UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

MARCELO ZAMBIASI

NovaStudio: Geração de Testes a partir de uma Modelagem UML

Trabalho de Graduação.

Prof. Dr. Cláudio Fernando Resin Geyer
Orientador

Porto Alegre, dezembro de 2010.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Zambiasi, Marcelo

NovaStudio: Geração de Testes a partir de uma Modelagem UML / Marcelo Zambiasi – Porto Alegre, 2010

79 f.: il.

Trabalho de Graduação (conclusão) – Universidade Federal do Rio Grande do Sul, Curso de Bacharelado em Ciência da Computação, Porto Alegre, BR, 2010.

Orientador: Prof. Dr. Cláudio Fernando Resin Geyer

1. Engenharia Dirigida por Modelos. 2. Arquitetura Dirigida por Modelos. 3. Teste Dirigido por Modelos. I. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. José Carlos Ferraz Hennemann
Vice-reitor: Prof. Rui Vicente Oppermann
Pró-Reitor de Graduação: Prof. Valquiria Link Bassani
Diretor do Instituto de Informática: Prof. Flávio Rech Wagner
Coordenador do Curso: Prof. João César Netto
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

“ You can never solve a problem on the
level on which it was created. ”

ALBERT EINSTEIN (1879-1955)

AGRADECIMENTOS

Primeiramente, gostaria de agradecer à minha família por todo apoio que me foi dado não somente nesta jornada de estudos, mas também em toda minha vida. Ao meu pai e a minha mãe pela atenção, a dedicação, o carinho e a busca de me proporcionarem sempre a melhor educação possível. Ao meu irmão por sua amizade e seu companheirismo para festas e viagens.

Agradeço à Universidade Federal do Rio Grande Sul, em especial ao corpo docente e funcionários do Instituto de Informática, pelo altíssimo nível de qualidade de ensino oferecido. Foi um grande privilégio ter estudado em um dos melhores cursos de Ciência da Computação de todo o Brasil. Um agradecimento especial a todos os professores do projeto SCISTEMA-UFRGS e a CAPES que me proporcionaram a chance de estudar por dois anos na França, através de um convênio de dupla-diplomação com o Grenoble INP-Ensimag.

Gostaria de agradecer aos funcionários da empresa BULL por terem me acolhido durante seis meses de estágio, o qual resultou neste trabalho. Em especial, eu agradeço ao Eng. Cédric Tran-Xuan, responsável por meu estágio, e ao Eng. Emmanuel Rias, chefe da equipe NovaStudio, por seus conselhos para a realização deste trabalho. Também agradeço ao Prof. Dr. Philippe Lalanda da Ensimag pela orientação deste trabalho na sua versão francesa.

Gostaria de expressar minha gratidão ao Prof. Dr. Cláudio Geyer pela orientação deste trabalho, além de todo o apoio concedido durante os dois anos que estive na França.

Agradeço a todos meus ex-colegas de trabalho da BULL, da equipe Adèle e do PET Computação UFRGS, em especial ao tutor do grupo Prof. Dr. Dante Barone, pelos momentos de aprendizado e também pelos momentos de descontração.

Não poderia deixar de agradecer a todos meus amigos pelas festas, conselhos, conversas, viagens, risadas e pelo apoio ou compreensão durante estes anos de estudo. Em especial, agradeço a todos meus amigos de Grenoble por termos constituído uma verdadeira família lá, ajudando a suprir a distância das nossas.

Por fim, gostaria de agradecer a todos que contribuíram de alguma forma para a realização deste trabalho ou para a minha formação como bacharel em Ciência da Computação.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS..8

LISTA DE FIGURAS...9

LISTA DE TABELAS...11

RESUMO...12

ABSTRACT...13

1.INTRODUÇÃO..14

1.1.Objetivos deste trabalho..15

1.2.Organização do documento..15

2.ENGENHARIA DIRIGIDA POR MODELOS..17

2.1.Introdução..17

2.2.Principais conceitos da Engenharia Dirigida por Modelos...18

2.3.Arquitetura Dirigida por Modelos (MDA)..21

2.3.1.Pontos de vista e modelos da MDA...22

2.4.Os Benefícios da Engenharia Dirigida por Modelos...23

2.4.1.Vantagens da Arquitetura Dirigida por Modelos..24

2.5.Considerações finais..24

3.TESTE DIRIGIDO POR MODELOS...25

3.1.Introdução..25

3.2.Teste de Software...26

3.2.1.Fases de teste...26

3.2.2.Técnicas de teste...27

3.3.Abordagens de teste baseadas em modelos...28

3.3.1.Teste Baseado em Modelos..28

3.3.2.Teste Dirigido por Modelos..29

3.4.Meta-modelagem de testes com UML 30
3.4.1.Meta-modelagem de testes com diagramas de sequência UML............................30
3.4.2.Meta-modelagem de testes com diagramas de estado UML.................................31
3.4.3.Comparação entre a utilização de diagramas de sequência e de estado................31
3.5.Considerações finais...32
4.NOVASTUDIO...33
4.1.Introdução...33
4.2.Funcionalidades 34
4.3.Concepção e desenvolvimento..34
4.4.Arquitetura lógica do Java EE...35
4.4.1.Subcamadas Business Service e Business Object...36
4.4.2.Interação entre as camadas...36
4.5.Implementação da arquitetura lógica..37
4.6.Motor de geração de código..38
4.7.Considerações finais...39
5.MODELAGEM DE TESTES...40
5.1.Escolha da modelagem...40
5.2.Diagramas de sequência...41
5.2.1.Nomenclatura dos diagramas...41
5.2.2.Diagramas de sequência de inicialização...42
5.2.3.Diagramas de sequência com asserção direta..44
5.2.4.Diagramas de sequência com asserção indireta...45
5.2.5.Definição das mensagens UML...46
5.3.Considerações finais...47
6.REALIZAÇÃO...48
6.1.Framework para execução dos testes...48
6.1.1.JUnit...48
6.1.2.Framework NovaStudio Java EE – Ejb3Unit...49
6.1.3.Framework NovaStudio WEB – JUnit..50
6.2.Adaptação de NovaStudio para o suporte do meta-modelo de testes............................51
6.2.1.Arquitetura de NovaStudio...51
6.2.2.Parser XMI...53
6.2.3.Modificação do meta-modelo de NovaStudio...53
6.2.4.Povoamento do meta-modelo..55
6.3.Geração de testes utilizando Acceleo..56
6.3.1.Geração de templates com Acceleo..56
6.3.2.Geração do código de testes..57
6.4.Desenvolvimento do plugin TestGeneration...58
6.5.Documentação do projeto...59
6.6.Considerações finais...59

7.RESULTADOS......60
7.1.Cenário de teste......60
7.2.Diagrama de sequência de inicialização......62
7.2.1.Estrutura do teste......63
7.2.2.Métodos setUp e tearDown......65
7.3.Diagramas de sequência para o teste de métodos......69
7.3.1.Método isSingle......69
7.3.2.Método findAllMarried......70
7.3.3.Método becomeOlder......71
7.4.Código gerado para a camada Business Object......72
7.5.Considerações finais......73
8.CONCLUSÃO......74
REFERÊNCIAS......76

LISTA DE ABREVIATURAS E SIGLAS

CIM	Computation Independent Model
DSML	Domain-Specific Modeling Languages
EJB	Enterprise JavaBeans
EMF	Eclipse Modeling Framework
GPML	General-Purpose Modeling Language
IEEE	Institute of Eletrical and Eletronics Engineers
J2EE	Java 2 Platform, Enterprise Edition
Java EE	Java Enterprise Edition
MBT	Model-Based Testing
MDA	Model-Driven Architecture
MDD	Model-Driven Developement
MDE	Model-Driven Engineering
MDT	Model-Driven Testing
MOF	Meta-Object Facility
MVC	Model-View-Controller
OCL	Object Constraint Language
OMG	Object Management Group
PIM	Plataform Independent Model
POJO	Plain Old Java Object
PSM	Plataform Specific Model
UML	Uniform Modeling Language
URI	Uniform Ressource Identifier
XMI	XML Metadata Interchange
XP	Extreme Programming

LISTA DE FIGURAS

Figura 2.1. Relações RepresentaçãoDe e ConformeA..19
Figure 2.2. Hierarquia de modelos..20
Figura 2.3. Transformação de modelos..21
Figura 2.4. Arquitetura dirigida por Modelos (OMG, 2010)..22
Figura 3.1. Processo de teste da MDT (figura baseada em Dai (2004))..29
Figura 4.1. Arquitetura lógica Java EE (BULL, 2007)..35
Figura 4.2. Interação entre as camadas (BULL, 2007)..37
Figura 5.1. Convenção de nomenclatura dos diagramas de sequência..42
Figura 5.2. Exemplo de diagrama de sequência de inicialização..43
Figura 5.3. Definição de um atributo de um objeto..43
Figura 5.4. Exemplo de diagrama de sequência com asserção direta..44
Figura 5.5. Exemplo de diagrama de sequência com asserção indireta..45
Figura 5.6. Exemplo de mensagem UML..46
Figura 6.1. Arquitetura de NovaStudio..52
Figura 6.2. Classes adicionadas ao meta-modelo de NovaStudio..54
Figura 6.3. O meta-modelo EMF após as modificações..54
Figura 6.4. O arquivo .generation completo (à esquerda) e em detalhe a parte deste arquivo relativa a geração de testes (à direita)..55
Figura 6.5. Geração de código por Acceleo..56
Figura 6.6. Exemplo de um template Acceleo..57
Figura 6.7. Interface "Generation Preferences"..58
Figura 6.8. Interface "Generate Test Code"..59
Figura 7.1. Diagrama de classe para a camada Business Object..61
Figura 7.2. Diagrama de classe para a camada Business Service..61
Figura 7.3. Diagrama de sequência de inicialização da classe Man..62
Figura 7.4. Diagrama de objetos da classe Man..62
Figura 7.5. Estrutura do código de teste para a classe ManTest..65
Figura 7.6. Trecho final do código para a classe ManTest..65
Figura 7.7. Código gerado para o método setUp da classe ManTest..67
Figura 7.8. Código gerado para o método tearDown da classe ManTest..68
Figura 7.9. Diagrama de sequência para o método isSingle..69
Figura 7.10. Código gerado para o método testIsSingle da classe ManTest..70
Figura 7.11. Diagrama de sequência para o método findAllMarried..70
Figura 7.12. Código gerado para o método findAllMarried da classe ManTest..71
Figura 7.13. Diagrama de sequência para o método becomeOlder..71

Figura 7.14. Código gerado para o método becomeOlder da classe ManTest..........................71
Figura 7.15. Código gerado para a camada Business Object da classe Man............................73

LISTA DE TABELAS

Tabela 4.1. MVC vs. Camadas da Arquitetura Java EE..36
Tabela 4.2. Implementações suportadas por NovaStudio..38

RESUMO

O processo de desenvolvimento de software é um dos processos mais complexos já realizados pelo homem e a tendência é que os sistemas de informação fiquem ainda maiores e mais complexos. Por causa desta complexidade, os sistemas de informação estão sujeitos a erros e por isso existe a necessidade de testá-los. Entretanto, o tempo para testar um sistema não pode ser negligenciado e pode representar mais da metade do tempo total de um projeto.

Com isso, surgem as necessidades do aumento da automatização e do nível de abstração no desenvolvimento e no teste de software, permitindo assim custos mais baixos nestes processos. Essas necessidades são respondidas pela utilização das abordagens da Engenharia Dirigida por Modelos e do Teste Dirigido por Modelos.

Neste contexto, a empresa francesa BULL S.A.S desenvolve o produto NovaStudio, um ambiente de desenvolvimento que permite gerar uma boa parte do código de aplicações Java a partir de uma modelagem UML. NovaStudio apoia-se na abordagem da Arquitetura Dirigida por Modelos, uma das principais variantes da Engenharia Dirigida por Modelos.

Este trabalho adiciona uma nova funcionalidade a NovaStudio, permitindo a geração automatizada do código de testes de uma aplicação. Os testes são gerados a partir de uma modelagem UML, seguindo os princípios do Teste Dirigido por Modelos. O meta-modelo proposto por este trabalho é baseado principalmente em diagramas de sequência.

Palavras-chave: Engenharia Dirigida por Modelos, Arquitetura Dirigida por Modelos, Teste Baseado em Modelos, Teste Dirigido por Modelos, NovaStudio, UML.

NovaStudio: Test Generation from an UML Modeling

ABSTRACT

Software development is one of the most complex processes ever conducted by man, and the trend is that information systems will become even larger and more complex. As result of this complexity, information systems may have errors and, therefore, there is a need to test them. However, the time to test a system can not be overlooked and may represent more than half of a project's total time.

So, there is a need for increasing the level of abstraction and automation in the software development and testing processes, which could lower costs. These needs are answered by using the approaches of Model-Driven Engineering and Model-Driven Test.

In this context, the french company BULL S.A.S develops the product NovaStudio. The development environment NovaStudio allows the generation of a good portion of the code for Java applications from an UML modeling. NovaStudio relies on Model-Driven Architecture approach, one of the main variants of Model-Driven Engineering.

This work adds a new feature to NovaStudio allowing the automated test code generation. Tests are generated from an UML modeling, following the principles of Model-Driven Test. The meta-model proposed by this wok is mainly based on sequence diagrams.

Keywords: Model-Driven Engineering, Model-Driven Architecture, Model-Driven Test, Model-Based Test, NovaStudio, UML.

1. INTRODUÇÃO

O processo de desenvolvimento de software é um dos processos mais complexos já realizados pelo homem. A tendência é que os sistemas de informação fiquem cada vez maiores e mais complexos, tornando assim o seu desenvolvimento ainda mais difícil a ser realizado (SELIC, 2006). Esse fenômeno de aumento de complexidade, além da demanda de software, não é novo na computação. Ele é conhecido como a crise do software.

A crise de software, no final dos anos 60, motivou a criação da Engenharia de Software que têm como um de seus principais objetivos a automatização do processo de desenvolvimento de software, permitindo assim contornar (gerenciar) o crescimento de complexidade deste processo. Além disso, com a automatização, é possível obter-se um ganho de produtividade e de qualidade no processo de desenvolvimento de software, já que o software é gerado através de técnicas automatizadas e não mais através da codagem manual por desenvolvedores.

Para contornar o aumento de complexidade e permitir a automatização do processo de desenvolvimento de software, a Engenharia Dirigida por Modelos se apresenta como uma possível solução. Esta abordagem de desenvolvimento de software se baseia em modelos para aumentar o nível de abstração do processo de desenvolvimento, além de permitir a automatização de tarefas, tal como a geração de código-fonte.

Uma outra grande preocupação da Engenharia de Software é a qualidade dos softwares que são produzidos, já que os sistemas de informação de boa qualidade são uma exceção: a norma são os sistemas defeituosos (WHITTAKER; VOAS, 2002). No intuito de obter-se sistemas com uma melhor qualidade (menos erros), uma das técnicas mais empregadas é o Teste de Software. Entretanto, o tempo necessário para testar um sistema não pode ser negligenciado: ele é estimado em 50% ou mais do tempo total de um projeto de software (BAO-LIN ET AL., 2007).

Com o objetivo de automatizar a geração de testes e assim reduzir os custos relacionados ao teste, a abordagem de Teste Dirigido por Modelos foi criada. Essa abordagem realiza a união entre as técnicas de Teste de Software e os princípios da Engenharia Dirigida por Modelos.

1.1. Objetivos deste trabalho

O contexto técnico deste trabalho é o ambiente de desenvolvimento NovaStudio, da empresa francesa BULL S.A.S. Este ambiente permite gerar uma boa parte do código de aplicações Java (J2EE) a partir de uma modelagem UML (*Unified Modeling Language*). NovaStudio utiliza a abordagem da Arquitetura Dirigida por Modelos, uma das principais variantes da Engenharia Dirigida por Modelos.

A equipe de NovaStudio deseja adicionar uma nova funcionalidade ao seu ambiente de desenvolvimento que permita a geração automatizada de testes. Portanto, é desejado que NovaStudio gere além do código de uma aplicação, também o seu código de testes. Os testes devem ser gerados a partir de uma modelagem UML, seguindo os princípios da abordagem de Teste Dirigido por Modelos (MDT).

O objetivo deste trabalho de graduação, portanto, consiste na geração automatizada de testes para o ambiente de desenvolvimento NovaStudio. Este objetivo principal pode ser dividido nas seguintes tarefas:

- estudo das técnicas de Engenharia Dirigida por Modelos e a sua principal variante, a Arquitetura Dirigida por Modelos.
- estudo das técnicas de Teste de Software e da abordagem de Teste Dirigido por Modelos, focando-se principalmente nas técnicas de modelagem de testes através da linguagem de modelagem UML.
- definição de um meta-modelo de testes para NovaStudio, que servirá como base para a geração do código de testes, e adaptação deste ambiente de desenvolvimento aos novos elementos UML adicionados por esta modelagem de testes.
- escolha de um *framework* para execução dos testes, sendo que os testes gerados por NovaStudio serão basicamente unitários e escritos na linguagem Java. Há um grande interesse por parte da equipe NovaStudio que estes testes sejam executados usando-se o conhecido *framework* de testes JUnit ou algum *framework* derivado deste.
- desenvolvimento de *templates* para a geração dos testes, baseados no Acceleo, que é o atual motor de geração de código de NovaStudio.
- integração completa e validação da nova funcionalidade de geração de testes a NovaStudio, além da escrita da documentação deste projeto.

1.2. Organização do documento

O restante deste trabalho encontra-se organizado da seguinte forma:

- O capítulo 2 descreve a Engenharia Dirigida por Modelos, além da sua principal variante a Arquitetura Dirigida por Modelos. São apresentados as principais características destas abordagens, além das vantagens da utilização de ambas.
- O capítulo 3 apresenta a abordagem de Teste Dirigido por Modelos. Neste capítulo são também introduzidos os principais conceitos de Teste de Software, além da abordagem de Teste Baseado em Modelos que originou o Teste Dirigido por Modelos. No final

deste capítulo, alguns trabalhos que utilizam modelos UML para a geração de testes são discutidos.

- O capítulo 4 introduz o ambiente de desenvolvimento NovaStudio e as suas principais características.
- O capítulo 5 apresenta o meta-modelo de testes proposto por este trabalho, que é baseado em diagramas de sequência e objetos UML.
- O capítulo 6 descreve como foi feita a realização, a implementação e a integração da funcionalidade de geração de testes ao ambiente de desenvolvimento NovaStudio,
- O capítulo 7 descreve os resultados deste trabalho através da apresentação de um caso de teste, utilizado para a validação da funcionalidade de geração de teste, com o respectivo código gerado por NovaStudio.
- O capítulo 8 apresenta as conclusões deste trabalho e as perspectivas de trabalhos futuros para o ambiente de desenvolvimento NovaStudio.

2. ENGENHARIA DIRIGIDA POR MODELOS

Este capítulo descreve a Engenharia Dirigida por Modelos (**MDE**). A primeira seção introduz essa abordagem de desenvolvimento de software. Em seguida, são apresentados os principais conceitos em torno da MDE. A terceira seção deste capítulo é dedicada a Arquitetura Dirigida por Modelos (**MDA**), variante do MDE proposta pelo OMG, que é a abordagem de desenvolvimento utilizada por NovaStudio. Os benefícios da utilização do MDE e mais particularmente do MDA são discutidos na quarta seção deste capítulo.

2.1. Introdução

O principal problema enfrentado pelos desenvolvedores de software é a complexidade deste processo, já que os sistemas de softwares estão entre os mais complexos já construídos pelo homem e a tendência é que fiquem cada vez mais complexos (SELIC, 2006). Esse fenômeno de aumento da complexidade dos sistemas, além da demanda de software, não é novo na computação. Ele ficou conhecido como a crise do software, que é discutida no célebre artigo "*The Humble Programmer*" de Djisktra (1972), e foi um dos motivadores para a criação da Engenharia de Software no final dos anos 1960. O termo Engenharia de Software apareceu pela primeira vez durante a conferência NATO (*North Atlantic Treat Organization*), em 1968, sendo definido como "o estabelecimento e uso de sólidos princípios de engenharia para obter software economicamente confiável e que trabalhe de forma eficiente em máquinas reais" (NAUR; RANDELL, 1969).

Na área de Engenharia de Software, muitas iniciativas, como metodologias, plataformas e tecnologias, surgiram com o intuito de auxiliar os desenvolvedores de software a gerenciar esse crescimento de complexidade. Entre essas iniciativas, podemos citar o aparecimento das primeiras linguagens de alto nível na década de 1950, como Fortran, a criação das linguagens orientadas a objetos, como Java, a utilização de padrões de projeto (GAMMA ET AL., 1994) e a eXtreme Programming (BECK, 1999). Mais recentemente, no início dos anos 2000, uma nova abordagem de desenvolvimento de software centrada na utilização de modelos foi proposta pela OMG (*Object Management Group*), através da Arquitetura Dirigida por Modelos (**MDA** – *Model-Driven Architecture*) (OMG, 2003). O propósito dessa iniciativa é de enfrentar a crescente complexidade no processo de criação e evolução de sistemas de software. A Arquitetura Dirigida por Modelos foi generalizada pela comunidade acadêmica de Engenharia de Software, criando-se assim a Engenharia Dirigida

por Modelos (**MDE** – *Model-Driven Engineering*), cuja primeira aparição do termo foi feita por Kent (2002).

A Engenharia Dirigida por Modelos é uma abordagem de desenvolvimento de software que se baseia em modelos para aumentar o nível de abstração do processo de desenvolvimento e também automatizar algumas de suas tarefas, notadamente a geração de código-fonte. A MDE pretende, através do aumento do nível de abstração e da automatização, contornar a crescente complexidade das plataformas, que já não é mais mascarada pelas linguagens orientadas a objeto (SCHIMIDT, 2006), além da obtenção de um aumento na produtividade do desenvolvimento de software (ATKINSON; KUHNE, 2003).

Para realizar esses objetivos, a MDE combina (SCHIMIDT, 2006):

- **Linguagens de modelagem específicas de domínio** (***Domain-specific modeling languages*** **– DSML**): estas linguagens formalizam a estrutura, o comportamento e os requisitos de uma aplicação num domínio particular (específico). As DSMLs são descritas usando-se meta-modelos que definem as relações entre os conceitos no domínio particular e especificam precisamente a semântica e as restrições associadas a estes conceitos (SCHIMIDT, 2006). As DSMLs se contrapõem às linguagens de modelagem de propósito generalista (**GPML** – *General-purpose modeling language*), como UML (*Unified Modeling Language*) (OMG, 2007), que podem ser aplicadas a múltiplos domínios.
- **Motores de transformação e geradores**: estes elementos são utilizados para a síntese de diversos artefatos, como código-fonte da aplicação, descritores de implantação XML, etc. Este processo automático de transformação de modelos em artefatos ajuda a assegurar a coerência entre a especificação (modelos) e a implementação (artefatos gerados, como o código-fonte). Este processo é conhecido como "correto por construção" e contrasta com o método mais convencional de desenvolvimento de software "construir por correção", mais propenso a erros já que é manual (SCHIMIDT, 2006).

Um termo utilizado como sinônimo para a MDE é o Desenvolvimento Dirigido por Modelos (**MDD** – *Model-Driven Developement*), que é definido como a abordagem de desenvolvimento de software aonde os modelos se tornam os elementos essenciais do processo de desenvolvimento (SELIC, 2006). Então podemos ver uma mudança de perspectiva na utilização dos modelos. Na MDE, os modelos não são mais usados como simples elementos documentação pelos programadores, que serão "perdidos" após a escrita do código. Pelo contrário, eles adquirem um importância central no processo de desenvolvimento e é através da criação e transformação de modelos que a aplicação final será gerada.

Como a Engenharia Dirigida por Modelos é uma abordagem muito recente, não existe ainda um consenso em torno suas definições. Na próxima seção, são apresentados os principais conceitos da MDE.

2.2. Principais conceitos da Engenharia Dirigida por Modelos

O principal conceito da MDE é o modelo, que pode ser definido como uma abstração de um sistema criado para um objetivo específico. Ele deve ser capaz de responder a questões no lugar do sistema real que ele modeliza e, para ser útil, deve ser mais simples que este último (BÉZIVIN; GERBÉ, 2001). Um modelo pode ser utilizado para a descrição de um

sistema em estudo (*system under study*) (SEIDEWITZ, 2003). Então, podemos ver um modelo como uma abstração permitindo o estudo de um conceito num contexto mais simples que o contexto real.

