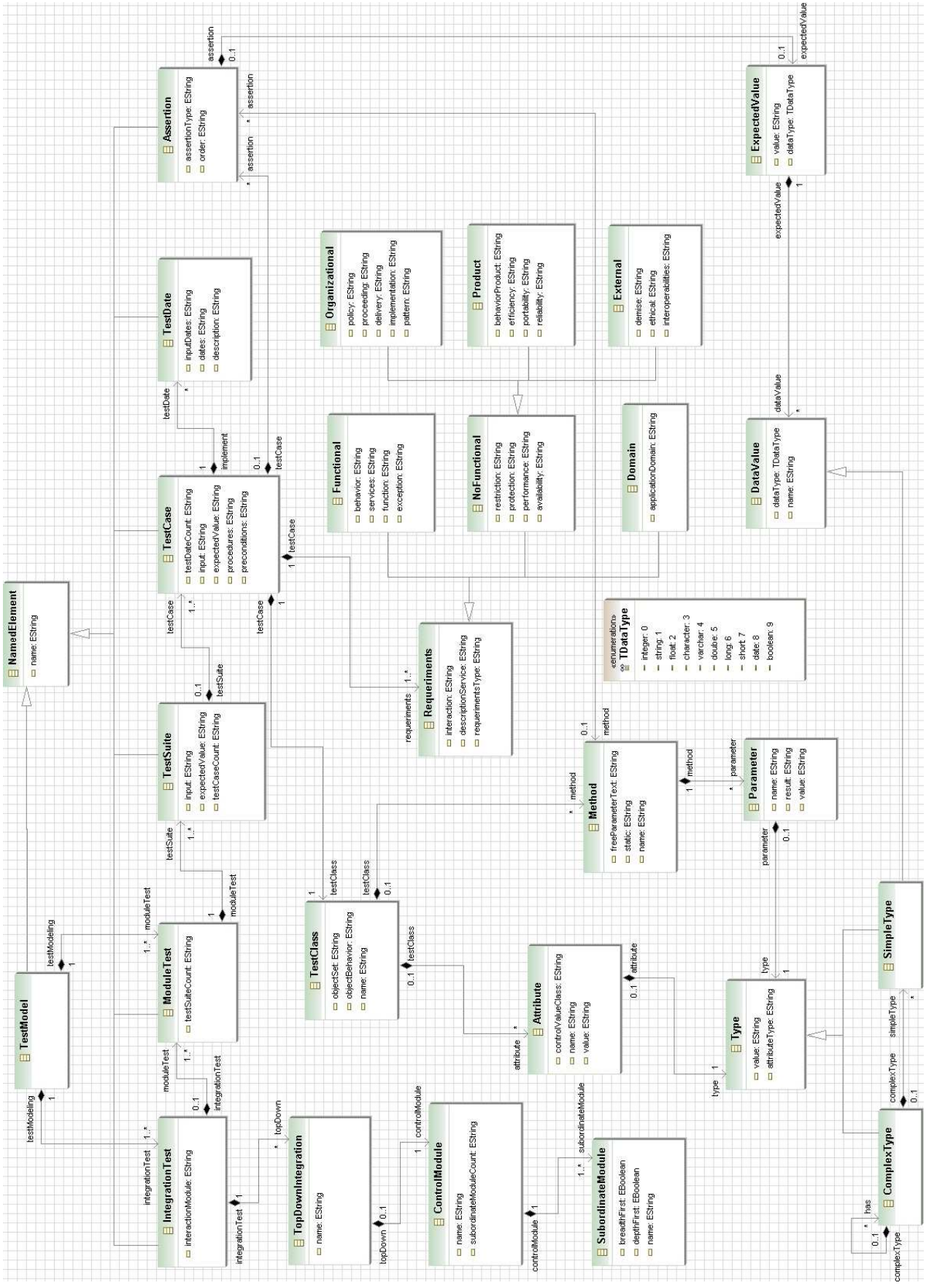
SWITHINBANK, Peter, CHESSELL, Mandy, GARDNER, Tracy et al, **Patterns: Model-Driven Development Using IBM Rational Software Architect**, Red Book, December 2005.

UTTING, M.; PRETSCHNER, A.; LEGEARD, B. **A Taxonomy of Model-Based Testing**. Department of Computer Science, University of Waikato, Hamilton, New Zealand, 2006.

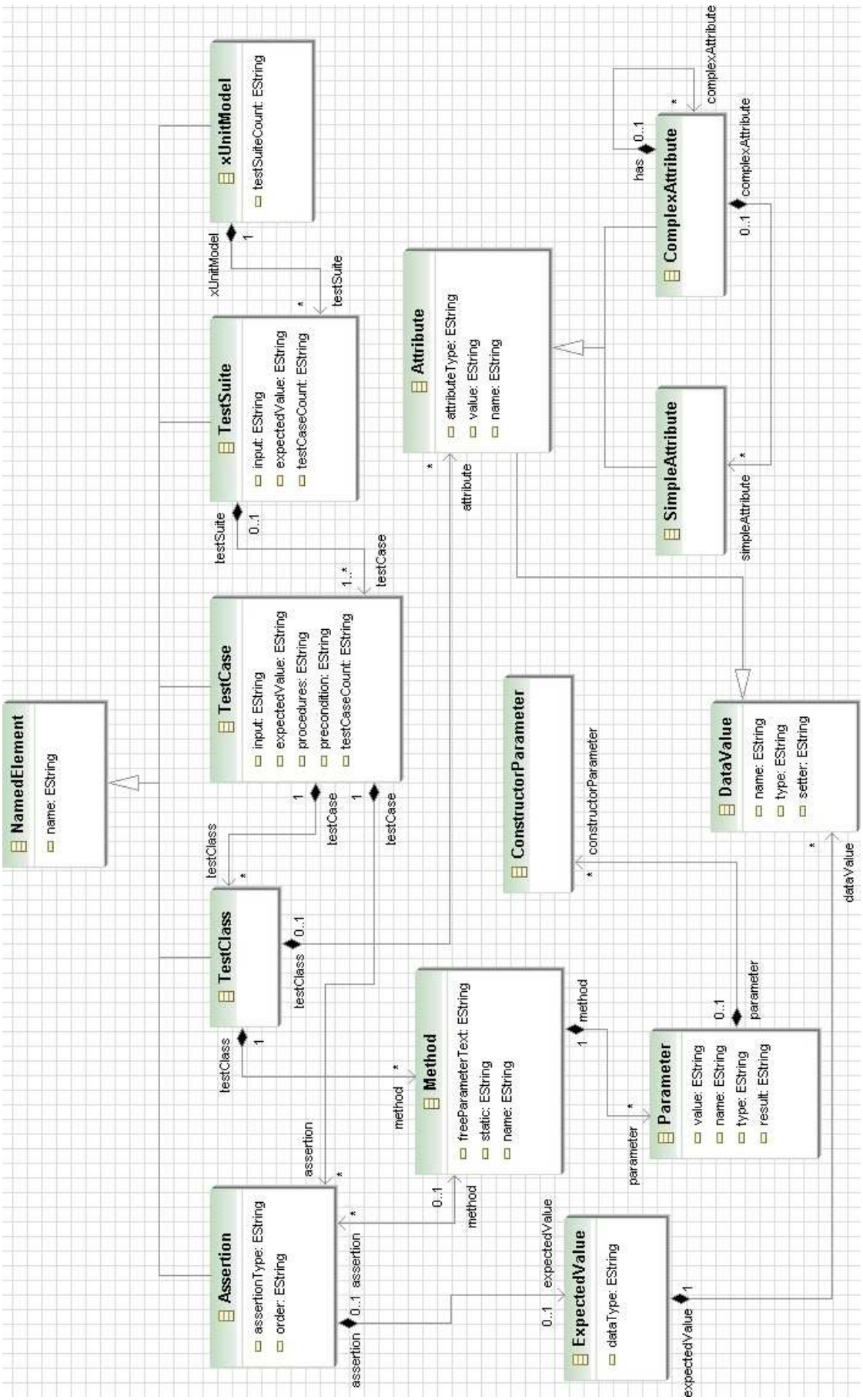
WIECZOREK, Sebastian, FRITZSCHE, Mathias, SCHNITTER, Joachim. **Enhancing Test Driven Development with Model Based Testing and Performance Analysis**. Testing: Academic & Industrial Conference - Practice and Research Techniques, 2008.

