JOSÉ EDUARDO BELIX

UM ESTUDO SOBRE MDA: SUPORTE FORNECIDO PELA UML E REUSO DE SOLUÇÕES PRÉ-DEFINIDAS

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para a obtenção do título de Mestre em Engenharia Elétrica

São Paulo

2006

JOSÉ EDUARDO BELIX

UM ESTUDO SOBRE MDA: SUPORTE FORNECIDO PELA UML E REUSO DE SOLUÇÕES PRÉ-DEFINIDAS

Dissertação apresentada à Escola Politécnica da Universidade de São Paulo para a obtenção do título de Mestre em Engenharia Elétrica

Área de Concentração:
Sistemas Digitais

Orientador:
Prof. Dr. Edison Spina

São Paulo

2006

FICHA CATALOGRÁFICA

Belix, José Eduardo
Um estudo sobre MDA: suporte fornecido pela UML e reuso de soluções pré-definidas / J.E. Belix. -- São Paulo, 2006.
127 p.

Dissertação (Mestrado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais.

1.Geração de código 2.Softwares (Reuso) 3.UML I.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Computação e Sistemas Digitais II.t.

Errata

Folha	Linha	Onde se lê	Leia-se
4	27	Uma comprovação da solução é oferecida no final desta dissertação, exemplificada com partes da implementação de um software de vendas varejo.	Uma comprovação da solução é oferecida no apêndice A, com exemplos extraídos de um software JAVA escrito para atender a especificação descrita no apêndice C, usando as soluções pré-definidas descritas no capítulo 6.

À Márcia, pelo carinho, incentivo e também por sua compreensão pelas incontáveis horas em que estive ausente, dedicado a esta pesquisa.

Agradecimentos

Ao Profº Dr. Edison Spina pela orientação e apoio para a realização deste trabalho.

Ao doutorando Sérgio Martins Fernandes, à Profª Dra. Selma Shin Shimizu Melnikoff e ao Profº Dr. Reginaldo Arakaki, pelas valiosas sugestões para o desenvolvimento desta pesquisa.

A todos os professores e colegas do Departamento de Engenharia de Computação e Sistemas Digitais, que de alguma forma tenham contribuído para a realização deste trabalho.

Resumo

Este trabalho tem por objetivo propor a utilização de soluções pré-definidas e comprovadas na abordagem MDA, auxiliando o desenvolvedor na resolução de problemas recorrentes ao desenvolvimento de software. A utilização destas soluções pré-definidas leva a um aumento de produtividade no contexto MDA e na geração de software baseado em boas práticas. Para que este objetivo seja cumprido, é empreendida uma análise sobre o MDA e sobre como operacionalizar as transformações entre modelos. Também é empreendida uma análise sobre o suporte fornecido pela UML e sobre reutilização em desenvolvimento orientado a modelos. Por fim este trabalho apresenta partes de uma aplicação protótipo, construída para ser uma prova de conceito de código gerado através da combinação de UML e soluções pré-definidas.

Abstract

The goal of this work is to propose the use of pre-defined solutions on MDA approach, supporting the developer in solving recurrent problems of software development. The use of these pre-defined solutions leads to an increase of productivity in MDA context, and in the generation of software based on best practices. To reach this goal, an analysis of MDA is undertaken, as well as an analysis of how to enable the transformations between models. It is also undertaken an analysis about the use of UML and the reuse in model driven development. Finally this work presents portions of a prototype application, constructed to be a proof-of-concept of generated code, combining UML and the pre-defined solutions.

Sumário

LISTA DE FIGURAS I

LISTA DE TABELAS II

LISTA DE ABREVIATURAS III

1 INTRODUÇÃO 1

1.1 Objetivos 2
1.2 Justificativa 3
1.3 Metodologia 3
1.4 Estrutura do trabalho 5

2 O PAPEL DA UML NO MDA 7

2.1 Uma breve descrição da UML 7
2.1.1 Os avanços da UML para contemplar o MDA 9
2.2 Definição de uma linguagem 11
2.2.1 Graus de formalismo 12
2.3 A sintaxe da UML 13
2.3.1 Metamodelagem 14
2.4 A semântica UML 16
2.4.1 Meios para a definição de semântica na UML 17
2.5 O problema de consistência dos modelos UML 18
2.5.1 Um exemplo de consistência com UML 20
2.6 Considerações finais do capítulo 2 24

3 O DESENVOLVIMENTO ORIENTADO POR MODELOS (MDD) 25

3.1 O cenário atual do MDD 25
3.2 A visão do OMG 28
3.3 A proposta MDA 29
3.3.1 Os benefícios do MDA 30
3.4 Os modelos-chave do MDA 32
3.4.1 Modelo Independente de Computação (CIM) 32
3.4.2 Modelo Independente de Plataforma (PIM) 33
3.4.2.1 Definição de plataforma 34
3.4.3 Modelo para Plataforma Específica (PSM) 34
3.5 Mapeamentos e transformações 34
3.5.1 Mapeamento de tipos do modelo 35
3.5.2 Mapeamento de instâncias do modelo 35
3.5.3 Transformações 36
3.5.4 Condições para as transformações 38
3.5.5 Tipos de transformação 38
3.5.6 Transformações inter-linguagem e intra-linguagem 39
3.5.6.1 Extensão de linguagens para transformações inter-linguagem 40
3.6 Engenharia direta e reversa 43

3.7 Considerações finais do capítulo 3 44

4 IMPLEMENTAÇÃO DAS TRANSFORMAÇÕES 45

4.1 Acesso aos modelos para implementar as transformações 45
4.2 Manipulação direta de modelos 46
4.3 Representação intermediária 50
4.3.1 Atuação baseada em XMI 51
4.3.2 Linguagens de *query* para XML 52
4.3.2.1 O XPATH 54
4.3.2.2 Geração de arquivos texto com XSLT 55
4.3.2.3 Peculiaridades do XSLT 57
4.4 Suporte de uma linguagem de transformação 59
4.4.1 Linguagens imperativas e declarativas 60
4.4.2 Query, Views, Transformations (QVT) 61
4.5 Considerações finais do capítulo 4 63

5 UTILIZAÇÃO DE SOLUÇÕES PRÉ-DEFINIDAS NO DESENVOLVIMENTO MDA 64

5.1 Soluções para a redução de esforços de modelagem 64
5.2 Classificação para soluções pré-definidas em MDA/MDD 64
5.3 O impacto das soluções apresentadas sobre o esforço de modelagem 66
5.3.1 Redução de esforços causada pelas soluções de caixa branca 66
5.3.2 Redução de esforços causada pelas soluções de caixa preta 67
5.3.3 Redução de esforços causada pelas soluções de caixa cinza 69
5.4 Utilização das soluções - um exemplo prático 73
5.4.1 Utilização das soluções de caixa branca 74
5.4.2 Aplicação das soluções de caixa preta e caixa cinza 76
5.5 Considerações sobre modelagem e o valor das restrições 76
5.5.1 Restrições são úteis 77
5.6 Considerações finais do capítulo 5 77

6 A APLICAÇÃO DAS SOLUÇÕES PRÉ-DEFINIDAS NO DESENVOLVIMENTO MDA 79

6.1 Abordagens *top-down* e *bottom-up* 79
6.2 Funcionalidades de uma ferramenta CASE 80
6.3 Considerações sobre a geração automática de código a partir de modelos 81
6.3.1 Regras para inter-relacionamento de modelos UML 83
6.3.2 Elaboração das soluções pré-definidas 84
6.3.3 Definição das transformações 85
6.4 Um exemplo de geração de código 88
6.4.1 Descrição de comportamento 96
6.5 Considerações finais do capítulo 6 98

7 CONSIDERAÇÕES FINAIS 99

7.1 Trabalhos futuros 99

APÊNDICE A - EXEMPLOS DE CÓDIGO-FONTE DE SOLUÇÕES DE CAIXA CINZA 101

A.1 Métodos de persistência (caixa cinza) 101

A.2 Fornecimento de conexões com o banco de dados (caixa preta) 104
APÊNDICE B - REPRESENTAÇÃO EM XMI DE UM MODELO UML..... 107
APÊNDICE C - ESPECIFICAÇÃO DO SOFTWARE EXEMPLO 112
C.1 Visão do sistema..... 112
C.2 Atores do sistema 113
C.3 Diagrama de casos de uso..... 114
C.4 Especificação de casos de uso 114
LISTA DE REFERÊNCIAS 119

Lista de Figuras

Figura 1 - Classificação das linguagens segundo grau de formalismo e especificidade de domínio (BARESI; PEZZÉ, 2005)..... 13
Figura 2 - As quatro camadas da arquitetura OMG (caso UML) (KLEPPE;WARMER;BAST, 2003)..... 15
Figura 3 - Um modelo UML complementado com sentenças OCL 18
Figura 4 - Diagrama de classes de um sistema de vendas, complementado por sentenças OCL 21
Figura 5 - Diagrama de seqüência de um sistema de vendas, exibindo o comportamento de cancelamento de pedidos..... 22
Figura 6 - Diagrama de seqüência de um sistema de vendas, exibindo o comportamento de abertura de pedidos..... 23
Figura 7 - Diagrama de estados de um sistema de vendas, exibindo o comportamento de inclusão de itens 23
Figura 8 - Múltiplos PSMs gerados a partir de um PIM..... 31
Figura 9 - Fragmento de um PIM marcado (OMG, 2001a)..... 37
Figura 10 - Um PSM CORBA derivado do PIM da figura 9 (OMG, 2001a)..... 37
Figura 11 - Transformação de modelos entre linguagens diferentes (CAPLAT; SOURROUILLE, 2005)..... 42
Figura 12 - Transformação bidirecional (KLEPPE;WARMER;BAST, 2003)..... 43
Figura 13 - Editor de *scripts* do Rational Rose..... 47
Figura 14 - Exemplo de utilização de eixo no XPath..... 55
Figura 15- Implementação de método de consulta para a classe Cliente..... 71
Figura 16 - Implementação de método de consulta para a classe *Dealer* 71
Figura 17 - Método de consulta em uma associação tipo agregação 73
Figura 18 - Diagrama de seqüência para uma consulta no sistema de vendas varejo proposto (BELIX *et al*, 2005) 74
Figura 19 - Diagrama de seqüência da consulta, mas com a aplicação das soluções de caixa branca (BELIX *et al*, 2005) 75
Figura 20 - Organização lógica do software exemplo 89
Figura 21 -Diagrama de classes antes da transformação PIM-PIM..... 90
Figura 22 - Diagrama de classes refinado pela transformação PIM-PIM..... 91
Figura 23 - Diagrama de classes PSM 92
Figura 24 - Comportamento da consulta por ID ao catálogo..... 97
Figura 25 - Classe Cliente com três atributos e um método 107
Figura 26 -Diagrama de Casos de Uso da aplicação exemplo..... 114

Lista de Tabelas

Tabela 1 - Script Rose para manipulação de atributos de *public* para *private* 48
Tabela 2 - Fragmento de código XSLT 52
Tabela 3 - Programa XSLT para geração de código-fonte em JAVA 55
Tabela 4 - Cálculo de fatorial em Prolog 60
Tabela 5 - Cálculo de fatorial em Visual Basic 6 60
Tabela 6 -Código-fonte da classe Cliente 92
Tabela 7- Código-fonte da consulta por ID da classe controladora de Caso de Uso da consulta ao catálogo 97
Tabela 8 – Código-fonte base do método getInfoPorID 101
Tabela 9 – Código-fonte base do método getInfo 102
Tabela 10 – Código-fonte base do método getInfoPorID para classes Todo 103
Tabela 11 - Classe para fornecimento de conexões DB 104
Tabela 12 - Classe com parâmetros para a conexão DB 105
Tabela 13 - Representação XMI gerada a partir do modelo da figura 25 107

Lista de Abreviaturas

ADO *ActiveX Data Objects*

API *Application Programming Interface*

BNF *Backus-Naur Form*

CASE *Computer Aided Software Engineering*

CRUD *Create, Read, Update, Destroy*

CWM *Common Warehouse Meta-model*

DAO *Data Access Object*

DDL *Data Definition Language*

DOM *Document Object Model*

DSL *Domain Specific Language*

FBD *Function Block Diagram*

HTML *HyperText Markup Language*

IDE *Interface Development Environment*

J2EE *Java 2 Entreprise Edition*

JAXP *Java API for XML Processing*

LEMMA *Language for Easy Medical Models Analysis*

MDA *Model-Driven Architecture*

MDD *Model-Driven Development*

MDSD *Model-Driven Software Development*

MER *Modelo Entidade-Relacionamento*

MOF *Meta-Object Facility*

OMG *Object Management Group*

OO *Object-Oriented*

OOPSLA *Object-Oriented Programming, Systems, Languages and Applications*

PIM *Platform Independent Model*

PSM *Platform Specific Model*

QVT *Query / Views / Transformations*

SAX *Simple API for Java*

SDL *Specification and Description Language*

SQL *Structured Query Language*

TI *Tecnologia de Informação*

VB *Visual Basic*

W3C *World Wide Web Consortium*

XML *eXtensible Markup Language*

XMI *XML Metadata Interchange*

XSL *eXtensible Stylesheet Language*

XSLT *XSL Transformation*

XVCL *XML-Based Variant Configuration Language*

1 INTRODUÇÃO

Um dos atuais desafios no desenvolvimento de software é o que se conhece como desenvolvimento orientado a modelos (MDD ou *model-driven development*), uma tentativa de elevar o nível de abstração no desenvolvimento de software para além das atuais linguagens de programação.

Desde que o homem começou a usar computadores, pesquisadores têm trabalhado no aumento do nível de abstração com o qual os engenheiros de software escrevem programas. O primeiro compilador Fortran foi um grande marco na ciência da computação por ter permitido pela primeira vez que os programadores especificassem o que a máquina deveria fazer, ao invés de especificar como deveria fazer (ATKINSON; KÜHNE, 2003).

MDD é um termo genérico para representar a idéia de um desenvolvimento orientado por modelos, que não exige a definição de todos os detalhes de implementação necessários pelas linguagens de programação.

Entretanto, não existe uma abordagem única para MDD. Há por exemplo o *Model-Driven Architecture* (MDA) do *Object Management Group* (OMG, 2003b), *Software Factories* da Microsoft (GREENFIELD *et al*, 2004) ou *Model-Driven Software Development* (MDSD) (BETTIN, 2004a) (BETTIN, 2004b).

(BETTIN, 2004b) discorre sobre as semelhanças e diferenças entre MDSD e MDA, notadamente sobre o uso da UML. Para Bettin, o MDA é baseado no *Meta Object Facility* (MOF) e na *Unified Modeling Language* (UML), ambos padrões do OMG. O MDSD é baseado nos princípios de modelagem e meta modelagem, mas não está vinculado a nenhum meta-modelo específico.

A questão da modelagem e da metamodelagem é um ponto central nas discussões sobre MDD. Um modelo precisa ser representado segundo uma linguagem bem definida, ou seja, que tenha forma (sintaxe) e significado (semântica) bem definidos, para que seja adequado para a interpretação automática por computador (KLEPPE;WARMER;BAST, 2003). Com relação a este último parágrafo, duas observações são pertinentes:

- É importante a interpretação automática por computador para que ferramentas CASE possam ser empregadas para suportar a automação necessária ao MDA;
- A UML – na qual o MDA é baseado – é criticada por autores como não sendo adequada para o MDD (MELLOR, 2002). (GREENFIELD *et al*, 2004) também considera a UML inadequada para o desenvolvimento orientado a modelos - especialmente no que se refere aos problemas semânticos – embora também reconheça os méritos da linguagem: “UML é a linguagem de modelagem orientada a objetos mais amplamente utilizada. Ela é descrita por mais de uma centena de livros e suportada por múltiplas ferramentas comercias”.

O OMG, por sua vez, trabalha na contínua evolução da UML. Recentemente foi lançada a UML 2.0, que segundo o OMG “... é esperado que esta grande revisão proposta da UML venha a desempenhar um importante papel no futuro do MDA” (OMG, 2003d). Entretanto vale a pena notar que descrição atual da UML 2.0 “... não é uma especificação completamente formal de uma linguagem, porque ao fazer isto significativa complexidade seria adicionada sem um claro benefício” (OMG, 2003d).

1.1 Objetivos

Reconhecendo a UML como a linguagem de modelagem dominante, a abordagem MDD analisada neste trabalho é a OMG-MDA.

Reconhecendo também que diferentes aplicações têm pontos em comum, esta dissertação defende que a abordagem MDA pode ser utilizada para fornecer soluções pré-definidas que resolvam estes pontos, desonerando o desenvolvedor da necessidade de modelar toda a estrutura e comportamento do software. Ou seja, a reutilização de soluções comprovadas não só resolve problemas recorrentes, mas também melhora a produtividade no desenvolvimento MDA através de um menor esforço de modelagem por parte dos desenvolvedores.

Este trabalho tem por objetivos:

- Defender, no contexto MDA, a utilização de soluções pré-definidas como forma de resolver problemas recorrentes no desenvolvimento de software;
- Mostrar que a utilização de soluções pré-definidas reduz a quantidade de esforços na modelagem de software;
- Apresentar exemplos da aplicação destas soluções, analisando seu resultado;
- Analisar o uso da UML no desenvolvimento MDA;
- Apresentar as maneiras pelas quais é possível atualmente executar as transformações necessárias ao MDA;

1.2 Justificativa

Respaldado pelo fato de muitas aplicações terem pontos em comum, este trabalho recomenda que as ferramentas CASE proporcionem transformações que forneçam soluções recorrentes e comprovadas para estes pontos. Tais soluções devem ser aplicadas da forma mais transparente possível.

O objetivo é desonerar o desenvolvedor da preocupação com problemas que podem ser resolvidos de forma automática, ou pelo menos parcialmente automatizada, resultando em um aumento da produtividade do desenvolvedor e na geração de um software de melhor qualidade.

O MDA é uma iniciativa do OMG, mas cabe à comunidade técnica propor soluções que o tornem atrativo para a comunidade de desenvolvedores. "... não podemos olhar para o OMG e esperar que ele forneça uma metodologia completa para a utilização das técnicas orientadas por modelos porque este não é o papel típico de um órgão padronizador" (BETTIN, 2004a).

1.3 Metodologia

Segundo (MARCONI; LAKATOS, 2004), o trabalho pretende seguir as etapas da concepção atual do método científico moderno, segundo o qual uma investigação alcança seus objetivos de forma científica quando cumpre ou se propõem a cumprir as seguintes etapas:

- Descobrimento do problema;
- Colocação precisa do problema;
- Procura de conhecimentos ou instrumentos relevantes ao problema;
- Tentativa de solução do problema com o auxílio dos meios identificados;
- Invenção de novas idéias;
- Produção de novos dados empíricos;
- Obtenção de uma solução;
- Investigação das conseqüências da solução obtida;
- Prova (comprovação) da solução;
- Correção das hipóteses, teorias, procedimentos ou dados empregados na obtenção da solução incorreta.