Um segundo conceito muito importante da MDE é o meta-modelo, que pode ser definido de modo simplista como "um modelo de modelos" (OMG, 2001). O meta-modelo tem como objetivo definir a semântica de um modelo. Ele é um modelo de uma linguagem de modelagem e define esse linguagem de modo explícito (BÉZIVIN ET AL., 2004). Um meta-modelo define um conjunto de conceitos e as relações entre esses conceitos, podendo ser utilizado como um filtro de abstração numa atividade de modelagem, visto que um mesmo sistema pode ser representado por diversos modelos (BÉZIVIN; GERBÉ, 2001).

Assim como a abordagem orientada a objetos preconiza a visão de que "tudo é um objeto", a Engenharia Dirigida por Modelos baseia-se principalmente no princípio de que "tudo é modelo" (BÉZIVIN, 2005). Este princípio básico da MDE define duas relações que são amplamente aceitas pela comunidade da Engenharia de Software (BÉZIVIN, 2004) (BÉZIVIN ET AL., 2004)(FAVRE, 2004):

- **RepresentaçãoDe**: um modelo é a representação de um sistema. Um exemplo clássico dessa relação é um mapa, que nada mais é que a representação de um território. Este exemplo é apresentado na figura 2.1.
- **ConformeA**: um modelo é conforme a um meta-modelo. Voltando ao exemplo clássico do mapa, podemos dizer que um mapa é conforme a sua legenda (figura 2.1). Um mapa é inútil sem uma legenda, já que ela indica como interpretá-lo (escala, símbolos, etc). Portanto, um mapa é desenhado seguindo a linguagem (gráfica) definida por sua legenda.

Figura 2.1. Relações RepresentaçãoDe e ConformeA

Além de modelos e meta-modelos, um terceiro conceito existe na MDE: o meta-meta-modelo. Ele tem como objetivo definir a semântica de um meta-modelo (relação de conformidade). Um meta-meta-modelo está para meta-modelo assim como este último está para um modelo (BÉZIVIN, 2005). Para que não tenhamos infinitas "meta-camadas", um meta-meta-modelo normalmente é conforme a si mesmo. Os conceitos apresentados anteriormente, juntamente com as duas relações, formam um hierarquia clássica de quatro camadas que é mostrada na figura 2.2.

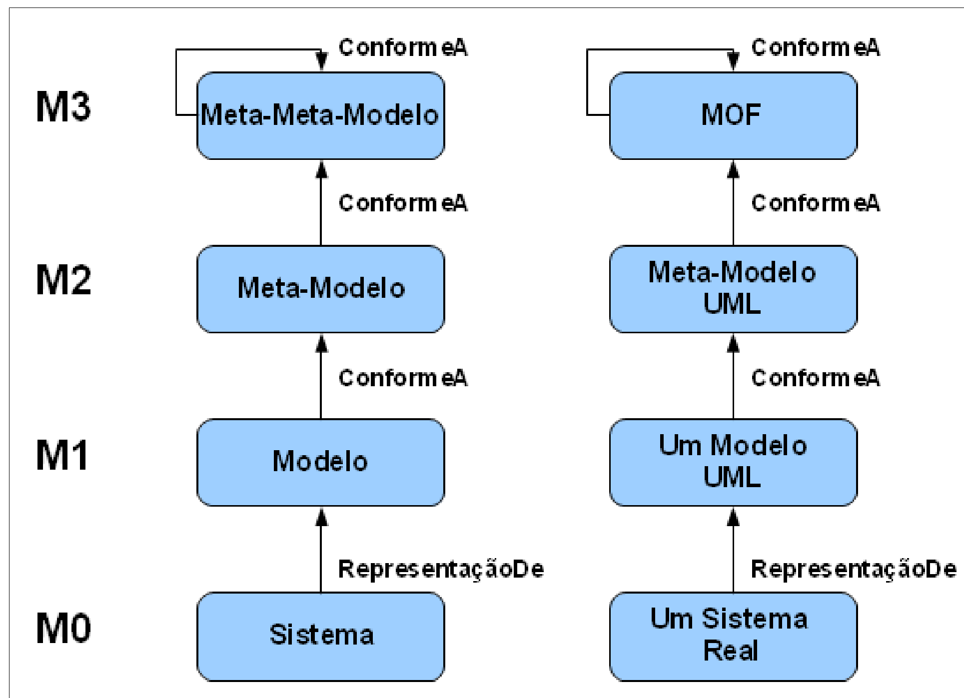

Figure 2.2. Hierarquia de modelos

No lado esquerdo da figura 2.2, podemos ver que um modelo (M1) representa um sistema (M0) e ele é conforme a um meta-modelo (M2). Por sua vez, um meta-modelo é conforme a um meta-meta-modelo (M3), sendo que este último é conforme a si mesmo. Em um exemplo mais concreto (lado direito da figura 2.2), um sistema real (M0) é representado por um modelo específico UML, linguagem de modelagem padronizada pela OMG. Este modelo UML é conforme a um meta-modelo UML (M2). Já o meta-modelo UML é conforme ao MOF (*Meta-Object Facility*) (OMG, 2006), que é o padrão da OMG para criação de meta-modelos (linguagem de meta-modelagem). O MOF é conforme a si mesmo.

Um último conceito central na Engenharia Dirigida por Modelos é a noção de transformação, tendo em vista que descrever como transformar um modelo é uma condição indispensável para torná-lo produtivo (BÉZIVIN ET AL., 2004). Na visão da MDE, os modelos são os artefatos principais do desenvolvimento de software e os desenvolvedores apoiam-se sobre tecnologias computacionais para transformá-los em sistemas executáveis (FRANCE; RUMPE, 2007).

A operação de transformação, ilustrada na figura 2.3, tem como objetivo produzir um modelo Y, conforme a um meta-modelo Y, a partir de um modelo X, conforme a um meta-modelo X. Então, as entradas dessa operação são os meta-modelos X e Y, além obviamente do modelo X que desejamos transformar. Já o resultado (saída) da operação é o modelo Y. Um exemplo dessa operação poderia ser a transformação de um modelo UML em um documento XML ou até mesmo em um outro modelo UML.

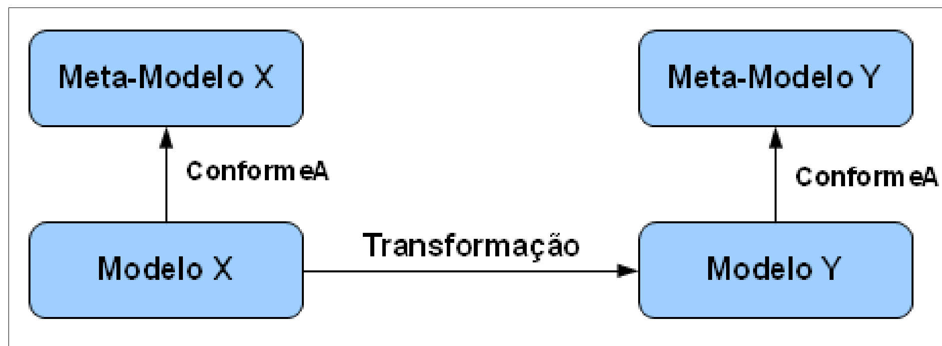

Figura 2.3. Transformação de modelos

Entretanto, mesmo que a relação **TransformaçãoEm** tenha sido introduzida por Favre (2004), ainda não existe um consenso na comunidade de Engenharia de Software nem sobre a existência dessa relação e nem sobre a exata definição da operação de transformação Por isso, as transformações são assuntos de pesquisas ativas na MDE (BÉZIVIN ET AL., 2004).

Existem muitas proposições de abordagens na área de Engenharia Dirigida por Modelos, como Fábricas de Software (*Software Factories*) (GREENFIELD ET AL., 2004), *Agile Model-Driven Developpement* (AMDD)[1] e *Domain-Oriented Programming* (THOMAS; BARRY, 2003). Na próxima seção, a principal iniciativa no contexto da MDE, que é a Arquitetura Dirigida por Modelos (MDA), será detalhada, já que esta é a abordagem de desenvolvimento utilizada por NovaStudio.

2.3. Arquitetura Dirigida por Modelos (MDA)

A primeira iniciativa e sem dúvida a mais importante até hoje no contexto das abordagens dirigidas por modelos é a Arquitetura Dirigida por Modelos (MDA), proposta pela OMG no início dos anos 2000. A MDA evolui, através da comunidade de pesquisa na área de Engenharia de Software, e foi generalizada como a Engenharia Dirigida por Modelos (MDE). Então, MDA pode ser considerado como uma variante particular da MDE (BÉZIVIN, 2004).

Os principais fundamentos da MDA são (BOOCH ET AL., 2004):

- **Representação direta** dos conceitos específicos do domínio do problema, em substituição aos conceitos dependentes de tecnologia. A redução da distância semântica entre o domínio do problema e sua representação, permite projetos mais precisos e com maior produtividade.
- **Automatização** através do uso de ferramentas computacionais que transformam modelos de domínio específico em código aplicativo. A automatização permite obter um código sem os erros que seriam introduzidos por um processo manual, além de um ganho de produtividade.
- Utilização de **padrões abertos**, já que estes ajudam a eliminar a diversidade e promovem um ecossistema de fornecedores e usuários, possibilitando o reúso de artefatos e a interoperabilidade entre as ferramentas desenvolvidas.

1 http://www.agilemodeling.com/essays/amdd.htm

A MDA parte de uma ideia bem conhecida e estabelecida de separar a especificação do sistema da forma como ele é implementado. Esta abordagem de desenvolvimento de software proposta pela OMG provê um método e as ferramentas necessárias para (OMG, 2003):

- especificar um sistema independentemente da plataforma que o suporta,
- especificar plataformas,
- escolher uma plataforma específica para um sistema, e
- transformar a especificação de um sistema para uma plataforma particular.

Os três principais objetivos da MDA são a portabilidade, a interoperabilidade e o reúso através de uma separação arquitetural das preocupações (OMG, 2003). Para atingir esse objetivos, a MDA baseia-se em padrões abertos também propostos pela OMG: MOF (*Meta Object Facility*), UML (*Unified Modeling Language*), CWM (*Common Warehouse Metamodel*) e XMI (XML *Metadata Interchange*).

A figura 2.4 apresenta uma visão clássica da Arquitetura Dirigida por Modelos, contendo os padrões citados acima e alguns exemplos de plataformas que podem ser suportadas pela MDA, como Java, .Net, etc. Além disso, alguns domínios de utilização da MDA também são mostrados, como finanças, telecomunicações, etc.

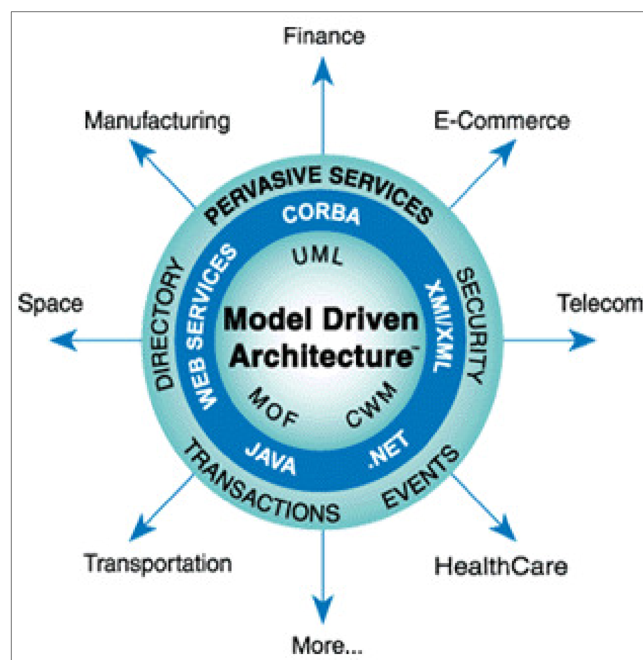

Figura 2.4. Arquitetura dirigida por Modelos (OMG, 2010)

2.3.1. Pontos de vista e modelos da MDA

Um ponto de vista (*viewpoint*) é uma abstração que permite focar num conjunto particular de aspectos, desconsiderando todos os detalhes irrelevantes. A MDA especifica três pontos de vista diferentes, além de um tipo de modelo para cada ponto de vista (OMG, 2003):

- **Ponto de vista independente de computação**: este ponto de vista foca no contexto e nos requisitos de um sistema, sem considerar sua estrutura e a forma como seu processamento será tratado. O Modelo Independente de Computação (**CIM** – *Computation Independent Model*) representa este ponto de vista. Ele é normalmente referido como um modelo de domínio ou de negócios, já que possui um vocabulário familiar com os especialistas do domínio em questão.

- **Ponto de vista independente de plataforma**: este ponto de vista foca na operação do sistema, escondendo os detalhes necessários para uma plataforma particular. Então, são abordadas neste ponto de vistas as características do sistemas que não mudam de uma plataforma para outra. Este ponto de vista é representado pelo Modelo Independente de Plataforma (**PIM** – *Plataform Independent Model*), que exibe um nível de independência suficiente para ser mapeado para uma ou várias plataformas diferentes.

- **Ponto de vista dependente de plataforma**: este ponto de vista combina o ponto de vista independente de plataforma com os detalhes necessários para especificar como um sistema utiliza uma plataforma específica. O Modelo Específico de Plataforma (**PSM** – *Plataform Specific Model*) representa este ponto de vista.

O princípio básico do funcionamento da MDA é a criação de PIMs que representam o sistema de modo independente de plataforma e a transformação destes, através de regras de transformação, em PSMs a fim de implementar o sistema. Embora essas transformações possam ser manuais, para tirarmos verdadeiro proveito da abordagem de desenvolvimento de software MDA, essas transformações (PIM → PSM) devem ser automatizadas.

2.4. Os Benefícios da Engenharia Dirigida por Modelos

A MDE baseia-se sobre dois princípios fundamentais que são os aumentos do níveis de abstração e de automação. O aumento do nível de abstração das especificações permite que estas estejam mais próximas do domínio do problema e mais longes dos detalhes de implementação. Por sua vez, o aumento do nível de automação permite contornar a lacuna semântica (*semantic gap*) entre a especificação (modelo) e a implementação (código gerado), já que este última será gerada de forma automatizada a partir do modelo (SELIC, 2006).

Os resultados práticos destes dois princípios fundamentais são: um produto (software) de melhor qualidade e um aumento no nível de produtividade (SELIC, 2006).

Como já discutido neste capítulo, o processo automatizado de transformação de modelos em artefatos ajuda a assegurar a coerência entre a especificação (modelos) e a implementação (código-fonte) (SCHIMIDT, 2006). Então, pode-se esperar como resultado um código de melhor qualidade (com presença de menos *bugs*) e que respeita a sua especificação. Com relação à eficiência deste código gerado em termos de utilização de memória e processamento, ele é similar a um código escrito manualmente com uma variação de 5% a 15% para melhor ou pior (SELIC, 2006). Além disso, a fim de permitir uma detecção de erros o mais cedo possível e assim reduzir os custos associados a correções, os modelos podem ser verificados (*model checking*) (SCHIMIDT, 2006). Essa verificação pode ser feita através da utilização de métodos matemáticos formais (SELIC, 2006).

Aumentando-se o nível de abstração, o esforço de desenvolvimento (tempo) e a complexidade do artefatos que são utilizados pelo desenvolvedor são reduzidos (HAILPERN;

TARR, 2006). Isto permite um aumento do nível de produtividade. Este aumento é principalmente ligado a automatização, já que a escrita do código-fonte pode ser feita de modo automático através de transformações de modelos.

2.4.1. Vantagens da Arquitetura Dirigida por Modelos

A MDA aporta muitos benefícios no desenvolvimento de sistemas de informação. Talvez a principal vantagem da MDA é a independência de plataforma: a partir de um PIM podemos implementar um sistema em diferentes plataformas, através de transformações (PIM → PSM). Com isto, há também um ganho de interoperabilidade, já que diversas plataformas podem ser suportados por um sistema.

Uma outra grande vantagem da MDA é o ganho de produtividade, já que a implementação pode ser obtida através de transformações automatizadas. Além disso, esta implementação, por ser gerada de forma automatizada, possui uma melhor qualidade, tendo em vista que ela é coerente com a especificação do sistema e erros não serão adicionados pela escrita manual de código.

Além destes principais benefícios, alguns outros, segundo Duby (2003), são:

- **Reação rápida a mudanças de requisitos funcionais e de plataformas:** a separação entre o PIM e o PSM permite reagir rapidamente a mudanças nos requisitos do ambiente de execução sem modificar o PIM. Por outro lado, se algum requisito funcional do programa for mudado, basta que o PIM seja modificado.
- **Aumento da longevidade do sistema:** como as funcionalidades e arquitetura do sistema são definidas separadamente na MDA, mudanças radicais podem ocorrer nestas duas, independentemente uma da outra, o que permite de estender o tempo de vida dos sistemas que utilizam MDA.
- **Custos de manutenção mais baixos:** Como o código-fonte é gerado a partir de modelos, estes dois elementos estão sincronizados. Então a manutenção pode ser feita sobre os modelos, que possuem um nível de abstração mais alto que os códigos-fonte e, portanto, sua manipulação é mais rápida.

2.5. Considerações finais

Neste capítulo, a Engenharia Dirigida por Modelos (MDE) foi apresentada como uma abordagem de desenvolvimento de software promissora a fim de contornar a crescente complexidade dos sistemas de software. Para alcançar este objetivo, a MDE visa aumentar o nível de abstração e automatizar certas tarefas, como o desenvolvimento de código, aumentando também assim o nível de produtividade e de qualidade do produto (software) gerado.

Experiências com a MDE na última década indicam que as tecnologias em torno dela já atingiram um nível de maturidade ao qual as permite de serem aplicadas com sucesso em projetos industriais de grande escala, obtendo-se aumentos nos níveis de qualidade e produtividade. No entanto, a MDE ainda está em fase inicial e muitas melhorias nos campos práticos e teóricos devem ocorrer antes que a MDE transforme-se em uma abordagem de desenvolvimento dominante (SELIC, 2006).

3. TESTE DIRIGIDO POR MODELOS

Este capítulo apresenta a abordagem de Teste Dirigido por Modelos. A primeira seção deste capítulo introduz esta abordagem, que combina a Engenharia Dirigida por Modelos com o teste de software, além de sua motivação. Em seguida, são apresentados os principais conceitos de teste de software. A terceira seção deste capítulo apresenta as abordagens de teste baseado em modelos, descrevendo o Teste Baseado em Modelos e o Teste Dirigido por Modelos. A quarta seção deste capítulo discute alguns trabalhos que utilizam modelos UML para gerar testes, já que uma modelagem deste tipo será proposta neste trabalho.

3.1. Introdução

Como discutido no capítulo anterior, os sistemas de softwares estão entre os mais complexos já construídos pelo homem e a tendência é que fiquem cada vez mais complexos (SELIC, 2006). Esse aumento de complexidade é causado, por exemplo, 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). Por causa de toda essa complexidade, a criação de software confiável e de boa qualidade é dificultada, implicando muitas vezes em projetos de desenvolvimento de software mal-sucedidos.

Uma das grandes preocupações da Engenharia de Software é a qualidade dos softwares que são produzidos, já que os sistemas de informação de boa qualidade são uma exceção: a norma são os sistemas defeituosos (WHITTAKER; VOAS, 2002). Para obter-se sistemas com uma melhor qualidade, uma das técnicas mais empregadas na Engenharia de Software é o Teste de Software. Quanto mais cedo começa-se a testar um sistema, mais cedo seus erros serão detectados e corrigidos, além de que menor será o custo associado a essas correções.

Entretanto, o tempo dispensado para testar um sistema não pode ser negligenciado: ele é estimado em 50% ou mais do tempo total de um projeto de software (BAO-LIN ET AL., 2007). Para automatizar a geração de testes e assim reduzir os custos de projetos relacionados, a abordagem de Teste Dirigido por Modelos (**MDT** – *Model-Driven Testing*) foi criada. Essa abordagem, que une o Teste de Software com a Engenharia Dirigida por Modelos, pode ser vista como uma evolução da abordagem de Teste Baseado em Modelos (**MBT** – *Model-Based Testing*).

Neste capítulo, são estudadas as abordagens de Teste Dirigido por Modelos e Baseado em Modelos. Mas, antes disso, são introduzidos na próxima seção alguns conceitos fundamentais de Teste de Software.

3.2. Teste de Software

O IEEE *Software Knowledge Body* define o teste como sendo “a atividade executada para avaliar e melhorar a qualidade de um produto, através da identificação dos defeitos e dos problemas” (ABRAN ET AL., 2004). Mais particularmente para o domínio da computação, essa mesma organização vê o teste de software como consistindo na verificação dinâmica do comportamento de um programa, para um conjunto finito de casos de testes, comparando-o com seu comportamento esperado (ABRAN ET AL., 2004). Testar todos os casos para um sistema real é impossível ou, ao menos, impraticável em um tempo finito. Além disso, o problema do oráculo[2], que consiste na tomada da decisão se o comportamento de um sistema é correto ou não, também aparece nessa definição de teste de software.

Como discutido anteriormente, desenvolver software é uma atividade complexa, na qual muitos erros podem ser introduzidos. A fim de minimizar a ocorrência de erros associados ao processo de desenvolvimento de software, atividades de garantia de qualidade são adicionada a esse processo, como VV&T (Verificação, Validação e Teste). Dentre as atividades de verificação e validação, umas das técnicas mais utilizadas é o teste (BARBOSA, E. F. ET AL., 2000).

O célebre livro “*The Art of Software Testing*” (MYERS ET AL., 2004) afirma que um teste bem executado é aquele que encontra erros (*bugs*). O teste, portanto, tem como objetivo de executar um programa a fim de encontrar suas anomalias e defeitos. Ele pode ser visto como uma atividade “destrutiva”, já que no teste procuramos encontrar as falhas de um programa, enquanto que na fase de desenvolvimento procuramos construir um programa correto.

3.2.1. Fases de teste

O teste de software é geralmente efetuado em diferentes fases (níveis) ao longo do processo de desenvolvimento e manutenção. Tendo isso em vista, o alvo do teste pode variar: um único módulo, um grupo de módulos ou o sistema completo. De maneira geral, três grandes fases de teste podem ser identificadas (ABRAN ET AL., 2004):

- **Teste de unidade ou teste unitário**: é a fase onde são testados os componentes mínimos de um sistema, como por exemplo uma classe ou um método, considerando-se a abordagem orientada a objetos. O objetivo básico é encontrar falhas em pequenas partes do sistema (unidades), que funcionam independentemente do todo (o sistema completo). Este tipo de teste concentra-se em erros de lógica e de implementação de cada módulo do software separadamente (BARBOSA, E. F. ET AL., 2000).
- **Teste de integração**: é o processo de testar a interação entre os componentes de software (ABRAN ET AL., 2004). Seu objetivo básico é encontrar as falhas

2 Um oráculo pode ser visto como a entidade que analisa o resultado do teste , comparando-o com o resultado esperado, para saber se o programa que está sendo testado se comporta corretamente. Portanto, ele produz um veredicto sobre um teste, dizendo se este “passou” ou “falhou” (ABRAN ET AL., 2004).

provenientes dessa interação, como por exemplo erros na transmissão de dados entre dois componentes. Ele visa portanto a descobrir erros associados às interfaces entre os módulos de um software (BARBOSA, E. F. ET AL., 2000).

- **Teste de sistema**: este teste preocupa-se com o comportamento do sistema como um todo (ABRAN ET AL., 2004). Nesta fase, o software desenvolvido é combinado com os outros elementos do sistema (*hardware*, banco de dados, etc) e assim podemos verificar se o sistema atende a seus requisitos (as necessidades do clientes). Este tipo de teste visa identificar erros de funções e características de desempenho que não estejam de acordo com as especificações do software (BARBOSA, E. F. ET AL., 2000).

Geralmente, iniciamos o processo de teste pelos testes de unidade. Após cada unidade do sistema ter sido validada, passa-se aos testes de integração. Finalmente, para testar o software como um todo e assegurar que ele atende ao requisitos especificados, efetua-se o teste de sistema.

Mais especificamente sobre os testes unitários, a metodologia ágil de gestão de projeto de software XP (*Extreme Programming*), de Beck (1999), prega como uma de suas principais práticas que os testes unitários devem ser escritos antes do módulo (unidades) a ser testada. Considerando-se uma abordagem dirigida a objetos, é recomendado, portanto, escrever-se os testes de uma classe antes mesmo do código que a implementa.

3.2.2. Técnicas de teste

Existem muitas técnicas para o teste de software, sendo que estas fornecem ao desenvolvedor uma abordagem sistemática e teoricamente fundamentada, além de um mecanismo que pode auxiliar a avaliar a qualidade e adequação da atividade de teste (BARBOSA, E. F. ET AL., 2000). As técnicas de teste de software são normalmente classificadas em:

- **Teste funcional (caixa-preta)**: avalia o comportamento externo do componente de software, sem considerar o comportamento interno do mesmo (MYERS ET AL., 2004). O software é considerado como uma caixa-preta, já que seus detalhes de implementação não são considerados para o teste. Este teste é baseado na especificação do software. Algumas técnicas conhecidas de teste funcional incluem: particionamento de equivalência, análise de valores limites e grafo de causa-efeito (BARBOSA, E. F. ET AL., 2000).