ANEXOS

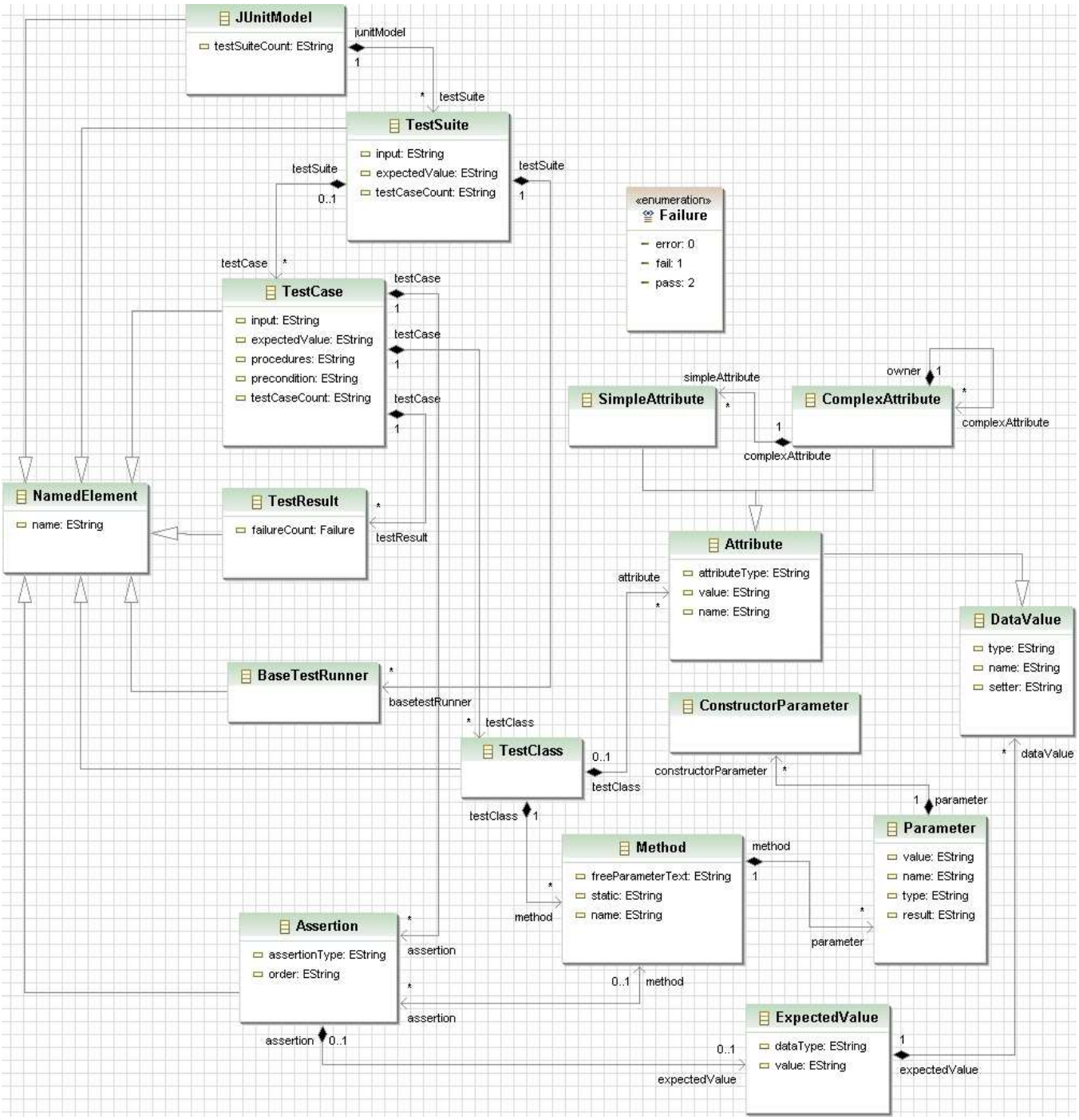
ANEXO A - Metamodelo de Teste Independente de Plataforma (MPIT)



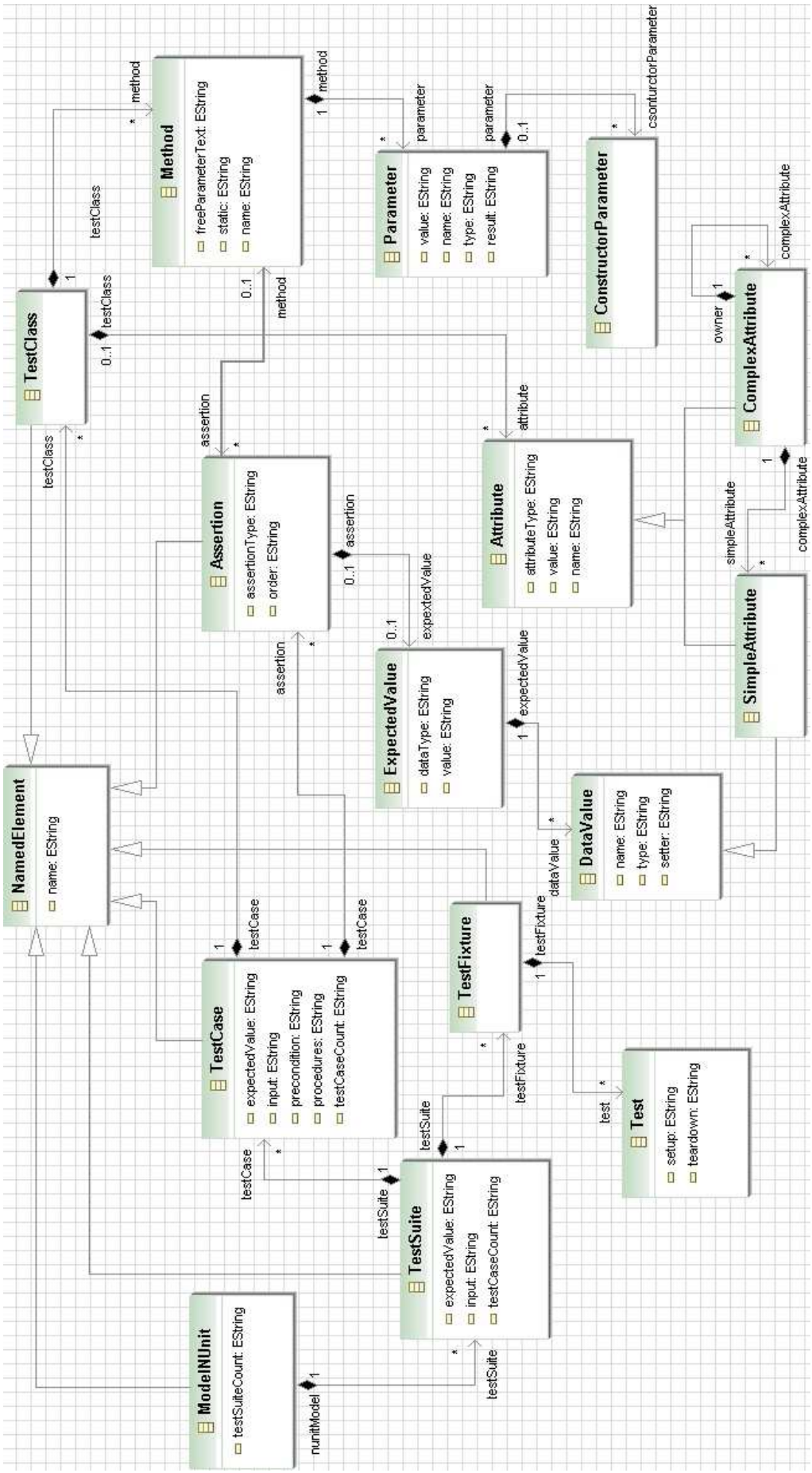
ANEXO B - Metamodelo de Teste Específico de Plataforma xUnit (MPST-xUnit)



ANEXO C - Metamodelo de Teste Específico de Plataforma JUnit (MPST-JUnit)



ANEXO D - Metamodelo de Teste Específico de Plataforma NUnit (MPST-NUnit)