Em conformidade com estes passos, o descobrimento do problema ocorreu com a tentativa de desenvolvimento de uma ferramenta de transformações - baseada em XSLT - para testar a aplicabilidade da abordagem MDA. Tal tentativa fez surgir um questionamento sobre a reutilização de soluções pré-definidas durante a modelagem do software, como forma de auxiliar o desenvolvedor. O problema a ser solucionado então é o reuso de soluções pré-definidas em desenvolvimento orientado a modelos.

Durante a procura de conhecimentos ou instrumentos relevantes ao problema em questão, foi necessário um estudo sobre a questão da reutilização baseada em modelos. Também foi necessário um estudo detalhado sobre o MDA, sobre o MOF e principalmente sobre as vantagens e limitações da UML como linguagem de modelagem.

A tentativa de solução do problema com os meios identificados levou a uma solução que prevê dois pontos: o primeiro e principal é a utilização de soluções pré-definidas, classificadas segundo a quantidade de implementação oferecida. O segundo ponto é uma prescrição de uso da UML para evitar inconsistências no modelo do software.

Uma comprovação da solução é oferecida no final desta dissertação, exemplificada com partes da implementação de um software de vendas varejo.

Seguindo esta ordem, o trabalho demandou pesquisa bibliográfica e de laboratório.

“Uma pesquisa bibliográfica pode visar um levantamento dos trabalhos realizados anteriormente sobre o mesmo tema estudado no momento. Pode identificar e selecionar os métodos e técnicas a serem utilizados, além de fornecer subsídios para a redação da introdução e revisão da literatura do projeto ou trabalho. Em suma, uma pesquisa bibliográfica leva ao aprendizado sobre uma determinada área” (CRUZ; RIBEIRO, 2003).

A pesquisa bibliográfica abrange as seguintes etapas:

- Escolha e delimitação do assunto (Tema);
- Determinação da disponibilidade do material para consulta;
- Levantamento e anotações das citações relevantes;
- Aprofundamento e expansão da pesquisa;
- Seleção das fontes a serem utilizadas;
- Leituras e sumarização;
- Redação.

“Uma pesquisa de laboratório, diferentemente de uma pesquisa bibliográfica ou de campo, permite ao pesquisador manipular suas variáveis, isolá-las ou até mesmo provocar eventos passíveis de controle” (CRUZ; RIBEIRO, 2003).

A pesquisa de laboratório será necessária para testar as possibilidades disponíveis para a realização das transformações MDA.

1.4 Estrutura do trabalho

O capítulo 2 da dissertação aborda o uso da UML no desenvolvimento MDA. Ele discorre sobre sintaxe e semântica da UML e apresenta as vantagens e desvantagens de seu uso em um processo automático de geração de software a partir de modelos.

O capítulo 3 inicia-se com uma breve descrição do cenário atual do desenvolvimento orientado a modelos. O objetivo do capítulo é a apresentação do OMG-MDA, os

conceitos desta abordagem, seus modelos principais e as vantagens que podem ser obtidas com seu uso.

No capítulo 4 são abordadas as maneiras atuais pelas quais as transformações MDA podem ser implementadas. São apresentadas três estratégias distintas de implementação das transformações (Manipulação Direta de Modelos, Representação Intermediária e Linguagens de Transformação) e uma avaliação empírica e teórica é feita sobre cada uma delas.

O capítulo 5 é a principal contribuição deste trabalho. Nele é apresentada uma taxonomia de soluções que auxilia o desenvolvedor na resolução de problemas recorrentes, liberando-o da preocupação com detalhes que podem ser resolvidos automaticamente. Isto possibilita a redução de esforços na modelagem de software via MDA.

O capítulo 6 aborda como aplicar conjuntamente as soluções propostas no capítulo 5 com os recursos da UML, permitindo liberdade ao arquiteto de software para estender as soluções propostas durante a modelagem de um software.

Finalmente o capítulo 7 apresenta as principais conclusões, ressaltando as contribuições deste trabalho e sugerindo futuras pesquisas que podem complementar a abordagem apresentada aqui.

2 O PAPEL DA UML NO MDA

Como esta dissertação trata questões relativas ao MDA – que utiliza UML - este capítulo tem por objetivo discutir a importância da UML nesta abordagem, ou seja, qual é o papel da UML e quais seus pontos positivos e negativos para apoiar o MDA. O capítulo foca o formalismo da UML, que é um ponto fraco da linguagem, mas cuja discussão é imprescindível para que as ferramentas CASE MDA possam ser feitas de forma a gerar código-fonte tanto mais completo e correto quanto possível.

Este trabalho não pretende esgotar a questão das deficiências da UML. Não há neste trabalho uma análise detalhada das omissões ou inconsistências da linguagem por duas razões: a primeira é que o foco deste trabalho não é uma análise completa das deficiências da linguagem; a segunda é o grande tamanho da especificação UML 2.0, o que tornaria tal análise um empreendimento enorme e demorado.

2.1 Uma breve descrição da UML

A UML firmou-se como a linguagem de modelagem dominante na comunidade de desenvolvimento de software. Sua popularidade também deve ser encarada como um fator adicional de sucesso para o MDA. A linguagem está em contínua evolução, sendo que a versão 2.0 propõe 13 diagramas diferentes para representar os aspectos estruturais, comportamentais e físicos de um software. Segundo (BOOCH; RUMBAUGH; JACOBSON, 2005), são eles:

- Diagrama de Classes;
- Diagrama de Objetos;
- Diagrama de Componentes;
- Diagrama de Estrutura Composta;
- Diagrama de Casos de Uso;
- Diagrama de Seqüência;
- Diagrama de Comunicação;

- Diagrama de Estados;
- Diagrama de Atividades;
- Diagrama de Implantação (*Deployment*);
- Diagrama de Pacotes;
- Diagrama de Tempo;
- Diagrama da Visão Geral da Interação.

A abordagem multi-visão da UML fornece aos desenvolvedores os benefícios da separação de áreas de interesse durante a modelagem de um software. Os diferentes diagramas servem a diferentes propósitos durante o processo de desenvolvimento

O Diagrama de Classes é o diagrama mais comumente encontrado em modelagem de sistemas OO. Ele é composto por classes, interfaces e seus relacionamentos. As classes contêm métodos e atributos. O Diagrama de Objetos fornece uma fotografia das instâncias das classes em um determinado momento (BOOCH; RUMBAUGH; JACOBSON, 2005).

Diagramas de Pacotes podem ser utilizados para definir pacotes que representam conjuntos de classes ou outros pacotes agrupados. Pretendem demonstrar a decomposição do próprio modelo em unidades de organização e suas dependências (BOOCH; RUMBAUGH; JACOBSON, 2005), (SCHATTKOWSKI, 2005).

A especificação comportamental na UML, em um nível mais alto, normalmente começa com a identificação e descrição de Casos de Uso para sistemas que possam ser descritos pela sua interação com atores. O comportamento detalhado costuma recair sobre Diagramas de Estado e Diagramas de Atividades, além dos Diagramas de Interação (Seqüência ou Comunicação), que explicitamente apresentam a interação entre objetos e atores através das mensagens trocadas entre eles (BOOCH; RUMBAUGH; JACOBSON, 2005), (SCHATTKOWSKI, 2005).

Os Diagramas de Componente e Implantação, por sua vez, podem ser utilizados respectivamente para descrever como classes e interfaces são encapsuladas e como

estes componentes são distribuídos em nós de processamento (BOOCH; RUMBAUGH; JACOBSON, 2005), (SCHATTKOWSKI, 2005).

Tanto no MDA quanto em qualquer abordagem MDD, opta-se pelo uso de modelos para a definição dos sistemas de software. No caso MDA, apesar da abordagem permitir outras linguagens, a UML é preferencialmente empregada para representar os modelos de um software.

Para que modelos sejam automaticamente processados por uma ferramenta CASE e gerem implementação em quantidades significativas, é necessário o uso uma linguagem de modelagem tão rigorosamente definida quanto uma linguagem de programação (GREENFIELD *et al*, 2004).

2.1.1 Os avanços da UML para contemplar o MDA

Conforme mencionado na introdução desta dissertação, recentemente foi lançada a UML 2.0 que, segundo o OMG "... é esperado que esta grande revisão proposta da UML venha a desempenhar um importante papel no futuro do MDA" (OMG, 2003d).

A versão 2.0 da linguagem suporta os mais proeminentes conceitos da abordagem MDA, que são:

- Família de Linguagens

  A UML pode ser configurada para uma grande variedade de domínios (financeiro, telecomunicações, etc...), plataformas e métodos (RUP, métodos ágeis, etc). Para tal, o mecanismo de definição de *profiles* foi refinado, tornando-se mais robusto, flexível e mais fácil de ser implementado (OMG, 2003d).

  Profile é uma forma para se fazer uma extensão leve (sem alterar o metamodelo) da UML. Ele é composto de três mecanismos de que permitem estender a linguagem de uma forma controlada, a saber: esteriótipos (*stereotypes*), valores marcados (*tagged values*) e restrições (*constraints*). Esteriótipos estendem o vocabulário da UML, permitindo a criação de novos tipos de blocos de construção, derivados a partir de outros existentes, porém

específicos para um problema determinado. Valores marcados estendem as propriedades de um esteriótipo UML, permitindo a criação de novas informações na especificação do elemento. As restrições estendem a semântica de um bloco de construção, adicionando novas regras ou modificando as existentes (BOOCH; RUMBAUGH; JACOBSON, 2005), (OMG, 2004b).

Um *profile* deve especializar o metamodelo de referência, mas não o contradizer, isto é, deve fornecer regras bem formadas que sejam mais restritivas, porém compatíveis com o metamodelo. Além disso, o *profile* deve ser intercambiável entre ferramentas através do XMI (OMG, 2004b).

- Especificação de um sistema independentemente da plataforma que o suporta

  Assim como suas versões anteriores, a UML 2.0 suporta métodos de software que fazem distinção entre modelos de análise e lógicos e entre modelos físicos e de projeto. Já há uma consistência com a abordagem MDA pois modelos de análise ou lógicos são tipicamente independentes de plataforma (OMG, 2003d).

- Especificação de plataforma

  A versão 2.0 fornece duas melhorias com relação às versões 1.x. A primeira é já mencionada melhora nos mecanismos de *profile*, o que possibilita *profiles* específicos para, por exemplo, J2EE ou .Net. A segunda é uma melhora para a especificação de ambientes de execução e nós de computação, permitindo assim a especificação completa dos ambientes de implantação (OMG, 2003d).

- Transformação da especificação do sistema para uma plataforma particular

  A UML tem opções de relacionamentos que podem ser utilizados para especificar as transformações entre modelos, incluindo Realização (*Realization*: um relacionamento especializado entre dois conjuntos de modelos, um representando especificação e o outro representando a implementação), Refino (*Refine*: especifica um relacionamento entre elementos do modelo em diferentes níveis semânticos, como análise e

projeto) e Rastro (*Trace*: indica o relacionamento entre dois elementos que representam o mesmo conceito) (OMG, 2003d), (OMG, 2005).

Segundo (OMG, 2003d), os objetivos das técnicas de especificação da UML 2.0 são:

- Correção: as técnicas de especificação devem melhorar a correção do metamodelo, auxiliando a validá-lo;
- Precisão: as técnicas de especificação devem aumentar a precisão tanto da sintaxe quanto da semântica;
- Concisão: as técnicas de especificação devem permitir que sintaxe e semântica sejam definidas sem detalhes supérfluos;
- Consistência: as técnicas de especificação devem complementar a abordagem de metamodelagem adicionando detalhes de uma maneira consistente;
- Compreensão: enquanto aumentando a precisão e concisão, as técnicas de especificação devem também melhorar a leitura da especificação. Por esta razão, um formalismo menos que estrito foi aplicado.

Porém, conforme mencionado na introdução deste trabalho, a UML 2.0 "... não é uma especificação completamente formal de uma linguagem, porque ao fazer isto seria adicionada significativa complexidade sem um claro benefício" (OMG, 2003d).

2.2 Definição de uma linguagem

A definição de uma linguagem é constituída de duas partes: a sintaxe e a semântica. A sintaxe cuida da forma das expressões permitidas na linguagem. A semântica cuida do significado qualquer expressão sintaticamente correta em uma linguagem (ZHANG; XU, 2004).

A sintaxe é dividida em sintaxe concreta e sintaxe abstrata. A primeira representa os símbolos da notação da linguagem e a segunda descreve a representação interna da máquina para uma determinada linguagem. A semântica, por sua vez, define o significado da notação e é dividida em semântica estática e em semântica dinâmica. A primeira lida com a estrutura de cada sentença e a segunda define comportamento dinâmico (BARESI; PEZZÈ, 2005).

A título de exemplo, semântica estática descreve propriedades de um programa que podem ser resolvidas antes da execução do programa, enquanto a semântica dinâmica descreve propriedades do programa durante a execução (ZHANG; XU, 2004). A semântica estática está especialmente relacionada com atividades de validação, como o descasamento de tipo (*type mismatch*) causado ao se tentar adicionar um caractere a um tipo aritmético real (MEEK, 1990).

Também a título de exemplo, os elementos que compõem a sintaxe abstrata da língua inglesa incluem substantivos, verbos e adjetivos, enquanto as regras para a combinação deles são dadas pela gramática da língua inglesa. A sintaxe concreta define como estes elementos aparecem de uma forma utilizável para o ser humano. No caso da língua inglesa, duas formas possíveis são: escrita (textual) e falada (GREENFIELD *et al*, 2004).

Se a linguagem tiver uma sintaxe gráfica, é importante definir primeiro a sintaxe abstrata da linguagem. A sintaxe concreta é então definida pelo mapeamento da notação para a sintaxe abstrata (OMG, 2003d).

2.2.1 Graus de formalismo

As linguagens podem ser classificadas segundo seu grau de formalismo. Para isso são previstas três classificações distintas (BARESI; PEZZÈ, 2005):

- Linguagens formais, onde sintaxe e semântica são formalmente definidas;
- Linguagens semi-formais, onde a sintaxe é formal, mas a semântica deixa espaço para diferentes interpretações;
- Linguagens informais, onde tanto a sintaxe quanto a semântica são definidas informalmente.

A figura 1 apresenta uma classificação das linguagens segundo sua semântica e especificidade de domínio:

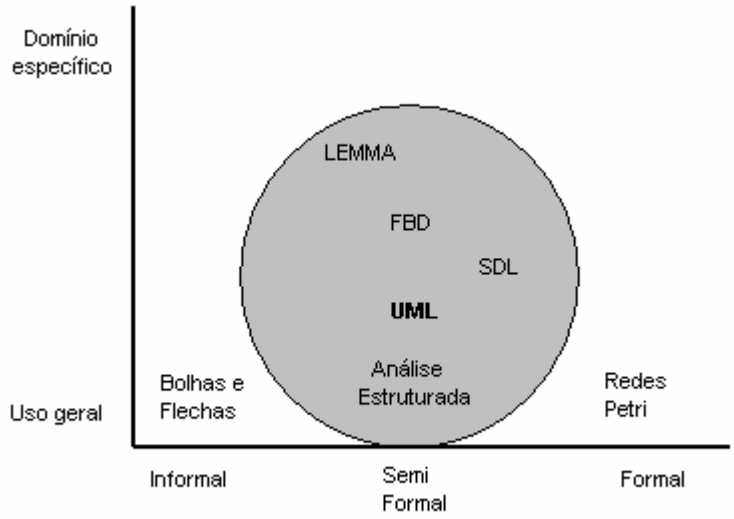

Figura 1 - Classificação das linguagens segundo grau de formalismo e especificidade de domínio (BARESI; PEZZÉ, 2005)

Além da UML, algumas outras linguagens relacionadas são: LEMMA (*Language for Easy Medical Models Analysis*), uma linguagem gráfica para modelagem de processos de diagnósticos. Tem uma sintaxe fixa, mas sua semântica depende da análise a ser realizada; FBD (*Function Block Diagram*), linguagem semi-formal para um domínio específico. Foi proposta para o projeto de controladores lógicos programáveis; SDL (*Specification and Description Language*), uma linguagem formal, orientada a objetos, definida pelo *International Telecommunications Union–Telecommunications Standardization Sector* (ITU–T). A linguagem é voltada para a especificação de aplicações complexas, orientadas a eventos, tempo-real e interativas, envolvendo atividades concorrentes (BARESI; PEZZÉ, 2005), (IEC, 1996).

2.3 A sintaxe da UML

Sintaxe e semântica das linguagens vêm sendo estudadas há vários anos. A sintaxe começou a ser definida mais cedo: já no desenvolvimento do ALGOL 60, Peter Naur introduziu a notação atualmente conhecida como Backus-Naur Form (BNF), uma notação formal para descrever a sintaxe de linguagens. Essa notação se tornou amplamente reconhecida e praticamente universal em seu uso, quase que suplantando

totalmente outras técnicas para descrição sintática de notações textuais (ZHANG; XU, 2004).

Uma sintaxe é descrita como um conjunto de caracteres (alfabeto) e suas possíveis seqüências (símbolos, seqüências de caracteres, sentenças) nas notações textuais convencionais. A situação se torna mais complexa quando há diagramas envolvidos: a sintaxe passa a não lidar com seqüências lineares de caracteres, mas com diferentes tipos de caixas (representando classes, estados, objetos, etc) e diferentes tipos de arcos (representando transições, associações, etc). Há quatro níveis para a definição das linguagens textuais convencionais (RUMPE, 1998):

- O primeiro nível é um conjunto de caracteres definido para formar um alfabeto;
- No segundo nível os caracteres são agrupados, dando origem a palavras, nomes, delimitadores, etc;
- O terceiro nível agrupa estas palavras em sentenças usando uma gramática livre de contexto;
- O quarto nível restringe estas sentenças através de um contexto.

O BNF não pode ser aplicado a UML porque a ela é uma linguagem visual. Neste caso, utiliza-se uma técnica diferente, conhecida como metamodelagem.

2.3.1 Metamodelagem

Um modelo é a descrição de um sistema. Uma linguagem define quais são os recursos disponíveis para se escrever um modelo. Na UML, por exemplo, há o conceito de classe, que pode ser utilizado para a definição de uma classe Cliente, que pode ter instâncias como José ou João. Assim, cada tipo de elemento de uma linguagem é definido pelo metamodelo da linguagem. No caso da UML, podem-se utilizar classes, associações e atributos porque o metamodelo da linguagem tem elementos que definem o que vem a ser classes, associações, atributos e quaisquer outros elementos da linguagem (KLEPPE;WARMER;BAST, 2003).

Como o metamodelo também é um modelo, ele é representado utilizando-se uma linguagem, que neste caso é chamada de metalinguagem. O BNF, por exemplo, é

uma metalinguagem. Desta forma, uma metalinguagem também precisa de um metamodelo próprio, ou seja, um meta-metamodelo (KLEPPE;WARMER;BAST, 2003).