- **Teste estrutural (caixa-branca)**: avalia o comportamento interno do componente de software (MYERS ET AL., 2004). O software é considerado como uma caixa-branca, aonde seus detalhes de implementação são conhecidos. Os casos de teste são gerados a partir da análise do código-fonte do software. Os critérios mais conhecidos de teste estrutural são os baseados em fluxo de controle e os baseados em fluxo de dados (ABRAN ET AL., 2004).

3.3. Abordagens de teste baseadas em modelos

A confiabilidade e a produtividade do processo de teste pode ser melhorada se as atividades de teste forem automatizadas (LIMA ET AL., 2007). A criação de casos de teste é talvez a etapa mais complicada do processo de teste, sendo que sua automatização pode reduzir o custo de desenvolvimento de software, já que a geração manual de casos de teste será eliminada, e melhorar a confiabilidade do processo através do aumento da cobertura do teste (LI ET AL., 2006). As abordagens de Teste Baseado em Modelos (MBT) e Teste Dirigido por Modelos (MDT), que é derivada da primeira, apresentam-se como uma boa solução para a construção automática e a execução de casos de testes.

3.3.1. Teste Baseado em Modelos

O Teste Baseado em Modelos (MBT) é abordagem de teste de software aonde os casos de teste são derivados automaticamente de um modelo, que especifica os aspectos funcionais do software. O objetivo do MBT é a verificação da conformidade entre a especificação do software (modelo) e sua implementação (EL-FAR; WHITTAKER, 2001). Este abordagem pode ser resumida como a "automatização do teste em caixa-preta" (UTTING; LEGEARD, 2007), já que os testes são obtidos a partir de modelos, sem levar em consideração portanto os detalhes de implementação (código-fonte) do software.

Os principais benefícios da MBT são (UTTING; LEGEARD, 2007):

- **Detecção de falhas**: o principal objetivo do teste é encontrar falhas no sistema testado (*system under test*). Experiências com o MBT comprovam que esta abordagem é tão ou mais eficaz do que técnicas manuais de teste para encontrar falhas (UTTING; LEGEARD, 2007).

- **Redução do tempo e do custo de testes**: a MBT pode reduzir o tempo e, por consequência, o custo para a geração de casos de testes (UTTING; LEGEARD, 2007). Experiências na IBM mostraram que o tempo completo para o teste de um projeto pode ser reduzido aproximadamente pela metade usando-se técnicas de MBT (FARCHI; HARTMAN; PINTER, 2002).

- **Aumento da qualidade do teste**: quando os testes são escritos manualmente, a qualidade destes, que é representada pela cobertura do teste, depende do conhecimento do engenheiro responsável pela escrita deles. Com o MBT, os testes são gerados automaticamente a partir de heurísticas e algoritmos. Desta forma, o MBT permite a geração de um número maior de casos de testes do que é possível com testes manuais, aumentando portanto a cobertura do teste (UTTING; LEGEARD, 2007).

- **Descoberta de erros de requisitos e evolução do sistema**: a criação do modelo de testes pode ajudar a clarificar os requisitos do sistema, expondo seus erros. Encontrar erros na fase de especificação, aonde os requisitos são definidos, poupa que estes sejam propagados para as fases de concepção e implementação, reduzindo-se assim os custos para corrigi-los. Portanto, quanto mais cedo criarmos os modelos de teste melhor. Além disso, se os requisitos do sistema mudam (o sistema evolui), o esforço para adaptar os testes a mudança é minimizado, já que o esforço para atualizarmos o modelos de teste, que gera os casos de teste automaticamente, é menor que o esforço para alterar todos casos de teste manualmente (UTTING; LEGEARD, 2007).

Entretanto, o MBT também apresenta limitações. Uma limitação fundamental é que o MBT não garante encontrar todas as diferenças (erros) entre o modelo e sua implementação. Além disso, uma de suas principais limitações práticas é que esta abordagem necessita de diferentes habilidades da equipe de testes: habilidades de modelagem para criar o modelo de testes e de programação para implementar os geradores de teste. Isto implica custos de treinamento e um aumento da curva inicial de aprendizagem do processo de teste (UTTING; LEGEARD, 2007).

Uma outra limitação do MBT é que ele não considera a diferenciação entre modelos independentes de plataforma (PIM) e modelos específicos de plataforma (PSM) (HECKEL, R.; LOHAMANN, 2003). Por isso, na próxima seção deste capítulo, será apresentado o Teste Dirigido por Modelos, que pode ser considerado como uma evolução do MBT a fim de beneficiar-se da separação entre PIM e PSM.

3.3.2. Teste Dirigido por Modelos

O Teste Dirigido por Modelos (MDT) é uma abordagem derivada do Teste Baseado em Modelos (MBT) aonde foram incorporadas as práticas da Engenharia Dirigida por Modelos (MDE). No MDT, os artefatos de teste são gerados automaticamente de acordo com regras de transformação pré-definidas que derivam, dos modelos de desenvolvimento, os casos de teste e a estrutura de execução necessária para múltiplas plataformas (LIMA ET AL., 2007). Além disso, no MDT existe a separação entre modelos independentes e específicos de plataforma, seguindo a filosofia da Arquitetura Dirigida por Modelos (MDA) (HECKEL, R.; LOHAMANN, 2003).

Portanto, a filosofia da MDA pode ser aplicada tanto para o desenvolvimento quanto para os testes. Desta forma, os modelos de desenvolvimento (PIM e PSM) podem ser transformados em modelos de teste e há uma separação entre modelos independentes e específicos de plataforma (DAI, 2004). A figura 3.1 apresenta essa visão do MDT.

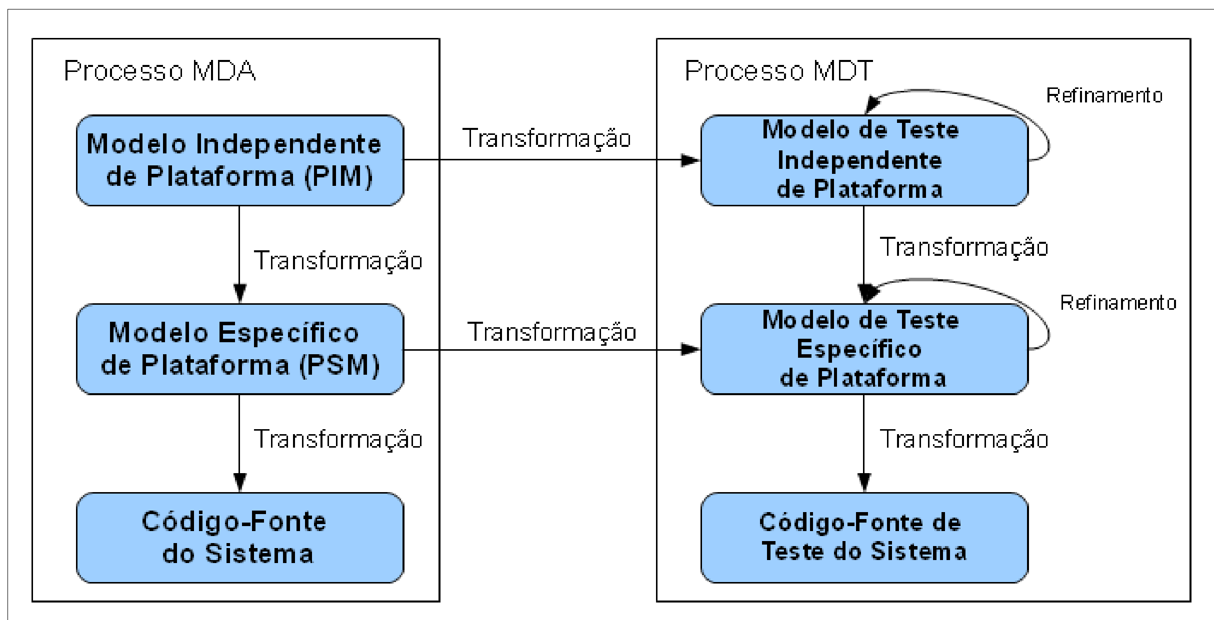

Figura 3.1. Processo de teste da MDT (figura baseada em Dai (2004))

Além dos benefícios do MBT (tempo e custo de teste reduzidos, aumento da detecção de falhas, etc), as vantagens esperadas do MDT através da utilização da MDE, segundo Lima et al. (2007), são:

- redução do custo de implantação do sistema e das ferramentas para executar seu teste em múltiplas plataformas, através do reúso de modelos;
- alcance de um nível maior de automatização na geração dos testes, através da especificação de regras de transformação entre modelos;

3.4. Meta-modelagem de testes com UML

Este trabalho concentra-se nas técnicas de MBT e MDT cujo os testes são modelizados utilizando-se a linguagem de modelagem UML (*Unified Modeling Language*) (OMG, 2007), tendo em vista que o principal objetivo deste trabalho é a proposição de um meta-modelo de testes nesta linguagem.

A maioria das modelizações de testes encontradas durante a etapa de pesquisa deste trabalho é baseada nos diagramas UML de sequência e de estado (*statechart).* Outros tipos de diagramas UML são também utilizados como suporte ao diagrama de modelagem principal (sequência ou estado), como os diagramas de classe e de objeto, a fim de prover mais informações. Além disso, a linguagem de restrições OCL (*Object Constraint Language*), que também faz parte da UML, também é frequentemente utilizada com o objetivo de formalizar os modelos.

No entanto, outras formas de modelagem de testes sem a utilização de diagramas de sequência ou de estados também existem. Por exemplo, diagramas de atividade foram utilizados por Rocha e Martins (2008). Uma modelagem de testes utilizando somente OCL combinado com informações de diagramas de classes foi proposta por Packevicius, Usaniov e Bareisa (2007). Mas, os diagramas UML mais populares para a meta-modelagem de testes são realmente os diagramas de sequência e de estado. Trabalhos que propõem a utilização desses dois diagramas serão discutidos na sequência.

3.4.1. Meta-modelagem de testes com diagramas de sequência UML

Fraikin e Leonhardt (2002) apresentam uma meta-modelagem de testes utilizando diagramas de sequência como meta-modelo principal. Os diagramas de sequência propostos neste trabalho possuem como informações complementares os parâmetros dos métodos e seus valores de retorno. Para contornar o problema de ter um estado inicial estável, existe uma restrição na modelagem que impõe que o primeiro método invocado de cada diagrama de sequência seja um método do tipo construtor. O objetivo desta restrição é de realizar a inicialização do objeto antes de efetuar testes sobre o mesmo.

Os diagramas de sequência são também utilizados como forma principal de meta-modelagem de testes no trabalho de Javed, Strooper e Watson (2007), cujo o objetivo principal é a geração de testes unitários para os *frameworks* da família xUnit[3].

3 *Frameworks* que automatizam a execução de testes unitários. O mais conhecido deles é o Junit, criado por Kent Back e Eric Gamma em 1997, para a linguagem de programação Java.

A utilização de diagramas de sequência juntamente com a linguagem de restrições OCL é apresentada por Bao-Lin et al. (2007). Neste trabalho, a partir de um diagrama de sequência é construída uma árvore com todos os caminhos possíveis da execução deste diagrama. Os valores das restrições OCL são utilizados para efetuar um teste de limites, ou seja, para verificar que os resultados dos métodos respeitam essas restrições. Por exemplo, os valores de um atributo de uma classe devem ser menores que X após a execução um método, especificado através de uma pós-condição (*post-condition*) OCL.

Por sua vez, Sokenou (2006) utiliza também diagramas de sequência como fonte principal de informações para a geração de testes. Para complementar o meta-modelo proposto neste trabalho, são utilizadas informações obtidas a partir de diagramas de estado para a inicialização de objetos. Além disso, esses diagramas de estado são combinados com restrições OCL para servir como oráculo para os testes.

3.4.2. Meta-modelagem de testes com diagramas de estado UML

Assim como os diagramas de sequência, os diagramas de estado UML são também utilizados para servir como meta-modelo principal para a geração de testes.

A ferramenta de geração de testes ParTeg (WEIßLEDER; SCHILINGLOFF, 2007) baseia-se na linguagem de restrições OCL, em diagramas de estado e diagramas de classe para derivar os valores dos parâmetros de entrada dos testes. Em seguida, essa ferramenta verifica, a partir do diagrama de estados, se o sistema em teste se comporta de modo correto: se o sistema está no estado correto do diagrama de estados dado os valores de entrada do teste.

Uma outra ferramenta que utiliza os diagramas de estados como meta-modelo principal para a geração de testes é SmartTesting[4]. Essa ferramenta foi desenvolvida na França e é conhecida mundialmente no domínio de Teste Baseado em Modelos.

Além disso, o trabalho de Crichton, Cavarra e Davies (2001) apresenta uma proposta para a geração de testes utilizando diagramas de estados em combinação com diagramas de classe e de objetos. Por sua vez, Nam, Mousset e Lecy (2006) definiram também uma meta-modelagem dos testes usando diagramas de estados, sendo que o objetivo desta modelagem é gerar testes para o *framework* JUnit.

3.4.3. Comparação entre a utilização de diagramas de sequência e de estado

Uma comparação entre a utilização dos diagramas de sequência e de estado como artefato principal para a geração de testes é feita por Kansomkeat et al. (2008). Na experiência realizada neste trabalho, foram criados diagramas de sequência e de estados para representar o sistema a ser testado. Como resultado desta comparação, os testes unitários gerados a partir de diagramas de estado encontraram 12% a mais de falhas que os testes unitários gerados a partir de diagramas de sequência. Por sua vez, os testes de integração gerados a partir des diagramas de sequência encontraram 27% mais de falhas que os testes de integração derivados de diagramas de estado. Entretanto, a metodologia utilizada para realizar esse estudo comparativo foi muito simples: somente um sistema foi utilizado como exemplo e os testes foram escritos à mão a partir dos diagramas (processo não-automatizado).

4 http://www.smartesting.com

A dificuldade para a criação de uma modelagem a partir de diagramas UML é discutida por Fraikin e Leonhardt (2002). Este trabalho afirma os diagramas de sequência são mais fáceis de serem utilizados para a geração de testes, já que são mais conhecidos e mais facilmente compreendidos pelos desenvolvedores de software. Portanto, os desenvolvedores possuem menos dificuldades para criar modelizações de testes a partir de diagramas de sequência que de diagramas de estado.

3.5. Considerações finais

Neste capítulo, o Teste de Software foi apresentado como uma das técnicas da Engenharia de Software para obter sistemas com uma melhor qualidade (menos falhas). Entretanto, o custo para se testar um sistema é muito alto (cerca de 50% do tempo total de um projeto). Para obter-se uma melhor produtividade no processo do teste, além de um aumento da confiabilidade, as atividades de teste devem ser automatizadas.

Na área de automatização de testes destaca-se a abordagem de Teste Baseado em Modelos (MBT). Além da MBT, destaca-se também a abordagem de Teste Dirigido por Modelos (MDT), que pode ser vista como uma união dos princípios da Engenharia Dirigida por Modelos (MDE) com a MBT. Essas duas abordagens (MBT e MDT) apresentam-se como boa solução para a construção automática e a execução de casos de testes. Elas possuem como principais vantagens em relação aos testes manuais: eficácia similar ou maior na descoberta de falhas, redução do custo e do tempo dos testes e aumento da cobertura dos testes.

Para a utilização das abordagens baseadas em (ou dirigidas por) modelos, é necessário a definição de um meta-modelo de testes. Neste capítulo, foram estudadas as formas de modelizar os testes baseadas na linguagem de modelagem UML, já que será sobre essa linguagem que será feita a nossa proposição de um meta-modelo de testes. Os dois principais diagramas UML utilizados para a geração de testes são os diagramas de sequência e os diagramas de estados, sendo que o primeiro pode ser considerado mais “popular”, já que ele permite uma modelagem mais simples.

4. NOVASTUDIO

Este capítulo descreve o produto NovaStudio da empresa BULL S.A.S, sociedade aonde este trabalho foi realizado, além das funcionalidades a serem implementadas nesta ferramenta por este trabalho. A primeira seção deste capítulo introduz o ambiente de desenvolvimento NovaStudio. A segunda seção apresenta as principais funcionalidades desta ferramenta. A terceira seção explica como é feita a concepção e o desenvolvimento de aplicações com NovaStudio. A quarta seção descreve a arquitetura lógica do Java EE que é suportada por NovaStudio e a quinta lista as implementações técnicas dessa arquitetura. Em seguida, o motor de geração de código de NovaStudio é descrito. As informações deste capítulo são baseadas no Guia de Utilização de NovaStudio (versão 3.1.10), que possui um acesso restrito aos clientes e desenvolvedores desta ferramenta.

4.1. Introdução

O produto NovaStudio[5] da empresa BULL[6] S.A.S fornece um ambiente de desenvolvimento, baseado em *plugins* Eclipse[7], que permite de gerar uma parte do código necessário as aplicações Java EE. O objetivo de NovaStudio é aumentar a produtividade das equipes de desenvolvimento através da utilização da abordagem da Arquitetura Dirigida por Modelos (MDA). NovaStudio faz interface com uma ferramenta de modelagem UML, por exemplo Enterprise Architect[8], através de arquivos XMI.

Em termos de geração do código Java, NovaStudio suporta diferentes *frameworks* implementados através do retorno de experiências das equipes de desenvolvimento de software da BULL. A implementação ou a escolha de um *framework* não é estruturante: a partir de uma mesma modelagem, é possível de se escolher a todo momento um outro tipo de implementação, já que NovaStudio segue o princípios da MDA, separando os modelos independentes de plataforma (PIM) dos modelos específicos de plataforma (PSM).

Além disso, NovaStudio apresenta a vantagem de ser completamente integrado ao conhecido ambiente de desenvolvimento integrado Eclipse, que é uma ferramenta livre de desenvolvimento muito utilizada em laboratórios de pesquisa e empresas.

5 http://www.bull.com/fr/middleware/novastudio.php
6 http://www.bull.com/
7 http://www.eclipse.org/
8 http://www.sparxsystems.com/products/ea/index.html

4.2. Funcionalidades

NovaStudio apresenta como principais funcionalidades:

- Geração do código de uma aplicação Java EE, além do banco de dados, através da utilização de diagramas UML seguindo os princípios da MDA. NovaStudio suporta também o processo inverso (*bottom up*): a partir de um banco de dados é possível obter os diagramas de classe para uma ferramenta de modelagem. Destes diagramas obtidos, NovaStudio permite gerar o código aplicativo, seguindo sua abordagem clássica MDA (*top down*).
- NovaStudio suporta diferentes tipos de *frameworks* para a geração de código: WEB, JDK 1.4 ou JDK 1.5, Hibernate 3, EJB3 ou JDO.
- NovaStudio suporta uma Arquitetura Orientada a Serviços (SOA). Ele oferece uma visão permitindo a gestão do registro de serviços de negócios de uma empresa e uma outra visão permitindo configurar o acesso a diferentes tipos de Sistemas de Informação Executivos da empresa ou de seus fornecedores.

Entretanto, NovaStudio não gera o código completo de uma aplicação. A lógica de negócios da aplicação deve ser adicionada posteriormente por um desenvolvedor, diretamente no ambiente de desenvolvimento integrado Eclipse. Isso significa que para alguns métodos da aplicação, aqueles que possuem a lógica de negócios, só será gerado por NovaStudio o esqueleto dos mesmos.

4.3. Concepção e desenvolvimento

Dois papéis principais podem ser distinguidos no processo de concepção e geração de código com NovaStudio:

- **Arquiteto**: utiliza uma ferramenta de modelagem UML para definir os objetos de negócios e os serviços que devem ser implementados. Ele também tem a função de exportar a modelagem concebida para um formato XMI e repassá-la ao Desenvolvedor.
- **Desenvolvedor**: utiliza NovaStudio para gerar o código a partir do modelo definido pelo Arquiteto. Ele também é responsável por adicionar a lógica (código) referente aos métodos de negócios da aplicação, já que para estes NovaStudio gera apenas um esqueleto. Além disso, o Desenvolvedor pode alterar algumas configurações do modelo, através da modificação direta de *tagged values*[9], e retornar o modelo UML no formato XMI para o Arquiteto.

Para a geração de código com NovaStudio, várias etapas devem ser respeitadas:

1. **Modelagem**: o Arquiteto cria o modelo que representa a aplicação a ser implementada em uma ferramenta de modelagem UML e exporta esse modelo em formato XMI.

9 Um tagged value é constituído de uma chave e um valor, sendo utilizado para adicionar uma informação complementar a um elemento de um modelo. Por exemplo, o tipo de banco de dados que será utilizado na implementação de uma aplicação será MySQL.

2. **Conversão do modelo para o modelo específico de NovaStudio**: o Desenvolvedor converte o arquivo XMI, exportado pelo Arquiteto, em um novo arquivo XMI baseado no meta-modelo específico de NovaStudio (arquivo de extensão ".generation"). Para isto, a funcionalidade "Enterprise Architect → Populate Generation Model" de NovaStudio deve ser utilizada.

3. **Geração do código**: a terceira etapa consiste na geração do código aplicativo a partir do arquivo ".generation" que foi criado na etapa anterior. Para isto, é utilizado a funcionalidade "Code Generation" de NovaStudio.

4. **Implementação do métodos associado a camada de negócios**: a última etapa consiste na adição da lógica específica à camada de negócios, ou seja, na implementação dos métodos que não gerados automaticamente por NovaStudio, aqueles aonde somente um esqueleto é gerado.

Podemos notar que dentre as etapas acima, somente a primeira deve ser executada pelo Arquiteto. Todas as demais etapas são de responsabilidade do Desenvolvedor. Outra característica de NovaStudio e dos projetos realizados com esta ferramenta é que normalmente os papéis de Arquiteto e Desenvolvedor não são exercidos por uma mesma pessoa.

4.4. Arquitetura lógica do Java EE

A arquitetura lógica do Java EE suportada por NovaStudio é baseada em uma separação de camadas (figura 4.1). Nesta arquitetura, existem três componentes principais: a camada de apresentação (*presentation layer*), a camada de negócios (*business layer*) e o banco de dados (*database*). Esta separação arquitetural é lógica, mas pode ser também física: dependendo da plataforma alvo, estas camadas podem residir num único servidor ou em servidores distintos.

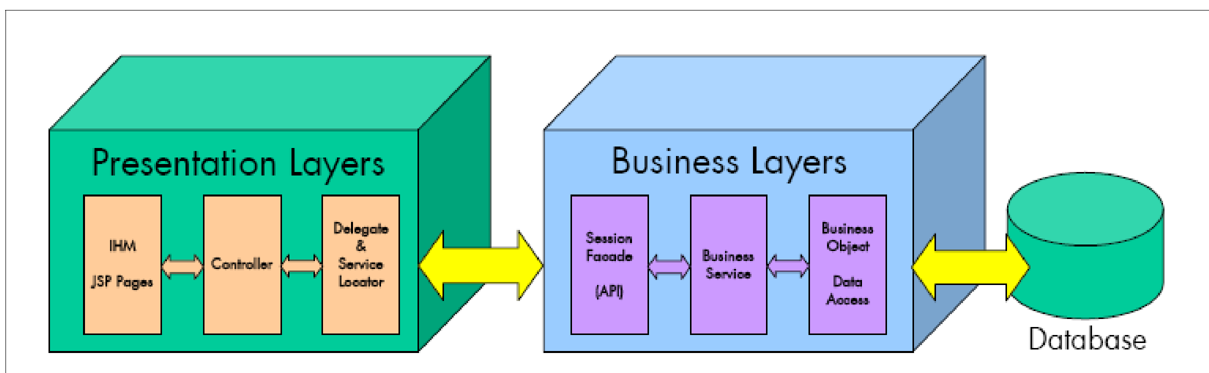

Figura 4.1. Arquitetura lógica Java EE (BULL, 2007)

Na figura acima, pode-se notar que arquitetura utilizada por NovaStudio baseia-se no conhecido padrão de arquitetura de software Modelo-Visão-Controle (MVC – *Model-View-Controller*). Em suma, o MVC visa separar a lógica de negócios da aplicação (Modelo), da interação com usuários e eventos (Controle) e a apresentação dos dados (Visão) (KRASNER; POPE, 1988).

As camadas do MVC e as correspondentes relações com as sub-camadas da arquitetura Java EE utilizada por NovaStudio são apresentadas na tabela 4.1.