ANEXO E - Regras de Transformação (modelo-a-modelo) de MPIT para MPST-JUnit

```
module MPIT2MPSTJUnit; -- Module Template
create OUT : JUnitMetaModel from IN : modeltesting;

rule TestModel2JUnitModel{
      from testmodel:modeltesting!TestModel
      to junitmodel:JUnitMetaModel!JUnitModel
      (
            name<- testmodel.name
      )
}

rule TestSuite2TestSuite{
      from _testsuite:modeltesting!TestSuite
      to testsuite:JUnitMetaModel!TestSuite
      (
            name<- _testsuite.name,
            input<- _testsuite.input,
            testCase<- _testsuite.testCase
      )
}

rule TestCase2TestCase{
      from _testcase:modeltesting!TestCase
      to testcase:JUnitMetaModel!TestCase
      (
            name<- _testcase.name,
            input<- _testcase.input,
            testSuite<- _testcase.testSuite,
            assertion<- _testcase.assertion,
            testClass<- _testcase.testClass
      )
}

rule Assertion2Assertion{
      from _assertion:modeltesting!Assertion
      to assertion:JUnitMetaModel!Assertion
      (
            name<- _assertion.name,
            assertionType<- _assertion.assertionType,
            expectedValue<- _assertion.expectedValue,
            testCase<- _assertion.testCase,
            method<- _assertion.method

      ),
        testresult:JUnitMetaModel!TestResult(
            name<- assertion.name,
            testCase<- assertion.testCase
      )
}

rule TestClass2TestClass{
      from _testclass:modeltesting!TestClass
      to testclass:JUnitMetaModel!TestClass
      (
            name<- _testclass.name,
            testCase <- _testclass.testCase,
            method<- _testclass.method,
```

```
            attribute<- _testclass.attribute
      )
}

rule Method2Method{
      from _method:modeltesting!Method
      to method:JUnitMetaModel!Method
      (
            name<- _method.name,
            testClass<- _method.testClass,
            assertion<- _method.assertion,
            parameter<- _method.parameter

      )
}


rule Parameter2Parameter{
      from _parameter:modeltesting!Parameter
      to parameter:JUnitMetaModel!Parameter
      (
            value<- _parameter.value,
            name<- _parameter.name,
            result<- _parameter.result,
            method<- _parameter.method

      )
}

rule ExpectedValue2ExpectedValue{
      from _expectedvalue:modeltesting!ExpectedValue
      to expectedvalue:JUnitMetaModel!ExpectedValue
      (
            value<- _expectedvalue.value,
            assertion<- _expectedvalue.assertion,
            dataValue<- _expectedvalue.dataValue
      )
}

rule DataValue2DataValue{
      from _datavalue:modeltesting!DataValue
      to datavalue:JUnitMetaModel!DataValue
      (
            name<- _datavalue.name,
            expectedValue<- _datavalue.expectedValue
      )
}

rule Attribute2Attribute{
      from _attribute:modeltesting!Attribute
      to attribute:JUnitMetaModel!Attribute
      (
            name<- _attribute.name,
            value<- _attribute.value,
            testClass<- _attribute.testClass
      )
}
```

ANEXO F - Modelo de Teste Específico de Plataforma JUnit em XMI

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:JUnitMetaModel="http://JUnitMetaModel">
  <JUnitMetaModel:JUnitModel name="ATM System"/>
  <JUnitMetaModel:TestSuite name="TestInquiry">
    <testCase name="TestInquiryCustomerConsole">
      <assertion name="transactionInquiry" assertionType="True"
method="/1/@testCase.0/@testClass.0/@method.0">
        <expectedValue value="true">
          <dataValue name="boolean"/>
        </expectedValue>
      </assertion>
      <testResult name="transactionInquiry"/>
      <testClass name="CustomerConsole">
        <method assertion="/1/@testCase.0/@assertion.0"
name="completTransactionInquiry">
          <parameter value="true" name="complet" result="complet"/>
        </method>
        <attribute value="helaine" name="from"/>
      </testClass>
    </testCase>
    <testCase name="TestInquiryPrinter">
      <assertion name="printer" assertionType="True"
method="/1/@testCase.1/@testClass.0/@method.0">
        <expectedValue value="true">
          <dataValue name="boolean"/>
        </expectedValue>
      </assertion>
      <testResult name="printer"/>
      <testClass name="Printer">
        <method assertion="/1/@testCase.1/@assertion.0" name="print">
          <parameter value="printer" name="printer" result="printer"/>
        </method>
      </testClass>
    </testCase>
  </JUnitMetaModel:TestSuite>
  <JUnitMetaModel:TestSuite name="TestDeposit">
    <testCase name="TestDepositCustomerConsole">
      <assertion name="amountMoney" assertionType="Equals"
method="/2/@testCase.0/@testClass.0/@method.2">
        <expectedValue value="1000">
          <dataValue name="integer"/>
        </expectedValue>
      </assertion>
      <assertion name="menuChoice" assertionType="Equals"
method="/2/@testCase.0/@testClass.0/@method.1">
        <expectedValue value="deposit">
          <dataValue name="varchar"/>
        </expectedValue>
      </assertion>
      <assertion name="transaction" assertionType="True"
method="/2/@testCase.0/@testClass.0/@method.0">
        <expectedValue value="true">
          <dataValue name="boolean"/>
        </expectedValue>
      </assertion>
      <testResult name="menuChoice"/>
      <testResult name="transaction"/>
```

```
      <testResult name="amountMoney"/>
      <testClass name="CustomerConsole">
        <method assertion="/2/@testCase.0/@assertion.2"
name="completTransactionDeposit">
          <parameter value="end" name="complet" result="end"/>
        </method>
        <method assertion="/2/@testCase.0/@assertion.1"
name="ReadMenuChoice">
          <parameter value="deposit" name="menu" result="deposit"/>
        </method>
        <method assertion="/2/@testCase.0/@assertion.0" name="ReadAmount">
          <parameter value="1000" name="amount" result="1000"/>
        </method>
        <attribute value="1000" name="amount"/>
        <attribute value="helaine" name="to"/>
        <attribute value="true" name="depositCash"/>
        <attribute value="false" name="depositCheque"/>
      </testClass>
    </testCase>
    <testCase name="TestDepositCashDispenser">
      <assertion name="DisponibleCash" assertionType="True"
method="/2/@testCase.1/@testClass.0/@method.1">
        <expectedValue value="true">
          <dataValue name="boolean"/>
        </expectedValue>
      </assertion>
      <assertion name="amountCash" assertionType="Equals"
method="/2/@testCase.1/@testClass.0/@method.0">
        <expectedValue value="1000">
          <dataValue name="integer"/>
        </expectedValue>
      </assertion>
      <testResult name="DisponibleCash"/>
      <testResult name="amountCash"/>
      <testClass name="CashDispenser">
        <method assertion="/2/@testCase.1/@assertion.1"
name="checkCashOnHand">
          <parameter value="1000" name="cash" result="1000"/>
        </method>
        <method assertion="/2/@testCase.1/@assertion.0"
name="dispenseCash">
          <parameter value="true" name="dispense" result="true"/>
        </method>
        <attribute value="1000" name="CashOnHand"/>
      </testClass>
    </testCase>
    <testCase name="TestDepositPrinter">
      <assertion name="printer" assertionType="True"
method="/2/@testCase.2/@testClass.0/@method.0">
        <expectedValue value="true">
          <dataValue name="boolean"/>
        </expectedValue>
      </assertion>
      <testResult name="printer"/>
      <testClass name="Printer">
        <method assertion="/2/@testCase.2/@assertion.0" name="print">
          <parameter value="printer" name="printer" result="printer"/>
        </method>
      </testClass>
    </testCase>
  </JUnitMetaModel:TestSuite>
```

```
  <JUnitMetaModel:TestSuite name="TestSession" input="">
    <testCase name="TestSessionCustomerConsole">
      <assertion name="StateOn" assertionType="True"
method="/3/@testCase.0/@testClass.0/@method.0">
        <expectedValue value="true">
          <dataValue name="boolean"/>
        </expectedValue>
      </assertion>
      <assertion name="StateOff" assertionType="False"
method="/3/@testCase.0/@testClass.0/@method.0">
        <expectedValue value="false">
          <dataValue name="boolean"/>
        </expectedValue>
      </assertion>
      <assertion name="StateInterrupted" assertionType="False"
method="/3/@testCase.0/@testClass.0/@method.0">
        <expectedValue value="false">
          <dataValue name="boolean"/>
        </expectedValue>
      </assertion>
      <testResult name="StateInterrupted"/>
      <testResult name="StateOff"/>
      <testResult name="StateOn"/>
      <testClass name="Session">
        <method assertion="/3/@testCase.0/@assertion.2
/3/@testCase.0/@assertion.1 /3/@testCase.0/@assertion.0"
name="performSession">
          <parameter value="true" name="Session" result="iniciate"/>
        </method>
        <attribute value="123" name="pin"/>
        <attribute value="true" name="state"/>
      </testClass>
    </testCase>
    <testCase name="TestSessionCardRead">
      <assertion name="SessionOn" assertionType="True"
method="/3/@testCase.1/@testClass.0/@method.4">
        <expectedValue value="true">
          <dataValue name="boolean"/>
        </expectedValue>
      </assertion>
      <assertion name="closeSession" assertionType="False"
method="/3/@testCase.1/@testClass.0/@method.3">
        <expectedValue value="false">
          <dataValue name="boolean"/>
        </expectedValue>
      </assertion>
      <assertion name="SessionOff" assertionType="False"
method="/3/@testCase.1/@testClass.0/@method.3">
        <expectedValue value="false">
          <dataValue name="boolean"/>
        </expectedValue>
      </assertion>
      <assertion name="ValidCard" assertionType="True"
method="/3/@testCase.1/@testClass.0/@method.3">
        <expectedValue value="true">
          <dataValue name="boolean"/>
        </expectedValue>
      </assertion>
      <assertion name="InvalidCard" assertionType="False"
method="/3/@testCase.1/@testClass.0/@method.3">
        <expectedValue value="false">
```