A princípio, este ciclo poderia não ter fim, mas o OMG criou um padrão chamado MOF para a definição dos metamodelos de seus padrões, como CWM, XMI e a própria UML. O OMG usa uma arquitetura em quatro camadas para seus padrões, chamadas de M0, M1, M2 e M3, conforme figura 2:

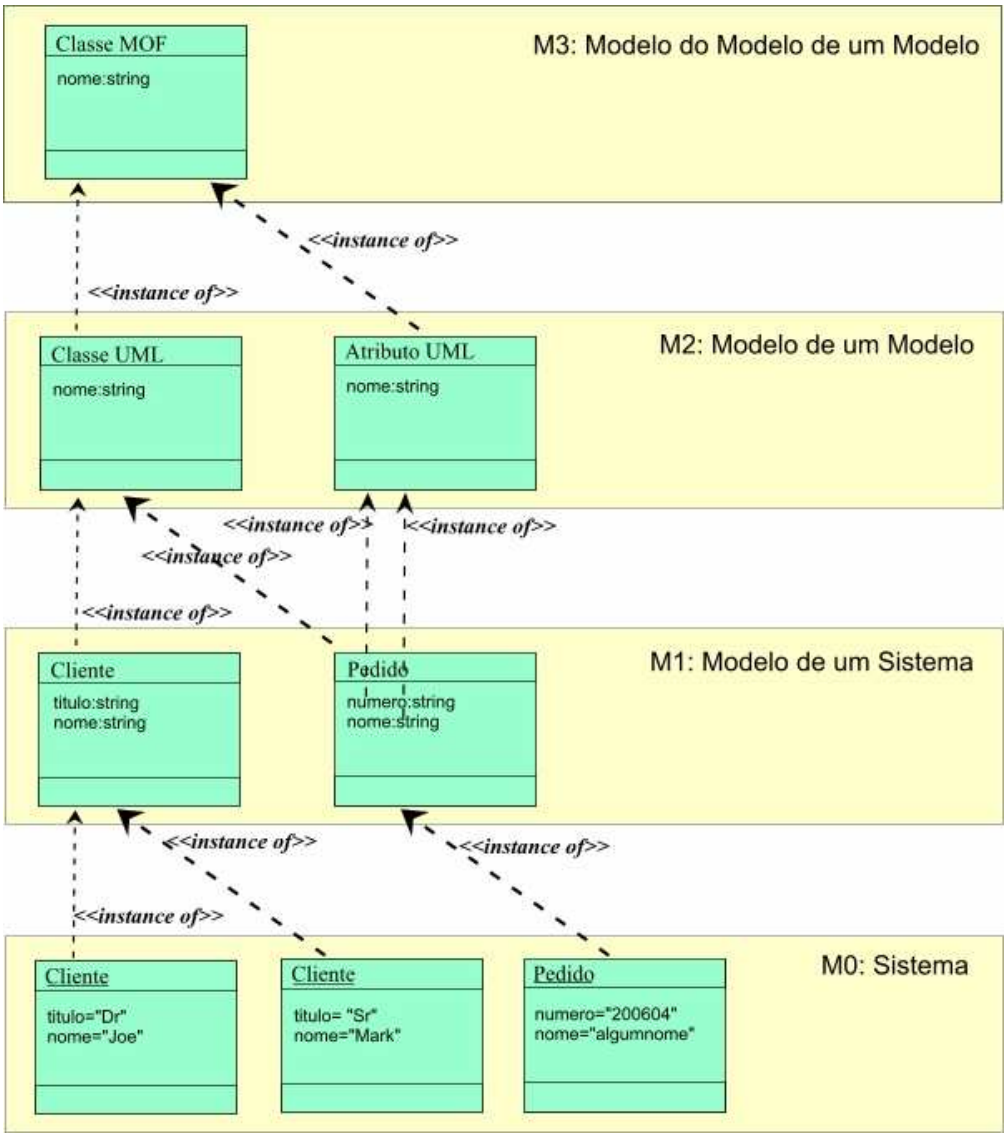

Figura 2 - As quatro camadas da arquitetura OMG (caso UML) (KLEPPE;WARMER;BAST, 2003)

O MOF também é utilizado para definir um formato de intercâmbio para os modelos M1. Sempre que uma linguagem de modelagem é definida utilizando-se um

metamodelo MOF, este define uma maneira padronizada de geração de formato de intercâmbio para modelos naquela linguagem. Este formato de intercâmbio é baseado em XML e é chamado XMI (XML *Metadata Interchange*). Como o MOF é definido com base em si próprio, XMI também pode ser utilizado para gerar formatos de intercâmbio padrão para metamodelos (KLEPPE;WARMER;BAST, 2003).

2.4 A semântica UML

Em contraste com a situação mais estável da sintaxe, depois de mais de 40 anos de desenvolvimento ainda não há uma notação formal universalmente aceita para descrição semântica – ao contrário: um grande número de notações formais foram criadas e novas surgem regularmente. A razão para isso reside no fato do comportamento dos programas apresentarem complexidade muito maior do que suas estruturas (ZHANG; XU, 2004).

Apesar deste último parágrafo explicitamente mencionar programas, pode-se entender esta mesma situação aplicada às linguagens de modelagem. Aliás, com o crescimento da importância do MDD, pode-se "... sugerir que a distinção entre linguagens de programação e linguagens de modelagem venha a se tornar irrelevante em pouco tempo" (GREENFIELD *et al*, 2004).

Uma abordagem semântica formal é vantajosa para o MDD / MDA porque dá uma definição não ambígua para uma linguagem. A essência de uma semântica formal é o seu tratamento matemático rigoroso. (ZHANG; XU, 2004).

Segundo a Figura 1, a UML é uma linguagem semi-formal e que pode ser utilizada em diferentes domínios. Esta flexibilidade no uso é decorrente da incompletude ou informalidade de sua definição original e resulta em diferentes interpretações (BARESI; PEZZÈ, 2005). Com relação a este último parágrafo, duas observações são pertinentes:

- Parte da definição da semântica na UML é definida utilizando-se linguagem natural (inglês) (OMG, 2003d).
- Com relação à classificação como linguagem para múltiplos domínios, vale esclarecer que a "UML é uma linguagem com escopo muito amplo que cobre

> um grande e diverso conjunto de aplicações. Nem toda a sua capacidade de modelagem é necessariamente útil em todos os domínios ou aplicações. Isto sugere que a linguagem deva ser estruturada de forma modular, com a habilidade de seleção de apenas as partes da linguagem que sejam de interesse direto" (OMG, 2004b).

2.4.1 Meios para a definição de semântica na UML

A especificação da semântica da linguagem UML usa uma combinação de linguagens – um subconjunto da UML chamado OCL e linguagem natural precisa (inglês). Fora isso também há um expediente chamado Ponto de Variação Semântica (OMG, 2003d).

As sentenças OCL permitem formalismo matemático à linguagem aumentando a precisão da UML. A razão de sua existência é que os diagramas da UML tipicamente não são suficientemente refinados para fornecer todos os aspectos relevantes de uma especificação (OMG, 2003c).

O exemplo da figura 3 é útil para entender tal conceito, mostrando ser possível a especificação de detalhes que não poderiam ser representados somente com o diagrama de classes:

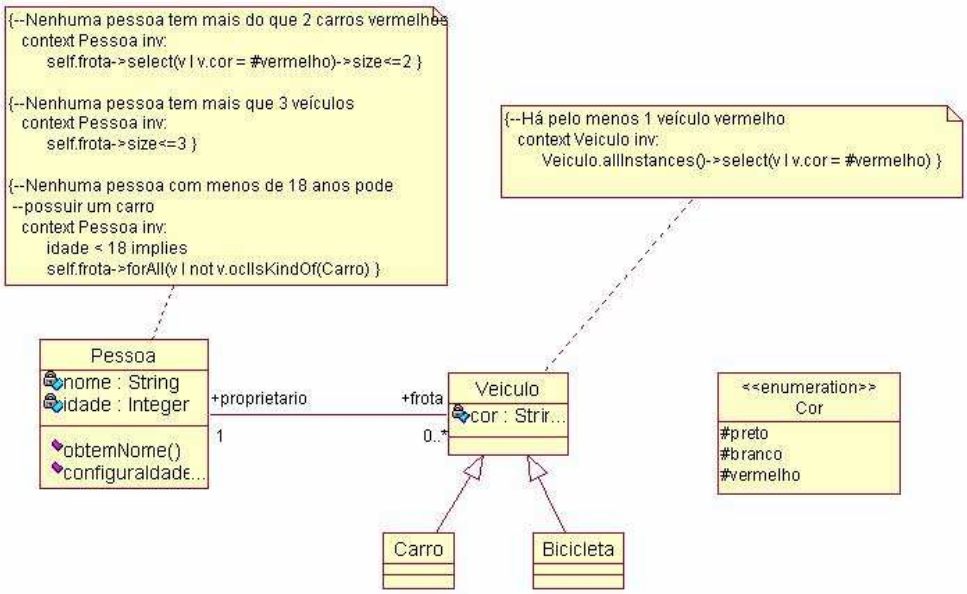

Figura 3 - Um modelo UML complementado com sentenças OCL

Pontos de Variação Semântica, por sua vez, indicam pontos onde a semântica é intencionalmente especificada de forma incompleta, para fornecer liberdade para um refinamento da UML para um domínio específico, por exemplo, utilizando-se *profiles* (OMG, 2003d).

A utilização de linguagem natural e de pontos de variação semântica é criticada por sua informalidade, fazendo com que a semântica da linguagem fique incompleta e ambígua (REGGIO; CERIOLI; ASTESIANO, 2000).

2.5 O problema de consistência dos modelos UML

O uso da UML pode levar à construção de descrições conflitantes, causada por suas múltiplas visões, as quais oferecem diferentes formas para capturar o mesmo aspecto de um sistema. Os aspectos comportamentais das especificações de um sistema podem se sobrepor, causando redundância no melhor dos casos e inconsistência no pior. Portanto, a existência de uma semântica formal para as sublinguagens da UML e suas interconexões, bem como ferramentas para análise e consistência de comportamento é crucial (KENT; EVANS; RUMPE, 1999), (HAREL; RUMPE, 2004).

O próprio nível de abstração de um diagrama UML permite flexibilidade, o que é interessante como forma de comunicar idéias entre pessoas, mas é um fator adicional de incertezas para a abordagem MDA. Por exemplo, em (BOOCH; RUMBAUGH; JACOBSON, 2005), os diagramas de atividade são prescritos para modelagem de aspectos dinâmicos de um sistema, porém podem ser utilizados para modelar em diferentes níveis de abstração, isto é, como um todo, em um subsistema, em uma operação ou mesmo em uma classe.

Ainda dentro desta questão de definição de níveis de abstração, os modelos UML utilizados na especificação de um sistema de software devem manter-se compatíveis. Por exemplo, em uma abordagem por Casos de Uso os requisitos funcionais descrevem o sistema segundo um nível de abstração muito alto e podem ser refinados por diagramas de seqüência ou colaboração. Estes Diagramas de interação (seqüência e colaboração) expressam o comportamento no que diz respeito à interação entre objetos. Para chegar ao nível da implementação, o projetista pode vir a utilizar um novo refinamento, desta vez com diagrama de estados. Estes modelos acabam por se tornar dependentes entre si, de tal forma que modificações em um modelo podem afetar outros em níveis diferentes (WHITTLE *et al*, 2000).

Mesmo que a questão de nível de abstração estivesse resolvida, deve-se considerar que os modelos UML comportamentais são tipicamente não ortogonais, mostrando comportamentos similares a partir de perspectivas diferentes, o que pode introduzir ambigüidade (WHITTLE et al, 2000).

De acordo com (ENGELS *et al*, 2001), os possíveis tipos de consistência são:

- Consistência horizontal

  É a consistência entre modelos que representam aspectos relacionados de um sistema em um mesmo nível de abstração. Problemas de consistência horizontal podem surgir em casos, como na UML, onde a especificação é constituída de diferentes partes porque o sistema é modelado a partir de diferentes visões. Nestes casos ocorre a sobreposição dos modelos, isto é, eles não se mantêm completamente independentes. É necessário que os diferentes modelos sejam mantidos consistentes, caso contrário, sua implementação torna-se inexeqüível (ENGELS et al, 2001).

- Consistência vertical

  É a consistência entre modelos que representam um mesmo aspecto de um sistema, em níveis de abstração diferentes. Problemas de consistência vertical podem surgir quando um modelo é transformado em outro, mais refinado. O modelo refinado precisa estar consistente com o que o originou, de forma a manter a especificação consistente como um todo (ENGELS *et al*, 2001).

Deficiências na definição de uma linguagem podem levar aos problemas descritos. A consistência sintática de uma linguagem assegura que uma especificação esteja em conformidade com a sintaxe abstrata do metamodelo, requerendo que o modelo como um todo seja bem formado. A consistência semântica é dividida em horizontal e vertical. A primeira requer que modelos de diferentes visões sejam semanticamente compatíveis no que diz respeito à descrição de um aspecto do sistema. A segunda (vertical) requer que um modelo refinado seja semanticamente compatível com o modelo que o originou (ENGELS *et al*, 2001). Pelo exposto neste capítulo, percebe-se que deficiências semânticas da UML podem levar a inconsistências horizontais e verticais na especificação de um software.

2.5.1 Um exemplo de consistência com UML

Pode-se observar estas questões de consistência numa especificação (parcial) de um sistema de vendas, representada pelas figuras 4, 5, 6 e 7.

O diagrama de classes com sentenças OCL da figura 4 apresenta aspectos estáticos e comportamentais do software. Ele representa entidades do sistema de software, suas propriedades (atributos), operações (métodos) e relacionamentos. Também contém um conjunto de restrições escritas em OCL, usado para representar regras de negócio e que tem influência sobre o comportamento do software, como as pré e pós condições de métodos de algumas classes.

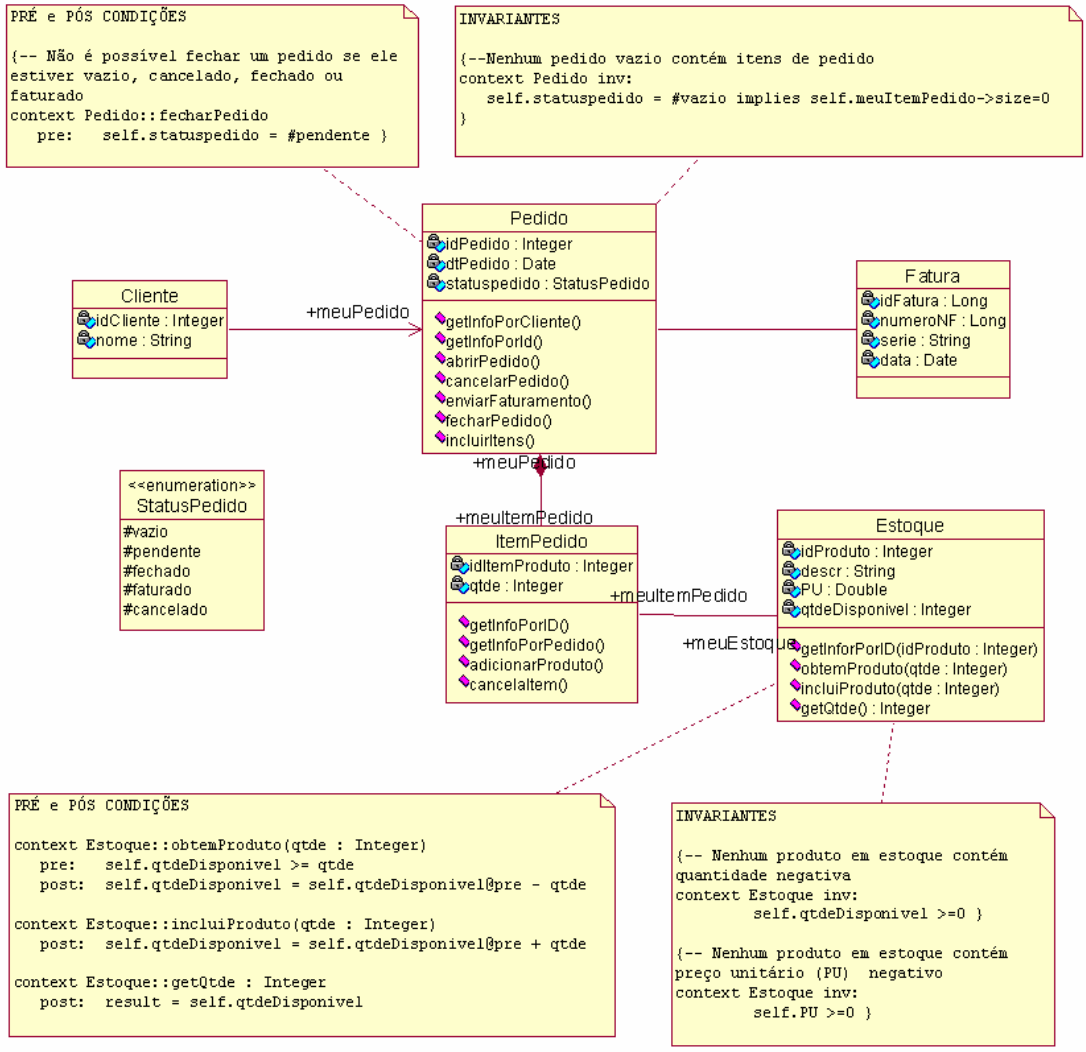

Figura 4 - Diagrama de classes de um sistema de vendas, complementado por sentenças OCL

O diagrama de seqüência da figura 5 representa o comportamento do sistema durante o cancelamento de um pedido. Ele exibe a seqüência de mensagens que são trocadas entre as entidades para permitir esta funcionalidade. Durante o cancelamento, o pedido recupera as informações de cada um de seus itens, cancelando-os um a um e recompondo o estoque.