Tabela 4.1. MVC vs. Camadas da Arquitetura Java EE

MVC	**Camadas da Arquitetura Java EE**
Modelo	*Business Delegate, Service Locator, Session Façade, Business Service* e *Business Object*
Visão	Interface Homem-Máquina (IHM)
Controle	*Controller*

4.4.1. Subcamadas *Business Service* e *Business Object*

No contexto deste trabalho, as duas subcamadas mais importantes são a *Business Service* e a *Business Object*. As outras subcamadas não serão detalhadas aqui, mas maiores informações sobre as mesmas podem ser obtidas em (MEDEIROS, 2009) ou (SUN, 2010).

- ***Business Service:*** esta subcamada contém a implementação dos serviços da lógica de negócios definidos na ferramenta de modelagem. É nesta subcamada que o desenvolvedor deve adicionar a implementação dos métodos específicos da aplicação que não são gerados por NovaStudio, conforme discutido na seção 4.3. Se decidido pelo designer, os métodos básicos de banco de dados do tipo CRUD (Criar, Ler, Atualizar e Deletar) também são adicionados nessa subcamada.

- ***Business Object***: esta subcamada contém os objetos de negócio (*Business Objects*) que são entidades definidas na ferramenta de modelagem e que são salvas no banco de dados.

Se considerarmos a tecnologia Java EJB3 (*Enterprise JavaBeans* versão 3), a subcamada *Business Service* seria constituída por *SessionBeans* do EJB3 e a subcamada *Business Object* por *EntityBeans.*

4.4.2. Interação entre as camadas

Uma visão da interação entre as camadas apresentadas na seção anterior é mostrada na figura 4.2. Esta figura representa um diagrama de sequência contendo a chamada de um cliente a um serviço de uma aplicação que foi desenvolvida usando a arquitetura Java EE suportada por NovaStudio.

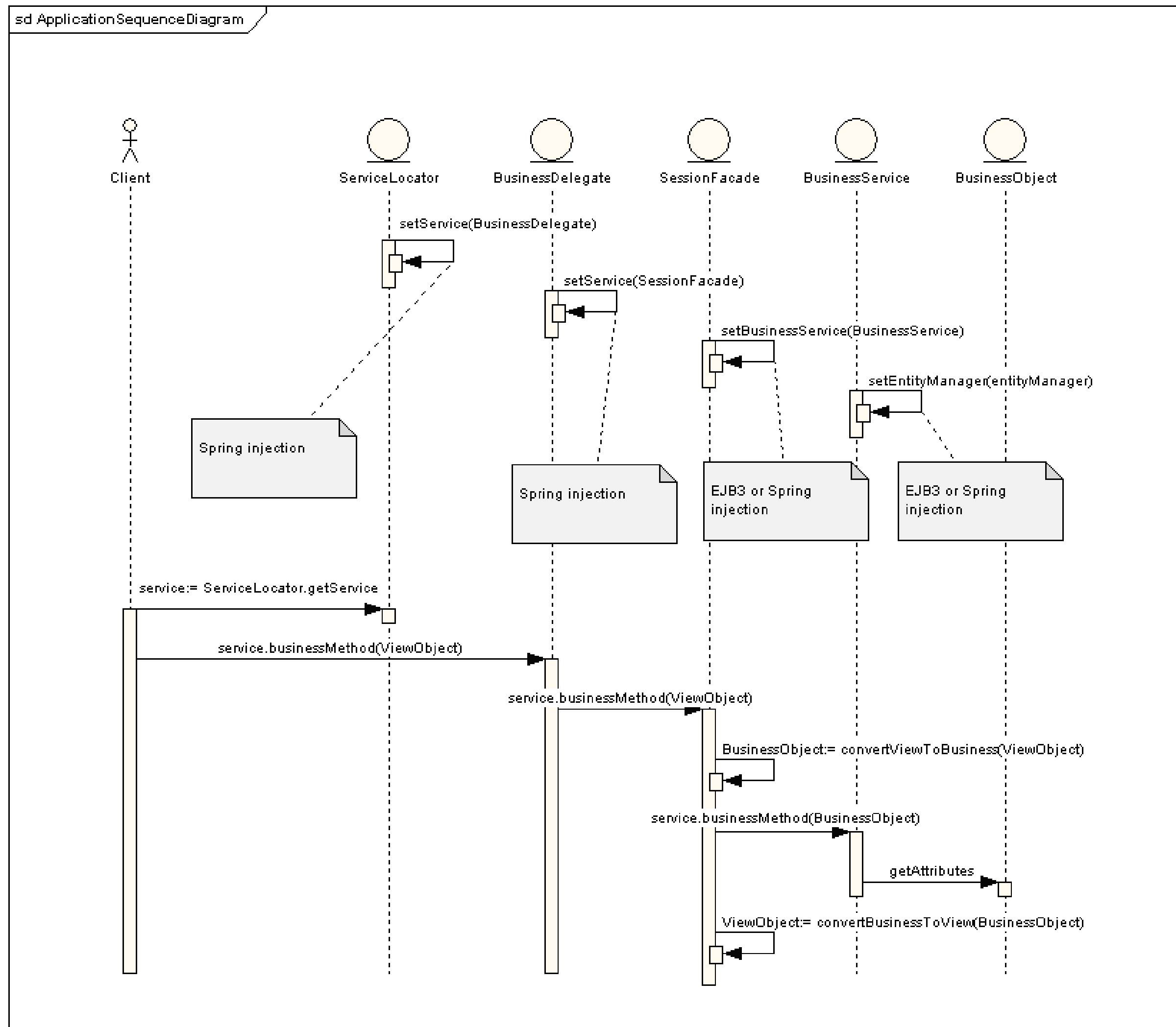

Figura 4.2. Interação entre as camadas (BULL, 2007)

4.5. Implementação da arquitetura lógica

NovaStudio suporta dois *frameworks* de desenvolvimento[10]: o *framework* Java EE e o *framework* Web. O primeiro é utilizado para o desenvolvimento de aplicações que necessitam de funcionalidades fornecidas pelo Java EE, tais como transações e segurança. Por sua vez, o segundo serve para o desenvolvimento de aplicações que podem ser executadas por servidores Web, como, por exemplo, o Apache Tomcat.

Uma vez que o *framework* de implementação foi escolhido, é necessário escolhermos qual será sua implementação técnica. A tabela 4.2 lista as implementações que são suportadas por NovaStudio.

10 Na sua última versão, NovaStudio começou a suportar a geração de código PHP também (MEDEIROS, 2009). Entretanto, por decisão da equipe NovaStudio. a geração de código PHP será desconsiderada neste trabalho.

Tabela 4.2. Implementações suportadas por NovaStudio

	JDK	Session Facade	Business Services	Persistent Framework
Java EE framework (EJB3)	1.5	EJB3	EJB3	EJB3
J2EE framework (EJB2) 1	1.4	EJB2	EJB2	Hibernate 3
J2EE framework (EJB2) 1	1.4	EJB2	EJB2	JDO (Speedo)
Web framework 1	1.4	Classes simples	Classes simples	Hibernate 3
Web framework 2	1.4	Classes simples	Classes simples	JDO (Speedo)

Como podemos ver na tabela acima, NovaStudio suporta diversos *frameworks*. Por decisão da equipe NovaStudio, somente serão gerados testes para os framework Java EE (EJB3) e WEB (Hibernate 3). Os outros *frameworks* suportados por NovaStudio não fazem parte dos objetivos deste trabalho.

4.6. Motor de geração de código

O motor de geração de código da versão atual de NovaStudio é Acceleo[11], que é um gerador de código para diferentes linguagens (Java, .Net, PHP, etc) que utiliza a abordagem MDA. Acceleo é completamente integrado ao Eclipse e ao *Eclipse Modeling Framework* (EMF) e compreende um conjunto de ferramentas que o permite de se adaptar facilmente a todo tipo de projetos e tecnologias (MUSSET; JULLIOT; LACRAMPE, 2009). A geração de código com Acceleo baseia-se na criação de *templates* (padrões de geração) que descrevem a estrutura dos arquivos (códigos) a serem gerados.

Acceleo substitui o antigo motor de geração de NovaStudio que era baseado em *templates* Velocity[12]. Esta tecnologia, por não ter sido criada para atender aos fins específicos da MDA, apresenta alguns inconvenientes. O principal deles é a não consideração de um meta-modelo pelos *templates* Velocitity, o que acarreta um trabalho extra do desenvolvedor para assegurar que as manipulações de modelos, a fim de gerar o código, não violem o seu meta-modelo (MEDEIROS, 2009).

Portanto, a geração do código de testes, um dos objetivos principais deste trabalho, será efetuada através do desenvolvimento de *templates* Acceleo, sendo que um ou vários *templates* devem ser escritos para cada *framework* suportado por NovaStudio.

11 http://www.acceleo.org/
12 http://velocity.apache.org/

4.7. Considerações finais

Neste capítulo, foi apresentado a ferramenta MDA NovaStudio da empresa BULL, que gera de forma automatizada uma grande parte do código de aplicações Java a partir de modelos UML - notadamente diagramas de classe UML. Estes modelos UML são obtidos de ferramentas de modelagem UML, como Enterprise Architect, através da leitura de arquivos do tipo XMI que são exportados por estas ferramentas.

Como já foi dito no capítulo 1, o principal objetivo deste trabalho é adicionar a funcionalidade de geração automatizada de testes a NovaStudio. Para que esse objetivo possa ser alcançado, é necessário, primeiramente, a definição de um meta-modelo UML de testes, que servirá como base para a geração do código de testes. Esse meta-modelo será apresentado no próximo capítulo.

Além da definição do meta-modelo, também é necessário adaptar NovaStudio aos novos elementos UML adicionados por esta meta-modelagem de testes, escolher um *framework* para executar os testes e escrever os *templates* Acceleo correspondentes aos testes a serem gerados. Por fim, deve-se integrar a funcionalidade de geração de testes a NovaStudio através da escrita de um *plugin* Eclipse. A descrição dessas tarefas será feita no capítulo 6.

5. MODELAGEM DE TESTES

Este capítulo propõe uma nova forma de modelagem de testes utilizando-se diagramas de sequência UML juntamente com objetos UML. Com a meta-modelagem proposta aqui, torna-se possível a geração de testes unitários (com dependências) para as classes que também são geradas automaticamente pelo ambiente de desenvolvimento NovaStudio.

5.1. Escolha da modelagem

Uma das primeiras etapas do nosso trabalho constitui-se no estudo de técnicas de geração de testes a partir de modelos, notadamente das abordagens de Teste Baseado em Modelos (MBT) e de Teste Dirigido por Modelos (MDT). Este estudo foi apresentado no capítulo 3. Por decisão da equipe NovaStudio, o meta-modelo a ser utilizado para a geração de testes deve ser baseado em diagramas UML, ou seja, o meta-meta-modelo deve ser o UML. Por essa razão, também apresentamos no capítulo 3, os diagramas UML mais utilizados como base para a modelagem de testes que são: os diagramas de sequência e os diagramas de estado.

Após esse estudo ter sido realizado, foi decidido utilizar-se os diagramas de sequência como base para a modelagem de testes, sendo que esta decisão foi apresentada para a equipe NovaStudio e aceita. As principais razões para a escolha de diagramas de sequência em detrimento dos diagramas de estado foram:

- **Simplicidade da modelagem**: os diagramas de sequência são mais conhecidos e mais utilizados por desenvolvedores e arquitetos de software que os diagramas de estado (DOBING; PARSONS, 2006). Eles são também de mais fácil compreensão para a criação de modelizações (FRAIKIN; LEONHARDT, 2002). Portanto, o critério da simplicidade da modelagem e de sua usabilidade pesou muito na escolha dos diagramas de sequência como base para o meta-modelo de testes. Esse critério foi muito importante, tendo em vista que se a modelagem é muito complexa para ser utilizada, então nenhum arquiteto de software desejará utilizá-la e o sucesso da nova funcionalidade de geração de testes de NovaStudio será comprometido.

- **Poder de expressão**: a equipe NovaStudio desejava que a meta-modelagem escolhida, que será descrita nas próximas seções, pudesse exprimir testes unitários e se possível testes mais complexos (com dependências reais). Portanto, foi verificado através de

diversos casos de testes, que o poder de expressão da meta-modelagem proposta permitia gerar os os tipos de testes especificados pela equipe NovaStudio. Como os requisitos da equipe foram respeitados (supridos), então a meta-modelagem com diagramas de sequência foi aceita.

Um último critério de escolha que pode ser citado, para a meta-modelagem de testes, é a compatibilidade com a ferramenta de modelagem Enterprise Architect[13]. Isso porque este é o único software de modelagem UML suportado atualmente por NovaStudio. Portanto, com a meta-modelagem de testes validada à mão (através da escrita de casos de testes), a próxima etapa realizada foi verificar se Enterprise Architect fornecia todas as funcionalidades necessárias para a descrição da nossa meta-modelagem: o que foi verificado com sucesso.

Além dos diagramas de sequência, os objetos UML, ou seja, as instâncias de uma classe, são utilizados para complementar a meta-modelagem proposta.

5.2. Diagramas de sequência

Os diagramas de sequência são a base principal para a meta-modelagem de testes proposta por este trabalho. Além deles, são também utilizados os objetos UML como complemento das informações necessárias para a criação das classes de teste. Nesta seção, será apresentado a nossa meta-modelagem e as regras a serem respeitadas para ela possa ser utilizada. Todos os diagramas e capturas de tela presentes neste capítulo foram gerados a partir do ambiente de modelagem Enterprise Architect, que como já citamos anteriormente, é o único ambiente suportado atualmente por NovaStudio.

A meta-modelagem de testes utilizando diagramas de sequência proposta por este trabalho apresenta as seguintes características:

- os diagramas de sequência devem respeitar uma forma de nomenclatura;
- deve ser criado um diagrama de sequência para cada método de cada classe a ser testada;
- existe um tipo de diagrama de sequência especial que representa a inicialização dos objetos a serem testados. Este diagrama chama-se diagrama de sequência de inicialização;
- existem dois tipos de diagramas de sequência para testes, sendo que um deles faz asserções diretas e o outro faz asserções indiretas para verificação dos métodos.

As características e condições acima serão detalhadas na sequência desta seção.

5.2.1. Nomenclatura dos diagramas

Os diagramas de sequência desenvolvidos pelo Arquiteto devem seguir a seguinte convenção de nomenclatura: XXX-YYY, onde XXX é a classe a ser testada pelo diagrama YYY representa o método a ser testado. Além disso, os diagramas de sequência devem ser colocadas no mesmo pacote (*package*) das classes que eles testam.

---

13 http://www.sparxsystems.com/products/ea/index.html

Como observação, quando YYY é igual a “Initialization”, então por convenção de nomenclatura, este diagrama de sequência será considerado como um diagrama de sequência de inicialização. Entretanto, isso é confirmado posteriormente por testes e verificações realizados por NovaStudio.

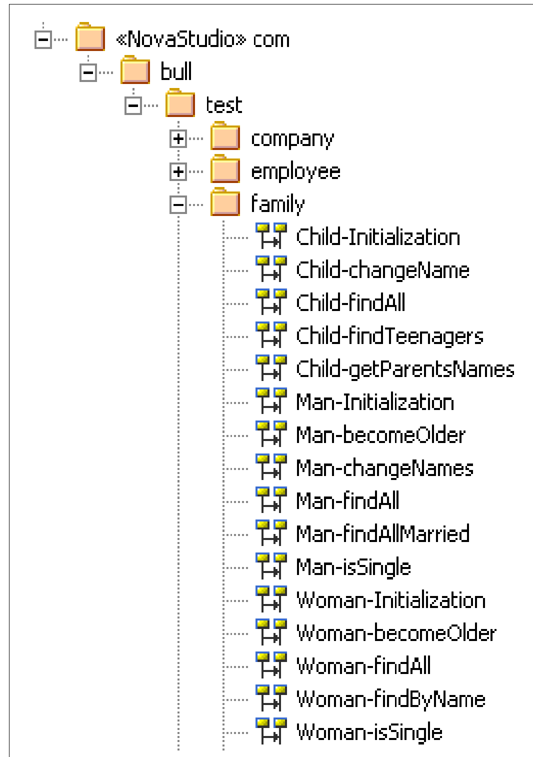

Figura 5.1. Convenção de nomenclatura dos diagramas de sequência

5.2.2. Diagramas de sequência de inicialização

Um dos principais problemas para a modelagem e geração automatizada de testes é a criação de um estado inicial consistente para a execução dos testes. Para resolver essa problemática, nossa meta-modelagem propõe a criação de um tipo especial de diagrama de sequência, cujo o objetivo é fazer a declaração e inicialização de todos os objetos necessários para cada classe a ser testada. Esse tipo de diagrama especial de sequência foi chamado de diagrama de sequência de inicialização.

A figura 5.2 mostra um exemplo de diagrama de sequência de inicialização, aonde ocorre a declaração e inicialização de cinco objetos do tipo (da classe) Man. Nos diagramas de sequência de inicialização, cada objeto criado deve possuir um identificador único (como m1, m2, etc) dentre todos os objetos de todas as classes do modelo de testes. Isso é necessário tendo em vista que os objetos podem ter associações entre eles e um objeto pode ser referenciado no diagrama de sequência de uma outra classe. Entretanto, um objeto só pode ser definido (declarado) no diagrama de sequência de inicialização da classe a qual ele pertence. Por exemplo, um objeto do tipo Man, como m1 ou m2, só pode ser declarado no diagrama de inicialização “Man-Initialization”, como mostrado na figura 5.2.

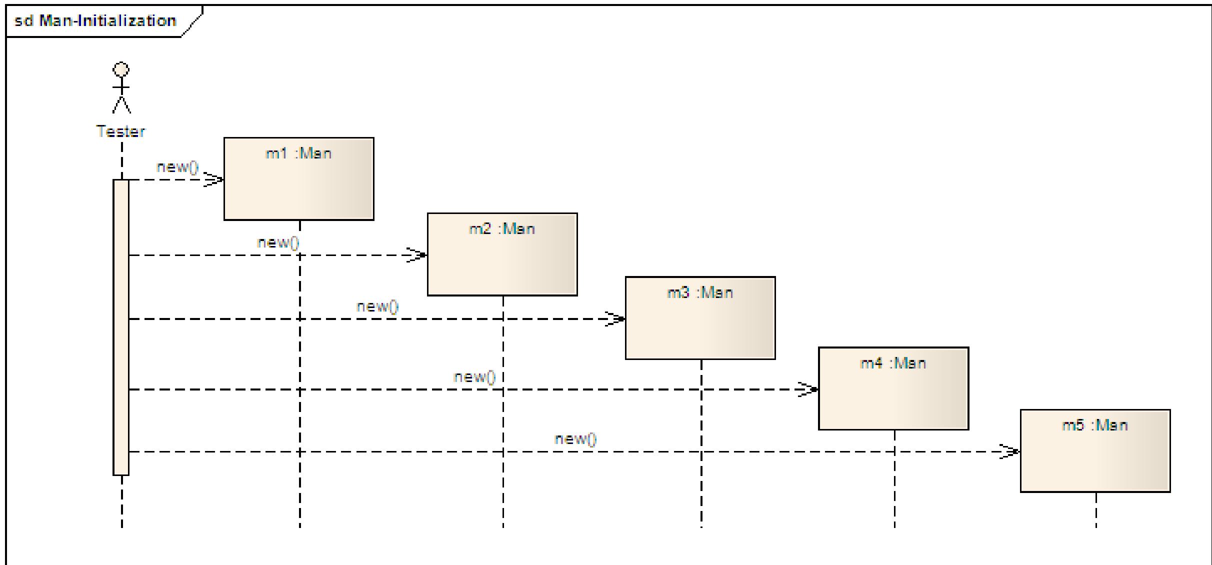

Figura 5.2. Exemplo de diagrama de sequência de inicialização

A figura 5.3 ilustra como são definidos os atributos de um objeto na ferramenta de modelagem Enterprise Architect. A associação entre objetos, suportada pelo meta-modelo proposto neste trabalho, é igualmente ilustrada nesta figura: os objetos c1 e c2, que são do tipo Children e foram declarados no diagrama de inicialização "Child-Initialization", podem ser referenciados como atributos de qualquer diagrama do modelo. Nesta figura, por exemplo, estes objetos (c1 e c2) são os filhos (atributo *childs*) de um dos objetos Man apresentados na figura 5.2.

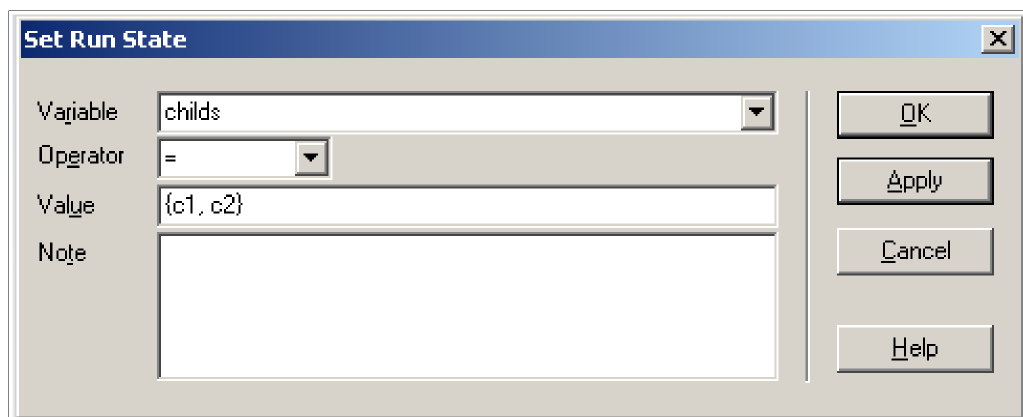

Figura 5.3. Definição de um atributo de um objeto

Na figura acima, podemos ver também a declaração de um atributo do tipo coleção (ou lista), aonde os elementos deste atributo são contidos entre chaves e separados por vírgulas. Para atributos do tipo String, estes devem ser declarados entre aspas (não-obrigatoriamente). Mais uma observação sobre a declaração de atributos é necessária: todos os atributos devem ser de tipo objeto Java, uma vez que NovaStudio não suporta os tipos primitivos do Java (int, char, etc) para a geração do código. Além disso, a palavra *null* é reservada para declaração de um objeto nulo.

5.2.3. Diagramas de sequência com asserção direta

Com a problemática da criação de um estado inicial já resolvida pelos diagramas de sequência de inicialização, podemos passar ao outro importante problema para a geração de testes: saber se o resultado de um teste está correto. Como já introduzido no capítulo 3, este problema é conhecido na literatura como o problema do oráculo (ABRAN ET AL., 2004).

Para a resolução deste problema, foi decidido que o Arquiteto que cria os diagramas de sequência deve definir o resultado esperado para cada chamada de método que ele deseja fazer uma asserção. Uma asserção constitui-se de um teste que verifica se o resultado esperado de um método (ou o valor de uma variável) é igual ao resultado encontrado durante a execução do programa.

A definição do resultado esperado de um método é fácil, já que o Arquiteto conhece todos os objetos utilizados no teste, tendo em vista que foi ele mesmo que declarou estes objetos e seus atributos nos diagramas de sequência de inicialização. Além disso, o Arquiteto conhece o comportamento esperado de cada método (sua funcionalidade) e os parâmetros passados como argumento.

Por outro lado, descobrir o resultado de um método de modo completamente automático, sem definição do resultado esperado, é uma tarefa impossível ou, ao menos, muito difícil, já que a geração dos testes no caso do NovaStudio é feita antes mesmo da implementação dos métodos a serem testados.

A figura abaixo apresenta um diagrama de sequência com asserção direta. Este tipo de diagrama de sequência considera que o resultado esperado do método pode ser comparado diretamente com o resultado real do método, já que este método não é do tipo *void* (sem retorno) e podemos testar o método diretamente a partir do valor (objeto) retorno por ele.