```
          <dataValue name="boolean"/>
        </expectedValue>
      </assertion>
      <assertion name="ApprovedCard" assertionType="True"
method="/3/@testCase.1/@testClass.0/@method.3">
        <expectedValue value="true">
          <dataValue name="boolean"/>
        </expectedValue>
      </assertion>
      <testResult name="closeSession"/>
      <testResult name="SessionOn"/>
      <testResult name="ApprovedCard"/>
      <testResult name="SessionOff"/>
      <testResult name="InvalidCard"/>
      <testResult name="ValidCard"/>
      <testClass name="CardRead">
        <method name="cardInsert">
          <parameter value="true" name="inserted" result="valid"/>
        </method>
        <method name="cardRead">
          <parameter value="true" name="reader" result="valid"/>
        </method>
        <method name="cardEject">
          <parameter value="false" name="ejected" result="invalid"/>
        </method>
        <method assertion="/3/@testCase.1/@assertion.1
/3/@testCase.1/@assertion.5 /3/@testCase.1/@assertion.2
/3/@testCase.1/@assertion.4 /3/@testCase.1/@assertion.3" name="VerifyCard">
          <parameter value="false" name="verified" result="invalid"/>
        </method>
        <method assertion="/3/@testCase.1/@assertion.0"
name="initiateSession">
          <parameter value="true" name="initiate" result="initiate"/>
        </method>
        <attribute value="123" name="pin"/>
      </testClass>
    </testCase>
    <testCase name="TestSessionTransaction">
      <assertion name="initiateTransaction" assertionType="True"
method="/3/@testCase.2/@testClass.0/@method.1">
        <expectedValue value="true">
          <dataValue name="boolean"/>
        </expectedValue>
      </assertion>
      <assertion name="verifyPIN" assertionType="True"
method="/3/@testCase.2/@testClass.0/@method.0">
        <expectedValue value="true">
          <dataValue name="boolean"/>
        </expectedValue>
      </assertion>
      <assertion name="closedTransaction" assertionType="False"
method="/3/@testCase.2/@testClass.0/@method.3">
        <expectedValue value="false">
          <dataValue name="boolean"/>
        </expectedValue>
      </assertion>
      <assertion name="interruptedTransaction" assertionType="False"
method="/3/@testCase.2/@testClass.0/@method.3">
        <expectedValue value="false">
          <dataValue name="boolean"/>
        </expectedValue>
```

```
      </assertion>
      <testResult name="verifyPIN"/>
      <testResult name="interruptedTransaction"/>
      <testResult name="initiateTransaction"/>
      <testResult name="closedTransaction"/>
      <testClass name="Transaction">
        <method assertion="/3/@testCase.2/@assertion.1" name="readPIN">
          <parameter value="123" name="pin" result="true"/>
        </method>
        <method assertion="/3/@testCase.2/@assertion.0"
name="makeTransaction">
          <parameter value="true" name="transaction" result="transaction"/>
        </method>
        <method name="completTransaction">
          <parameter value="true" name="transaction" result="transaction"/>
        </method>
        <method assertion="/3/@testCase.2/@assertion.3
/3/@testCase.2/@assertion.2" name="peformInvalidPinExtension">
          <parameter value="123" name="pin" result="true"/>
        </method>
        <attribute value="true" name="state"/>
        <attribute value="123" name="pin"/>
      </testClass>
    </testCase>
  </JUnitMetaModel:TestSuite>
  <JUnitMetaModel:TestSuite name="TestWithdrawal">
    <testCase name="TestWithdrawalCustomerConsole">
      <assertion name="transactionWithdrawal" assertionType="Equals"
method="/4/@testCase.0/@testClass.0/@method.0">
        <expectedValue value="500">
          <dataValue name="integer"/>
        </expectedValue>
      </assertion>
      <testResult name="transactionWithdrawal"/>
      <testClass name="CustomerConsole">
        <method assertion="/4/@testCase.0/@assertion.0"
name="completTransactionWithdrawal">
          <parameter value="500" name="complet" result="amount"/>
        </method>
        <attribute value="A" name="from"/>
        <attribute value="500" name="amount"/>
      </testClass>
    </testCase>
    <testCase name="TestWithdrawalCashDispenser">
      <assertion name="DisponibleCash" assertionType="Equals"
method="/4/@testCase.1/@testClass.0/@method.1">
        <expectedValue value="500">
          <dataValue name="integer"/>
        </expectedValue>
      </assertion>
      <assertion name="amountCash" assertionType="Equals"
method="/4/@testCase.1/@testClass.0/@method.0">
        <expectedValue value="500">
          <dataValue name="integer"/>
        </expectedValue>
      </assertion>
      <testResult name="amountCash"/>
      <testResult name="DisponibleCash"/>
      <testClass name="CashDispenser">
        <method assertion="/4/@testCase.1/@assertion.1"
name="checkCashOnHand">
```

```
          <parameter value="500" name="cash" result="amount"/>
        </method>
        <method assertion="/4/@testCase.1/@assertion.0"
name="dispenseCash">
          <parameter value="500" name="dispense" result="amount"/>
        </method>
        <attribute value="500" name="cashOnHand"/>
      </testClass>
    </testCase>
  </JUnitMetaModel:TestSuite>
  <JUnitMetaModel:TestSuite name="TestTransfer">
    <testCase name="TestTransferCustomerConsole">
      <assertion name="transactionTransfer" assertionType="Equals"
method="/5/@testCase.0/@testClass.0/@method.0">
        <expectedValue value="500">
          <dataValue name="integer"/>
        </expectedValue>
      </assertion>
      <testResult name="transactionTransfer"/>
      <testClass name="CustomerConsole">
        <method assertion="/5/@testCase.0/@assertion.0"
name="completTransactionTransfer">
          <parameter value="500" name="complet" result="transfer"/>
        </method>
        <attribute value="A" name="from"/>
        <attribute value="500" name="amount"/>
        <attribute value="B" name="to"/>
      </testClass>
    </testCase>
    <testCase name="TestTransferCashDispenser">
      <assertion name="DisponibleCash" assertionType="Equals"
method="/5/@testCase.1/@testClass.0/@method.1">
        <expectedValue value="500">
          <dataValue name="integer"/>
        </expectedValue>
      </assertion>
      <assertion name="amountCash" assertionType="Equals"
method="/5/@testCase.1/@testClass.0/@method.0">
        <expectedValue value="500">
          <dataValue name="integer"/>
        </expectedValue>
      </assertion>
      <testResult name="amountCash"/>
      <testResult name="DisponibleCash"/>
      <testClass name="CashDispenser">
        <method assertion="/5/@testCase.1/@assertion.1"
name="checkCashOnHand">
          <parameter value="500" name="cash" result="500"/>
        </method>
        <method assertion="/5/@testCase.1/@assertion.0"
name="dispenseCash">
          <parameter value="500" name="dispense" result="500"/>
        </method>
        <attribute value="500" name="cashOnHand"/>
      </testClass>
    </testCase>
    <testCase name="TestTransferPrinter">
      <assertion name="printer" assertionType="True"
method="/5/@testCase.2/@testClass.0/@method.0">
        <expectedValue value="true">
          <dataValue name="boolean"/>
```

```
        </expectedValue>
      </assertion>
      <testResult name="printer"/>
      <testClass name="Printer">
        <method assertion="/5/@testCase.2/@assertion.0" name="print">
          <parameter value="printer" name="printer" result="printer"/>
        </method>
      </testClass>
    </testCase>
  </JUnitMetaModel:TestSuite>
</xmi:XMI>
```

ANEXO G - Regras de Transformação (modelo-a-texto) JUnit

```
query modelJUnit2SourceCode = JUnitMetaModel!NamedElement.allInstances()->
      select(e | e.oclIsTypeOf(JUnitMetaModel!TestSuite) or
e.oclIsTypeOf(JUnitMetaModel!TestCase))->
    collect(x | x.toString().writeTo('C:/SourceCode/JUnit/' + x.name +
'.java')); -- Query Template

uses modelJUnit2SourceCode;

                                    ***

library modelJUnit2SourceCode; -- Library Template

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

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

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

helper context JUnitMetaModel!Assertion def: toString() : String =
    '\t@Test\n'+
      '\tpublic void ' +
       self.name + '(){\n' +
       self.generateAssertionType() +
       '\n\t}\n';

helper context JUnitMetaModel!TestCase  def: toString() : String =
      'public class ' +
       self.name +
      ' extends junit.framework.TestCase{\n\n' +
     self.assertion->iterate(i; acc:String=''| acc +
     i.toString()+'\n') +
      '\n}\n';

helper context JUnitMetaModel!TestSuite  def: toString() : String =
      'public class ' +
       self.name + ' ' +
```