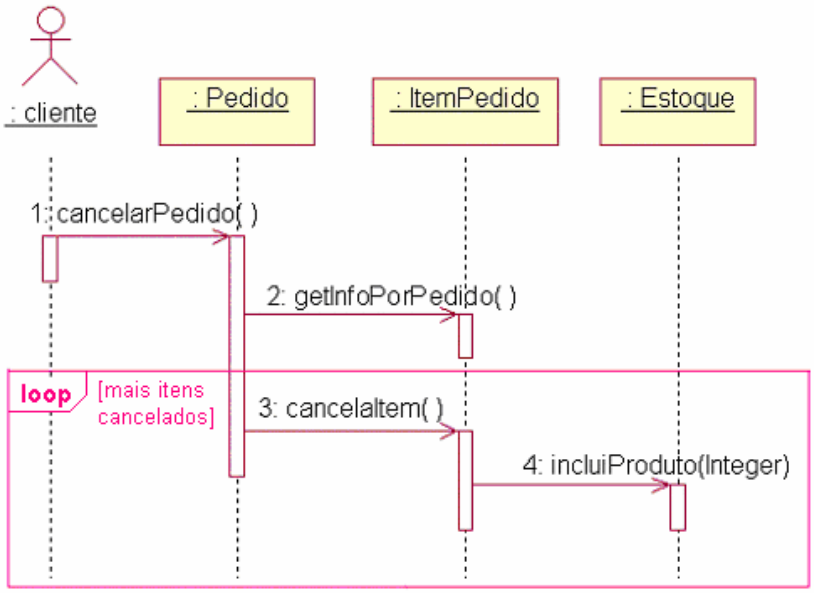

Figura 5 - Diagrama de seqüência de um sistema de vendas, exibindo o comportamento de cancelamento de pedidos

O diagrama de seqüência da figura 6 exibe a troca de mensagens entre as entidades do sistema durante a abertura de um novo pedido. O cliente solicita a abertura de um pedido, incluindo quantos itens de pedido ele quiser. Encerra-se a inclusão de novos itens com o fechamento do pedido

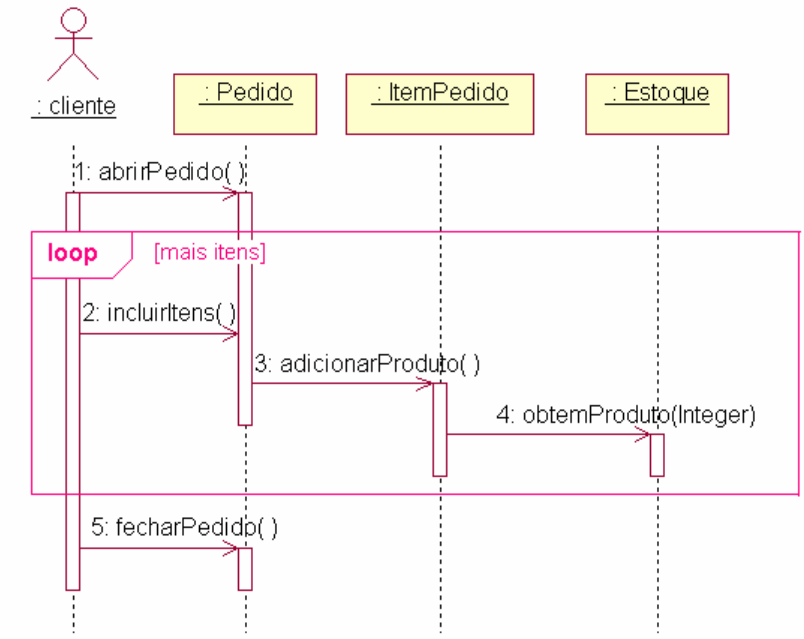

Figura 6 - Diagrama de seqüência de um sistema de vendas, exibindo o comportamento de abertura de pedidos

O diagrama de estados da figura 7 exibe o comportamento do método *incluirItens*, da classe Pedido, detalhando seu comportamento.

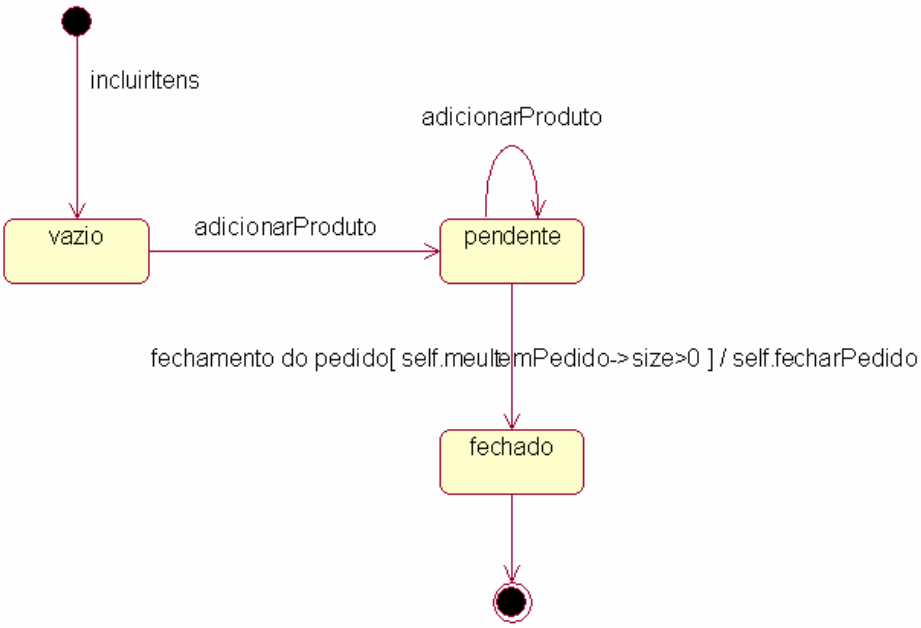

Figura 7 - Diagrama de estados de um sistema de vendas, exibindo o comportamento de inclusão de itens

O exemplo exposto mostra a questão da sobreposição na UML. Percebe-se que o diagrama de estados da figura 7 sobrepõe-se ao invariante e à pré condição da classe

Pedido. Além disso, os diagramas de seqüência e o diagrama de estados têm que estar em conformidade com o diagrama de classes no que diz respeito aos métodos e aos relacionamentos das classes.

2.6 Considerações finais do capítulo 2

A UML apresenta algumas deficiências para ser utilizada na abordagem MDA. Ela pode levar a inconsistências horizontais e verticais, que introduzem ambigüidade em uma especificação de software, tornando inexeqüível o processo de geração de código-fonte a partir de modelos.

Entretanto, definindo-se um relacionamento formal entre os diagramas utilizados, ou seja, definindo-se claramente o nível de abstração de cada diagrama (para consistência vertical) e o inter-relacionamento entre diagramas do mesmo nível de abstração (consistência horizontal), é possível fazer uma verificação de consistência de uma especificação em tempo real durante a modelagem. Observado este ponto, a UML pode ser utilizada na abordagem MDA.

3 O DESENVOLVIMENTO ORIENTADO POR MODELOS (MDD)

Este capítulo fornece uma breve visão do cenário atual, onde o Desenvolvimento Orientado por Modelos (MDD) está em seu início. O capítulo apresenta um esboço deste cenário, mostrando algumas das tendências para o MDD sem esgotar o tema. Apresenta também, com mais detalhes, a abordagem MDA mostrando os seus principais conceitos.

3.1 O cenário atual do MDD

Não existe atualmente apenas uma única abordagem para o Desenvolvimento Orientado por Modelos. Neste item da dissertação, três abordagens convenientemente escolhidas são brevemente comentadas e comparadas. As abordagens escolhidas para tal são: OMG-MDA, *Software Factories* e *Model-Driven Software Development*. Duas delas foram escolhidas por sua relevância: a primeira é a OMG-MDA, proposta do OMG que conta com o apoio de gigantes da área de informática como IBM e Compuware. A segunda é *Software Factories*, abordagem defendida por outro gigante da informática: a Microsoft. A terceira abordagem, o *Model-Driven Software Development*, foi escolhido por ser apresentada como uma abordagem mais abrangente que o MDA, porém compatível. (BETTIN; VÖLTER, 2005). O MDSD vêm sendo discutido em *wokshops* em recentes edições da conferência OOPSLA (edições de 2002 até 2005. A página do workshop na edição 2005 é http://www.oopsla.org/2005/ShowEvent.do?id=199). Vale observar que na OOPSLA 2005 também houve um *workshop* sobre *Software Factories* (na página http://www.oopsla.org/2005/ShowEvent.do?id=203).

As principais diferenças entre as abordagens estão relacionadas a seguir:

- Diferenças com relação à Linguagem de Modelagem

  Uma importante diferença existente entre as abordagens está relacionada ao uso da UML e do MOF. É a partir do MOF que o metamodelo da UML é definido. A UML é a linguagem utilizada pelo usuário MDA para a

representação de modelos (PIM e PSM. Estes modelos serão vistos adiante neste capítulo).

O MDSD é baseado em princípios de modelagem e metamodelagem e não está vinculado a nenhum metamodelo específico (como o MOF) e a nenhuma linguagem de modelagem específica (como a UML). Isto não significa que a UML não possa ser utilizada no MDSD: a proposta é que o aumento no nível de abstrações utilize Linguagens para Domínios Específicos (DSL), que podem ou não ser baseadas em UML ou em MOF (BETTIN, 2004b).

Software Factories, por sua vez, é clara em não adotar a UML. Argumenta, segundo sua visão, que a UML é largamente inconsistente para ser utilizada por uma abordagem de desenvolvimento orientado por modelos. Também argumenta que, sendo uma linguagem de modelagem de uso geral, a UML não pode atender a todos os possíveis domínios de uma forma que forneça significado real. A abordagem também descarta o uso do MOF, o que acarreta o descarte do padrão XMI, baseado no MOF. Segundo a abordagem, *Software Factories* utiliza modelos simples, porém formais e altamente focados em aspectos específicos de requisitos, arquitetura e implementação para famílias de produtos. Tais modelos são baseados em Linguagens para Domínios Específicos (DSL) (GREENFIELD *et al*, 2004).

Linguagens para Domínios Específicos (DSL) são linguagens processáveis por máquina cujos termos são derivados de um modelo de domínio específico. As DSL são feitas para domínios particulares e capturam precisamente a semântica daquele domínio (THOMAS; BARRY, 2003).

A UML é uma linguagem de modelagem de uso geral que pode ser especializada através de mecanismos de extensão, tornando-se uma Linguagem de Domínio Específico. Por Exemplo: UML *Profile for Software Radio Waveform Applications* (OMG, 2001a) (OMG, 2004a).

Linguagens para Domínios Específicos são utilizadas no contexto de Linha de Produção de Software e é daí que surge esta ênfase do MDSD e da *Software Factories* com relação a elas: ambas abordagens declaram explicitamente ter suas raízes na Linha de Produção de Software (*Software*

Product Line). Já a documentação disponível no OMG não faz esta explícita vinculação entre MDA e Linha de Produção de Software.

Linha de Produção de Software é um conjunto de sistemas de software que compartilham funcionalidades comuns e que satisfazem uma necessidade específica de um segmento de mercado. São desenvolvidas a partir de um conjunto comum de artefatos, tornando-a atrativa pela economia oferecida (CLEMENTS; NORTHROP, 2001).

Cada sistema é formado agregando-se componentes a partir de uma base comum de artefatos, configurados quando necessário através de mecanismos de variação pré-planejados, como parametrização ou herança. Construir um novo sistema torna-se mais uma questão de composição ou geração do que criação; a atividade predominante é a integração ao invés da programação. Para cada Linha de Produção de Software há um guia pré-definido que especifica a exata abordagem de construção do produto (CLEMENTS; NORTHROP, 2001).

- Processo de Desenvolvimento de Software

  Outra importante diferença que distingue o MDA das abordagens MDSD e *Software Factories* é o fato do MDA não ser um processo de desenvolvimento de software. O MDA fornece linhas gerais para um processo de desenvolvimento, mas não fornece uma metodologia clara que possa ser utilizada para aplicar o desenvolvimento orientado a modelos na prática. *Software Factories* apresenta-se como um processo de desenvolvimento de software pouco prescritivo - ao contrário do RUP- e o MDSD adota práticas do desenvolvimento ágil (o que também significa ser um processo pouco prescritivo) (BETTIN, 2004a) (BETTIN, 2004b) (BROWN; CONALLEN, 2005b), (FERNANDES *et al*, 2005), (GREENFIELD *et al*, 2004).

- Independência da Plataforma

  A abordagem MDA prevê o desenvolvimento de um modelo inicial, livre de considerações sobre a plataforma onde o sistema será implantado. Este

modelo inicial é então transformado - segundo um conjunto de transformações - em um modelo que considera as características específicas da plataforma, este último sendo subseqüentemente transformado em código-fonte (OMG, 2003b).

Assim como o MDA, o MDSD também utiliza a noção de modelos dependentes e independentes da plataforma de implantação. *Software Factories*, por sua vez, não utiliza tal abordagem. Segundo depoimento de Jack Greenfield - principal nome da abordagem *Software Factories* - apesar de algum isolamento da tecnologia da plataforma ser atingível, ele não acredita que seja exeqüível a produção de software seguro, utilizável, confiável e com desempenho, a partir de especificações que não levem em conta qualquer tipo de dependência de plataforma. De fato ele questiona esta posição sobre independência (BETTIN, 2004b), (GREENFIELD, 2005).

- Padrões e código abertos

  O MDA faz uso dos padrões abertos que o OMG vêm desenvolvendo ao longo dos últimos anos, como UML, MOF e CWM. O MDSD pode fazer uso desses padrões, mas não se restringe a eles (BETTIN, 2004b).

  Ainda com relação aos padrões abertos, o OMG está trabalhando no desenvolvimento de uma linguagem padronizada (QVT) para a definição das transformações necessárias ao MDD. O MDSD vai mais longe e além de encorajar o desenvolvimento do padrão QVT, também apóia o desenvolvimento e uso de uma infra-estrutura de código aberto (*Open Source*) (BETTIN, 2004b).

  Software Factories, ao contrário dessas iniciativas, apresenta ferramentas e padrões proprietários.

3.2 A visão do OMG

Segundo a visão do OMG (*Object Management Group*), a infraestrutura de TI de qualquer organização atual é um sistema de computação distribuída. Para integrar e usar informação corretamente é necessário acesso através de departamentos, através

da companhia, através do mundo e, principalmente através da cadeia de suprimentos ou serviços dos fornecedores. Dentro deste contexto, o OMG afirma não haver mais espaço para as aplicações isoladas (*standalone*). (OMG, 2003b).

Também segundo a visão do OMG, apesar de se ter como certo que todas as aplicações devam ser feitas para durar, para serem integradas e atualizadas, os desenvolvedores de software ignoram estes fatos e preocupam-se apenas com as especificações que lhes são entregues. Além disso, a maior parte dos sistemas de *software* continua sendo feita ignorando-se a realidade de constantes mudanças de infraestrutura e de requisitos (OMG, 2003b).

Simultaneamente o OMG considera que houve uma mudança de pensamento no mercado de TI: a partir do momento em que os métodos e linguagens de modelagem finalmente começaram a convergir, o interesse em modelagem de dados e aplicações aumentou significativamente. Neste cenário, o OMG sustenta que toda construção de software precisa de um cuidadoso trabalho de projeto. Também sustenta que modelos baseados em padrões do próprio OMG, como UML, MOF e CWM, podem automatizar a construção de aplicações e bancos de dados e a interligação entre diferentes aplicações (OMG, 2003b).

O OMG adota esta abordagem firmemente focada em modelos, defendendo um novo passo evolucionário no desenvolvimento de *software*: a automação por modelos é um novo nível de compilação, ou seja, o *software* deve ser diretamente gerado a partir de modelos (OMG, 2003b).

Vale observar que o MDD não é uma unanimidade entre a comunidade de *software*, onde há grupos que discutem o papel e importância dos modelos no desenvolvimento de aplicações (FERNANDES *et al*, 2005).

3.3 A proposta MDA

Com base em sua visão, o OMG propõe o MDA – *Model Driven Architecture* – com a intenção de garantir flexibilidade em longo prazo para (OMG, 2003b):

- Implementação: projetos existentes podem ser implementados utilizando-se novas infra-estruturas de implementação;

- Integração: pode-se automatizar a produção de pontes para integração das diferentes infra-estruturas utilizadas por um sistema de *software*;
- Manutenção: a disponibilidade do design de uma forma legível pela máquina dá ao desenvolvedor acesso direto à especificação do sistema, tornando a manutenção muito mais simples;
- Teste e simulação: uma vez que os modelos desenvolvidos podem ser utilizados para gerar código, eles também podem ser validados contra requisitos, testados contra várias infraestruturas e podem ser usados para simular diretamente o comportamento do sistema em projeto.

O MDA apóia a idéia de separar a especificação do sistema da forma como o sistema utiliza as facilidades da plataforma. O MDA fornece uma abordagem que possibilita (OMG, 2003b):

- Especificação de um sistema independente da plataforma que o suporta;
- Especificação de plataformas;
- Escolha de uma plataforma particular para um sistema;
- Transformação das especificações de um sistema para uma particular plataforma.

3.3.1 Os benefícios do MDA

(KLEPPE;WARMER;BAST, 2003) apresentam uma relação com quatro diferentes benefícios que podem ser obtidos com a utilização do MDA:

- Produtividade

  No MDA, o foco do desenvolvedor passa a ser o desenvolvimento do PIM. Os PSMs necessários são gerados pela transformação de PIM para PSM. É necessário que alguém defina a exata transformação - uma tarefa difícil e especializada – mas que definida uma única vez, pode ser aplicada ao desenvolvimento de vários sistemas.

  Com as transformações, detalhes técnicos são adicionados automaticamente do PIM para o PSM aumentando a produtividade no desenvolvimento do

sistema. Entretanto tais ganhos somente serão obtidos se a geração do PSM a partir do PIM for automatizada por ferramentas.

- Portabilidade

  No MDA a portabilidade é alcançada pelo foco no desenvolvimento do PIM, que é, por definição, independente de plataforma. Um mesmo PIM pode ser automaticamente transformado em múltiplos PSMs para diferentes plataformas. Tudo o que for especificado no PIM é completamente portável.

- Interoperabilidade

  Múltiplos PSMs gerados a partir de um PIM podem ter relacionamentos, os quais são chamados de pontes (*bridges*) dentro do vocabulário MDA. Tais relacionamentos são resolvidos pela geração dos PSMs e das necessárias pontes, conforme figura 8:

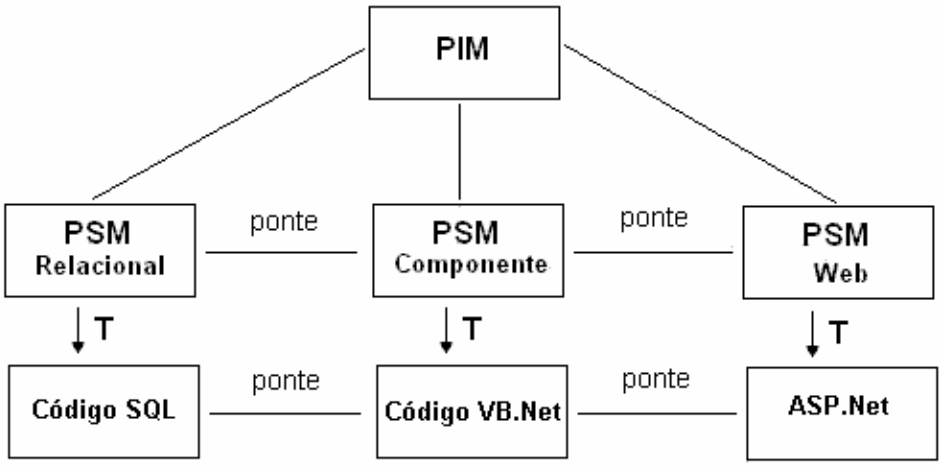

Figura 8 - Múltiplos PSMs gerados a partir de um PIM

No exemplo da figura 8, um PIM gera três PSMs. Considerando-se que haja uma classe persistente Cliente no PIM, ela será transformada em uma classe Cliente em VB.Net e uma tabela tb_cliente no banco de dados SQL. Construir uma ponte entre o objeto Cliente do PSM-VB.Net e o objeto Cliente do PSM-Relacional é criar um procedimento para recuperar as informações do cliente do DB e instanciar um objeto Cliente no VB.Net. Da

mesma forma, para persistir as informações de um cliente, recuperam-se os dados a partir de um objeto Cliente no VB e armazenam-se estes dados na tabela tb_cliente do DB.