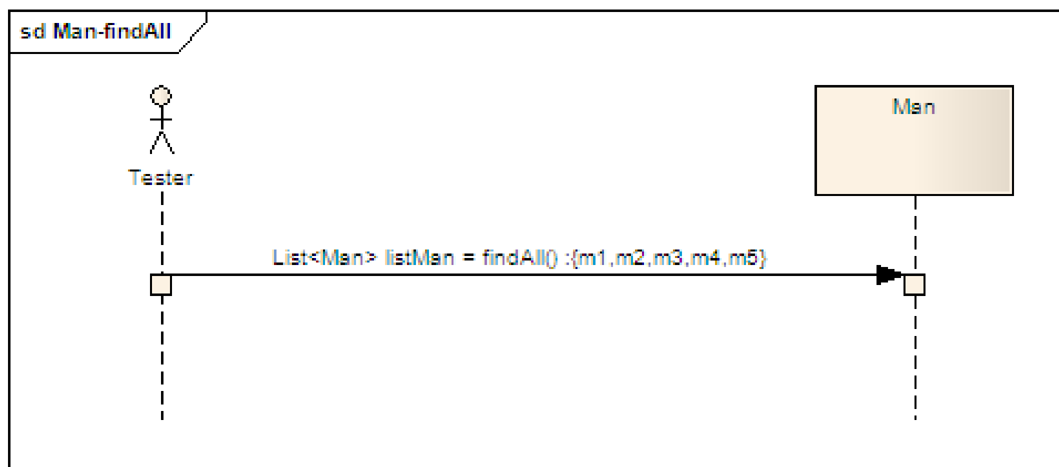

Figura 5.4. Exemplo de diagrama de sequência com asserção direta

Na figura 5.4, o método findAll da classe Man é invocado pelo ator Tester, que representa o *framework* de teste. O resultado deste método é guardado na variável listMan, que é também declarada (List<Man>). O resultado esperado é uma coleção (*Collection*) que contém os objetos m1, m2, m3, m4 e m5 que foram definidos anteriormente no diagrama de sequência de inicialização da classe Man (figura 5.2).

A partir do valor de resposta de um método definido pelo Arquiteto, é declarada uma

asserção diretamente depois deste método, comparando o resultado esperado com o valor contido na variável que contém o resultado deste método. No caso da figura 5.4, será comparado através de uma asserção o resultado do método findAll, que está contido na variável listMan, com a coleção composta por {m1, m2, m3, m4, m5}.

5.2.4. Diagramas de sequência com asserção indireta

Um tipo de diagrama de sequência para efetuar-se um teste indireto também foi concebido no meta-modelo de testes proposto por este trabalho. Neste tipo de teste, uma asserção não é declarada diretamente sobre o resultado de um método a ser testado. A asserção é efetuada sobre um outro elemento do modelo definido pelo Arquiteto. O teste indireto é importante principalmente:

- quando o objeto retornado pelo método a ser testado não é importante, o que é fundamental é a mudança de estados (dos atributos) do objeto obtido como resposta;
- no caso em que o método a ser testado não retorna valores – método do tipo *void*. Neste caso, só é possível serem verificadas as modificações que foram feitas nos elementos do modelo.

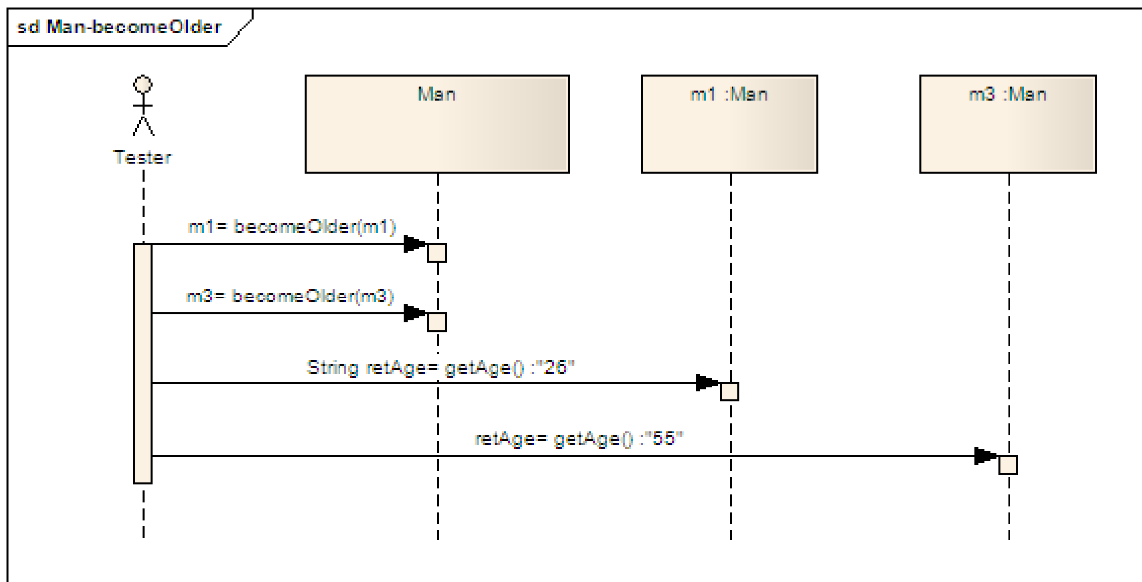

Figura 5.5. Exemplo de diagrama de sequência com asserção indireta

A figura 5.5 apresenta um diagrama de sequência com asserção indireta. Nesta figura, podemos ver o caso do teste de um método chamado “becomeOlder”, que tem como funcionalidade incrementar em uma unidade a idade (*age*) de uma pessoa. Este método é invocado no diagrama acima duas vezes pelo ator Tester com os parâmetros (objetos) m1 e m3. A resposta dos métodos é escrita também nas mesmas variáveis (objetos) que foram utilizadas para invocar o método. Portanto, não existe necessidade de declarar m1 e m3, já que estas variáveis já foram declaradas no diagrama de sequência de inicialização apresentado na figura 5.2.

Como o valor retornado pelo método becomeOlder não é importante para o teste, já que trata-se do objeto dado como parâmetro com o atributo *age* modificado, então nenhuma asserção é feita automaticamente. Por isso, o Arquiteto deve definir os testes (asserções) a

serem efetuados para verificar se o método está correto. No caso da figura 5.5, o Arquiteto definiu duas invocações ao *getter* dos objetos m1 e m3 e indicando quais são os valores de retorno esperados. A partir disso, NovaStudio pode gerar duas asserções comparando se os valores obtidos na respostas desses getters são iguais aos valores esperados.

Uma observação importante sobre a declaração de variáveis dentro dos diagramas de sequência pode ser feita com base na figura 5.5: após a declaração de uma variável num diagrama de sequência, ela pode ser utilizada várias vezes neste mesmo diagrama, da mesma forma como em um código-fonte. No caso específico do diagrama da figura 5.5, a variável retAge foi declarada na terceira mensagem UML (invocação de método) e reutilizada na quarta mensagem UML.

5.2.5. Definição das mensagens UML

No meta-modelo de testes proposto neste trabalho, as mensagens UML servem basicamente para representar a invocação de um método. Além disso, elas podem indicar qual é o valor esperado de retorno de um método, a fim de gerar uma asserção por NovaStudio, ou seja, para um teste ser efetuado sobre este método. O modo de definir uma mensagem UML na ferramenta de modelagem Enterprise Architect é ilustrado na figura abaixo.

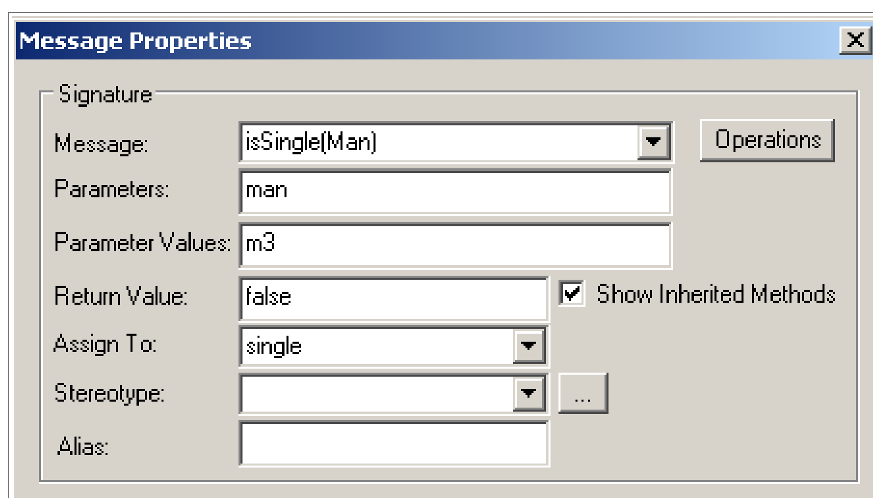

Figura 5.6. Exemplo de mensagem UML

Na figura 5.6 é mostrada a definição da mensagem UML que representa a invocação do método isSingle, ou seja, que verifica se um homem (Man) está solteiro. Neste caso, chamamos o método isSingle utilizando como parâmetro a variável m3, que já foi definida no diagrama de inicialização mostrado neste capítulo (figura 5.2). O resultado desta invocação será armazenado na variável single. O valor esperado de retorno deste método é false e é com esse valor, definido pelo Arquiteto, que se torna possível efetuar uma asserção sobre o método isSingle.

Os campos de propriedades de uma mensagem UML no Enterprise Architect, ilustrados na figura 5.6, que são utilizados pela funcionalidade de geração de testes são:

- **Message**: representa a assinatura do método (nome e tipo de parâmetros). Este campo é preenchido automaticamente por Enterprise Architect e não deve ser alterado pelo Arquiteto.

- **Parameters**: o nome dos parâmetros do método. Este campo também é preenchido automaticamente por Enterprise Architect, não devendo ser alterado.
- **Parameter Values**: este campo contém o valor dos parâmetros. No caso de múltiplos parâmetros, estes devem ser separados por vírgulas.
- **Return Value**: este campo contém o valor esperado de resposta do método invocado.
- **Assign To**: este campo contém a variável na qual a resposta do método invocado será atribuída.

A diferenciação básica entre os diagramas de sequência com asserção direta e indireta está no preenchimento do campo Return Value. No caso deste campo estar preenchido, então uma asserção entre o valor definido neste campo e a variável definida no campo Assign To será efetuada.

Um comentário importante sobre o ator UML, como o contido nos diagramas de sequência apresentados neste capítulo, deve ser feito: o ator Tester é emissor (a fonte) de todas as mensagens UML (chamadas a métodos) contidas nos diagramas de sequência, uma vez que ele representa o *framework* que executa o teste. Portanto, a troca direta de mensagens UML entre dois elementos do modelo (como classes ou objetos) não é suportada. No que diz respeito ao nome do ator UML, ele deve se chamar Tester, mas isso é apenas uma convenção – não é obrigatório.

5.3. Considerações finais

Neste capítulo foi apresentado a meta-modelagem proposta por este trabalho para a geração automatizada de testes. Esta meta-modelagem é baseada em diagramas de sequência e objetos UML. Três tipos de diagramas de sequência podem ser definidos em uma modelagem baseada no meta-modelo proposto aqui[14]: inicialização, com asserção direta e com asserção indireta.

O meta-modelo criado neste trabalho é principalmente inspirado nas meta-modelizações propostas por Fraikin e Leonhardt (2002) e Javed, Strooper e Watson (2007). Entretanto, a nossa proposta de meta-modelagem é original por ter criado os diagramas de sequência de inicialização. Além disso, o modo de conceber os testes, que permite de testar o resultado de um método diretamente ou indiretamente, através do uso de diagramas de sequência com asserção direta e indireta, também pode ser considerado como inovador.

O próximo capítulo descreve como foi feita a implementação da meta-modelagem para ser suportada pelo ambiente de desenvolvimento NovaStudio. Além disso, o capítulo 7 apresenta um caso completo da utilização da meta-modelagem proposta por este trabalho, através de um exemplo de modelagem, juntamente com o código gerado a partir deste exemplo (resultado do trabalho). Alguns fragmentos deste exemplo já foram utilizados neste capítulo para ilustração dos tipos de diagramas de sequência suportados pela meta-modelagem proposta.

14 NovaStudio também suporta somente a geração de esqueletos de métodos de testes, que devem ser preenchidos posteriormente pelo Desenvolvedor. Para isso, o Arquiteto deve criar um diagrama de sequência vazio, respeitando as regras de nomenclatura definidas neste capítulo. Maiores informações sobre essa funcionalidade são apresentadas no próximo capítulo.

6. REALIZAÇÃO

Este capítulo descreve como foi feita a realização e a implementação da funcionalidade de geração de testes sobre o ambiente de desenvolvimento NovaStudio. Serão apresentados aqui: os *frameworks* escolhidos para a execução dos testes e alguns conceitos fundamentais sobre eles; as adaptações feitas em NovaStudio para o suporte do meta-modelo de testes; como foi feita a geração dos *templates* e do código de testes utilizando-se Acceleo; o desenvolvimento do *plugin* Eclipse para a nova funcionalidade de geração de testes de NovaStudio; e um comentário sobre a documentação do projeto.

6.1. *Framework* para execução dos testes

Com a meta-modelagem definida, que foi apresentada no capítulo anterior, as etapas seguintes deste trabalho foram a pesquisa e escolha de um *framework* adaptado para a execução dos testes gerados. Por decisão da equipe NovaStudio, o *framework* Java EE com EJB3 era o alvo prioritário da equipe NovaStudio para geração de testes. Em um segundo momento, o *framework* WEB com Hibernate 3 deveria ser contemplado. Além disso, a equipe NovaStudio desejava que o *framework* escolhido fosse o JUnit ou algum *framework* derivado deste. Portanto, antes de passarmos a escolha do *framework*, apresentaremos na próxima seção alguns conceitos importantes de JUnit.

6.1.1. JUnit

JUnit[15] é um *framework* de teste para a linguagem de programação Java criado no ano de 1997 por Kent Beck e Eric Gamma. JUnit é um dos *frameworks* de teste Java mais utilizados. Ele é um *framework* de código-livre (*open-source*) e possui diversas extensões, como por exemplo DbUnit[16] para o teste de banco de dados.

O objetivo principal do JUnit é a automatização da execução de testes unitários. Uma das justificativas para isso é que tornando-se o processo de testes mais simples e rápido, permite que os desenvolvedores se interessem mais por testes: eles são infectados pelos testes e desenvolvem assim cada vez mais testes. Com isso, o código produzido se torna mais

15 http://www.junit.org/
16 http://www.dbunit.org/

estável e o programador mais produtivo, criando-se assim um ciclo vicioso (BECK; GAMMA, 1998).

O termo xUnit é utilizado para definir qualquer estrutura de testes unitários automáticos (BRANDÃO ET AL, 2005). JUnit é o *framework* xUnit mais conhecido. Os *frameworks* xUnit são baseados nos seguintes conceitos:

- ***Test fixture***: é o contexto do teste, ou seja, o conjunto de pré-condições (estado inicial) para o teste. Antes da execução de cada teste, o contexto deve ser inicializado (*setUp*) e após a execução de cada teste o contexto deve limpo (*tearDown*).
- ***Test suite***: é o conjunto dos testes que utilizam o mesmo *test fixture*. A ordem de execução dos testes não deve influenciar nos resultados.
- **Asserção**: é uma função que verifica o comportamento ou o estado de uma unidade do sistema durante o teste. É através de asserções, comparando-se o valor esperado de uma unidade do sistema com o valor durante a execução, que é possível verificar se um teste "passou" ou não.

6.1.2. *Framework* NovaStudio Java EE – Ejb3Unit

Como foi dito anteriormente, a equipe NovaStudio escolheu o EJB3 como implementação prioritária para a geração dos testes. A tecnologia Enterprise JavaBeans (EJB) é uma arquitetura de componentes de software do lado do servidor para a plataforma Java EE. No EJB, os objetos são gerenciados por um *container*, que os cria (instanciação), os utiliza, os reutiliza, ou seja, gerencia o ciclo de vida dos mesmos.

Para realizar o teste de um objeto Java "clássico", também conhecido como POJO (*Plain Old Java Object*), a instanciação (criação) deste objeto é feita juntamente com o preenchimento de seus atributos e associações com outros objetos. Após, no caso do teste de um método deste objeto, a invocação deste método é feita e as asserções podem ser efetuadas através do *framework* de testes para verificar se este método possui erros.

Por outro lado, o teste de EJBs é mais complicado que o teste de POJOs, já que neste caso necessitamos do *container* para realizar o gerenciamento do objeto a ser testado (KOSTELA, 2008). Portanto, não é possível utilizarmos somente o *framework* de teste JUnit, já que é necessário implantar (*deploy*) o *container* EJB, abordagem chamada de *in-container testing*, ou contornar a sua utilização, abordagem conhecida como *out-of-container testing*. Para esta decisão de escolha entre um teste com implantação (*deployment*) ou não do *container*, deve-se levar em consideração que um teste que necessita implantar todo o *container* antes de sua execução é normalmente mais lento do que um teste que consegue simular as funcionalidades do *container*.

A geração automática de testes impõe que configuração do *framework* seja simples e automatizada, já que não é desejado que o utilizador de NovaStudio perca seu tempo configurando o *framework* para executar os testes. Após a consideração de várias opções de *frameworks* de teste, como o Cactus[17] ou o DbUnit[18], Ejb3Unit[19] foi escolhido como *framework* de teste para a implementação NovaStudio Java EE. As justificativas para a escolha de Ejb3Unit foram:

17 http://jakarta.apache.org/cactus/
18 http://www.dbunit.org/
19 http://ejb3unit.sourceforge.net/

- Ejb3Unit é um *framework* de testes livre, sendo uma extensão do JUnit. Ele utiliza o *runner* de JUnit para a execução dos testes, suportando todas as asserções que JUnit suporta.
- Ejb3Unit utiliza a abordagem de *out-of-container testing*, não sendo necessário implantar todo *container* EJB3, o que permite uma execução mais rápida dos testes. Isso também facilita a utilização do *framework*, fazendo com que Ejb3Unit possua uma configuração simples (somente um arquivo).
- Ejb3Unit cria automaticamente um banco de dados em memória para a execução de testes, sem necessidade de configurações. Esta funcionalidade não foi encontrada nos outros *frameworks* analisados.
- Embora não se trate de um grande projeto de software livre, Ejb3Unit foi utilizado pelo conhecido software CodePro[20] para o teste de EJB3. Com isso, as preocupações da equipe NovaStudio sobre a confiabilidade deste projeto foram contentadas.

  Sobre as características mais técnicas de Ejb3Unit, ele propõe quatro classes bases de teste, sendo que duas são utilizadas por NovaStudio:

- **BaseEntityFixture**: é a classe utilizada para testar os EntityBeans de EJB3 (a camada *Business Object* de NovaStudio). Este teste é completamente automático, tendo como limitação o não suporte de associações *ManyToMany*. No capítulo 7, são mostrados alguns exemplos usando essa classe de teste.
- **BaseSessionBeanFixture**: é a classe utilizada para testar os SessionBeans (a camada *Business Service* de NovaStudio). Neste caso, Ejb3Unit nos fornece o ferramental necessário para realizar o teste, como por exemplo, um EntityManager para fazer a persistência dos objetos. No capítulo 7, também são mostrados exemplos de código utilizando essa classe de teste.

Partindo-se dos casos de teste padrão de NovaStudio, alguns destes foram reescritos à mão utilizando-se o *framework* Ejb3Unit. Com isso, foi possível testarmos as capacidades e limitações de Ejb3Unit, além de gerarmos bons exemplos para a posterior criação de *templates* Acceleo para a geração de código.

6.1.3. *Framework* NovaStudio WEB – JUnit

A escolha do *framework* de teste para a implementação NovaStudio WEB foi mais simples que a apresentada anteriormente (Java EE), já que neste caso não havia toda a problemática do *container*, uma vez que os objetos eram POJOs. Como podíamos utilizar JUnit diretamente para execução dos testes, então este *framework* foi priorizado por desejo da equipe NovaStudio.

Entretanto, na implementação WEB de NovaStudio, os objetos também são armazenados num banco de dados através do *framework* de mapeamento objeto-relacional Hibernate. Portanto, necessitávamos de uma banco de dados que fosse leve, para obtermos um bom desempenho nos teste, e que nos permitisse uma configuração simples. O banco de dados escolhido foi o HSQLDB[21].

20 http://www2.instantiations.com/codepro/analytix/default.htm
21 http://hsqldb.org/

O HSQLDB é um servidor de banco de dados de código aberto e escrito na linguagem Java. Ele possui como características ocupar pouco espaço em disco, ser facilmente configurável e possuir um bom desempenho na execução. Além disso, ele suporta a criação de um banco de dados em memória.

Para a implementação WEB de NovaStudio, utilizamos a mesma metodologia de trabalho que com a Java EE: escrevemos testes à mão para validarmos a nossa escolha de uso de JUnit juntamente com HSQLDB, além destes testes nos proverem bons exemplos para a criação de *templates* Acceleo para a posterior geração de código. A parte mais complicada da criação dos testes à mão foi a necessidade de escrevermos completamente os testes para a camada *Business Object* de NovaStudio, como, por exemplo, testes de inserção e supressão de objetos no banco de dados, já que no caso de Ejb3Unit esses testes eram completamente automatizados (bastava somente fazer uma instanciação da classe de testes específica de Ejb3Unit com os parâmetros corretos). Para os testes da camada *Business Service*, apenas algumas alterações foram efetuadas, como por exemplo o não-uso de Ejb3Unit e sim de JUnit com HSQLDB.

6.2. Adaptação de NovaStudio para o suporte do meta-modelo de testes

Com o meta-modelo validado pela equipe NovaStudio e os *frameworks* de teste escolhidos, a próxima etapa deste trabalho constitui na adaptação de NovaStudio para o suporte da nova meta-modelagem, que é composta por diagramas de sequência e objetos UML.

As etapas de adaptação de NovaStudio, que serão descritas nas próximas subseções, foram:

- modificação de sua arquitetura, para adicionar os módulos (componentes) que tratam da nova funcionalidade de teste;
- modificação do parser XMI para o suporte à leitura de diagramas de sequência e de objetos UML;
- modificação do meta-modelo de NovaStudio;
- modificação do módulo que efetua o povoamento do meta-modelo;

Além das etapas citadas acima, o módulo que faz o gerenciamento das preferências para a geração do código, tais como o tipo do *framework* (EJB, WEB) e sua persistência, também foi modificado para contemplar a nova funcionalidade de geração de testes. Por fim, foi criado um módulo que gera o código de testes, cujo os elementos ainda serão descritos neste capítulo.

6.2.1. Arquitetura de NovaStudio

Para o suporte da nova meta-modelagem por NovaStudio, tendo em vista que antes NovaStudio só suportava diagramas de classe e de atividade, foram necessárias modificações na arquitetura deste ambiente de desenvolvimento.

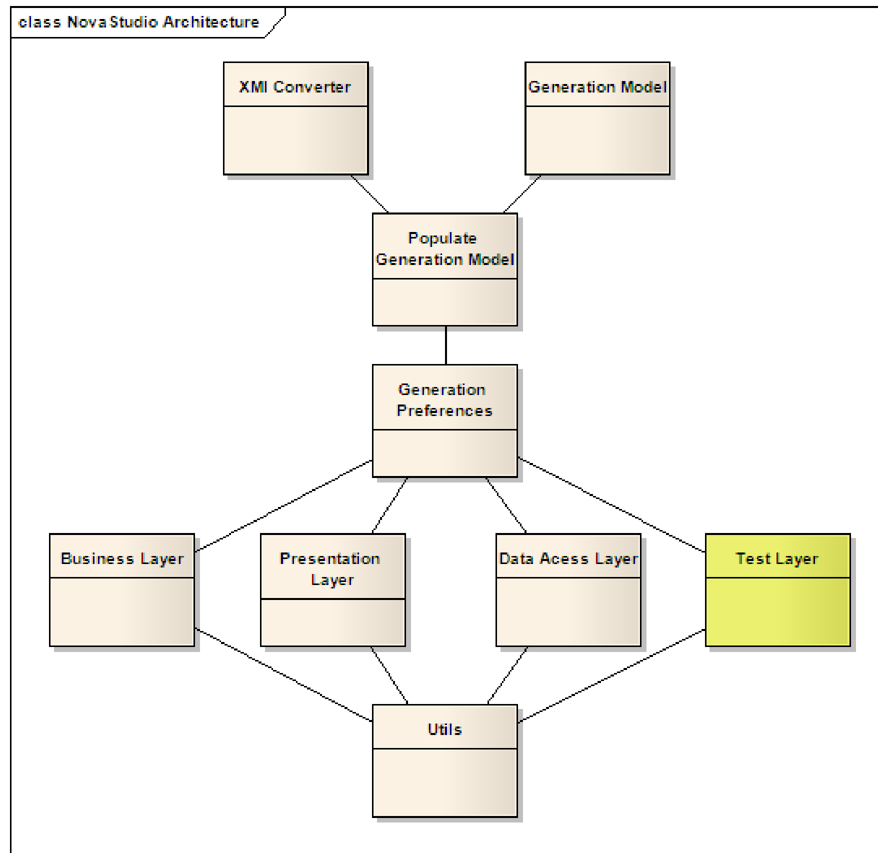

Figura 6.1. Arquitetura de NovaStudio

A figura 6.1 apresenta a visão lógica da nova arquitetura de NovaStudio. Nesta figura são apresentados os principais módulos de NovaStudio, além do módulo que foi adicionado por este trabalho, que está destacado em amarelo na figura acima (Test Layer).