```
'extende' + ' ' +
'Test Case' + ' ' +
'{\n\n'+
'\tTestSuite suite = new TestSuite("'+ self.name +'");\n'+
self.testCase->iterate(i; acc:String=''| acc +
'\tsuite.addTestCase('+ i.name+'.class' + ');\n') +
'\treturn suite;\n' +
'\n}\n\n';
```

ANEXO H - Código-Fonte de Teste em JUnit para o Sistema ATM

```
public class TestSession extends Test Case {

	TestSuite suite = new TestSuite("TestSession");
	suite.addTestCase(TestSessionCustomerConsole.class);
	suite.addTestCase(TestSessionCardRead.class);
	suite.addTestCase(TestSessionTransaction.class);
	return suite;
}
```

***

```
public class TestSessionCardRead extends junit.framework.TestCase{

	@Test
	public void SessionOn(){

		assertTrue(true, new CardRead().initiateSession(initiate));
	}

	@Test
	public void closeSession(){

		assertFalse(false, new CardRead().VerifyCard(verified));
	}

	@Test
	public void SessionOff(){

		assertFalse(false, new CardRead().VerifyCard(verified));
	}

	@Test
	public void ValidCard(){

		assertTrue(true, new CardRead().VerifyCard(verified));
	}

	@Test
	public void InvalidCard(){

		assertFalse(false, new CardRead().VerifyCard(verified));
	}

	@Test
	public void ApprovedCard(){

		assertTrue(true, new CardRead().VerifyCard(verified));
	}
}
```

```
public class TestSessionCustomerConsole extends junit.framework.TestCase{

	@Test
	public void StateOn(){

		assertTrue(true, new Session().performSession(Session));
```

```
	}

	@Test
	public void StateOff(){

		assertFalse(false, new Session().performSession(Session));
	}

	@Test
	public void StateInterrupted(){

		assertFalse(false, new Session().performSession(Session));
	}
}
```

```
public class TestSessionTransaction extends junit.framework.TestCase{

	@Test
	public void initiateTransaction(){

		assertTrue(true, new
Transaction().makeTransaction(transaction));
	}

	@Test
	public void verifyPIN(){

		assertTrue(true, new Transaction().readPIN(pin));
	}

	@Test
	public void closedTransaction(){

		assertFalse(false, new
Transaction().peformInvalidPinExtension(pin));
	}

	@Test
	public void interruptedTransaction(){

		assertFalse(false, new
Transaction().peformInvalidPinExtension(pin));
	}
}
```

***

```
public class TestTransfer extends Test Case {

	TestSuite suite = new TestSuite("TestTransfer");
	suite.addTestCase(TestTransferCustomerConsole.class);
	suite.addTestCase(TestTransferCashDispenser.class);
	suite.addTestCase(TestTransferPrinter.class);
	return suite;
}
```

```
public class TestTransferCashDispenser extends junit.framework.TestCase{
```

```
	@Test
	public void DisponibleCash(){

		assertEqual (500, new CashDispenser().dispenseCash(dispense));
	}

	@Test
	public void amountCash(){

		assertEqual (500, new CashDispenser().checkCashOnHand(cash));
	}
}
```

```
public class TestTransferCustomerConsole extends junit.framework.TestCase{

	@Test
	public void transactionTransfer(){

		assertEqual (500, new
CustomerConsole().completTransactionTransfer(complet));
	}
}
```

```
public class TestTransferPrinter extends junit.framework.TestCase{

	@Test
	public void printer(){

		assertTrue(true, new Printer().print(printer));
	}
}
```

***

```
public class TestWithdrawal extends Test Case {

	TestSuite suite = new TestSuite("TestWithdrawal");
	suite.addTestCase(TestWithdrawalCustomerConsole.class);
	suite.addTestCase(TestWithdrawalCashDispenser.class);
	return suite;
}
```

```
public class TestWithdrawalCashDispenser extends junit.framework.TestCase{

	@Test
	public void DisponibleCash(){

		assertEqual (500, new CashDispenser().dispenseCash(dispense));
	}

	@Test
	public void amountCash(){

		assertEqual (500, new CashDispenser().checkCashOnHand(cash));
	}
}
```

```
public class TestWithdrawalCustomerConsole extends
junit.framework.TestCase{

      @Test
      public void transactionWithdrawal(){

            assertEqual (500, new
CustomerConsole().completTransactionWithdrawal(complet));
      }
}
```

***

```
public class TestDeposit extends Test Case {

      TestSuite suite = new TestSuite("TestDeposit");
      suite.addTestCase(TestDepositCustomerConsole.class);
      suite.addTestCase(TestDepositCashDispenser.class);
      suite.addTestCase(TestDepositPrinter.class);
      return suite;
}
```

```
public class TestDepositCashDispenser extends junit.framework.TestCase{

      @Test
      public void DisponibleCash(){

            assertTrue(true, new CashDispenser().dispenseCash(dispense));
      }

      @Test
      public void amountCash(){

            assertEqual (1000, new CashDispenser().checkCashOnHand(cash));
      }
}
```

```
public class TestDepositCustomerConsole extends junit.framework.TestCase{

      @Test
      public void amountMoney(){

            assertEqual (1000, new CustomerConsole().ReadAmount(amount));
      }

      @Test
      public void menuChoice(){

            assertEqual (deposit, new
CustomerConsole().ReadMenuChoice(menu));
      }

      @Test
      public void transaction(){

            assertTrue(true, new
CustomerConsole().completTransactionDeposit(complet));
```

```
	}
}
```

```
public class TestDepositPrinter extends junit.framework.TestCase{

	@Test
	public void printer(){

		assertTrue(true, new Printer().print(printer));
	}
}
```

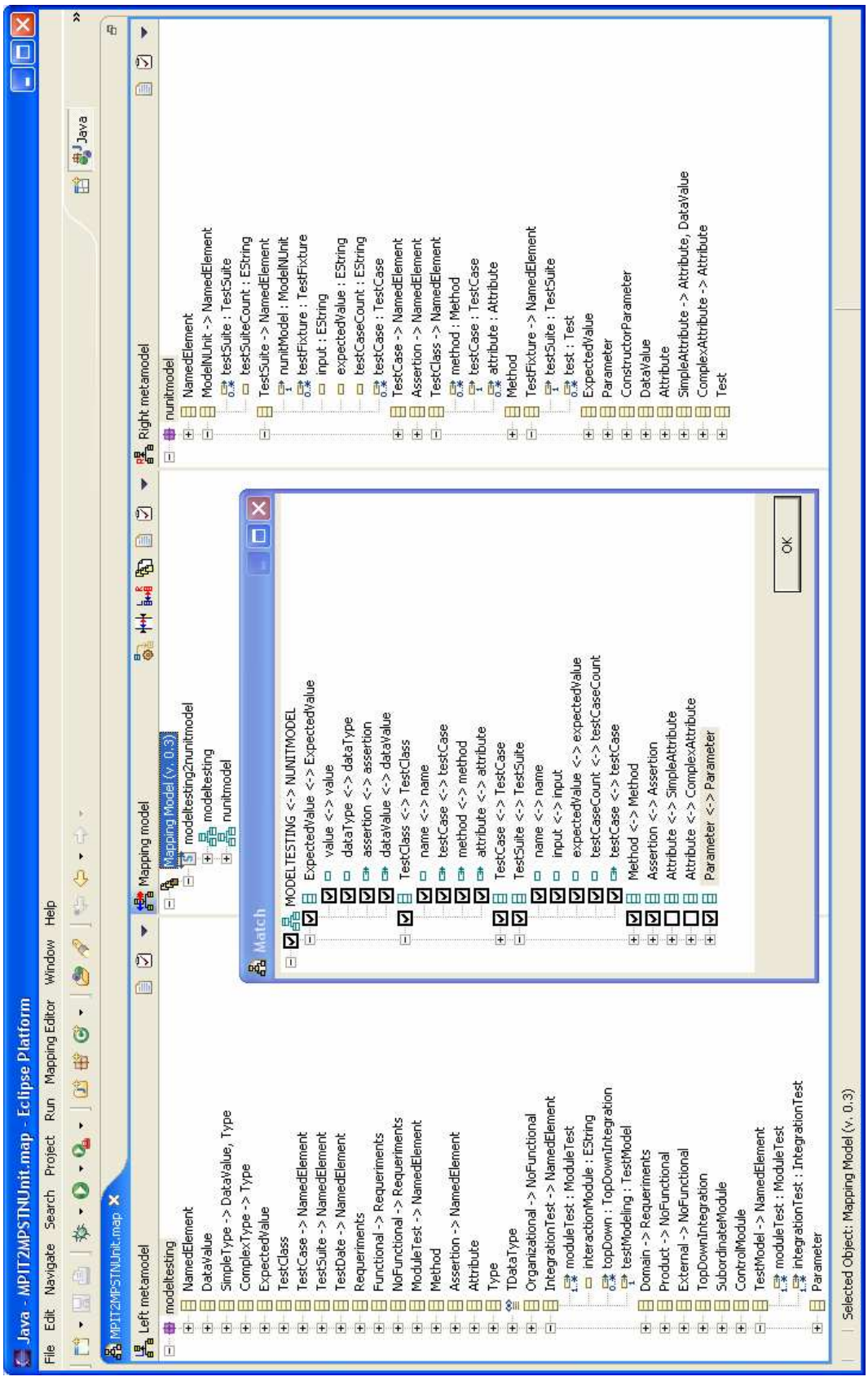
***