- Manutenção e Documentação

  Na abordagem MDA os desenvolvedores focam no PIM, que é usado para gerar PSM, que por sua vez é usado para gerar código-fonte. O modelo é uma exata representação do código, portanto o PIM preenche a funcionalidade de documentação de alto nível.

  Como o PIM não é descartado no final do desenvolvimento, mudanças que eventualmente sejam feitas no sistema podem ser realizadas alterando-se o PIM e gerando-se novamente o PSM e o código-fonte.

3.4 Os modelos-chave do MDA

O OMG definiu um conjunto específico de camadas e transformações que fornecem um framework conceitual e um vocabulário para o MDA. Pode-se identificar três principais modelos: Modelo Independente de Computação (CIM), Modelo Independente de Plataforma (PIM), Modelo para Plataforma Específica (PSM) (BROWN, 2004).

O código-fonte não é explicitamente tratado pelo OMG como um modelo, mas é um modelo já que ele próprio não é diretamente executado pelo computador, mas transformado em uma linguagem que, esta sim, pode ser executada.

3.4.1 Modelo Independente de Computação (CIM)

O CIM não mostra detalhes da estrutura do sistema e às vezes é chamado de modelo de domínio. O vocabulário usado em sua especificação é familiar aos praticantes do domínio em questão. O CIM exerce um papel importante entre os especialistas do domínio de negócio e os especialistas em design e construção dos artefatos (OMG, 2003b).

O CIM, em termos MDA, pode ser considerado o modelo do Ponto de Vista Empresa (*Enterprise Viewpoint*) do RM-ODP, que tem a perspectiva de modelo de negócio e é

extremamente útil para comunicar as necessidades dos clientes para os arquitetos. Ele deve ser diretamente compreensível pelos envolvidos e focar no propósito, escopo e políticas de um sistema (ALMEIDA; VAN SINDEREN; PIRES, 2004), (PUTMAN, 2001).

Certas partes do CIM podem ser suportadas por um sistema de *software*, mas o próprio CIM permanece independente de *software*. A derivação automática de CIM para PIM não é possível porque as escolhas de quais partes do CIM devem ser suportadas por sistema de *software* são sempre humanas. Um PIM precisa ser desenvolvido para cada sistema de suporte de parte de um CIM (KLEPPE;WARMER;BAST, 2003).

3.4.2 Modelo Independente de Plataforma (PIM)

Todo o projeto de desenvolvimento MDA começa com a criação de um Modelo Independente de Plataforma (PIM), preferencialmente expresso em UML. Construído por especialistas de modelagem e de negócio, este modelo expressa as regras de negócio e funcionalidades o menos distorcidas possível pela tecnologia. Por conta desta independência tecnológica o PIM retém seu valor ao longo do tempo, requerendo mudanças apenas quando as condições de negócio assim o determinam (SIEGEL; OMG, 2001).

Entretanto esta questão de independência em relação à plataforma de implementação não é trivial. É necessário considerar que a fronteira entre o PIM e o PSM não é intuitiva pois, embora detalhes específicos de plataforma estejam ausentes, o PIM incorpora alguns aspectos tecnológicos. Conceitos como persistência, nível de segurança e suporte transacional podem ser tratados neste modelo, possibilitando um mapeamento mais preciso para o PSM (SIEGEL; OMG, 2001).

(ALMEIDA *et al*, 2004) observa que há pouco suporte metodológico para a distinção entre as questões dependentes e independentes de plataforma, e que isso pode acarretar problemas quanto aos benefícios prometidos pelo MDA através da separação de áreas de interesse entre PIM e PSM.

3.4.2.1 Definição de plataforma

Independência de plataforma é a qualidade de um modelo de abstrair-se das características tecnológicas de uma determinada plataforma.

O significado do termo *plataforma* é relativo a um particular ponto de vista, ou seja, o PIM para um contexto pode ser o PSM de outro. Por exemplo, um modelo pode ser um PIM no que diz respeito à escolha de um *middleware* de comunicação, se o mesmo não prescrever uma escolha tecnológica de *middleware*. Entretanto, se for tomada a decisão de utilização de um *middleware* particular como o CORBA, o modelo será um PSM CORBA. O novo modelo continua sendo um PIM no que diz respeito à escolha do ORB, e certamente no que diz respeito ao sistema operacional alvo (BROWN, 2004).

3.4.3 Modelo para Plataforma Específica (PSM)

O PSM (*Platform Specific Model*) é gerado a partir do PIM através de um conjunto de transformações. Como a implementação de um sistema pode cobrir diferentes plataformas (banco de dados, camada de apresentação *web*, servidor de aplicações, etc), diferentes PSM podem ser gerados a partir de um único PIM.

PSMs podem ser representados em UML padrão (*plain* UML) ou em UML estendida por *profiles*, o que aumenta o poder de representação da linguagem (SIEGEL; OMG, 2001).

Um PSM pode fornecer mais ou menos detalhes, dependendo de sua finalidade. Para que um PSM seja utilizado para a geração de código, ele precisa fornecer toda a informação necessária para a construção e operação de um sistema. Alternativamente ele pode ser utilizado como entrada para mais um conjunto de transformações, para refino e transformação no PSM utilizado para implementação (OMG, 2003b).

3.5 Mapeamentos e transformações

Um dos pontos-chave da abordagem MDA é a noção de mapeamento. Mapeamento é um conjunto de regras e técnicas usadas para modificar um modelo obtendo-se outro.

Os mapeamentos são utilizados pelas transformações entre PIM e PSM (OMG, 2001a).

Há duas formas diferentes de se fazer o mapeamento: uma é o mapeamento de tipos do modelo (*Model Type Mappings*) e a outra e o mapeamento de instâncias do modelo (*Model Instance Mappings*) (OMG, 2003b).

3.5.1 Mapeamento de tipos do modelo

O mapeamento de tipos do modelo especifica o mapeamento de qualquer modelo construído com tipos especificados na linguagem PIM para modelos expressos com tipos especificados na linguagem PSM (OMG, 2003b).

Ou seja, não é necessário relacionar todos os elementos do modelo-fonte com todos os elementos do modelo-alvo, mas sim as metaclasses dos elementos no modelo-fonte com as metaclasses dos elementos no modelo-alvo.

Para resolver questões de persistência, por exemplo, pode-se definir um conjunto de mapeamentos entre classes persistentes e elementos em um banco de dados relacional. Resumidamente, toda classe persistente *CL* pode ser mapeada no banco de dados para uma tabela T com o nome tb_*CL*. Cada atributo da classe em questão pode ser mapeado como uma coluna na respectiva tabela.

3.5.2 Mapeamento de instâncias do modelo

O mapeamento de tipos de modelo é apenas capaz de expressar transformações em termos de regras sobre elementos de um tipo no PIM, resultando na geração de elemento(s), de um ou mais tipos, no PSM, ou seja, é determinístico e depende unicamente das informações contidas no PIM para a geração do PSM (OMG, 2003b).

Uma maneira de se especificar transformações diferenciadas para elementos no modelo-fonte é através de marcas (*marking a model*). Elas representam conceitos no PSM e são aplicadas em elementos do PIM. Em um caso simples, um elemento do PIM é marcado uma vez, indicando que um certo mapeamento deve ser usado para transformar aquele elemento em um ou mais elementos do PSM. De forma mais

geral, um elemento no PIM pode receber diversas marcas, de diferentes mapeamentos, indicando que o elemento desempenha papéis em mais que um mapeamento (OMG, 2003b).

3.5.3 Transformações

A transformação de modelos, por sua vez, é a geração de um modelo-alvo, a partir de um modelo-fonte, ambos representando o mesmo sistema. Ambos os modelos estão representados por suas respectivas linguagens, por exemplo, UML para o PIM e Modelo Entidade-Relacionamento (MER) para um modelo de dados PSM. Isto não quer dizer que não haja transformações entre modelos descritos na mesma linguagem. A transformação PIM para PIM, prevista no MDA Guide Version (OMG, 2003b), é uma transformação de UML para UML (*plain UML*).

A transformação utiliza os mapeamentos, mas também contém outras informações, como, por exemplo, a condição no modelo-fonte para disparar a transformação, a linguagem do modelo-alvo, a linguagem-fonte, etc... (KLEPPE; WARMER; BAST, 2003).

É importante notar que PIM, PSM e código-fonte são modelos diferentes que representam o mesmo sistema. Isto não significa que o detalhamento das informações seja o mesmo em todos os modelos: persistência pode aparecer em um PIM marcado apenas como um esteriótipo. No PSM pode aparecer um conjunto de classes DAO e uma interface padronizada de acesso a dados. No código-fonte pode haver código JDBC de acesso a dados (no caso Java), sem que este código esteja no PIM ou PSM.

O grau de detalhamento aumenta com refinamento e incorporação automática de soluções pré-definidas (este ponto é abordado no capítulo 5). As figuras 9 e 10 a seguir, extraídas de (OMG, 2001a), mostram uma sugestão de transformação de um PIM marcado para um PSM Corba:

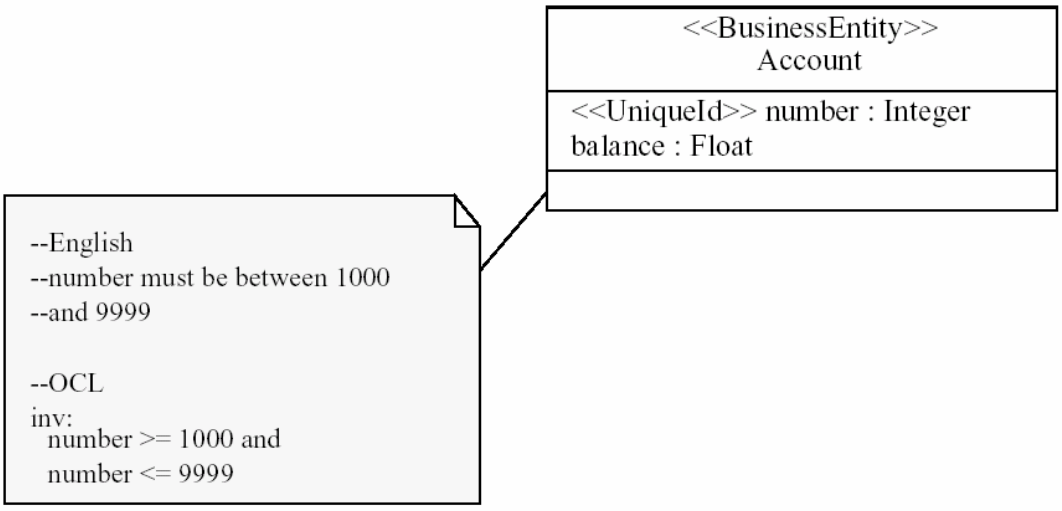

Figura 9 - Fragmento de um PIM marcado (OMG, 2001a)

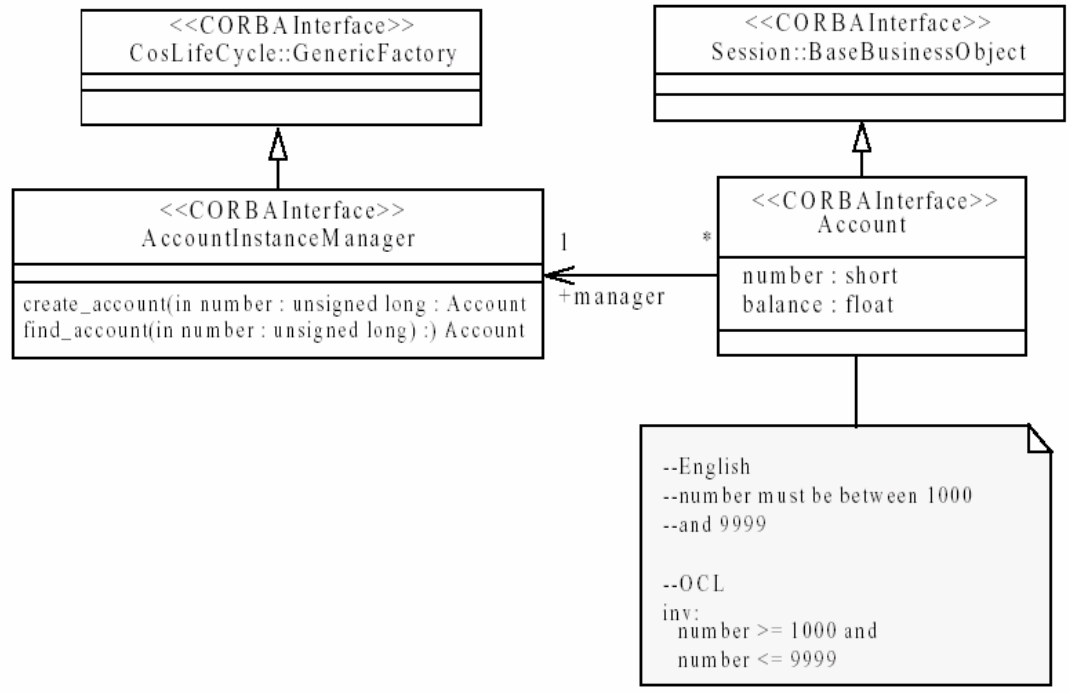

Figura 10 - Um PSM CORBA derivado do PIM da figura 9 (OMG, 2001a)

A idéia de produtividade do MDA advém deste aumento sucessivo e automático no detalhamento. Ganha-se tempo se as transformações forem capazes de complementar automaticamente os modelos, desonerando o desenvolvedor da preocupação com os detalhes de implementação.

3.5.4 Condições para as transformações

É importante notar que não é sempre possível conseguir o automatismo das transformações. Para que isso seja possível, é necessário que haja uma firme conexão semântica entre os elementos nos modelos, com regras claras e não ambíguas. Se as regras que definem estas conexões - e subseqüentemente as regras para construção de novos elementos em outros modelos - requeiram experiência ou julgamento do desenvolvedor, então a automação pode ser desconsiderada (BROWN; CONALLEN, 2005a). Um exemplo desta situação foi descrito no capítulo 2: segundo (BOOCH; RUMBAUGH; JACOBSON, 2005) um diagrama de atividades pode descrever o comportamento de um sistema completo, de um subsistema, de uma operação ou mesmo de uma classe. Havendo um contexto para entender o nível da abstração escolhido, essa indeterminação não chega a ser um problema para a cognição humana, porém não pode ser resolvida pelas transformações. Para evitar inconsistências, antes é necessário formalizar o inter-relacionamento entre os diversos diagramas utilizados para a representação de um sistema de *software*.

Outra razão para desconsiderar a automação é a inabilidade para acessar os elementos necessários dos próprios modelos de uma forma computacional. Passos de transformações que envolvam a leitura de documentos em linguagem natural, independentemente do nível de formalidade neles, tipicamente não são adequados à automação MDA como, por exemplo, a transformação automática de um documento de especificação de Caso de Uso em um modelo de análise. Por outro lado, a transformação de um diagrama de seqüência que acompanhe um Caso de Uso pode ser adequada para a automação (BROWN; CONALLEN, 2005a).

3.5.5 Tipos de transformação

Os três principais modelos no MDA são o CIM, PIM e PSM e não há transformações automáticas de CIM para PIM. Sendo assim, os tipos de transformação entre modelos previstos no MDA são (OMG, 2001a):

- PIM para PIM

O objetivo é aprimorar, filtrar ou especializar modelos, sem introduzir detalhes dependentes de plataforma;

- PIM para PSM

  O objetivo é utilizar um PIM suficientemente refinado e projetá-lo para um modelo de ambiente de execução;

- PSM para PSM

  O objetivo é refinar o modelo específico de uma plataforma;

- PSM para PIM

  O objetivo é extrair modelos independentes de plataforma a partir de implementações existentes.

Mesmo não estando explicitamente declarado em (OMG, 2001a) ou em (OMG, 2003b), há transformações entre PSM e código-fonte. O termo Transformações PSM-Código é referenciado no próprio *site* do OMG, no tópico MDA FAQ (*Frequently Asked Questions*) (http://www.omg.org/mda/faq_mda.htm).

Há diferenças entre as transformações PIM para PSM e PSM para Código-fonte. O primeiro tipo é normalmente uma transformação intra-linguagem (de UML para UML estendida). O segundo tipo é sempre inter-linguagem.

3.5.6 Transformações inter-linguagem e intra-linguagem

A transformação de modelos é um ponto essencial do MDA. Pode ocorrer entre modelos expressos na mesma linguagem ou em linguagens diferentes, mas no final do processo sempre se tem a transformação de modelos - preferencialmente expressos em UML - para código-fonte.

Nos mapeamentos inter-linguagem, a linguagem-alvo e a linguagem-fonte são diferentes. Nestes casos não se deve apenas descrever as ações de modelagem para a partir do modelo-fonte atingir o modelo-alvo, mas também comparar as duas linguagens para se determinar a extensão com a qual é possível transferir a semântica da linguagem do modelo-fonte para a linguagem do modelo-alvo. Obviamente quando a linguagem-fonte contém conceitos que a linguagem-alvo não é capaz de

representar, alguns mapeamentos do modelo-fonte para o modelo-alvo não são possíveis (CAPLAT; SOURROUILLE, 2005).

3.5.6.1 Extensão de linguagens para transformações inter-linguagem

Conforme mencionado no capítulo 2, os *profiles* UML são importantes para a abordagem MDA. A razão disso é a possibilidade de estender a linguagem para permitir a inclusão de conceitos que facilitem as transformações, principalmente entre linguagens diferentes.

As extensões desempenham um papel essencial ao lidar com as diferenças entre linguagens. Por exemplo, o vocabulário necessário para modelar um sistema depende de seu domínio. Uma linguagem pode então requerer novos elementos para aumentar o seu poder de expressão para domínios diferentes (CAPLAT; SOURROUILLE, 2005).

No caso específico da UML, o OMG define duas estratégias para estender a linguagem. A primeira, conhecida como extensão *leve* (*lightweight*), utiliza os *profiles*. Nesta estratégia, não é possível alterar a semântica dos elementos do metamodelo (apenas refiná-la), tão pouco mudar sua estrutura ou incluir novos elementos no metamodelo (PÉREZ-MARTINEZ, 2003).