Com exceção dos módulos Business, Presentation e Data Access Layer, todos os outros módulos ilustrados na figura 6.1 sofreram modificações por causa da adição da funcionalidade de geração de teste para NovaStudio. As funcionalidades de cada módulo são:

- **XMI Converter**: efetua a leitura dos arquivos XMI oriundos da ferramenta de modelagem UML e faz a primeira representação dos elementos do meta-modelo.
- **Generation Model**: este módulo contém todas as classes geradas pelo *Eclipse Modeling Framework* (EMF) para a gestão do meta-modelo.
- **Populate Generation Model**: é o módulo responsável pelo povoamento do meta-modelo EMF a partir das leitura efetuada pelo módulo XMI Converter.
- **Generation Preferences**: este módulo administra as preferências escolhidas pelo utilizador de NovaStudio para a geração de código.
- **Business Layer**: gera o código da camada *Business Service*.
- **Data Acess Layer**: gera o código da camada *Business Object*.
- **Presentation Layer**: gera o código das camadas de *Business Delegate* e *Service Locator.*

- **Test Layer**: este módulo é o responsável pela geração do código de testes para o código gerado por NovaStudio para as camadas *Business Service* e *Business Object*.
- **Utils**: este módulo possui diversas funções utilitárias usadas por diversos outros módulos. Uma funcionalidade interessante provida por este módulo é a formatação padrão do código gerado com auxílio da ferramenta Jalopy[22].

6.2.2. *Parser* XMI

Para a troca de informações entre a ferramenta de modelagem (Enterprise Architect) e NovaStudio, é o utilizado o padrão da OMG XML *Metadata Interchange* (XMI). NovaStudio suporta as versões 1.1 e 2.1 de XMI, que são bem distintas. Por isso, para a adaptação do *parser*, que antes só sabia realizar a leitura de diagramas de classe e de atividade UML, foi necessária a compreensão de ambos formatos XMI e do módulo XMI Converter.

A adaptação do parser era uma tarefa crítica, já que erros neste módulo são propagados para todos os outros módulos da arquitetura de NovaStudio. Portanto, foram efetuados diversos testes de não-regressão nesta etapa, a partir de arquivos XMI.

6.2.3. Modificação do meta-modelo de NovaStudio

Com o parser XMI alterado, passou-se a modificação do meta-modelo de NovaStudio baseado no *Eclipse Modeling Framework* (EMF). O EMF é constituído por diversas classes EMF que representam o meta-modelo, sendo normalmente representado por um arquivo XMI. O EMF gera automaticamente todo o código necessário (conjunto de classes) para a gestão de um meta-modelo, além um editor básico para esse meta-modelo.

Os elementos adicionados ao meta-modelo de NovaStudio, ilustrados na figura 6.2, para a representação da funcionalidade de geração de testes são:

- **TestClass**: contém toda a informação necessária para a geração de uma classe de teste. Ela herda diversos atributos da classe GenerationClass, que constitui a classe abstrata para a geração do código de uma classe (de persistência, de negócios, etc).
- **Sequence**: representa um diagrama de sequência (que posteriormente será um método de uma classe de teste).
- **AClass**: representação simplificada de uma classe UML, contendo somente os elementos necessários para a geração do teste.
- **AnObject**: representação de um objeto UML.
- **Message**: representação de uma mensagem UML.
- **Receiver**: contém as informações sobre a identidade do receptor de uma mensagem UML (classe ou objeto).
- **Variable**: pode representar ou um atributo de um objeto UML ou um parâmetro de uma mensagem UML.

---

22 http://jalopy.sourceforge.net/

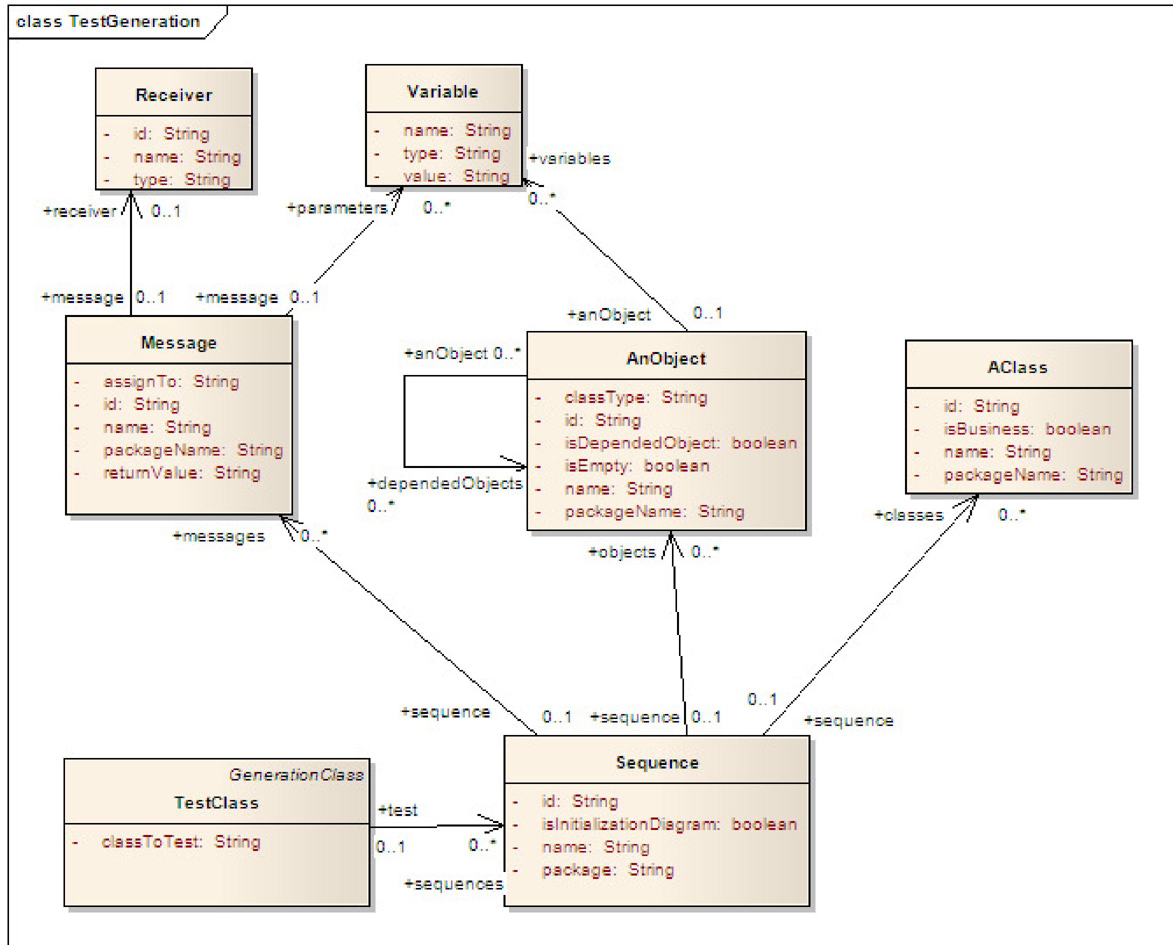

Figura 6.2. Classes adicionadas ao meta-modelo de NovaStudio

A figura 6.2 apresenta em detalhe as classes adicionadas ao meta-modelo de NovaStudio. Por sua vez, a figura 6.3 mostra o meta-modelo EMF resultante de NovaStudio após a adição dessas classes.

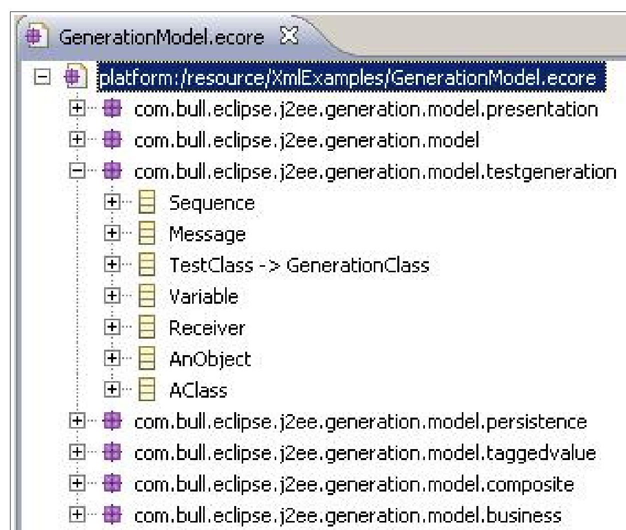

Figura 6.3. O meta-modelo EMF após as modificações

6.2.4. Povoamento do meta-modelo

Com o meta-modelo de NovaStudio modificado, a próxima etapa do trabalho realizada foi a modificação do módulo Populate Generation Model, que povoa este meta-modelo. O ato de povoar o meta-modelo consiste em instanciá-lo, ou seja, gerar um modelo a partir do meta-modelo e das informações obtidas do parser XMI.

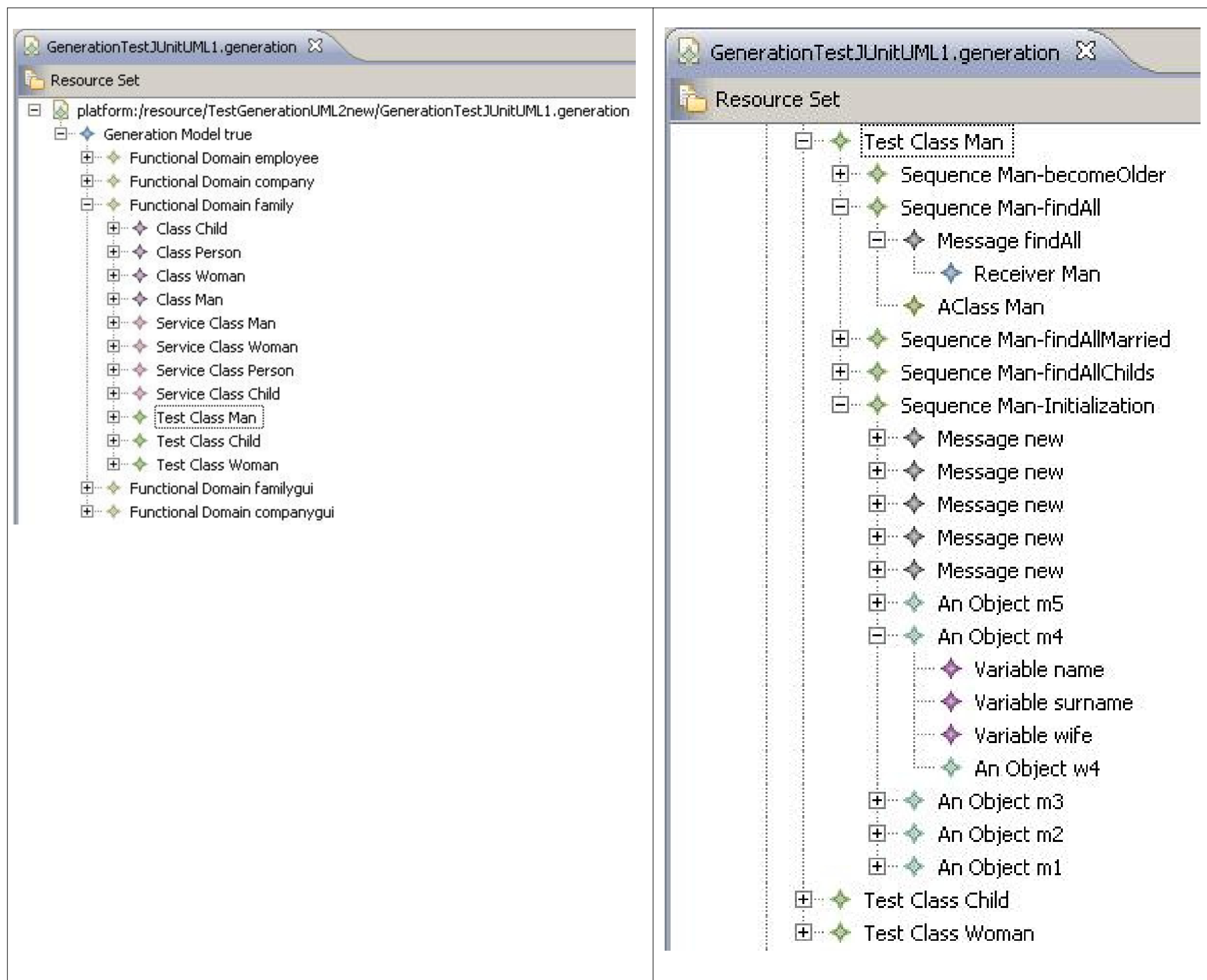

Figura 6.4. O arquivo .generation completo (à esquerda) e em detalhe a parte deste arquivo relativa a geração de testes (à direita)

A figura 6.4 mostra um exemplo de povoamento, no caso, o arquivo .generation que é resultado deste povoamento. A partir das informações lidas do parser XMI e do meta-modelo de NovaStudio, é gerado um arquivo .generation (EMF), que consiste no modelo (instância do meta-modelo de NovaStudio). Este arquivo .generation é utilizado para a posterior geração de código por NovaStudio.

6.3. Geração de testes utilizando Acceleo

Conforme já foi apresentado no capítulo 4, a equipe NovaStudio escolheu Acceleo como motor para a geração do código de teste. Nesta seção, apresentaremos como é feita a geração de *templates* Acceleo e em seguida detalhes sobre a geração do código de testes para os *frameworks* NovaStudio EJB3 e WEB através desses *templates*.

6.3.1. Geração de *templates* com Acceleo

A geração de código com Acceleo baseia-se na criação de *templates* (padrões de geração) que descrevem a estrutura dos arquivos (códigos) a serem gerados. Um *template* é baseado em um meta-modelo e, desta forma, o *template* pode manipular elementos de modelos derivado deste meta-modelo.

O meta-modelo de um *template* é definido por seu *Uniform Ressource Identifier* (URI) no cabeçalho deste *template*. Os *templates* contém o ferramental necessário para gerar o código-fonte a partir de um modelo (derivado de um meta-modelo). A figura 6.5 ilustra o modo de funcionamento de Acceleo.

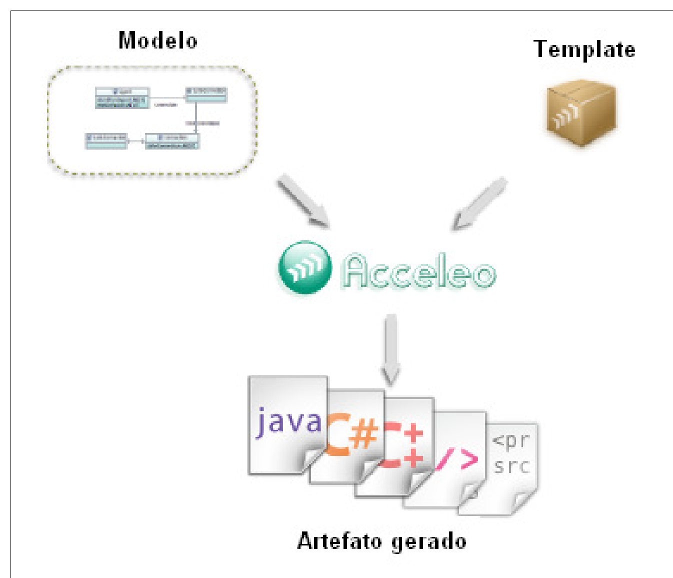

Figura 6.5. Geração de código por Acceleo

Um *template* é um arquivo composto por um ou vários *scripts* Acceleo que permitem gerenciar a geração de código (MUSSET; JULLIOT; LACRAMPE, 2009). Um *script* Acceleo começa pela baliza <%script%>. Já o nome do arquivo a ser gerado é definido no atributo file da baliza *script*.

As funcionalidades mais complexas a serem implementados através de macros da linguagem Acceleo podem ser importadas de classes Java. Este mecanismo de importação e utilização dentro de um script Acceleo se chama de serviços Java.

A figura 6.6 mostra um exemplo simplificado de um *template* Acceleo usado para a geração de testes. Neste exemplo, para cada elemento do modelo de testes será gerado um arquivo Java com seu nome e este será armazenado na pasta “test/exemple/simple”.

```
<%
metamodel com.bull.eclipse.j2ee.generation.model.testgeneration
import com.bull.eclipse.testgeneration.templates.ejb3.constants
import com.bull.eclipse.testgeneration.templates.ejb3.utilsEJB3
import com.bull.eclipse.testgeneration.GenerationServices
%>

<%script type="testgeneration.TestClass" name="fileName"%>
test/exemple/simple/<%name%>.java

<%script type="testgeneration.TestClass" name="testEJB3JUnit4" file="<%fileName%>"%>
package <%package%>.<%packageTest%>;

<%addImports()%>

/**
 * This is the test for the <%name%><%businessImplSuffix%> class.
 * <p> Test framework: - EJB3Unit (JUnit3) </p>
 */
public class <%name%><%testSuffix%> extends
    BaseSessionBeanFixture<<%name%><%businessImplSuffix%>>{

    private static final Class[] usedBeans = { <%addUsedBeans%> };

    public <%name%><%testSuffix%>() {
        super(<%name%><%businessImplSuffix%>.class, usedBeans);
    }

} <%-- End of the class --%>
```

URI do meta-modelo
Importação de um template Acceleo
Importação de um serviço Java
Elemento do meta-modelo
Nome do script
Nome do arquivo a gerar
Utilização de um serviço Java
Chamada a um script Acceleo
Um comentário

Figura 6.6. Exemplo de um *template* Acceleo

6.3.2. Geração do código de testes

Para a geração do código de testes, foi escrito um *template* para cobrir a geração de código para cada tipo de teste desejado. Por exemplo, para o teste da camada *Business Object* do *framework* EJB3 foi escrito um *template* específico para este caso. Além disso, foram criados *templates* com funções auxiliares além de classes Java com mesmo propósito.

Como Ejb3Unit suporta testes para o *framework* JUnit nas versões 3 e 4, então foram escritos *templates* para cobrir ambas versões no caso do *framework* NovaStudio EJB3. Além disso, também foram escritos *templates* para as versões 3 e 4 de JUnit para o *framework* NovaStudio WEB.

Além da geração de código de testes baseado nas informações definidas no diagramas de sequência e objetos UML pelo Arquiteto, também foi desenvolvido um conjuntos de *templates* para gerar somente esqueletos de testes (estrutura do teste com métodos de teste vazios) para todos os tipos de teste suportados por NovaSudio. Neste caso, as informações sobre diagramas de sequência, se existirem, são ignoradas. O principal objetivo dessa funcionalidade é proporcionar a geração de código de testes para os casos em que nenhum diagrama de sequência foi criado pelo Arquiteto. Além disso, os esqueletos são gerados somente para os testes da camada *Business Service*. Já os testes da camada *Business Object* são gerados de forma completa, já que estes são independentes das informações dos diagramas de sequência.

6.4. Desenvolvimento do *plugin* TestGeneration

NovaStudio é um ambiente de desenvolvimento composto por um conjunto de *plugins* Eclipse. Portanto, a integração da funcionalidade de geração de testes foi realizada através da criação de um *plugin*, que foi chamado de TestGeneration. Este *plugin* possui dependências com os módulos (*plugins*) GenerationPreferences e Utils de NovaStudio, como foi ilustrado na figura 6.4[23] que apresenta a arquitetura de NovaStudio. Além disso, ele também possui dependência com o *plugin* Acceleo.

O *plugin* TestGeneration contém todos os *templates* Acceleo e as classes Java necessárias para a geração do código de testes para as camadas *Business Object* e *Business Service* de NovaStudio. Além disso, ele possui as bibliotecas necessárias para a compilação e a execução dos testes, como Ejb3Unit e JUnit (nas versões 3 e 4).

O desenvolvimento da funcionalidade de geração de testes, através da criação do *plugin* TestGeneration, implicou em modificações no módulo GenerationPreferences, que gerencia as preferências escolhidas pelo usuário de NovaStudio para a geração de código. Este módulo possui entre seus elementos a interface do ambiente de desenvolvimento NovaStudio, desenvolvida com o Standard Widget Toolkit (SWT) do Java.

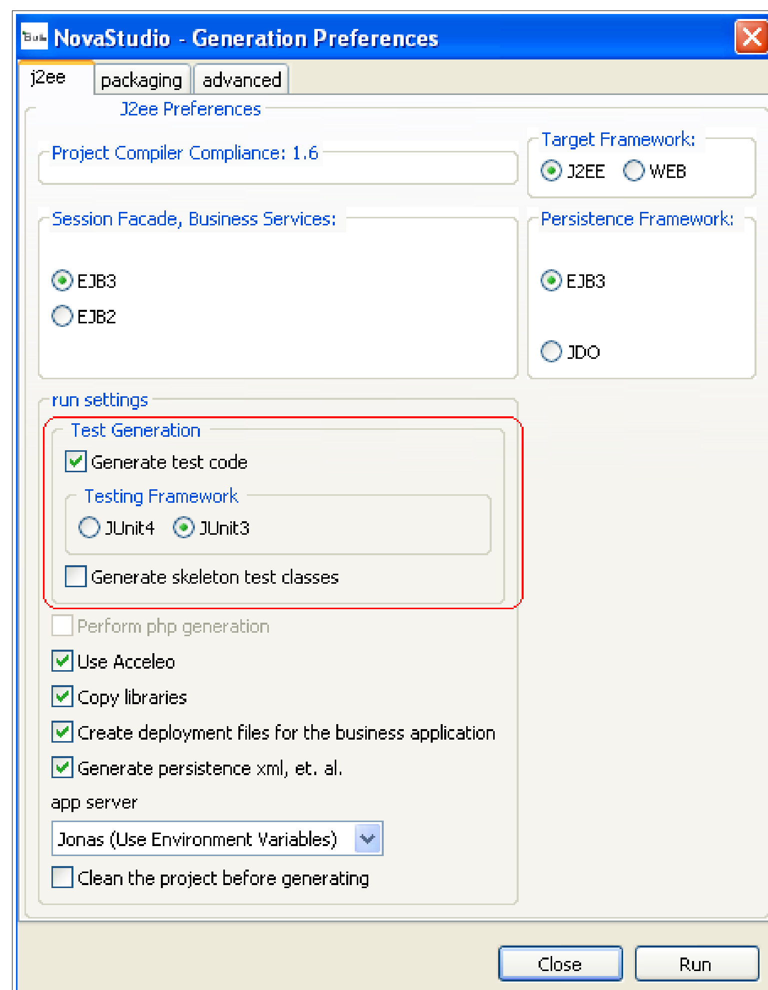

Figura 6.7. Interface "Generation Preferences"

23 Nesta figura, o *plugin* TestGeneration foi apresentado como Test Layer.

A figura 6.7 apresenta a interface principal para geração de código de NovaStudio e as modificações nela efetuadas (destacadas em vermelho). Com essa interface, o Desenvolvedor escolhe como o código será gerado, ou seja, qual será o *framework* (EJB3, WEB) e sua persistência, entre outras opções. Em seguida, ele aperta o botão Run e o código será gerado para todas as camadas da arquitetura suportada por NovaStudio[24].

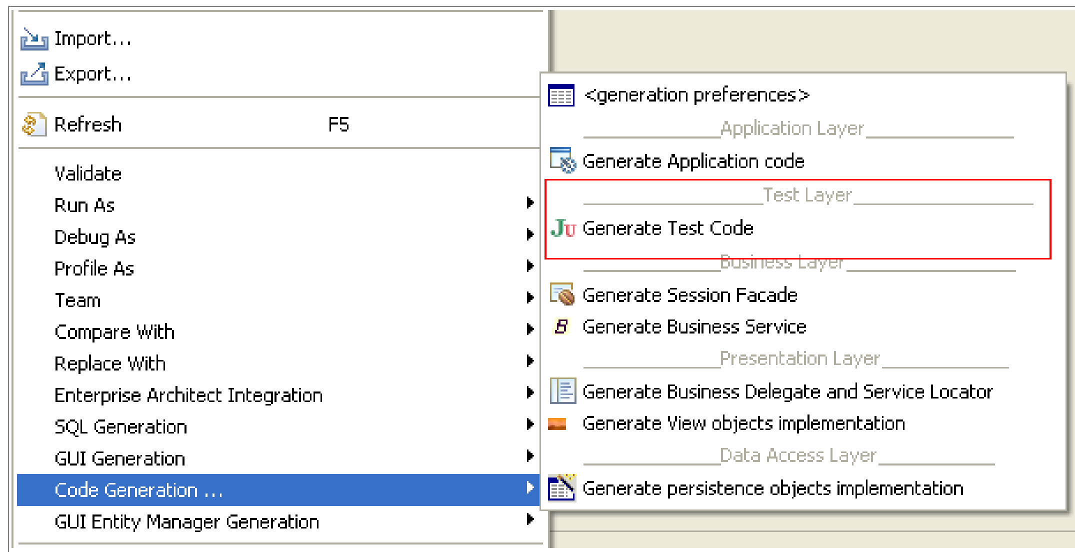

Figura 6.8. Interface “Generate Test Code”

NovaStudio também suporta a geração de código somente para um camada. A interface para essa funcionalidade, com sua modificação (destacada em vermelho), é apresentada na figura 6.8.