```
public class TestInquiry extends Test Case {

	TestSuite suite = new TestSuite("TestInquiry");
	suite.addTestCase(TestInquiryCustomerConsole.class);
	suite.addTestCase(TestInquiryPrinter.class);
	return suite;
}
```

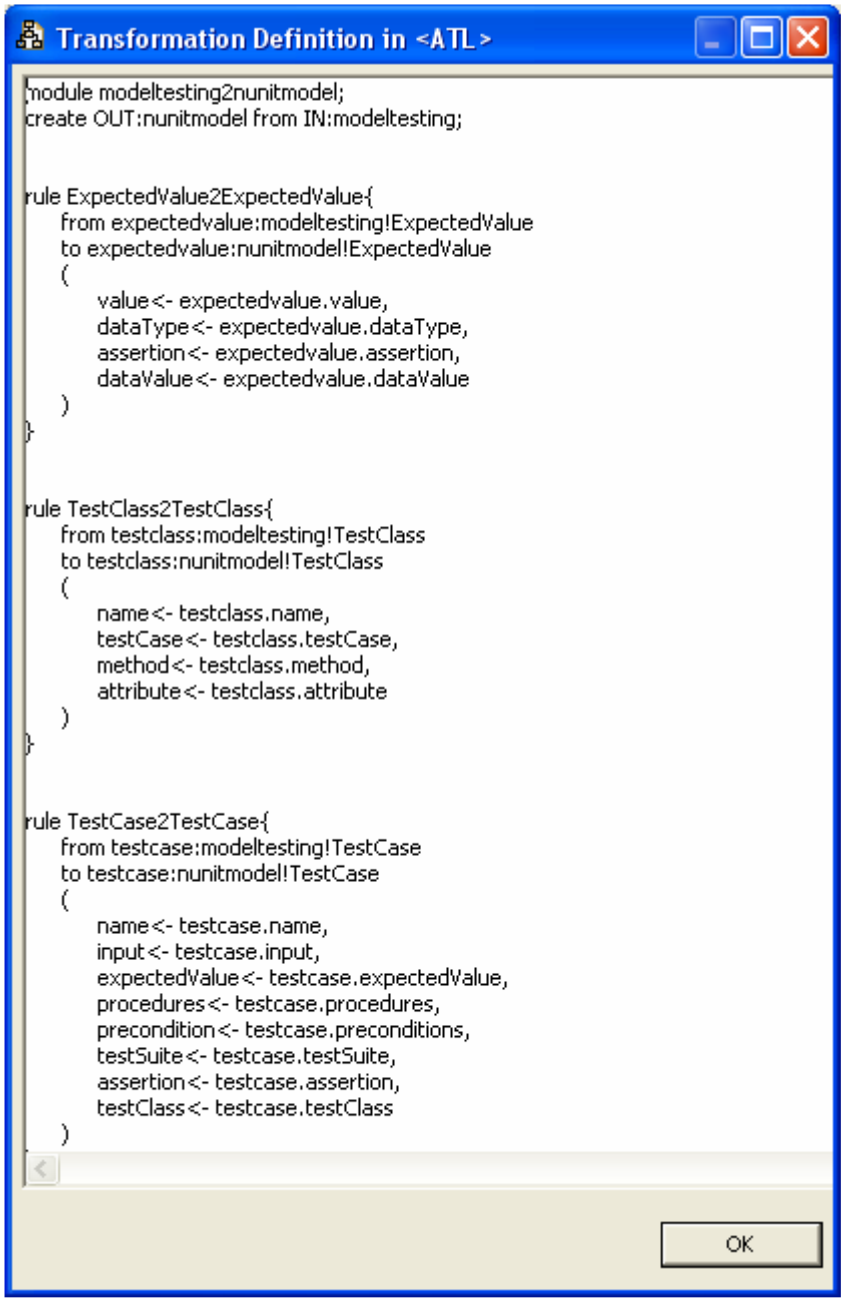
```
public class TestInquiryCustomerConsole extends junit.framework.TestCase{

	@Test
	public void transactionInquiry(){

		assertTrue(true, new
CustomerConsole().completTransactionInquiry(complet));
	}
}
```

```
public class TestInquiryPrinter extends junit.framework.TestCase{

	@Test
	public void printer(){

		assertTrue(true, new Printer().print(printer));
	}
}
```

ANEXO I - Especificação de Correspondência entre MPIT e MPST-NUnit



ANEXO J - Regras de Transformação (modelo-a-modelo) de MPIT para MPST-NUnit



***

```
module MPIT2MPSTNUnit; -- Module Template
create OUT : nunitmodel from IN : modeltesting;

rule TestModel2ModelNUnit{
	from testmodel:modeltesting!TestModel
	to modelnunit:nunitmodel!ModelNUnit
	(
		name<- testmodel.name
	)
}
```

```
rule TestSuite2TestSuite{
      from _testsuite:modeltesting!TestSuite
      to testsuite:nunitmodel!TestSuite
      (
            name<- _testsuite.name,
            input<- _testsuite.input,
            testCaseCount<- _testsuite.testCaseCount,
            testCase<- _testsuite.testCase
      )
}

rule TestCase2TestCase{
      from _testcase:modeltesting!TestCase
      to testcase:nunitmodel!TestCase
      (
            name<- _testcase.name,
            input<- _testcase.input,
            procedures<- _testcase.procedures,
            precondition<- _testcase.preconditions,
            testSuite<- _testcase.testSuite,
            testClass<- _testcase.testClass,
            assertion<- _testcase.assertion
      )
}

rule Assertion2Assertion{
      from _assertion:modeltesting!Assertion
      to assertion:nunitmodel!Assertion
      (
            name<- _assertion.name,
            assertionType<- _assertion.assertionType,
            order<- _assertion.order,
            expectedValue<- _assertion.expectedValue,
            testCase<- _assertion.testCase,
            method<- _assertion.method
      )
}
rule TestClass2TestClass{
      from _testclass:modeltesting!TestClass
      to testclass:nunitmodel!TestClass
      (
            name<- _testclass.name,
            testCase <- _testclass.testCase,
            method<- _testclass.method,
            attribute<- _testclass.attribute
      )
}

rule Method2Method{
      from _method:modeltesting!Method
      to method:nunitmodel!Method
      (
            name<- _method.name,
            freeParameterText<- _method.freeParameterText,
            static<- _method.static,
            assertion<- _method.assertion,
            testClass<- _method.testClass,
            parameter<- _method.parameter
      )
}
```

```
rule Parameter2Parameter{
	from _parameter:modeltesting!Parameter
	to parameter:nunitmodel!Parameter
	(
		value<- _parameter.value,
		method<- _parameter.method
	)
}


rule ExpectedValue2ExpectedValue{
	from _expectedvalue:modeltesting!ExpectedValue
	to expectedvalue:nunitmodel!ExpectedValue
	(
		value<- _expectedvalue.value,
		assertion<- _expectedvalue.assertion,
		dataValue<- _expectedvalue.dataValue
	)
}

rule DataValue2DataValue{
	from _datavalue:modeltesting!DataValue
	to datavalue:nunitmodel!DataValue
	(
		name<- _datavalue.name,
		expectedValue<- _datavalue.expectedValue
	)
}