A segunda estratégia, também conhecida como extensão *pesada* (*heavyweight*), atua na extensão do metamodelo da UML, explicitamente adicionando novas metaclasses e metaconstruções. A diferença entre as extensões *leve* e *pesada* vem da existência de restrições na forma com que os *profiles* podem estender o metamodelo da UML. As restrições da extensão leve impõem que qualquer extensão definida por um *profile* deva ser puramente aditiva, isto é, as extensões não podem conflitar como a semântica padrão. Estas restrições não se aplicam as extensões pesadas (PÉREZ-MARTINEZ, 2003).

Pode-se considerar a mesma situação para linguagens de programação. Extensões pesadas poderiam ser obtidas através da alteração das definições da linguagem, exigindo uma nova versão da linguagem e dos compiladores (como a que ocorreu

com a mudança de versão do VB6 para VB.Net, quando da inclusão da herança). Mudanças menos drásticas podem ser obtidas com mecanismos de extensão, como o VBX (Visual Basic eXtensions), disponível em algumas versões Visual Basic pré-.Net.

Apesar das alterações pesadas darem mais flexibilidade para a extensão de uma linguagem, elas têm um grande ponto negativo: a alteração do metamodelo exige uma nova versão da linguagem. No caso de linguagens de programação, isto implica em novos compiladores. No caso UML, isto impede a compatibilidade entre as diferentes ferramentas CASE para UML.

Para o mapeamento de modelos, a extensão de uma linguagem representa tanto um ônus quanto uma vantagem. Para apresentar estes dois pontos, utilizar-se-á a seguinte notação, sugerida em (CAPLAT; SOURROUILLE, 2005):

- $m(s)/l_1$, denota um modelo *m* de um sistema *s*, na linguagem *l*
- $m_1(s)/l_1 \rightarrow m_2(s)/l_2$ representa a transformação do modelo m_1, na linguagem l_1, para o modelo m_2, na linguagem l_2.

Isto posto, pode-se dizer que a extensão é um ônus porque a adição da extensão l_{ext} a l, acarreta a precisa definição da semântica de l_{ext}, para permitir $m_1(s)/(l_1 + l_{ext}) \rightarrow m_2(s)/l_2$. Por outro lado, é uma vantagem porque aumenta o poder de expressão da linguagem. Considere-se um *profile* UML P_{li-l2}, que defina noções de l_1 que não existam em l_2. A linguagem $l_2 + P_{l1-l2}$ teria o poder de expressão de l_1 e o mapeamento $m_1(s)/l_1 \rightarrow m_2(s)/(l_2 + P_{l1-l2})$ seria direto (CAPLAT; SOURROUILLE, 2005).

Os casos gerais para transformação de modelos entre linguagens diferentes podem ser vistos na figura 11:

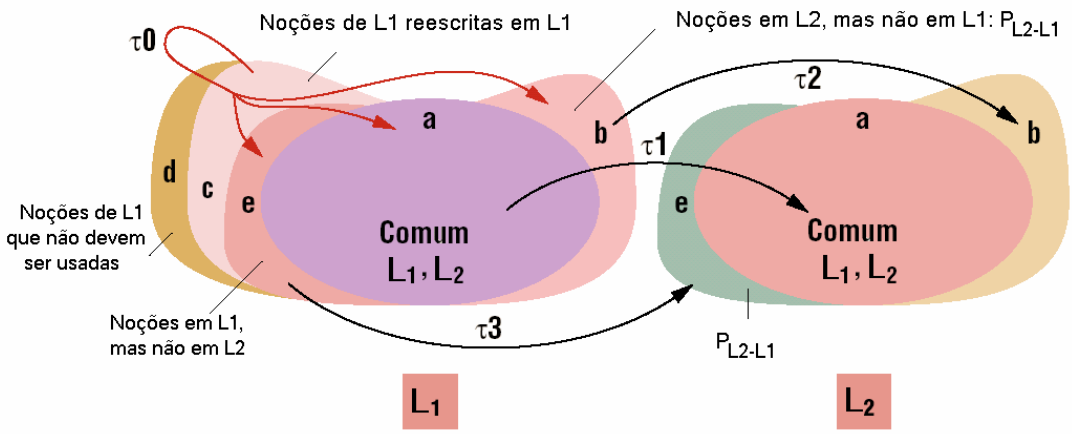

Figura 11 - Transformação de modelos entre linguagens diferentes (CAPLAT; SOURROUILLE, 2005)

- A transformação $\tau 1$ transforma noções comuns a l_1 e l_2. Nenhuma extensão de linguagem é necessária (CAPLAT; SOURROUILLE, 2005). Este é o caso de uma transformação de classes UML para classes em Java.
- A transformação $\tau 2$ utiliza noções de l_2 que não existem em l_1. Neste caso, a tradução requer uma extensão de l_1 para capturar as noções de l_2 (CAPLAT; SOURROUILLE, 2005).. Por exemplo: estender a UML com o *UML Profile for Java and EJB*, para representar melhor os conceitos da plataforma Java e com isso tornar as transformações mais diretas.
- Para as noções que existem em l_1, mas não em l_2 (*c*, *d* e *e* na figura 11), há três situações diferentes:
    - A primeira situação ($\tau 0$) é a reescrita das noções de l_1 em l_1. Por exemplo, a transformação de um relacionamento n-ário em n relacionamentos binários, preservando a semântica (CAPLAT; SOURROUILLE, 2005).;
    - A segunda situação (*d*) é quando a reescrita não é possível porque não há como representar o conceito em l_2. Neste caso, tais noções de l_1 não devem ser utilizadas. Por exemplo, a UML permite heranças

múltiplas, mas o Java ou o .Net não (CAPLAT; SOURROUILLE, 2005).

- A terceira situação (*e*), é a definição de uma extensão em l_2 que inclua as noções existentes em l_1 e a realização da tradução em função desta extensão ($\tau 3$). Por exemplo, há noções na UML e que não existem no C++ como mensagens assíncronas. O C++ poderia ser estendido através de uma biblioteca para permitir tal noção (CAPLAT; SOURROUILLE, 2005).

O uso de extensões também é conveniente para as transformações intra-linguagem. Por exemplo, a técnica de marcar o PIM requer um *profile* com valores marcados (*tagged values*) para representar conceitos do PSM.

3.6 Engenharia direta e reversa

A engenharia reversa pode ser conseguida através da bidirecionalidade nas transformações MDA. Esta é uma característica desejável, mas difícil de ser conseguida. A bidirecionalidade das transformações está representada na figura 12:

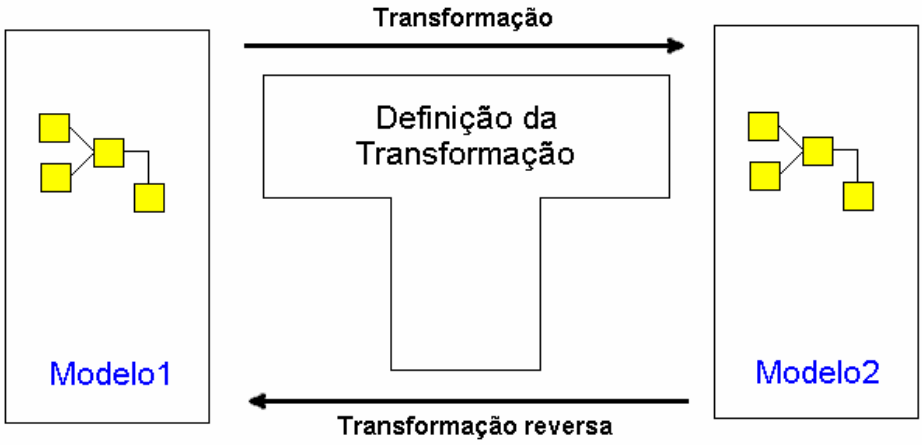

Figura 12 - Transformação bidirecional (KLEPPE;WARMER;BAST, 2003)

Há duas formas de obter-se a bidirecionalidade das transformações:

- Ambas as transformações poderem ser aplicadas de acordo com uma única definição de transformação.

Por conta das diferenças entre as linguagens fonte e alvo, é muito difícil construir uma definição de transformação que funcione dos dois lados. Por exemplo, na transformação de uma máquina de estados de um modelo de negócios em código Java, um estado pode ser transformado em uma variável booleana. Geralmente não é possível gerar a máquina de estados a partir do código Java. Não há como - a partir do código Java - descobrir quais atributos deveriam ser estados no modelo de negócio (KLEPPE;WARMER;BAST, 2003).

- Especificar duas definições de transformações de tal forma que uma seja inversa da outra.

  É muito difícil ter certeza que as transformações sejam inversas uma da outra. Por exemplo, se a primeira transformação realiza uma conversão de atributos públicos para atributos privados, não se pode simplesmente criar uma segunda transformação que os converta de privados para públicos. Se houvesse atributos privados no modelo-fonte, todos seriam convertidos para públicos na transformação reversa. Não haveria como distinguir os dois tipos a partir do modelo-alvo (KLEPPE;WARMER;BAST, 2003).

Resumidamente, apesar de desejável, é muito difícil implementar a engenharia reversa no MDA.

3.7 Considerações finais do capítulo 3

Pelo exposto neste capítulo, percebe-se que grandes empresas fornecedoras de ferramentas de software, como IBM e Microsoft, estão envolvidas com o aperfeiçoamento do MDD, o que sugere que esta forma de desenvolvimento de software pode se popularizar nos próximos anos.

Também pelo exposto, percebe-se que qualquer abordagem orientada a modelos aplica transformações inter e intralinguagens. Para facilitar estas transformações, extensões de linguagens são imprescindíveis.

4 IMPLEMENTAÇÃO DAS TRANSFORMAÇÕES

Preferencialmente PIMs e PSMs são escritos em UML, logo é importante que haja meios para realizar as transformações de modelos descritos em UML. Também é importante que haja meios para realizar transformação de UML para texto, como forma de gerar código-fonte.

Este capítulo apresenta as maneiras pelas quais é possível implementar as transformações UML para UML e UML para código-fonte. O capítulo apresenta uma classificação de três diferentes abordagens para operacionalizar as transformações, apresentando resultados práticos de duas destas abordagens, implementadas com recursos tecnológicos disponíveis atualmente. Também são feitos comentários sobre o QVT (*Query / Views / Transformations*), a nova e inacabada iniciativa do OMG que promete fornecer uma linguagem de transformações padronizada para a comunidade MDA.

4.1 Acesso aos modelos para implementar as transformações

Para que uma ferramenta de transformação possa implementar as transformações UML para UML ou UML para código-fonte, ela precisa ter acesso aos elementos do modelo do sistema. Precisa ser capaz de navegar pelos vários elementos que compõem o modelo, incluindo novos elementos ou excluindo elementos que existam. Ser capaz de atribuir valor aos elementos do modelo, se assim for preciso, bem como recuperar informações a partir do modelo para alimentar as transformações.

Sendall e Kozaczynski (SENDALL; KOZACZYNSKI, 2003) apresentam uma classificação composta de três abordagens diferentes para as implementar as transformações necessárias ao desenvolvimento orientado a modelos. São elas:

- Manipulação direta de modelos

  Acesso a uma representação interna de modelos e a habilidade de manipular a representação usando um conjunto de APIs procedurais das ferramentas de modelagem.

- Representação intermediária

  Exportar o modelo para um formato padronizado, tipicamente XMI, de tal forma que uma ferramenta externa possa transformá-lo.

- Suporte de uma linguagem de transformação

  Uma linguagem que forneça um conjunto de construções para explicitamente expressar, compor e aplicar transformações.

As três abordagens serão comentadas ao longo deste capítulo.

4.2 Manipulação direta de modelos

A manipulação direta de modelos é a primeira forma de manipulação de modelos abordada neste capítulo. Acessar as APIs (*Application Programming Interface*) de uma ferramenta CASE UML convencional pode ser a forma de se obter acesso aos elementos do modelo de um sistema e assim implementar as transformações MDA.

Este item será exemplificado com a ferramenta Rose da IBM-Rational.

Uma parte integrante do Rational Rose é a *Rational Rose Extensibility Interface* (REI), a qual permite que o usuário tenha acesso às APIs da ferramenta através de uma linguagem de *scripts* orientada a objetos, bastante semelhante ao *Visual Basic* pré .Net. A ferramenta fornece um editor destes *scripts*, conforme figura 13:

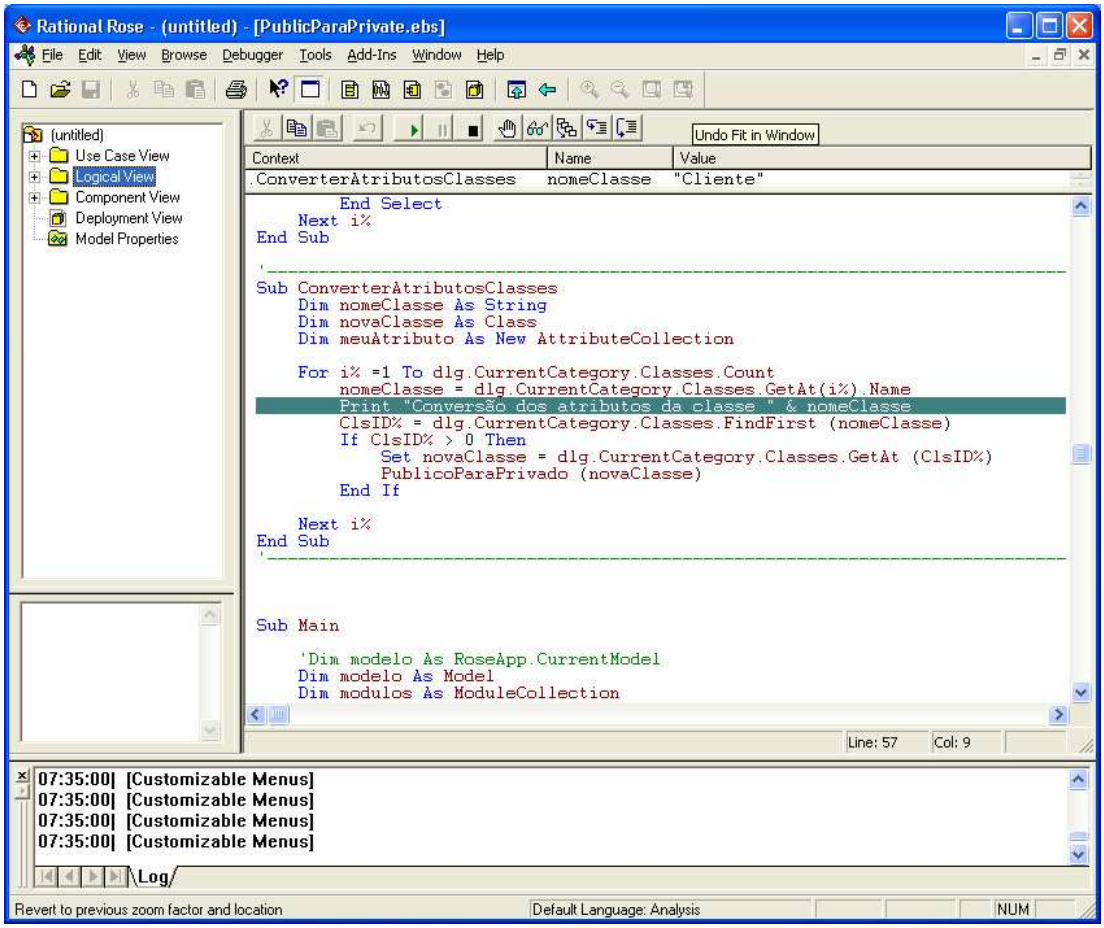

Figura 13 - Editor de *scripts* do Rational Rose

Tal linguagem é utilizada para estender e/ou alterar funcionalidades da ferramenta, bem como criar outras funcionalidades.

Um exemplo de extensão / alteração de funcionalidade está em (REED, 2000), onde é apresentada uma alteração do *script* DDLGEN.EBS, responsável por fazer a geração dos *scripts* de banco de dados. Esta alteração permitiu que a ferramenta tivesse maior flexibilidade na geração dos scripts DDL (*Data Definition Language*), considerando a classificação dos atributos de classe em persistentes ou transientes.

Esta linguagem de scripts pode ser utilizada para a implementação de transformações de modelos. O código da tabela 1 utiliza as APIs da ferramenta para criar uma transformação que atua sobre todos os atributos de todas as classes do modelo, convertendo-os de público para privado.

Tabela 1 - Script Rose para manipulação de atributos de *public* para *private*

```
'------------------------------
Type BrowserDialogType
      CurrentCategory As Category
      CurrentClass As Class
End Type
Private dlg As BrowserDialogType
'------------------------------
Sub SetCategoriaCorrente(Cat As Category)
      Set dlg.CurrentCategory = Cat
      Set dlg.CurrentClass = Nothing
End Sub
'------------------------------
Sub SetClasseCorrente (Class As Class)
      Set dlg.CurrentClass = Class
End Sub
'------------------------------
Sub SelecionarCategoria
      'Seleciona apenas a Logical View
      Dim theCategories As CategoryCollection
      Set theCategories = RoseApp.CurrentModel.GetAllCategories ()
      CatID% = theCategories.FindFirst ("Logical View")
      Set dlg.CurrentCategory = theCategories.GetAt (CatID%)
      SetCategoriaCorrente dlg.CurrentCategory
End Sub
'------------------------------
Sub PublicoParaPrivado (minhaClasse As Class)
      Dim meuAtributo As Attribute
      Dim mensagem As String

      For i% = 1 To minhaClasse.Attributes.Count
            Set meuAtributo = minhaClasse.Attributes.GetAt (i%)
            Select Case meuAtributo.ExportControl
                  Case rsPublicAccess
                        meuAtributo.ExportControl =
rsPrivateAccess
            End Select
      Next i%
End Sub
'------------------------------
Sub ConverterAtributosClasses
      Dim nomeClasse As String
      Dim novaClasse As Class
      Dim meuAtributo As New AttributeCollection
```

```
      For i% =1 To dlg.CurrentCategory.Classes.Count
           nomeClasse = dlg.CurrentCategory.Classes.GetAt(i%).Name
           Print "Conversão dos atributos da classe " & nomeClasse
           ClsID% = dlg.CurrentCategory.Classes.FindFirst
(nomeClasse)
           If ClsID% > 0 Then
            Set novaClasse = dlg.CurrentCategory.Classes.GetAt
(ClsID%)
            PublicoParaPrivado (novaClasse)
           End If
      Next i%
End Sub
'------------------------------
Sub Main

      Viewport.Open
      Print "Script para a conversão de atributos"
      Print "do tipo public para private"
      SelecionarCategoria
      ConverterAtributosClasses

End Sub
```

A linguagem de *scripts* do Rose é poderosa o bastante para permitir a geração de arquivos texto de uma forma idêntica ao *Visual Basic*. Sendo assim, é possível gerar o código-fonte a partir do modelo UML.