6.5. Documentação do projeto

O desenvolvimento da documentação constitui uma parte importante deste trabalho. Para a documentação técnica do código-fonte, foi utilizado a ferramenta Javadoc[25] para todas as classes escritas na linguagem Java, além de outras formas de comentários. Além disso, NovaStudio possui um guia de utilização com mais de 200 páginas. A complementação deste guia com a descrição da nova funcionalidade de geração de testes também foi efetuada.

6.6. Considerações finais

Neste capítulo foi descrita a realização e a implementação da funcionalidade de geração de testes para o ambiente de desenvolvimento NovaStudio. Esta nova funcionalidade foi testada através de um conjunto de casos de teste, sendo que o próximo capítulo apresenta um caso de teste completo para melhor compreensão do trabalho efetuado. Todos os objetivos deste trabalho foram alcançados e a funcionalidade de geração de testes a partir de uma modelagem UML foi validada pela equipe NovaStudio.

24 As camadas da arquitetura suportada por NovaStudio foram apresentadas na seção 4.4.
25 http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html

7. RESULTADOS

Este capítulo descreve os resultados obtidos neste trabalho através da apresentação de um dos casos de teste utilizados para a validação da funcionalidade de geração automatizada de testes no ambiente de desenvolvimento NovaStudio. A primeira seção deste capítulo apresenta o cenário de teste. A segunda seção mostra a estrutura do código de testes gerada a partir deste cenário. A terceira seção apresenta os diagramas de sequência criados pelo Arquiteto e o código correspondente gerado por NovaStudio. A quarta seção mostra o código de teste gerado para a camada *Business Object*. A quinta seção apresenta as considerações finais sobre os resultados obtidos.

7.1. Cenário de teste

Para uma melhor compreensão do que foi feito neste trabalho, este capítulo apresenta um caso de teste completo, que foi utilizado na validação da funcionalidade de geração de testes de NovaStudio. Este caso de teste já existia para a validação da geração de código por NovaStudio e foi estendido para validação da nova funcionalidade de geração de testes, através da escrita de diagramas de sequência e criação de objetos UML. A implementação de NovaStudio escolhida para o caso de teste descrito neste capítulo é a Java EE EJB3, juntamente com o *framework* de testes Ejb3Unit com JUnit na versão 4.

Esta seção apresenta o cenário deste caso de teste, ou seja, os diagramas de classe que são utilizados para geração de código por NovaStudio. O código gerado para estes diagramas não será apresentado aqui, já que não é resultado deste trabalho. Entretanto, um exemplo de NovaStudio para este código gerado a partir de diagramas de classse pode ser consultado em (MEDEIROS, 2009).

A figura 7.1 apresenta o diagrama de classe que originará o código para a camada *Business Object* da arquitetura lógica do Java EE suportada por NovaStudio (mostrada no capítulo 4). Neste tipo de diagrama de classes, são definidos os atributos das classes e as relações entre elas. No caso da implementação utilizando-se EJB3, o código gerado por NovaStudio será baseado nos *EntityBeans* do EJB3.

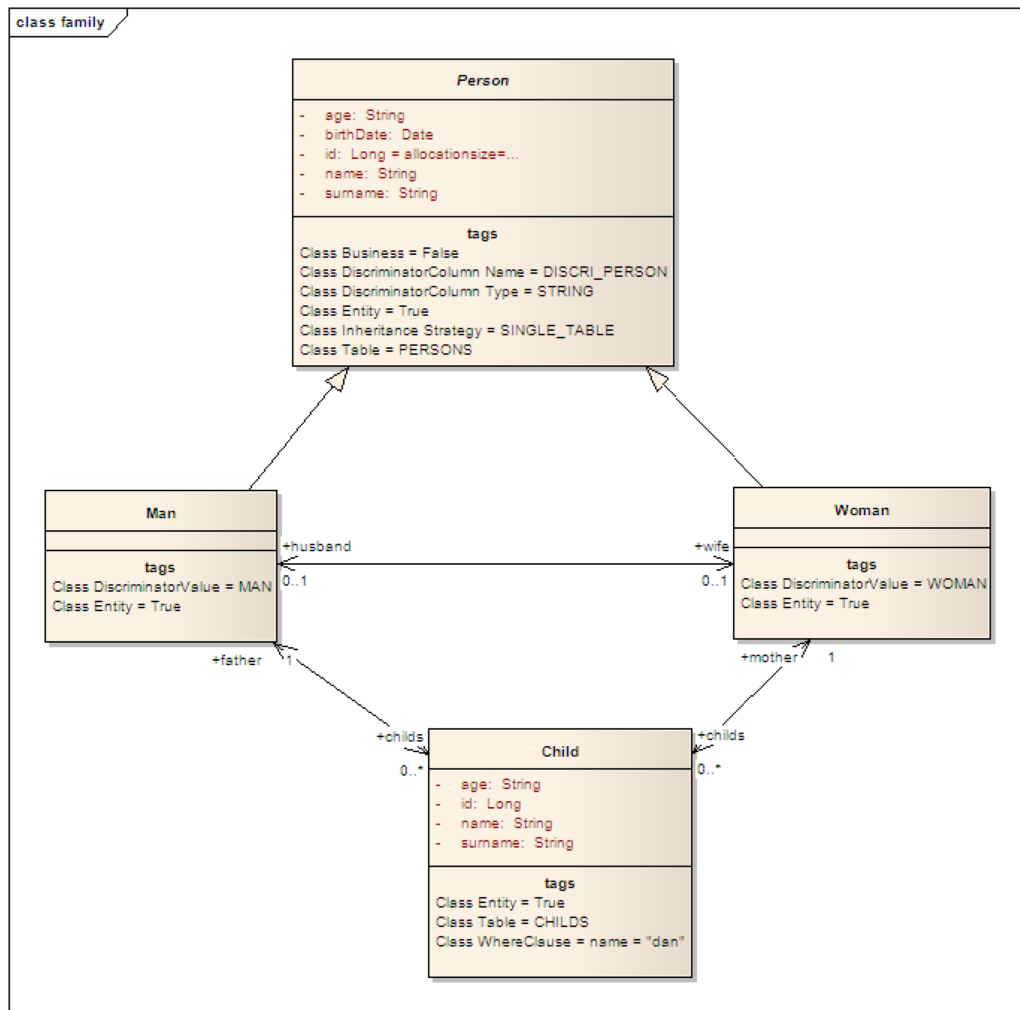

Figura 7.1. Diagrama de classe para a camada *Business Object*

Por sua vez, a figura 7.2 apresenta o diagrama de classes a partir do qual é gerado o código para a camada *Business Service*. Neste tipo de diagrama de classes, são definidos os métodos da lógica de negócios da aplicação. No caso da implementação utilizando-se EJB3, o código gerado utilizará os *SessionBeans* de EJB3.

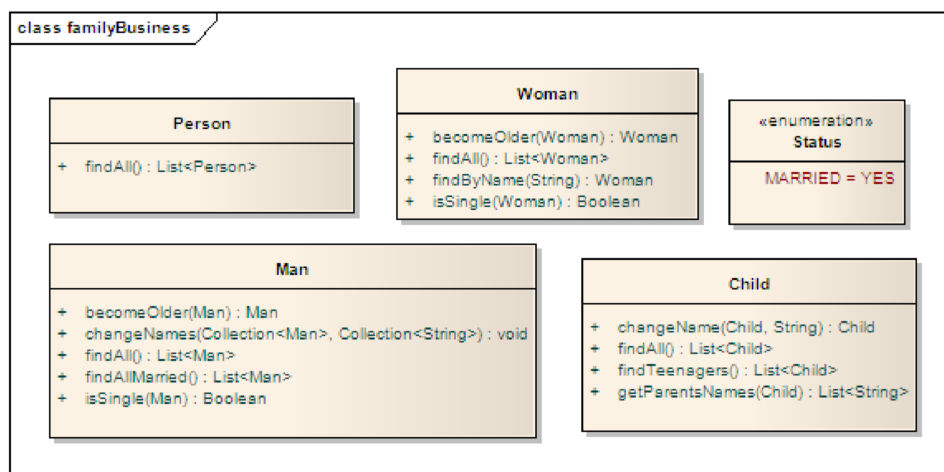

Figura 7.2. Diagrama de classe para a camada *Business Service*

7.2. Diagrama de sequência de inicialização

A partir dos diagramas de classes apresentados na seção anterior, pode-se inferir quais são os atributos e métodos que compõem cada classe. Nesta seção mostramos que a partir dos diagramas de sequência de inicialização, podemos gerar o código que declara os objetos que compõem o teste, além de também declarar suas dependências com outros objetos.

Neste caso de teste, será apresentado o resultado da geração de código de teste para a classe Man que foi definida nos diagramas das figuras 7.1 e 7.2. Além disso, conforme já foi dito, será considerado a implementação Java EE EJB3 de NovaStudio com o *framework* de testes Ejb3Unit com a versão 4 de JUnit.

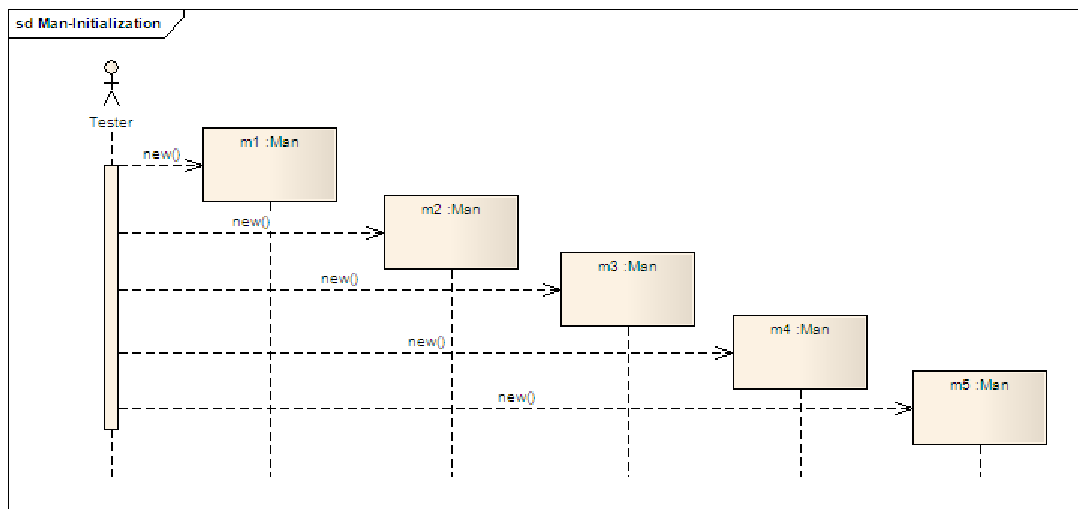

Figura 7.3. Diagrama de sequência de inicialização da classe Man

A figura 7.3 reapresenta o diagrama de sequência de inicialização para a classe Man, já mostrado no capítulo 5. A partir deste diagrama é gerada a estrutura do código de testes e também os métodos setUp e tearDown, que, respectivamente, inicializam e limpam o contexto do teste.

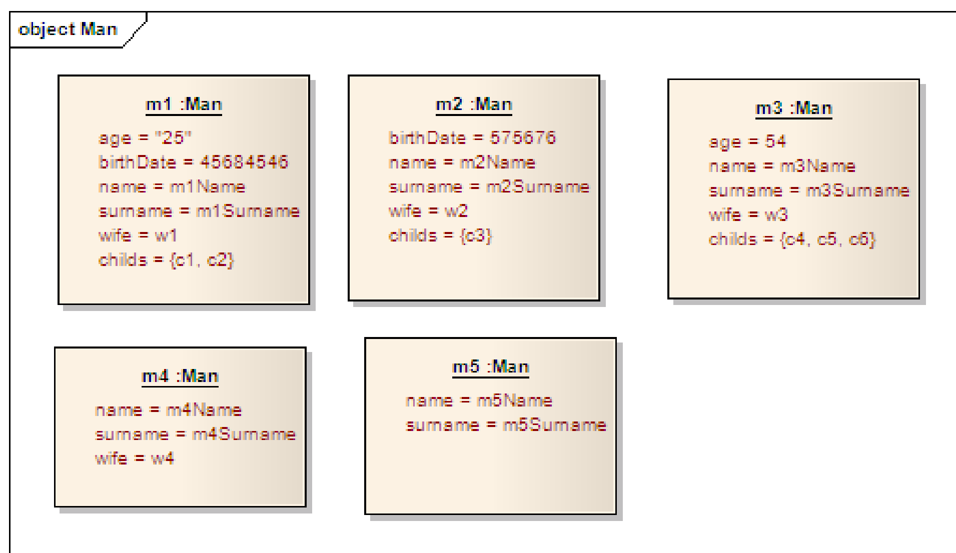

Figura 7.4. Diagrama de objetos da classe Man

Por sua vez, a figura 7.4 apresenta o diagrama de objetos da classe Man, mostrando quais são os valores dos atributos para cada um dos objetos declarados no diagrama de sequência de inicialização da classe Man.

7.2.1. Estrutura do teste

A partir do diagrama de sequência de inicialização, além da informação obtida dos diagramas de classe, é possível gerar-se a estrutura do teste. Isto significa a definição de todos os *imports* de classes e bibliotecas, além da declaração dos objetos que compõem o teste e também dos objetos associados a estes. Além disso, na estrutura do teste também existe um método especial gerado por NovaStudio (testInitialization) que verifica se as classes que geram a persistência foram bem inicializadas. Se este teste não passar, então nenhum outro teste passará.

A figura 7.5 apresenta a estrutura do código de teste gerado para a classe Man.

```
/*
 * This code was generated by NovaStudio©. Copyright BULL reserved.
 * ONLY edit the parts signaled by Start of user code.
 * File: ManTest.java
 * Template: testEJB3JUnit4.mt
 */
package com.bull.test.family.test;

import com.bm.testsuite.junit4.BaseSessionBeanJUnit4Fixture;

import com.bull.test.family.dto.Child;
import com.bull.test.family.dto.Man;
import com.bull.test.family.dto.Woman;
import com.bull.test.family.dto.persistence.ChildBean;
import com.bull.test.family.dto.persistence.ChildFactory;
import com.bull.test.family.dto.persistence.ManBean;
import com.bull.test.family.dto.persistence.ManFactory;
import com.bull.test.family.dto.persistence.WomanBean;
import com.bull.test.family.dto.persistence.WomanFactory;
import com.bull.test.family.service.business.ManBusiness;

import org.junit.After;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertNull;
import static org.junit.Assert.assertTrue;

import org.junit.Before;
import org.junit.Test;

import java.util.Date;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import javax.persistence.EntityManager;
import javax.persistence.Query;

//Start of user code @Enter your imports here @warning: do not remove this comm
ent

//End of user code @End of user imports @warning: do not remove this comment
```

```
/**
 * This is the test for the ManBusiness class.
 * <p> Test framework: - EJB3Unit (JUnit4)  </p>
 */
@SuppressWarnings("unchecked")
public class ManTest extends BaseSessionBeanJUnit4Fixture<ManBusiness> {
    /**
     * The EntityBean of the ManBusiness class and its associations
     * (if any) and the associations of its associations (recursive).
     * This is used by EJB3Unit to perform the EJB3 injection.
     */
    private static final Class[] usedBeans = {
                                                ChildBean.class, ManBean.class,
                                                WomanBean.class
                                             };

    /**
     * The EntityManager
     */
    EntityManager em;

    /**
     * The class to test
     */
    ManBusiness manBusiness;

    /*
     * The objects declared in the ManInitialization sequence diagram
     * from the UML model.
     */
    Man m5;
    Man m4;
    Man m3;
    Man m2;
    Man m1;

    /*
     * The objects declared in the ChildInitialization sequence diagram
     * from the UML model.
     * Child is an associated class with the Man class.
     */
    Child c7;
    Child c6;
    Child c4;
    Child c5;
    Child c3;
    Child c2;
    Child c1;

    /*
     * The objects declared in the WomanInitialization sequence diagram
     * from the UML model.
     * Woman is an associated class with the Man class.
     */
    Woman w5;
    Woman w4;
    Woman w3;
    Woman w2;
    Woman w1;

    //Start of user code @Enter your global objects here @warning: do not remov
e this comment

    //End of user code @End of user global objects @warning: do not remove this
comment
```

```
    /**
     * The constructor for the ManTest
     */
    public ManTest() {
        super(ManBusiness.class, usedBeans);
    }

    /**
     * This is a default test generated by NovaStudio. The objective is to
     * test if the dependency injection performed by EJB3Unit works well.
     */
    @Test
    public void testInitialization() {
        assertNotNull("The class to test manBusiness must not be null",
                      manBusiness);
        assertNotNull("The EntityManager must not be null", em);
        assertTrue("The EntityManager must be open", em.isOpen());
    }
```

Figura 7.5. Estrutura do código de teste para a classe ManTest

Para termos uma noção da ordem do código, após o código apresentado acima, é gerado o código para cada método a ser testado que possui um diagrama de sequência definido. Em seguida, é gerado o código para os métodos setUp e tearDown que serão apresentados na próxima subseção. Para completar o código de teste, é gerado este pequeno trecho (figura 7.6) que permite que o Desenvolvedor escreva seus próprios métodos de testes. Acceleo suporta a geração continuada, mas o código escrito entre as *tags* “Start of user code” e “End of user code” não é alterado por uma nova geração de código.

```
    //Start of user code @Enter your test methods here @warning: do not remove
this comment

    //End of user code @End of user test methods @warning: do not remove this c
omment
}
```

Figura 7.6. Trecho final do código para a classe ManTest

7.2.2. Métodos setUp e tearDown

Os métodos setUp, que faz a inicialização do contexto do teste, e tearDown, que limpa o contexto, também são gerados a partir das informações obtidas no diagrama de sequência de inicialização. A figura 7.7 mostra o código gerado para o método setUp.

```
   /**
     * The setUp method -> executed before each test method.
     * @throws Exception if any problem occurred while executing the
     * setUp method
     */
    @Before
    public void setUp() throws Exception {
        super.setUp();
        manBusiness = this.getBeanToTest();
        em = this.getEntityManager();
        if (em.getTransaction().isActive()) {
            em.getTransaction().commit();
        }
```

```
        em.getTransaction().begin();

        /*
         * Constructing all objects of the tested class and
         * setting its variables.
         */
        m5 = ManFactory.create();
        m5.setName("m5Name");
        m5.setSurname("m5Surname");
        em.persist(m5);

        m4 = ManFactory.create();
        m4.setName("m4Name");
        m4.setSurname("m4Surname");
        em.persist(m4);

        m3 = ManFactory.create();
        m3.setAge("54");
        m3.setName("m3Name");
        m3.setSurname("m3Surname");
        em.persist(m3);

        m2 = ManFactory.create();
        m2.setBirthDate(new Date(575676L));
        m2.setName("m2Name");
        m2.setSurname("m2Surname");
        em.persist(m2);

        m1 = ManFactory.create();
        m1.setAge("25");
        m1.setBirthDate(new Date(45684546L));
        m1.setName("m1Name");
        m1.setSurname("m1Surname");
        em.persist(m1);
        em.flush();


        /*
         * Constructing all associated objects and setting
         * its relationships with the tested class.
         */
        w4 = WomanFactory.create();
        w4.setName("w4Name");
        w4.setSurname("m4Surname");
        w4.setHusband(m4);
        em.persist(w4);

        w3 = WomanFactory.create();
        w3.setName("w3Name");
        w3.setSurname("w3Surname");
        w3.setHusband(m3);
        em.persist(w3);

        c6 = ChildFactory.create();
        c6.setAge("3");
        c6.setName("c6Name");
        c6.setSurname("m3Surname");
        c6.setFather(m3);
        em.persist(c6);
        c5 = ChildFactory.create();
        c5.setName("c5Name");
        c5.setSurname("m3Surname");
        c5.setFather(m3);
        em.persist(c5);

        c4 = ChildFactory.create();
        c4.setName("c4Name");
```

```
        c4.setSurname("m3Surname");
        c4.setAge("3");
        c4.setFather(m3);
        em.persist(c4);

        w2 = WomanFactory.create();
        w2.setAge("45");
        w2.setName("w2Name");
        w2.setSurname("m2Surname");
        w2.setHusband(m2);
        em.persist(w2);

        c3 = ChildFactory.create();
        c3.setName("c3Name");
        c3.setSurname("m2Surname");
        c3.setAge("15");
        c3.setFather(m2);
        em.persist(c3);

        w1 = WomanFactory.create();
        w1.setAge("32");
        w1.setBirthDate(new Date(4564779L));
        w1.setName("w1Name");
        w1.setSurname("m1Surname");
        w1.setHusband(m1);
        em.persist(w1);

        c1 = ChildFactory.create();
        c1.setAge("12");
        c1.setName("c1Name");
        c1.setSurname("m1Surname");
        c1.setFather(m1);
        em.persist(c1);

        c2 = ChildFactory.create();
        c2.setName("c2Name");
        c2.setSurname("m1Surname");
        c2.setFather(m1);
        em.persist(c2);

        /*
         * Setting the relationships of the tested class with
         * the associated objects.
         */
        m4.setWife(w4);
        m3.setWife(w3);
        m3.addChilds(c4);
        m3.addChilds(c5);
        m3.addChilds(c6);
        m2.setWife(w2);
        m2.addChilds(c3);
        m1.setWife(w1);
        m1.addChilds(c1);
        m1.addChilds(c2);
        em.flush();

        //Start of user code @Enter your setUp code here @warning: do not remov
e this comment

        //End of user code @End of user setUp code @warning: do not remove this
comment
    }
```

Figura 7.7. Código gerado para o método setUp da classe ManTest

Por sua vez, a figura 7.8 apresenta o código gerado para o método setUp.

```
    /**
     * The tearDown method -> executed after each test method.
     * @throws Exception if any problem occurred while executing the
     * tearDown method
     */
    @After
    public void tearDown() throws Exception {
        if (em.getTransaction().isActive()) {
            em.getTransaction().commit();
        }
        em.getTransaction().begin();
        Query query;
        /*
         * Set null all associations of the ManBean
         * class.
         */
        query = em.createQuery("from ManBean");
        List<ManBean> listManBean = query.getResultList();
        for (ManBean bean : listManBean) {
            if (bean != null) {
                bean.setChilds(null);
                bean.setWife(null);
                em.persist(bean);
            }
        }
        /*
         * Set null all associations of the classes associated with
         * ManBean class.
         */
        query = em.createQuery("from ChildBean");
        List<ChildBean> listChildBean = query.getResultList();
        for (ChildBean bean : listChildBean) {
            if (bean != null) {
                bean.setFather(null);
                em.persist(bean);
            }
        }
        query = em.createQuery("from WomanBean");
        List<WomanBean> listWomanBean = query.getResultList();
        for (WomanBean bean : listWomanBean) {
            if (bean != null) {
                bean.setHusband(null);
                em.persist(bean);
            }
        }
        em.flush();
        /*
         * Deleting all beans of the ManBean class
         * from the "database" and its associations.
         */
        query = em.createQuery("delete from ManBean");
        query.executeUpdate();
        query = em.createQuery("delete from ChildBean");
        query.executeUpdate();
        query = em.createQuery("delete from WomanBean");
        query.executeUpdate();
        em.getTransaction().commit();

        //Start of user code @Enter your tearDown code here @warning: do not remove
this comment

        //End of user code @End of user tearDown code @warning: do not remove this
comment
        super.tearDown();
    }
```

Figura 7.8. Código gerado para o método tearDown da classe ManTest

7.3. Diagramas de sequência para o teste de métodos

Nesta seção, é mostrado três diagramas de sequência e o código de teste correspondente gerado por NovaStudio para esses diagramas. Lembrando que cada diagrama de sequência, com exceção do diagrama de sequência de inicialização, representa o teste de um método de uma classe. São apresentados os diagramas de sequência com asserção direta que representam o teste do métodos isSingle e findAllMarried da classe Man, além do diagrama de sequência com asserção indireta que representa o teste do método becomeOlder da classe Man.

7.3.1. Método isSingle

A figura 7.9 apresenta o diagrama de sequência para o método isSingle. Por sua vez, a figura 7.10 mostra o código gerado a partir deste diagrama de sequência.