rule Attribute2Attribute{
	from _attribute:modeltesting!Attribute
	to attribute:nunitmodel!Attribute
	(
		name<- _attribute.name,
		value<- _attribute.value,
		testClass<- _attribute.testClass
	)
}
```

ANEXO L - Modelo de Teste Específico de Plataforma NUnit em XMI

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:nunitmodel="http://nunitmodel">
  <nunitmodel:ModelNUnit name="ATM System"/>
  <nunitmodel:TestSuite name="TestSession" input="" testCaseCount="">
    <testCase name="TestSessionCustomerConsole">
      <assertion name="StateOn" assertionType="True"
method="/1/@testCase.0/@testClass.0/@method.0">
        <expectedValue value="true">
          <dataValue name="boolean"/>
        </expectedValue>
      </assertion>
      <assertion name="StateOff" assertionType="False"
method="/1/@testCase.0/@testClass.0/@method.0">
        <expectedValue value="false">
          <dataValue name="boolean"/>
        </expectedValue>
      </assertion>
      <assertion name="StateInterrupted" assertionType="False"
method="/1/@testCase.0/@testClass.0/@method.0">
        <expectedValue value="false">
          <dataValue name="boolean"/>
        </expectedValue>
      </assertion>
      <testClass name="Session">
        <method assertion="/1/@testCase.0/@assertion.0
/1/@testCase.0/@assertion.1 /1/@testCase.0/@assertion.2"
name="performSession">
          <parameter value="true"/>
        </method>
        <attribute value="123" name="pin"/>
        <attribute value="true" name="state"/>
      </testClass>
    </testCase>
    <testCase name="TestSessionCardRead">
      <assertion name="SessionOn" assertionType="True"
method="/1/@testCase.1/@testClass.0/@method.4">
        <expectedValue value="true">
          <dataValue name="boolean"/>
        </expectedValue>
      </assertion>
      <assertion name="closeSession" assertionType="False"
method="/1/@testCase.1/@testClass.0/@method.3">
        <expectedValue value="false">
          <dataValue name="boolean"/>
        </expectedValue>
      </assertion>
      <assertion name="SessionOff" assertionType="False"
method="/1/@testCase.1/@testClass.0/@method.3">
        <expectedValue value="false">
          <dataValue name="boolean"/>
        </expectedValue>
      </assertion>
      <assertion name="ValidCard" assertionType="True"
method="/1/@testCase.1/@testClass.0/@method.3">
        <expectedValue value="true">
          <dataValue name="boolean"/>
        </expectedValue>
```

```
      </assertion>
      <assertion name="InvalidCard" assertionType="False"
method="/1/@testCase.1/@testClass.0/@method.3">
        <expectedValue value="false">
          <dataValue name="boolean"/>
        </expectedValue>
      </assertion>
      <assertion name="ApprovedCard" assertionType="True"
method="/1/@testCase.1/@testClass.0/@method.3">
        <expectedValue value="true">
          <dataValue name="boolean"/>
        </expectedValue>
      </assertion>
      <testClass name="CardRead">
        <method name="cardInsert">
          <parameter value="true"/>
        </method>
        <method name="cardRead">
          <parameter value="true"/>
        </method>
        <method name="cardEject">
          <parameter value="false"/>
        </method>
        <method assertion="/1/@testCase.1/@assertion.1
/1/@testCase.1/@assertion.2 /1/@testCase.1/@assertion.3
/1/@testCase.1/@assertion.4 /1/@testCase.1/@assertion.5" name="VerifyCard">
          <parameter value="false"/>
        </method>
        <method freeParameterText=""
assertion="/1/@testCase.1/@assertion.0" name="initiateSession">
          <parameter value="true"/>
        </method>
        <attribute value="123" name="pin"/>
      </testClass>
    </testCase>
    <testCase name="TestSessionTransaction">
      <assertion name="initiateTransaction" assertionType="True"
method="/1/@testCase.2/@testClass.0/@method.1">
        <expectedValue value="true">
          <dataValue name="boolean"/>
        </expectedValue>
      </assertion>
      <assertion name="verifyPIN" assertionType="True"
method="/1/@testCase.2/@testClass.0/@method.0">
        <expectedValue value="true">
          <dataValue name="boolean"/>
        </expectedValue>
      </assertion>
      <assertion name="closedTransaction" assertionType="False"
method="/1/@testCase.2/@testClass.0/@method.3">
        <expectedValue value="false">
          <dataValue name="boolean"/>
        </expectedValue>
      </assertion>
      <assertion name="interruptedTransaction" assertionType="False"
method="/1/@testCase.2/@testClass.0/@method.3">
        <expectedValue value="false">
          <dataValue name="boolean"/>
        </expectedValue>
      </assertion>
      <testClass name="Transaction">
```

```
        <method assertion="/1/@testCase.2/@assertion.1" name="readPIN">
          <parameter value="123"/>
        </method>
        <method assertion="/1/@testCase.2/@assertion.0"
name="makeTransaction">
          <parameter value="true"/>
        </method>
        <method name="completTransaction">
          <parameter value="true"/>
        </method>
        <method assertion="/1/@testCase.2/@assertion.2
/1/@testCase.2/@assertion.3" name="peformInvalidPinExtension">
          <parameter value="123"/>
        </method>
        <attribute value="true" name="state"/>
        <attribute value="123" name="pin"/>
      </testClass>
    </testCase>
  </nunitmodel:TestSuite>
  <nunitmodel:TestSuite name="TestDeposit">
    <testCase name="TestDepositCustomerConsole">
      <assertion name="amountMoney" assertionType="Equals"
method="/2/@testCase.0/@testClass.0/@method.2">
        <expectedValue value="1000">
          <dataValue name="integer"/>
        </expectedValue>
      </assertion>
      <assertion name="menuChoice" assertionType="Equals"
method="/2/@testCase.0/@testClass.0/@method.1">
        <expectedValue value="deposit">
          <dataValue name="varchar"/>
        </expectedValue>
      </assertion>
      <assertion name="transaction" assertionType="True"
method="/2/@testCase.0/@testClass.0/@method.0">
        <expectedValue value="true">
          <dataValue name="boolean"/>
        </expectedValue>
      </assertion>
      <testClass name="CustomerConsole">
        <method assertion="/2/@testCase.0/@assertion.2"
name="completTransactionDeposit">
          <parameter value="end"/>
        </method>
        <method assertion="/2/@testCase.0/@assertion.1"
name="ReadMenuChoice">
          <parameter value="deposit"/>
        </method>
        <method assertion="/2/@testCase.0/@assertion.0" name="ReadAmount">
          <parameter value="1000"/>
        </method>
        <attribute value="1000" name="amount"/>
        <attribute value="helaine" name="to"/>
        <attribute value="true" name="depositCash"/>
        <attribute value="false" name="depositCheque"/>
      </testClass>
    </testCase>
    <testCase name="TestDepositCashDispenser">
      <assertion name="DisponibleCash" assertionType="True"
method="/2/@testCase.1/@testClass.0/@method.1">
        <expectedValue value="true">
```

```
          <dataValue name="boolean"/>
        </expectedValue>
      </assertion>
      <assertion name="amountCash" assertionType="Equals"
method="/2/@testCase.1/@testClass.0/@method.0">
        <expectedValue value="1000">
          <dataValue name="integer"/>
        </expectedValue>
      </assertion>
      <testClass name="CashDispenser">
        <method assertion="/2/@testCase.1/@assertion.1"
name="checkCashOnHand">
          <parameter value="1000"/>
        </method>
        <method assertion="/2/@testCase.1/@assertion.0"
name="dispenseCash">
          <parameter value="true"/>
        </method>
        <attribute value="1000" name="CashOnHand"/>
      </testClass>
    </testCase>
    <testCase name="TestDepositPrinter" precondition="">
      <assertion name="printer" assertionType="True"
method="/2/@testCase.2/@testClass.0/@method.0">
        <expectedValue value="true">
          <dataValue name="boolean"/>
        </expectedValue>
      </assertion>
      <testClass name="Printer">
        <method assertion="/2/@testCase.2/@assertion.0" name="print">
          <parameter value="printer"/>
        </method>
      </testClass>
    </testCase>
  </nunitmodel:TestSuite>
  <nunitmodel:TestSuite name="TestInquiry">
    <testCase name="TestInquiryCustomerConsole">
      <assertion name="transactionInquiry" assertionType="True"
method="/3/@testCase.0/@testClass.0/@method.0">
        <expectedValue value="true">
          <dataValue name="boolean"/>
        </expectedValue>
      </assertion>
      <testClass name="CustomerConsole">
        <method assertion="/3/@testCase.0/@assertion.0"
name="completTransactionInquiry">
          <parameter value="true"/>
        </method>
        <attribute value="helaine" name="from"/>
      </testClass>
    </testCase>
    <testCase name="TestInquiryPrinter" precondition="">
      <assertion name="printer" assertionType="True"
method="/3/@testCase.1/@testClass.0/@method.0">
        <expectedValue value="true">
          <dataValue name="boolean"/>
        </expectedValue>
      </assertion>
      <testClass name="Printer">
        <method assertion="/3/@testCase.1/@assertion.0" name="print">
          <parameter value="printer"/>
```

```
        </method>
      </testClass>
    </testCase>
  </nunitmodel:TestSuite>
  <nunitmodel:TestSuite name="TestWithdrawal">
    <testCase name="TestWithdrawalCustomerConsole">
      <assertion name="transactionWithdrawal" assertionType="Equals"
method="/4/@testCase.0/@testClass.0/@method.0">
        <expectedValue value="500">
          <dataValue name="integer"/>
        </expectedValue>
      </assertion>
      <testClass name="CustomerConsole">
        <method assertion="/4/@testCase.0/@assertion.0"
name="completTransactionWithdrawal">
          <parameter value="500"/>
        </method>
        <attribute value="A" name="from"/>
        <attribute value="500" name="amount"/>
      </testClass>
    </testCase>
    <testCase name="TestWithdrawalCashDispenser">
      <assertion name="DisponibleCash" assertionType="Equals"
method="/4/@testCase.1/@testClass.0/@method.1">
        <expectedValue value="500">
          <dataValue name="integer"/>
        </expectedValue>
      </assertion>
      <assertion name="amountCash" assertionType="Equals"
method="/4/@testCase.1/@testClass.0/@method.0">
        <expectedValue value="500">
          <dataValue name="integer"/>
        </expectedValue>
      </assertion>
      <testClass name="CashDispenser">
        <method assertion="/4/@testCase.1/@assertion.1"
name="checkCashOnHand">
          <parameter value="500"/>
        </method>
        <method assertion="/4/@testCase.1/@assertion.0"
name="dispenseCash">
          <parameter value="500"/>
        </method>
        <attribute value="500" name="cashOnHand"/>
      </testClass>
    </testCase>
  </nunitmodel:TestSuite>
  <nunitmodel:TestSuite name="TestTransfer">
    <testCase name="TestTransferCustomerConsole">
      <assertion name="transactionTransfer" assertionType="Equals"
method="/5/@testCase.0/@testClass.0/@method.0">
        <expectedValue value="500">
          <dataValue name="integer"/>
        </expectedValue>
      </assertion>
      <testClass name="CustomerConsole">
        <method assertion="/5/@testCase.0/@assertion.0"
name="completTransactionTransfer">
          <parameter value="500"/>
        </method>
        <attribute value="A" name="from"/>
```

```
        <attribute value="500" name="amount"/>
        <attribute value="B" name="to"/>
      </testClass>
    </testCase>
    <testCase name="TestTransferCashDispenser">
      <assertion name="DisponibleCash" assertionType="Equals"
method="/5/@testCase.1/@testClass.0/@method.1">
        <expectedValue value="500">
          <dataValue name="integer"/>
        </expectedValue>
      </assertion>
      <assertion name="amountCash" assertionType="Equals"
method="/5/@testCase.1/@testClass.0/@method.0">
        <expectedValue value="500">
          <dataValue name="integer"/>
        </expectedValue>
      </assertion>
      <testClass name="CashDispenser">
        <method assertion="/5/@testCase.1/@assertion.1"
name="checkCashOnHand">
          <parameter value="500"/>
        </method>
        <method assertion="/5/@testCase.1/@assertion.0"
name="dispenseCash">
          <parameter value="500"/>
        </method>
        <attribute value="500" name="cashOnHand"/>
      </testClass>
    </testCase>
    <testCase name="TestTransferPrinter" precondition="">
      <assertion name="printer" assertionType="True"
method="/5/@testCase.2/@testClass.0/@method.0">
        <expectedValue value="true">
          <dataValue name="boolean"/>
        </expectedValue>
      </assertion>
      <testClass name="Printer">
        <method assertion="/5/@testCase.2/@assertion.0" name="print">
          <parameter value="printer"/>
        </method>
      </testClass>
    </testCase>
  </nunitmodel:TestSuite>
</xmi:XMI>
```