Como exemplo desta capacidade de geração de arquivos texto, no *Rose Enterprise Edition, Release Version* 2003.06.12.208.000, há um script chamado BROWSER.EBS, que gera código-fonte C++ para as classes do modelo UML. Este mesmo script cria telas para a participação interativa do usuário durante as transformações.

Os pontos fortes desta abordagem são:

- A possibilidade de manipulação do modelo de UML para UML, permitindo a implementação das regras de transformação entre PIM e PSM;
- A possibilidade de geração de arquivos texto permite a implementação das regras de transformação PSM para código-fonte;

- Todas as transformações podem ser realizadas dentro da ferramenta de modelagem, não sendo necessário exportar o modelo para que seja processado em outra ferramenta. Sendo assim, não é necessário utilizar padrões de intercâmbio de modelos, como o XMI.

Os pontos fracos desta abordagem são:

- O recurso de programação por *scripts* é proprietário e dependente da ferramenta. Regras de transformação escritas em uma ferramenta não podem ser transportadas para outra, mesmo que esta última tenha também uma linguagem de programação que dê acesso às suas APIs;
- Não se deve considerar que o fornecedor esteja preocupado em manter a compatibilidade dos scripts nas novas versões. O exemplo de extensão de funcionalidade para a geração de scripts DDL usou o script DDLGEN.EBS, o qual estava disponível em uma versão anterior da ferramenta. A partir da versão 2000, este script não está mais disponível;
- O editor de scripts do Rose é funcional, mas está longe da comodidade do IDE (*Interface Development Environment*) do *Visual Basic* 6. Não há *IntelliSense* e nem o conceito de projeto;
- O *Help* da ferramenta é deficiente se comparado com o do *Visual Basic* 6;
- A linguagem não fornece abstrações de um nível adequado para a especificação das transformações: atua-se sobre APIs da ferramenta CASE para manipular o modelo, quando o certo seria atuar sobre os próprios elementos do modelo UML.

4.3 Representação intermediária

Conforme mencionado no início deste capítulo, as transformações MDA atuam tanto na modificação do modelo (em transformações UML para UML) quanto na geração de código-fonte a partir deste modelo (arquivos texto que representam o código-fonte).

A representação intermediária é a segunda forma de manipulação de modelos abordada neste capítulo. Ela é exeqüível porque o OMG desenvolveu um formato

padronizado de intercâmbio de modelos chamado *XML Metadata Interchange* (XMI) (OMG, 2003a), permitindo que os fornecedores das ferramentas comerciais UML desenvolvam uma funcionalidade universal de exportação de modelos. Isto tornou possível o acesso ao modelo UML fora das ferramentas de modelagem.

O XMI é baseado no formato XML, sendo uma estrutura de armazenamento em árvores. Um exemplo de modelo UML convertido em XMI está no apêndice B desta dissertação.

4.3.1 Atuação baseada em XMI

Como o arquivo XMI é um arquivo texto, a princípio qualquer linguagem que pudesse manipular arquivos texto seria capaz de implementar as transformações UML para UML ou UML para código-fonte. Entretanto essa é uma simplificação excessiva. Linguagens de uso geral poderiam ser utilizadas, mas o desenvolvimento prévio de funcionalidades para a manipulação XML é imprescindível. Por exemplo, não é possível manipular XML sem operações para localização de início e final de *tags* ou consistências para verificação de arquivo de entrada e de arquivo de saída bem-formados. A navegação pelos nós da árvore XML também é necessária.

Algumas linguagens de uso geral, como o Java ou o *Visual Basic.Net* suportam APIs que dão acesso a funcionalidades para manipulação de documentos XML.

No caso específico do Java há duas principais APIs padronizadas para esta finalidade: a SAX (*Simple API for Java*) e DOM (*Document Object Model*), além de outras menos divulgadas, como JDOM, dom4j, ElectricXML e XMLPULL (HUNTER, 2002).

As versões Java 1.4 ou superiores apresentam o JAXP (*Java API for XML*), uma API que não fornece funcionalidades de *parsing* ou quaisquer outras necessárias para manipular XML, porém torna mais fácil lidar com o DOM e SAX (HUNTER, 2002), (MCLAUGHLIN, 2000).

O Framework .Net também tem uma biblioteca de classes (`System.Xml`) que fornece suporte para processamento de XML.

4.3.2 Linguagens de *query* para XML

Entretanto, por conta da importância do XML como padrão de representação e intercâmbio de dados na Internet, linguagens para extração e reestruturação de conteúdo XML foram propostas. Em (BONIFATI; CERI, 2000) há um comparativo de cinco destas linguagens de *query* embutidas, a saber: LOREL, XML-QL, XML-GL, XSL (XSLT) e XQL.

Destas linguagens, o XSLT (*XSL Transformations*) é a mais conhecida. É uma recomendação W3C (W3C, 1999a), apresentada como uma linguagem para a transformação de documentos XML.

A XSLT é uma linguagem interpretada (apesar da Sun Microsytems ter proposto um compilador), que faz uso de um processador XSLT para transformar documentos XML em documentos XML (default), HTML ou texto. É possível gerar outros formatos de saída se o processador utilizado assim o permitir (FUNG, 2001).

Nesta dissertação, a implementação de transformações através da representação intermediária, é testada com o uso de XSLT. Os exemplos XSLT apresentados nesta dissertação foram executados com o processador XT de James Clark (Clark foi o editor da versão 1.0 do XSLT. O processador está disponível em http://www.jclark.com/xml/xt-old.html).

O exemplo da tabela 2 é um fragmento de programa XSLT para manipulação de um modelo UML, representado na forma de um documento XMI.

Tabela 2 - Fragmento de código XSLT

```
<?xml version='1.0'?>
<xsl:stylesheet
   xmlns:xsl='http://www.w3.org/1999/XSL/Transform' version='1.0'
   xmlns:UML='href://org.omg/UML/1.3'>

 <xsl:output method='xml' indent='yes'/>

 <xsl:template match='/'>
          <xsl:apply-templates/>
 </xsl:template>
```

```
 ....

<xsl:template match='UML:Class'>
  <UML:Class>
      <xsl:attribute name='xmi.id'>
          <xsl:value-of select='@xmi.id'/>
      </xsl:attribute>
      <xsl:attribute name='name'>
          <xsl:value-of select='@name'/>
      </xsl:attribute>
      <xsl:attribute name='visibility'>
          <xsl:value-of select='@visibility'/>
      </xsl:attribute>
      <xsl:attribute name='isSpecification'>
          <xsl:value-of select='@isSpecification'/>
      </xsl:attribute>
      <xsl:attribute name='isRoot'>
          <xsl:value-of select='@isRoot'/>
      </xsl:attribute>
      <xsl:attribute name='isLeaf'>
          <xsl:value-of select='@isLeaf'/>
      </xsl:attribute>
      <xsl:attribute name='isAbstract'>
          <xsl:value-of select='@isAbstract'/>
      </xsl:attribute>
      <xsl:attribute name='isActive'>
          <xsl:value-of select='@isActive'/>
      </xsl:attribute>
      <xsl:attribute name='namespace'>
          <xsl:value-of select='@namespace'/>
      </xsl:attribute>
      <xsl:apply-templates select='node()'/>
  </UML:Class>
</xsl:template>

<xsl:template match='UML:Attribute'>
  <UML:Attribute>
      <xsl:attribute name='xmi.id'>
          <xsl:value-of select='@xmi.id'/>
      </xsl:attribute>
      <xsl:attribute name='name'>
          <xsl:value-of select='@name'/>
      </xsl:attribute>
      <xsl:attribute name='visibility'>private</xsl:attribute>
      <xsl:attribute name='isSpecification'>
          <xsl:value-of select='@isSpecification'/>
```

```
        </xsl:attribute>
        <xsl:attribute name='ownerScope'>
            <xsl:value-of select='@ownerScope'/>
        </xsl:attribute>
        <xsl:attribute name='changeability'>
            <xsl:value-of select='@changeability'/>
        </xsl:attribute>
        <xsl:attribute name='targetScope'>
            <xsl:value-of select='@targetScope'/>
        </xsl:attribute>
        <xsl:attribute name='type'>
            <xsl:value-of select='@type'/>
        </xsl:attribute>
        <xsl:apply-templates select='node()'/>
    </UML:Attribute>
 </xsl:template>

.....

</xsl:stylesheet>
```

Há dois pontos em destaque no código:

- O primeiro (xsl:output method = ‘xml’) indica que o resultado deste programa XSLT será um documento XML.
- O segundo (*private* para *visibility*, dentro do gabarito UML:Attribute), indica que todos os atributos de classes deste modelo terão visibilidade *private*, independentemente do que tenha sido especificado no modelo UML original.

4.3.2.1 O XPATH

O XSLT também contempla a especificação XPath, uma recomendação W3C que define uma “linguagem para endereçar partes de um documento XML, e que foi projetada para ser usada tanto pelo XSLT quanto pelo XPointer” (W3C, 1999b). O XPath permite que o XSLT tenha a habilidade de navegar pelos nós das árvores dos arquivos XMI.

Para endereçar os nós das árvores de uma arquivo XML, o XPath permite 13 diferentes eixos (FUNG, 2001), (W3C, 1999b). Por exemplo, se a partir do nó de contexto (nó selecionado), for necessário selecionar os nós filhos, utiliza-se o eixo-filho do nó de contexto (`child::nó`), conforme figura 14:

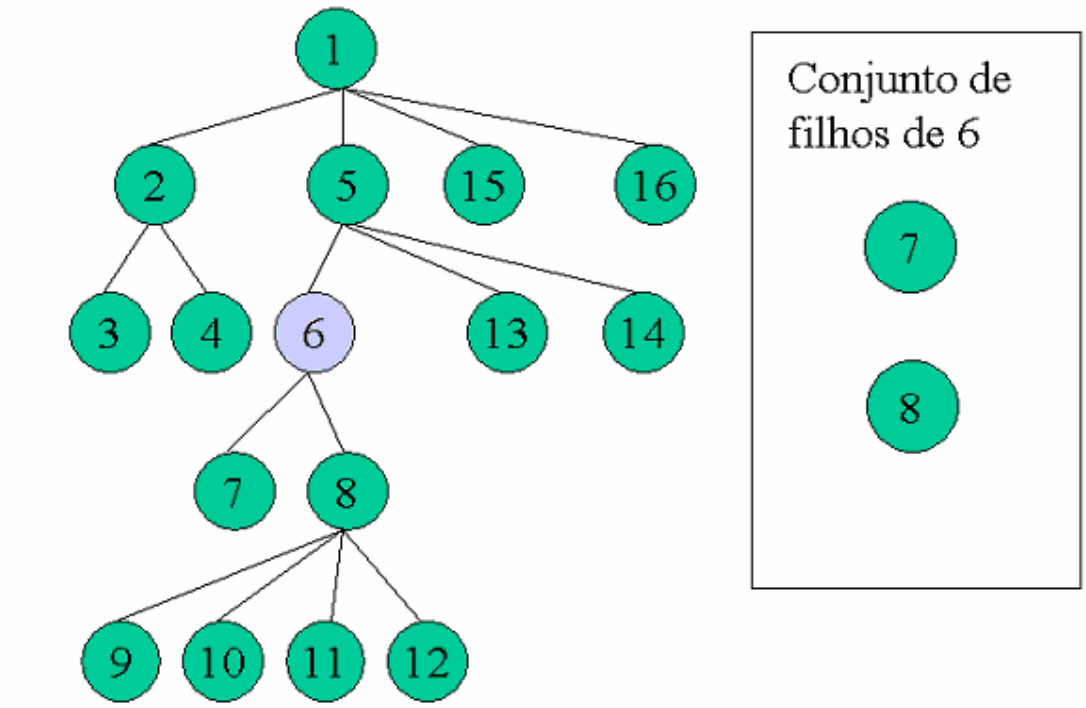

Figura 14 - Exemplo de utilização de eixo no XPath

4.3.2.2 Geração de arquivos texto com XSLT

O XSLT também pode ser utilizado para a geração de arquivos texto, possibilitando a geração de código-fonte a partir de um modelo PSM.

O exemplo da tabela 3 gera o texto de classes Java, com seus atributos, a partir de um documento XMI:

Tabela 3 - Programa XSLT para geração de código-fonte em JAVA

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xsl:stylesheet
    xmlns:xsl='http://www.w3.org/1999/XSL/Transform' version='1.0'
    xmlns:UML='href://org.omg/UML/1.3'>

<xsl:output method='html'  indent='no'/>
```

```
<xsl:preserve-space elements='UML:Attribute'/>

<xsl:template match='/'>
    <xsl:apply-templates/>
</xsl:template>

<xsl:template match='XMI.header'>
</xsl:template>

<xsl:template match='UML:Namespace.ownedElement'>
<xsl:apply-templates/>
</xsl:template>

<xsl:template match='UML:Class'>
/*  gerador automatico
** de codigo Java
*/
package com.java.teste;
public class  <xsl:value-of select='@name'/> {
   <xsl:apply-templates select='node()'/>
}
</xsl:template>

<xsl:template match='UML:Attribute'>
   <xsl:value-of select='@visibility'/>
   <!-- inclusão de espaço em branco -->
   <xsl:text>&#x20;</xsl:text>
   <xsl:call-template name='tipoDeDado'>
       <xsl:with-param name='type' select='@type'/>
   </xsl:call-template>
   <!-- inclusão de espaço em branco -->
   <xsl:text>&#x20;</xsl:text>
   <xsl:value-of select='@name'/>;
</xsl:template>

<xsl:template name='tipoDeDado'>
   <xsl:param name='type'/>
   <xsl:for-each select='ancestor::XMI/XMI.content/UML:Model
   /UML:Namespace.ownedElement/UML:DataType'>
      <xsl:if test="@xmi.id=$type">
         <xsl:value-of select='@name'/>
      </xsl:if>
   </xsl:for-each>
</xsl:template>

</xsl:stylesheet>
```

Pode-se observar que o método de saída (xsl:output method) deste programa foi definido como sendo html, quando o aparentemente certo seria modo texto. Com o uso do processador XT, a saída modo texto de um programa XSLT gerou uma formatação pior do que a saída modo HTML.

4.3.2.3 Peculiaridades do XSLT

O XSLT tem duas peculiaridades que tornam o seu aprendizado mais difícil, a saber:

- Linguagem de gabaritos (*template*)

  O XSLT não é uma linguagem procedural convencional: é formada por um conjunto de regras de gabarito (*template*) que definem a saída produzida em função da identificação de padrões na entrada (KAY, 2002).

- Tratamento de Variáveis

  O XSLT permite a definição de variáveis globais ou locais de uma forma parecida que numa linguagem procedural, mas uma vez que um valor tenha sido atribuído a uma variável, ele não poderá ser alterado (KAY, 2002).

Esta última característica interfere na maneira de se codificar programas em XSLT. A linguagem não tem instruções de atribuição porque se baseia nos conceitos da programação funcional.

A programação funcional é um caso particular do paradigma de programação declarativa e sua forma de codificação difere das linguagens imperativas, mais comuns entre nós.

As linguagens imperativas são caracterizadas por terem um estado implícito, o que cria nestas linguagens a noção de seqüência para permitir um controle preciso e determinístico sobre este estado. Dentro deste conceito, as instruções de atribuição são comuns, uma vez que seu efeito é alterar o armazenamento implícito, resultando em diferentes atribuições para uma variável (HUDAK, 1989).

Ao contrário das linguagens imperativas, “as linguagens declarativas não contêm um estado implícito e a ênfase é colocada inteiramente na programação com expressões. Em particular, linguagens funcionais são linguagens declarativas cujo modelo fundamental de computação é a Função, em contraste com a Relação, que forma a base para as linguagens de programação em lógica” (HUDAK, 1989).

Não existem instruções de atribuição no XSLT para que esta linguagem não tenha o caráter seqüencial das linguagens imperativas (KAY, 2002).

Pontos fortes da abordagem:

- Não é necessário testar se o documento XML de entrada é bem-formado. Um processador como o XT faz tal consistência. Da mesma forma, a menos que o implementador opte por controlar manualmente os caracteres de saída (isso se faz usando o comando `xsl:text disable-output-escaping ="yes"`), um documento XML de saída também será bem formado.
- Há abstrações (como `apply templates`) que facilitam a manipulação dos arquivos XMI.
- Não é difícil encontrar referências bibliográficas sobre XSLT por conta de sua importância em desenvolvimento *web;*
- Permite não somente a conversão UML para UML (XML para XML) como também a conversão UML para código (UML para texto), uma vez que gera arquivo texto.

Pontos fracos da abordagem:

- Longo tempo de aprendizado por conta das peculiaridades da linguagem;
- O processamento é batch e não permite a intervenção do arquiteto durante as transformações;
- Pode ser utilizada para gerar código-fonte (PSM – Código), mas com uma limitação: irá gerar todo o código das classes em um único arquivo (um única transformação) e será necessário particionar este arquivo (possivelmente usando uma linguagem de uso geral).

4.4 Suporte de uma linguagem de transformação

Segundo (SENDALL; KOZACZYNSKI, 2003), “a chave para se projetar uma linguagem de transformações é oferecer abstrações de transformações que sejam intuitivas e que cubram a maior parte possível de situações e, por esta razão, um comprometimento entre a típica exatidão de uma linguagem declarativa e a compreensão”.

Ainda segundo estes autores, as seguintes características seriam desejáveis para uma linguagem de transformação:

- Pré-condições

  Descrever as condições sob as quais as transformações devam ser implementadas.

- Composição

  Permitir a composição de transformações existentes para construir novas transformações, porque é normalmente mais fácil compor componentes do que construir novos a partir de partes básicas.

- Forma

  A acessibilidade e aceitação de uma linguagem dependem de sua forma. Um dos apelos da UML é a sua notação gráfica.

- Usabilidade

  A usabilidade de uma linguagem de transformação é fortemente afetada se a linguagem é declarativa ou imperativa. Uma abordagem declarativa torna a linguagem mais concisa, simplificando a descrição das regras de transformação. A abordagem imperativa oferece um paradigma mais familiar para compor regras de transformação, ou seja, seqüência, seleção e iteração. Uma linguagem que misture estas duas abordagens pode apresentar vantagens adicionais.

4.4.1 Linguagens imperativas e declarativas

(SENDALL; KOZACZYNSKI, 2003) comentam sobre as vantagens de uma linguagem de transformação que tenha características declarativas e imperativas. Tais categorias de linguagens representam a especificação de um programa de forma diferente:

- As linguagens imperativas dizem ao computador como atingir uma meta através de uma seqüência de instruções. Exemplos deste tipo de linguagem são: Cobol, Fortran, Basic, etc...
- As linguagens declarativas, por sua vez, dizem ao computador qual meta deve ser atingida, e o fazem através de um conjunto de especificações. Exemplos deste tipo de linguagem são: Prolog, LISP, SQL, etc...(BURKS, 2001).

O exemplo da tabela 4 mostra a implementação de uma rotina de cálculo de fatorial em Prolog (declarativa) (Coenen, 1999)

Tabela 4 - Cálculo de fatorial em Prolog