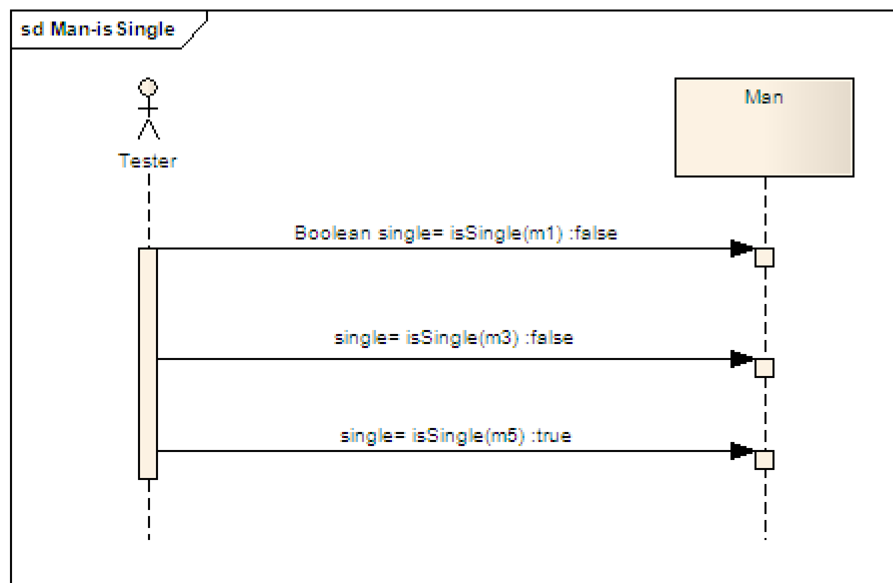

Figura 7.9. Diagrama de sequência para o método isSingle

```
    /**
    * The test for the method isSingle
     */
    @Test
    public void testIsSingle() {
        Boolean single = manBusiness.isSingle(m1);
        em.flush();
        assertEquals("The expected value [false] is not equal to the object sin
gle.", new Boolean(false), single);

        single = manBusiness.isSingle(m3);
        em.flush();
        assertEquals("The expected value [false] is not equal to the object sin
gle.", new Boolean(false), single);
        single = manBusiness.isSingle(m5);
        em.flush();
        assertEquals("The expected value [true] is not equal to the object sing
le.", new Boolean(true), single);
```

```
        //Start of user code @Enter your implementation for the method testIsSi
ngle here @warning: do not remove this comment

        //End of user code @End of user implementation for the method testIsSin
gle @warning: do not remove this comment
    }
```

Figura 7.10. Código gerado para o método testIsSingle da classe ManTest

7.3.2. Método findAllMarried

A figura 7.11 apresenta o diagrama de sequência para o método findAllMarried. Por sua vez, a figura 7.12 mostra o código gerado a partir deste diagrama de sequência.

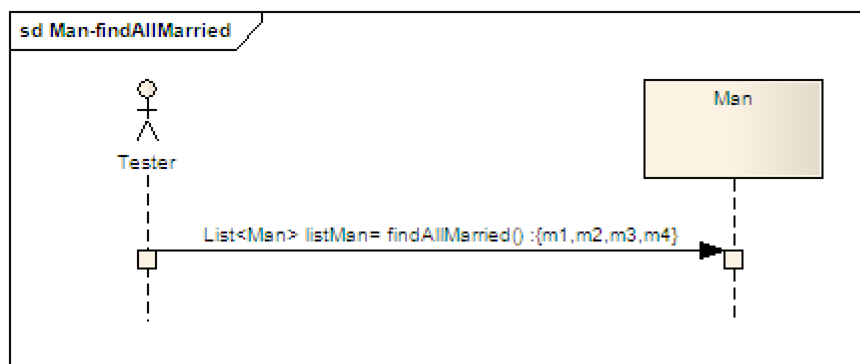

Figura 7.11. Diagrama de sequência para o método findAllMarried

```
    /**
     * The test for the method findAllMarried
     */
    @Test
    public void testFindAllMarried() {
        List<Man> listMan = manBusiness.findAllMarried();
        em.flush();

        //Start of user code @Enter your assertCollection1 implementation for t
he method findAllMarried() here @warning: do not remove this comment
        /*
         * These following asserts only assure the equality of the simple
         * attributes (JavaClasses) through the method toString() generated
         * by NovaStudio. The relations between user defined classes are not
         * tested. If desired, you can implement your own assert code below.
         */
        Map<String, Man> mans2 = new HashMap<String, Man>(listMan.size());
        for (Man man : listMan) {
            mans2.put(man.toString(), man);
        }
        assertTrue("m1 is not contained in the map.",
                   mans2.containsKey(m1.toString()));
        assertTrue("m2 is not contained in the map.",
                   mans2.containsKey(m2.toString()));
        assertTrue("m3 is not contained in the map.",
                   mans2.containsKey(m3.toString()));
        assertTrue("m4 is not contained in the map.",
                   mans2.containsKey(m4.toString()));

        //End of user code @End of user assertCollection1 implementation @warni
ng: do not remove this comment
```

```
        //Start of user code @Enter your implementation for the method testFindAll
Married here @warning: do not remove this comment

        //End of user code @End of user implementation for the method testFindAllM
arried @warning: do not remove this comment
    }
```

Figura 7.12. Código gerado para o método findAllMarried da classe ManTest

7.3.3. Método becomeOlder

A figura 7.13 apresenta o diagrama de sequência para o método becomeOlder. Por sua vez, a figura 7.14 mostra o código gerado a partir deste diagrama de sequência.

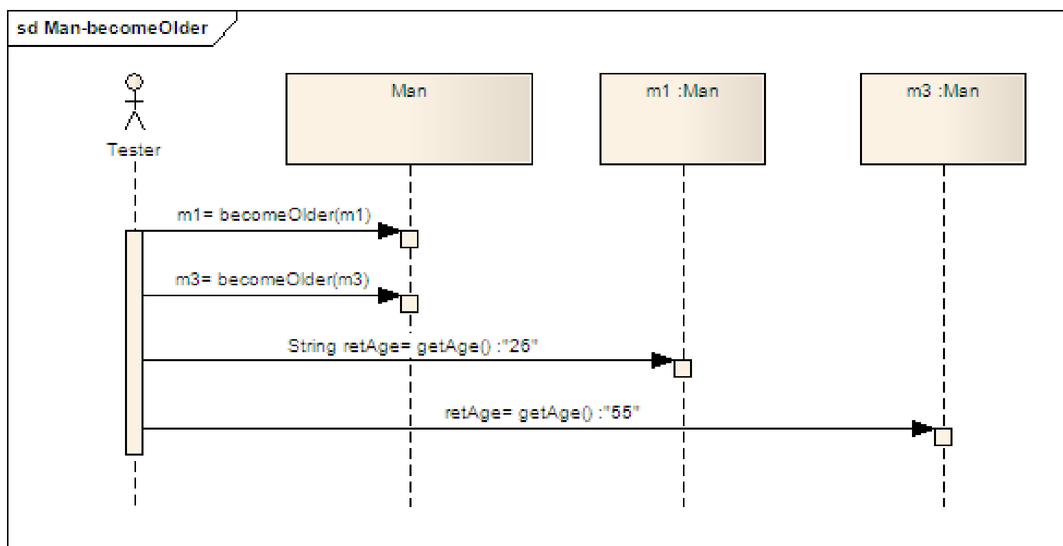

Figura 7.13. Diagrama de sequência para o método becomeOlder

```
    /**
     * The test for the method becomeOlder
     */
    @Test
    public void testBecomeOlder() {
        m1 = manBusiness.becomeOlder(m1);
        em.flush();
        m3 = manBusiness.becomeOlder(m3);
        em.flush();
        String retAge = m1.getAge();
        assertEquals("The expected value [26] is not equal to the object retAge.",
"26", retAge);
        retAge = m3.getAge();
        assertEquals("The expected value [55] is not equal to the object retAge.",
"55", retAge);

        //Start of user code @Enter your implementation for the method testBecomeO
lder here @warning: do not remove this comment

        //End of user code @End of user implementation for the method testBecomeOl
der @warning: do not remove this comment
    }
```

Figura 7.14. Código gerado para o método becomeOlder da classe ManTest

7.4. Código gerado para a camada *Business Object*

Nesta seção, apresentamos o código que é gerado para a camada *Business Object* de NovaStudio. Como foi dito no capítulo 6, o teste para esta camada é feito de modo completamente automático por Ejb3Unit, bastando apenas efetuar a criação do construtor deste teste com os parâmetros adequados.

A figura 7.15 apresentada o resultado da geração de código para a camada *Business Object* da classe Man.

```
/*
 * This code was generated by NovaStudio©. Copyright BULL reserved.
 * DO NOT EDIT this file!
 * File: ManBeanTest.java
 * Template: testEntityEJB3JUnit4.mt
 */
package com.bull.test.family.test;

import com.bm.datagen.Generator;
import com.bm.datagen.annotations.GeneratorType;
import com.bm.datagen.relation.BeanCollectionGenerator;
import com.bm.datagen.relation.SingleBeanGenerator;
import com.bm.testsuite.junit4.BaseEntityJunit4Fixture;
import com.bull.test.family.dto.persistence.ChildBean;
import com.bull.test.family.dto.persistence.ManBean;
import com.bull.test.family.dto.persistence.WomanBean;
import org.junit.Test;
import java.util.Collection;

/**
 * This is the test for the ManBean class.
 * EJB3Unit will perform automatic tests to check if the implementation of
 * this Entity Bean class is correct. The tests for classes that have
 * ManyToMany relations are not generated (not supported yet by EJB3Unit).
 * <p> Test framework: - EJB3Unit (JUnit4)  </p>
 */
@SuppressWarnings("unchecked")
public class ManBeanTest extends BaseEntityJunit4Fixture<ManBean> {
    /**
     * The EntityBean of the ManBean class and its associations
     * (if any) and the associations of its associations (recursive).
     * This is used by EJB3Unit to perform the EJB3 injection.
     */
    private static final Class[] usedBeans = {
                                               ChildBean.class, ManBean.class,
                                               WomanBean.class
                                             };

    /**
     * The beans generators for the associations (OneToOne, OneToMany
     * and ManyToOne).
     */
    private static final Generator[] specialGenerators = {
                                                         new ChildBeanCreator(),
                                                         new WomanBeanCreator()
                                                          };

    /**
     * The constructor for the ManBeanTest class
     */
    public ManBeanTest() {
        super(ManBean.class, specialGenerators, usedBeans);
    }
```

```
    /**
     * This is a default blank test generated by NovaStudio.
     */
    @Test
    public void testNothing() {
        //Blank test. Do not delete this method.
    }

    /**
     * The special generator for the relation with ChildBean contained
     * in the field childs. EJB3Unit will create 10 random
     * ChildBean objects to perform the test.
     */
    @GeneratorType(className = Collection.class, field = "childs")
    private static final class ChildBeanCreator extends BeanCollectionGenerator
<ChildBean> {
        private ChildBeanCreator() {
            super(ChildBean.class, 10);
        }
    }

    /**
     * The special generator for the relation with WomanBean contained
     * in the field wife. EJB3Unit will create one
     * random WomanBean object to perform the test.
     */
    @GeneratorType(className = WomanBean.class, field = "wife")
    private static final class WomanBeanCreator extends SingleBeanGenerator<Wom
anBean> {
        private WomanBeanCreator() {
            super(WomanBean.class);
        }
    }
}
```

Figura 7.15. Código gerado para a camada *Business Object* da classe Man

7.5. Considerações finais

Este capítulo apresentou um caso de teste, utilizado para a validação de NovaStudio, mostrando os diagramas de sequência e o código de testes gerado correspondente a cada diagrama.

Como já foi comentado no capítulo 6, todos os objetivos deste trabalho foram alcançados. A funcionalidade de geração de testes foi validada pela equipe NovaStudio após diversos casos de teste, semelhantes ao apresentado neste capítulo, terem sido executados por NovaStudio com sucesso. Portanto, a funcionalidade de geração de testes, a partir de uma modelagem UML, foi completamente integrada ao ambiente de desenvolvimento NovaStudio.

8. CONCLUSÃO

Este trabalho propôs um meta-modelo baseado em diagramas de sequência e objetos UML para a geração de testes unitários. A principal contribuição deste trabalho foi a implementação da funcionalidade de geração de teste unitários (com dependências), a partir da meta-modelagem UML proposta, para o ambiente de desenvolvimento NovaStudio, que constitui o contexto técnico deste trabalho. Esta funcionalidade de geração de testes a partir de uma modelagem UML foi validada pela equipe NovaStudio, após a execução de diversos casos de testes. Todos os objetivos deste trabalho foram alcançados.

A primeira parte deste trabalho (capítulos 2 e 3) apresentou um estudo teórico (Estado da Arte) sobre as abordagens de Engenharia Dirigida por Modelos (MDE) e sua principal variante a Arquitetura Dirigida por Modelos (MDA). Também foi apresentado um estudo sobre as técnicas de Teste de Software e das abordagens de Teste Baseado em Modelos (MBT) e Teste Dirigido por Modelos (MDT). Além disso, um estudo sobre as principais técnicas de modelagem de testes a partir de diagramas UML também foi apresentado, sendo que estas incluem principalmente os diagramas de sequência e os diagramas de estados.

A segunda parte deste trabalho (capítulo 4) apresentou o ambiente de desenvolvimento NovaStudio, que utilizando uma abordagem da MDA, gera boa parte do código de uma aplicação a partir de diagramas de classe UML. Além disso, neste capítulo foram lembrados os objetivos deste trabalho, sendo que o principal deles é a implementação da funcionalidade de geração de testes para a ferramenta NovaStudio, seguindo os princípios da MDT.

A terceira parte deste trabalho (capítulos 5, 6 e 7) apresentou uma proposta de meta-modelagem para geração de testes unitários, além da descrição de sua realização e implementação para o ambiente de desenvolvimento NovaStudio e os resultados que foram obtidos. A meta-modelagem proposta por este trabalho é baseada principalmente em diagramas de sequência. Ela pode ser considerada inovadora por propor os diagramas de sequência com asserção direta e indireta, além do diagrama de sequência de inicialização.

Para a realização e implementação da funcionalidade de geração de testes, foi necessário adaptar NovaStudio visando o suporte da meta-modelagem proposta. Com NovaStudio adaptado para o suporte de diagramas de sequência e objetos UML, foram escritos *templates* Acceleo para a geração de código de teste para as implementações Java EE e WEB de NovaStudio. Como *frameworks* de teste para essas implementações de NovaStudio, foram escolhidos Ejb3Unit e JUnit (versões 3 e 4). A implementação da funcionalidade de geração automatizada de testes, a partir de uma modelagem UML, foi

completamente integrada ao ambiente de desenvolvimento NovaStudio através da escrita de um *plugin* Eclipse.

Os resultados deste trabalho foram descritos através da apresentação de um caso de teste, contendo os diagramas de sequência (modelo) e o respectivo código gerado por NovaStudio. Este caso de teste foi um dos utilizados para a validação da funcionalidade de geração de testes pela equipe NovaStudio.

Portanto, este trabalho propôs um nova forma de meta-modelagem de testes unitários para uma aplicação. A implementação desta meta-modelagem, permitindo a geração automatizada de testes a partir de uma modelagem UML, foi efetuada com sucesso para o ambiente de desenvolvimento NovaStudio.

Como perspectivas de trabalhos futuros para o ambiente de desenvolvimento NovaStudio, pode-se pensar no desenvolvimento de uma nova funcionalidade de geração dos testes de integração de uma aplicação. Outra perspectiva interessante seria a melhoria da funcionalidade de geração de interfaces gráfica de uma aplicação, que é hoje efetuada por NovaStudio a partir de diagramas de atividade UML, ou uma nova proposta de meta-modelagem com este objetivo e a sua respectiva realização e implementação.

REFERÊNCIAS

ABRAN, A. et al. Guide to the Software Engineering Body of Knowledge (SWEBOK), **IEEE Press,** 2004.

ATKINSON, C; KUHNE, T. Model-driven development: a metamodeling foundation. **IEEE Software**, vol.20, no. 5, pp. 36-41, 2003.

BAO-LIN, L. et al. Test Case Automate Generation from UML Sequence Diagram and OCL Expression. **2007 International Conference on Computational Intelligence and Security (CIS 2007)**, pp. 1048-1052, 2007.

BARBOSA, E. F. et al. Introdução ao teste de software. **XIV Simpósio Brasileiro de Engenharia de Software**, pp. 330–378, 2000

BECK, K; GAMMA, E. Test Infected: Programmers Love Writing Tests. **Java Report**, Volume 3, Number 7, 1998.

BECK, K. Extreme Programming Explained: Embrace Change. **Addison-Wesley Professional**, 1999.

BÉZIVIN, J.; GERBÉ, O.; Towards a Precise Definition of the OMG / MDA (TM) Framework. **Automated Software Engineering (ASE'01),** 2001.

BÉZIVIN, J. In Search of a Basic Principle for Model-Driven Engineering. **Upgrade**, vol. 5, no. 2, pp. 21-24, 2004.

BÉZIVIN, J. et al. Rapport de synthèse de l'AS CNRS sur le MDA (Model Driven Architecture). **CNRS**, 2004. Disponível em: <http://www-list.cea.fr/labos/fr/LLSP/accord_uml/docs/fr/as_mda_rapport_synthese_fin2.pdf >. Acesso em: out. 2010.

BÉZIVIN, J. On the Unification Power of Models. **Software and Systems Modeling**, vol. 4, no. 2, pp.171-188, 2005.

BOOCH, G. et. al. An MDA manifesto. **MDA Journal**, 2004. Disponível em: < http://www.bptrends.com/publicationfiles/05%2D04%20COL%20IBM%20Manifesto%20%2D%20Frankel%20%2D3%2Epdf > . Acesso em: out. 2010.

BRANDÃO, H. A. et al. **xUnit – Testes Unitários Automatizados**. 2005. Disponível em: < http://paginas.fe.up.pt/~aaguiar/es/artigos%20finais/es_final_6.pdf >. Acesso em: nov. 2010.

BULL. **NovaStudio User Guide**, v. 3.1.10, 2007.

CRICHTON, C; CAVARRA, A.; DAVIES, J. Using UML for Automatic Test Generation. **16th IEEE International Conference on Automated Software Engineering (ASE 2001), IEEE Computer Society**, 2001.

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

DIJKSTRA, E. W. The humble programmer. **Commun. ACM**, vol. 15, no. 10, pp. 859-866, 1972.

DOBING, J.; PARSONS, J. How UML is used. **Commun. ACM**, vol. 49, no. 5, pp. 109-113, 2006.

DUBY, C. K. Accelerating Embedded Software Development with a Model Driven Architecture. **Pathfinder Solutions**, 2003. Disponível em: < www.omg.org/mda/mda_files/MDA_overview.pdf >. Acesso em: out. 2010.

EL-FAR, I. K.;WHITTAKER, A. Model-based software testing, **Encyclopedia on Software Engineering**, 2001.

FARCHI, E.; HARTMAN, A.; PINTER, S. S. Using a model based test generator to test for standard conformance. **IBM Systems Journal**, Vol. 41, No. 1, pp. 89–110, 2002.

FAVRE, J. M. Towards a Basic Theory to Model Model Driven Engineering. **Workshop on Software Model Engineering (WISME)**, 2004.

FRAIKIN, F.; LEONHARDT, T. SeDiTeC — Testing Based on Sequence Diagrams. **17th IEEE International Conference on Automated Software Engineering (ASE'02)**, 2002

FRANCE, R.; RUMPE, B. Model-driven Development of Complex Software: A Research Roadmap. **ICSE Future of Software Engineering**, pp. 37-54, 2007

GAMMA, E. et al. Design Patterns: Elements of Reusable Object-Oriented Software. **Addison-Wesley Professional**, 1st ed., 1994.

GREENFIELD, J. et al. Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. **Wiley Publishing**, 2004.

HAILPERN, B.; TARR, P. Model-driven development: The good, the bad, and the ugly. **IBM Systems Journal**, vol. 45, no. 3, pp. 451-461, 2006.

HECKEL, R.; LOHAMANN, M. Towards Model-Driven Testing. **International Workshop on Test and Analysis of Component-Based Systems, Electronic Notes in Theoretical Computer Science**, p.33-43, 2003

JAVED, A. Z.; STROOPER, P. A.; WATSON, G. N. Automated Generation of Test Cases Using Model-Driven Architecture. **Second International Workshop on Automation of Software Test (AST '07)**, 2007.

KANSOMKEAT, S. et al. A Comparative Evaluation of Tests Generated from Different UML Diagrams, **2008 Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing**, pp.867-872, 2008.

KENT, S. Model Driven Engineering. **Third International Conference on Integrated Formal Methods (IFM)**, pp. 286-298, 2002.

KOSTELA, L. Test Driven: Pratical TDD and Acceptance TDD for Java Developers. **Manning Publications**, 2008.

LIMA, H. et al. Automatic Generation of Platform Independent Built-in Contract Testers. **Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software (SBCARS2007)**, pp. 47-60, 2007.

LI, N. et al. A Framework of Model-Driven Web Application Testing", **30th Annual International Computer Software and Applications Conference**, COMPSAC '06, p. 157-162, 2006.

MEDEIROS, E. R. **NovaStudio : gerador de código usando a arquitetura dirigida pelos modelos (MDA)**. 2009. 70 f. Trabalho de Diplomação (Bacharelado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

MUSSET, J.; JULLIOT, E.; LACRAMPE, S. Acceleo User Guide – Acceleo Version 2.6. 2009. Disponível em: < http://www.acceleo.org/doc/obeo/en/acceleo-2.6-user-guide.pdf >. Acesso em: out. 2010.

MYERS, G. J. et al. The Art of Software Testing. Second Edition, **John Wiley & Sons**, 2004.

NAM, H. D.H.; MOUSSET, E. C.; LECY, D. C. Automating the Testing of Object Behaviour: A Statechart-Driven Approach. Transactions on Engineering, **Computing and Technology**, 2006.

NAUR, P.; RANDELL, B. Software Engineering: Report of a conference sponsored by the NATO Science Committee, **NATO**, 1969.

OBJECT MANAGEMENT GROUP – OMG. **Model Driven Architecture (MDA)**. Versão 1.0.1, 2003. Disponível em: < http://www.omg.org/cgi-bin/doc?omg/03-06-01 >. Acesso em: out. 2010.

OBJECT MANAGEMENT GROUP – OMG. **Meta-Object Facility (MOF).** Versão 2.0, 2006. Disponível em: < http://www.omg.org/cgi-bin/doc?formal/06-01-01.pdf >. Acesso em: out. 2010.

OBJECT MANAGEMENT GROUP – OMG. **Unified Modeling Language (UML)**. Versão 2.2, 2007. Disponível em: < http://www.omg.org/cgi-bin/doc?formal/07-11-04.pdf >. Acesso em: out. 2010.

OBJECT MANAGEMENT GROUP – OMG. **MDA**. Disponível em: < http://www.omg.org/mda/>. Acesso em: out. 2010.

PACKEVICIUS, Š.; USANIOV, A; BAREISA, E. The Use of Model Constraints as Imprecise Software Test Oracles. Information Technology And Control, Kaunas, **Technologija**,Vol. 36, No. 2, pp. 246–252, 2007.

ROCHA, C.; MARTINS, E. A Method for Model Based Test Harness Generation for Component Testing. **Journal of the Brazilian Computer Society**, v. 14, pp. 7-23, 2008.

SCHIMIDT, D. C. Model-Driven Engineering. **IEEE Computer**, vol. 39, no. 2, pp. 25-31, 2006.

SEIDEWITZ, E. What models mean. **IEEE Software**, Volume 20, Issue 5, pp. 26-32, 2003.

SELIC, B. Model-Driven Development: Its Essence and Opportunities. **Ninth IEEE InternationalSymposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06)**, pp.313-319, 2006.

SOKENOU, D. Generating Test Sequences from UML Sequence Diagrams and State Diagrams. **GI Jahrestagung**, pp. 236-240, 2006.

SUN. **Core J2EE Patterns: Patterns Index Page**. Disponível em : <http://java.sun.com/blueprints/corej2eepatterns/Patterns/index.html> Acesso em: out. 2010.

THOMAS, D.; BARRY, B. M. Model Driven Development: The Case for Domain Oriented Programming. Companion of the 18th OOPSLA, **ACM Press**, pp. 2-7, 2003.

UTTING, M.; LEGEARD, B. Practical Model-Based Testing: A Tools Approach. **Morgan-Kaufmann**, 2007.

WEIßLEDER, S.; SCHILINGLOFF, B. Deriving Input Partitions from UML Models for Automatic Test Generation. **MoDeVVa2007**, 2007.

WHITTAKER, J. A.; VOAS, J. M. 50 years of software: key principles for quality. **IT Professional**, Vol 28, pp. 28-35, 2002.