ANEXO M - Definição de Transformação (modelo-a-texto) MPST-NUnit a Código-Fonte de Teste NUnit

```
query modelNUnit2SourceCode = nunitmodel!NamedElement.allInstances()->
      select(e | e.oclIsTypeOf(nunitmodel!TestSuite))->
    collect(x | x.toString().writeTo('C:/SourceCode/NUnit/' + x.name +
'.cs')); -- Query Template

uses modelNUnit2SourceCode;


                                    ***


library modelNUnit2SourceCode; -- Library Template

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

helper context nunitmodel!Assertion def: getNameAssertion() : String =
      if self.assertionType='Equals' then 'Assert.AreEquals'
      else if self.assertionType='True' then 'Assert.IsTrue'
      else if self.assertionType='False' then 'Assert.IsFalse'
      else if self.assertionType='NotNull' then 'Assert.IsNotNull'
      else if self.assertionType='AreSame' then 'Assertion.AreSame'
      else if self.assertionType='Fail' then 'Assertion.Fail'
      else 'undefinedAssertionType' endif endif endif endif endif endif;

helper context nunitmodel!Assertion def: toString() : String =
       '\t\t\t\t'+ self.getNameAssertion() +
       '(' + self.expectedValue.value +
       ', '+ self.method.callsMethod() +
       ');\n';

helper context nunitmodel!TestSuite  def: toString() : String =
      'using "name"; \n' +
      'using NUnit.Framework; \n\n' +
      'namespace "name" \n' +
      '{\n\n' +
      '\t[TestFixure]\n' +
      '\tpublic class ' + self.name + '{\n\n' +
      self.testCase->iterate (i; acc:String=''| acc +
      '\t\t\t[Test]\n' +
      '\t\t\tpublic void '+ i.name+ '(){\n\n'+
            i.assertion->iterate (i; acc:String=''| acc + i.toString()) +
       '\t\t\t}\n\n')   +
      '\t\t}\n' +
      '}\n';
```

ANEXO N - Código-Fonte de Teste para NUnit

```
using System ATM;
using NUnit.Framework;

namespace System ATM
{

      [TestFixure]
      public class TestSession{

                  [Test]
                  public void TestSessionCustomerConsole(){

                        Assert.IsTrue(true, new
Session().performSession(true));
                        Assert.IsFalse(false, new
Session().performSession(true));
                        Assert.IsFalse(false, new
Session().performSession(true));
                  }

                  [Test]
                  public void TestSessionCardRead(){

                        Assert.IsTrue(true, new
CardRead().initiateSession(true));
                        Assert.IsFalse(false, new
CardRead().VerifyCard(false));
                        Assert.IsFalse(false, new
CardRead().VerifyCard(false));
                        Assert.IsTrue(true, new
CardRead().VerifyCard(false));
                        Assert.IsFalse(false, new
CardRead().VerifyCard(false));
                        Assert.IsTrue(true, new
CardRead().VerifyCard(false));
                  }

                  [Test]
                  public void TestSessionTransaction(){

                        Assert.IsTrue(true, new
Transaction().makeTransaction(true));
                        Assert.IsTrue(true, new
Transaction().readPIN(true));
                        Assert.IsFalse(false, new
Transaction().peformInvalidPinExtension(false));
                        Assert.IsFalse(false, new
Transaction().peformInvalidPinExtension(false));
                  }
            }
}
```

***

```
using System ATM;
using NUnit.Framework;

namespace System ATM
```

```
{

      [TestFixure]
      public class TestTransfer{

                  [Test]
                  public void TestTransferCustomerConsole(){

                        Assert.AreEquals(500, new
CustomerConsole().completTransactionTransfer(500));
                  }

                  [Test]
                  public void TestTransferCashDispenser(){

                        Assert.AreEquals(500, new
CashDispenser().dispenseCash(500));
                        Assert.AreEquals(500, new
CashDispenser().checkCashOnHand(500));
                  }

                  [Test]
                  public void TestTransferPrinter(){

                        Assert.IsTrue(true, new Printer().print(true));
                  }
            }
}
```

***

```
using System ATM;
using NUnit.Framework;

namespace System ATM
{

      [TestFixure]
      public class TestInquiry{

                  [Test]
                  public void TestInquiryCustomerConsole(){

                        Assert.IsTrue(true, new
CustomerConsole().completTransactionInquiry(true));
                  }

                  [Test]
                  public void TestInquiryPrinter(){

                        Assert.IsTrue(true, new Printer().print(true));
                  }
            }
}
```

***

```
using System ATM;
using NUnit.Framework;

namespace System ATM
```

```
{

      [TestFixure]
      public class TestDeposit{

                  [Test]
                  public void TestDepositCustomerConsole(){

                        Assert.AreEquals(1000, new
CustomerConsole().ReadAmount(1000));
                        Assert.AreEquals(deposit, new
CustomerConsole().ReadMenuChoice(deposit));
                        Assert.IsTrue(true, new
CustomerConsole().completTransactionDeposit(true));
                  }

                  [Test]
                  public void TestDepositCashDispenser(){

                        Assert.IsTrue(true, new
CashDispenser().dispenseCash(true));
                        Assert.AreEquals(1000, new
CashDispenser().checkCashOnHand(1000));
                  }

                  [Test]
                  public void TestDepositPrinter(){

                        Assert.IsTrue(true, new Printer().print(true));
                  }
            }
}
```

***

```
using System ATM;
using NUnit.Framework;

namespace System ATM
{

      [TestFixure]
      public class TestWithdrawal{

                  [Test]
                  public void TestWithdrawalCustomerConsole(){

                        Assert.AreEquals(500, new
CustomerConsole().completTransactionWithdrawal(500));
                  }

                  [Test]
                  public void TestWithdrawalCashDispenser(){

                        Assert.AreEquals(500, new
CashDispenser().dispenseCash(500));
                        Assert.AreEquals(500, new
CashDispenser().checkCashOnHand(500));
                  }
            }
}
```