```
factorial(0,1):-
        !.

factorial(N1,T2):-
        N2 is N1-1,
        factorial(N2,T1),
        T2 is N1*T1.
```

O exemplo da tabela 5 mostra a implementação de uma rotina de cálculo de fatorial em *Visual Basic* 6 (imperativa).

Tabela 5 - Cálculo de fatorial em Visual Basic 6

```
Function Fatorial(numero As Integer) As Long

    Dim fat, contador As Integer
    fat = 1
```

```
    For contador = 1 To numero
        fat = fat * contador
    Next
    Fatorial = fat
End Function
```

Vale notar que a ordem de execução das sentenças não altera o resultado de execução de uma linguagem declarativa, mas afeta o de uma linguagem imperativa.

Muitas linguagens contêm funcionalidades declarativas e imperativas simultaneamente (Prolog e LISP contém funcionalidades imperativas para E/S (BURKS, 2001)). O melhor que se pode fazer em muitos casos e dizer que uma linguagem é principalmente declarativa ou principalmente imperativa (GREENFIELD *et al*, 2004).

4.4.2 Query, Views, Transformations (QVT)

As definições de transformações usadas no framework MDA são atualmente definidas de uma forma não padronizada. Para permitir a padronização destas definições de transformações, o OMG está atualmente trabalhando em uma linguagem padrão para escrever definições de transformações. Este padrão é chamado QVT (*Query, Views, Transformations*) (KLEPPE;WARMER;BAST, 2003).

Em 2002, o OMG lançou um RFP (*Request for Proposal*) para receber propostas para o QVT (OMG, 2002). Segundo este RFP, alguns dos requisitos obrigatórios para o padrão são:

- As propostas devem definir uma linguagem para selecionar modelos. Esta deve facilitar seleções *ad-hoc* e filtragem para modelos, assim como a seleção de modelos que sejam origem das transformações;
- As propostas devem definir uma linguagem para a definição das transformações. As definições das transformações devem descrever relacionamentos entre um metamodelo MOF origem e um metamodelo MOF destino. O metamodelo origem e destino podem ser o mesmo;

- A sintaxe abstrata para definição das linguagens de transformação, visualização e seleção deve ser definida com o metamodelo MOF;
- A linguagem de definição de transformação deve ser capaz de expressar toda a informação requerida para gerar automaticamente um modelo alvo a partir de um modelo fonte.

Este padrão continuava em desenvolvimento durante o preparo desta dissertação. A falta de uma linguagem padronizada para a representação das transformações, bem como a falta de um processador para tal linguagem, obriga a utilização de outros meios para a aplicação prática dos conceitos do MDA. Uma opção é utilizar linguagens de transformação proprietárias de ferramentas CASE MDA disponíveis atualmente. Um exemplo é a linguagem de transformação do OptimalJ da Compuware.

Em (KLEPPE;WARMER;BAST, 2003), os autores propõem uma linguagem para definir as regras de transformação e afirmam que cada regra deve conter as seguintes informações:

- A linguagem do modelo-fonte e do modelo-alvo;
- Um parâmetro opcional para, por exemplo, definir constantes que possam ser utilizadas na transformação;
- Elementos do metamodelo no modelo-fonte;
- Elementos do metamodelo no modelo-alvo;
- Um indicador bidirecional para indicar se um modelo-fonte pode ser (re)gerado a partir do modelo-alvo;
- Condição expressa na linguagem-fonte para determinar se a transformação deve ser aplicada ou não;
- Condição expressa na linguagem-alvo para determinar as condições que devem existir no modelo-alvo para que a transformação seja aplicada. Alternativamente indica quais condições devem existir no modelo-alvo após a aplicação da transformação;
- Um conjunto de mapeamentos;

4.5 Considerações finais do capítulo 4

Este capítulo apresenta uma classificação com três formas diferentes de implementação das transformações de UML para UML e de UML para código-fonte. Mesmo inexistindo uma linguagem padronizada e aberta para descrição das transformações, é possível implementá-las com os recursos tecnológicos atuais. Este capítulo apresenta os resultados de uma tentativa de implementação das transformações com a linguagem de *scripts* do Rational Rose e com a linguagem XSLT, um padrão W3C para manipulação de arquivos XML.

5 UTILIZAÇÃO DE SOLUÇÕES PRÉ-DEFINIDAS NO DESENVOLVIMENTO MDA

Conforme comentado no capítulo 3, é possível aumentar a produtividade no processo MDA se as transformações forem capazes de complementar automaticamente os modelos, desonerando o desenvolvedor dos detalhes de implementação.

O objetivo deste capítulo é propor que ferramentas CASE MDA devam contemplar uma estratégia pré-estabelecida para a construção de software. As ferramentas devem conter um conjunto de transformações que contemplem a aplicação de soluções definidas de antemão e em conformidade com a arquitetura mencionada, de tal forma que a ferramenta possa fornecer soluções para problemas recorrentes com o mínimo de esforço de modelagem por parte de seu usuário.

5.1 Soluções para a redução de esforços de modelagem

Segundo (BELIX *et al*, 2005), ferramentas de transformação MDA devem ser feitas considerando-se que diferentes aplicações têm pontos em comum e, portanto, surgem duas considerações:

- É vantajoso para o desenvolvedor que as ferramentas CASE forneçam soluções que possam auxiliá-lo a resolver estes pontos em comum;
- Sendo que algumas destas soluções podem estabelecer partes do *design* do software, cabe ao desenvolvedor fazer a modelagem para apenas o que efetivamente necessite de definições de *design*.

5.2 Classificação para soluções pré-definidas em MDA/MDD

Conscientemente ou não, um conjunto de decisões sempre é tomado no desenvolvimento de um software. Estas decisões podem tratar de questões gerais da organização de um software, como a escolha de um padrão de arquitetura; ou de questões mais específicas, como persistência de dados.

Estas diferentes decisões levam a soluções com características distintas no que se refere à implementação de um *software*. Ênfase na facilidade de manutenção, mesmo em detrimento do desempenho, é um fator que pode contribuir para que uma equipe de desenvolvimento opte pelo padrão de arquitetura em camadas (MENDES, 2002). Um padrão de arquitetura "descreve um problema de design particular e recorrente que surge em contextos específicos e apresenta um esquema genérico e provado para sua solução" (BUSCHMANN *et al*, 1996). O padrão fornece uma solução de um problema, mas não fornece nenhuma implementação da solução.

Decisões podem resultar em soluções acompanhadas da respectiva implementação, completa ou não. Decisões sobre *front-end web* de uma aplicação podem levar à adoção de uma solução como o *framework* STRUTS, que fornece *design* e porções de código; é uma solução que tem implementação parcial, que precisa ser complementada.

(GREENFIELD *et al*, 2004) apresenta uma taxonomia para classificar esta questão de quantidade de implementação fornecida por uma solução:

- Solução de Caixa Branca

  Fornece a descrição de uma solução, mas não a sua implementação. Pode ser aplicada em vários contextos diferentes.

  Padrões (*patterns*) são exemplos de soluções de caixa branca.

- Solução de Caixa Preta

  Fornece a completa implementação de uma solução, transparente para o usuário.

  Web services, componentes ou funcionalidades (*features*) fornecidas por linguagens de programação são exemplos de soluções de caixa preta.

- Solução de Caixa Cinza

  Representa o ponto intermediário: fornece código-fonte e *design* para uma implementação parcial ou completa de uma solução que pode ser completada ou modificada se necessário.

  Frameworks são exemplos de soluções de caixa cinza.

A principal contribuição deste trabalho é aproveitar a taxonomia apresentada em (GREENFIELD *et al*, 2004) e aplicá-la explicitamente ao desenvolvimento MDA. São dois os resultados esperados: o primeiro é a reutilização de soluções pré-definidas, que podem ser utilizadas para a solução de problemas recorrentes ou para auxiliar o desenvolvedor. A aplicação destas soluções é feita de forma transparente ou pelo menos bastante facilitada para o desenvolvedor. O segundo – e mais importante – é derivado da idéia que estas soluções podem estabelecer partes do *design* do software, cabendo ao desenvolvedor fazer a modelagem para apenas o que efetivamente necessite de definições de *design*. Ou seja, possibilitam uma redução de esforços de modelagem por parte do desenvolvedor.

5.3 O impacto das soluções apresentadas sobre o esforço de modelagem

A taxonomia apresentada serve para classificar soluções em um espectro que varia de nenhuma implementação até uma implementação completa da solução. A aplicação destas soluções necessita de reflexão: soluções com implementação completa só podem ser utilizadas em um contexto que necessite exatamente daquela solução. A razão é que não se pode alterar design ou implementação deste tipo de solução. Soluções de caixa branca, por sua vez, permitem um grau maior de liberdade em sua configuração, adequando-se a um número maior de contextos. Por conta destas características, cada tipo de solução desempenha papéis diferentes na implementação de um sistema de software e na redução de esforços de modelagem no MDD.

5.3.1 Redução de esforços causada pelas soluções de caixa branca

As soluções caixa branca fornecem uma descrição da solução, mas não a sua respectiva implementação. O desenvolvedor fornece a implementação e pode configurar a solução com um bom grau de liberdade. Por tais características, as soluções de caixa brancas têm um grande escopo de aplicação.

Padrões *(Patterns)* de arquitetura são exemplos de soluções de Caixa Branca. Eles suportam a construção de um software com propriedades definidas. Apesar de fornecerem a estrutura básica para resolver um problema particular de *design*, eles não especificam uma solução completamente detalhada. Ao invés de fornecer um módulo pré-fabricado que deva ser usado da forma que se encontra, um padrão fornece um esquema de solução genérica de uma família de problemas (BUSCHMANN *et al*, 1996).

A economia de esforços de modelagem causada pelas soluções de caixa branca é conseguida pela automática aplicação delas por uma ferramenta CASE, obtendo-se assim a definição da estrutura do software sem a intervenção do usuário. Naturalmente considerações do usuário da ferramenta são necessárias para determinar se as pré-definidas soluções de caixa branca são adequadas ao software a ser construído.

Estando atreladas às definições arquiteturais de alto nível, as soluções de caixa branca restringem muito as possíveis soluções de caixa preta ou cinza que podem vir a ser empregadas.

5.3.2 Redução de esforços causada pelas soluções de caixa preta

Fornece a total implementação da solução e seu funcionamento é transparente para o usuário. Ao fornecer a implementação completa da solução, tal solução torna-se inflexível e pode ser aplicada apenas aos problemas que requeiram exatamente essa solução. Seu escopo de aplicação é reduzido em comparação com as soluções de Caixa Branca (GREENFIELD *et al*, 2004).

Em contrapartida seu uso dispensa esforços de modelagem por parte de quem a utiliza, uma vez que toda a estrutura e comportamento da solução já estão definidos.

Soluções de caixa preta podem ser utilizadas para solução de regras de negócio ou solução de questões técnicas, sendo esta última dentro de um contexto arquitetural específico:

- Soluções para regras de negócio

Enquanto modelam software, arquitetos podem querer usar um componente compilado existente e que atenda requisitos do projeto como, por exemplo, um componente para cálculo de ativos financeiros. Neste caso não é necessário modelar o componente (ele já existe), mas apenas modelar o software para utilizar este componente.

Existe uma grande diversidade de definições para componente de software Neste trabalho, considera-se componente como sendo um elemento compilado de software e que respeite a definição do RUP, ou seja, ser "uma parte não trivial, substituível e quase independente de um sistema e que preencha uma clara função no contexto de uma bem definida arquitetura" (IBM-RATIONAL, 2003).

Alguns problemas de plataforma podem ocorrer nesta tentativa de reuso. Por exemplo, durante o desenvolvimento de um projeto com tecnologia .Net, pode existir um componente que preencha exatamente um particular serviço, mas que esteja escrito em CORBA. Neste caso, não se pode diretamente reutilizar tal componente. É necessário construir um apropriado código (*wrapper*) para adaptar o componente ao ambiente real (BILLIG *et al*, 2004). A ferramenta CASE poderia fornecer este código.

- Soluções para questões técnicas

  Dentro do espírito de fornecer o máximo de soluções prontas para o usuário de uma ferramenta CASE MDA, e também focando na redução de esforços na modelagem de software, a ferramenta pode fornecer soluções para resolver questões técnicas pertinentes a uma estratégia de desenvolvimento específica. Por exemplo, classes com persistência baseada em DAO (*Data Acess Object*) requerem um componente que forneça conexão com o banco de dados e execução de sentenças SQL. Tal componente poderia ser fornecido e automaticamente aplicado ao software pela ferramenta CASE.

  Uma ferramenta CASE também pode aplicar funcionalidades fornecidas por linguagens de programação. Por exemplo, *recordset* desconectado (ADO em projetos .Net e *CachedRowSet* em projetos Java) é uma funcionalidade fornecida por linguagens de programação para tratar persistência de dados.

> Sem a necessidade de implementação adicional, ele também pode ser utilizado como um meio para o transporte de dados entre camadas ou para prevenir problemas causados por multi-acesso descontrolado. Entretanto somente é possível tirar vantagem destas duas últimas características se a arquitetura da aplicação assim o permitir. Esta restrição é causada pelo já mencionado limitado escopo das soluções de caixa preta.

O ponto em questão é que o uso de soluções de caixa preta gera economia de esforços de modelagem porque não é necessário modelar as soluções - elas já existem de antemão. Mais importante, o uso de soluções de caixa preta representa uma economia de modelagem não apenas dos aspectos estruturais do software, mas também dos aspectos comportamentais.

O conceito de soluções de caixa preta também permite encapsular todo um design complexo, de tal forma que um projetista não precise se preocupar com seus detalhes: "...requerer que projetistas mantenham em mente *designs* complexos limitaria o que pode ser atingido e não é necessário fazê-lo se usarmos corretamente as soluções de caixa preta" (COLWELL, 2005).

No caso específico de resolução de regras de negócio, o usuário deve conhecer a solução de caixa preta (sua funcionalidade, não os detalhes de implementação) e deve conhecer o contexto no qual a solução funciona. Caso contrário essa solução não se adequará ou não produzirá os resultados esperados.

No caso de resolução de questões técnicas o cenário muda: a ferramenta deve automaticamente aplicar estas soluções baseada no contexto arquitetural, e de tal forma que o usuário sequer tenha que conhecer as soluções ou verificar se as escolhas da arquitetura estão adequadas. É axiomático que as decisões tomadas pela ferramenta devem ser convenientes para todos os tipos de soluções aplicadas.

5.3.3 Redução de esforços causada pelas soluções de caixa cinza

Sendo uma implementação parcial, as soluções de caixa cinza permitem alterações de código e design. *Framewoks* são exemplos de soluções de caixa cinza

Um *framework* é um *design* reutilizável de todo ou parte de um sistema. Ele é o esqueleto de uma aplicação e pode ser configurado pelo desenvolvedor da aplicação. Um *framework* fornece reuso de *design* e reuso de código (JOHNSON, 1997).

De forma similar às soluções de caixa preta, essa dissertação afirma que, baseada na estratégia pré-estabelecida de suas transformações, a ferramenta CASE pode automaticamente aplicar essas soluções de caixa cinza para domínios técnicos. A aplicação automática de um *framework*, além de representar uma oportunidade de redução nos esforços de modelagem, é uma maneira de evitar os custos da curva de aprendizagem deste *framework*. "*Frameworks* são mais configuráveis que a maior parte dos componentes e tem mais interfaces complexas. Programadores devem aprender estas interfaces antes que possam utilizar o *framework*" (JOHNSON, 1997).

Efetivamente, o usuário não precisa saber quais soluções de caixa cinza serão aplicadas pela ferramenta CASE para lidar com questões técnicas. Em certos contextos de arquitetura de software, é possível identificar funcionalidades recorrentes que podem ser fornecidas parcialmente, necessitando apenas configuração para a geração completa do código-fonte. Transformações podem ser desenvolvidas para automaticamente resolver estes pontos.

As figuras 15 e 16 apresentam métodos de consulta em duas classes de um mesmo sistema de software escrito em Visual Basic 6. O sistema implementa a persistência de dados baseado no padrão DAO (*Data Access Object*) e com uso de uma interface CRUD (*Create, Read, Update, Destroy*).

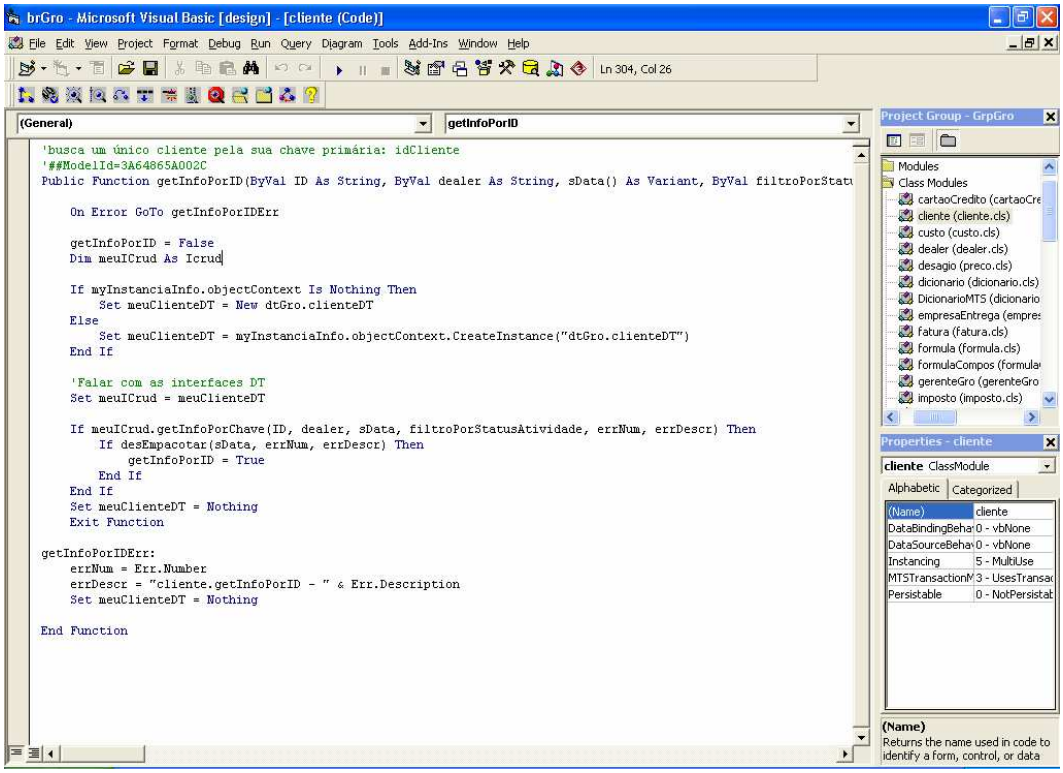

Figura 15- Implementação de método de consulta para a classe Cliente

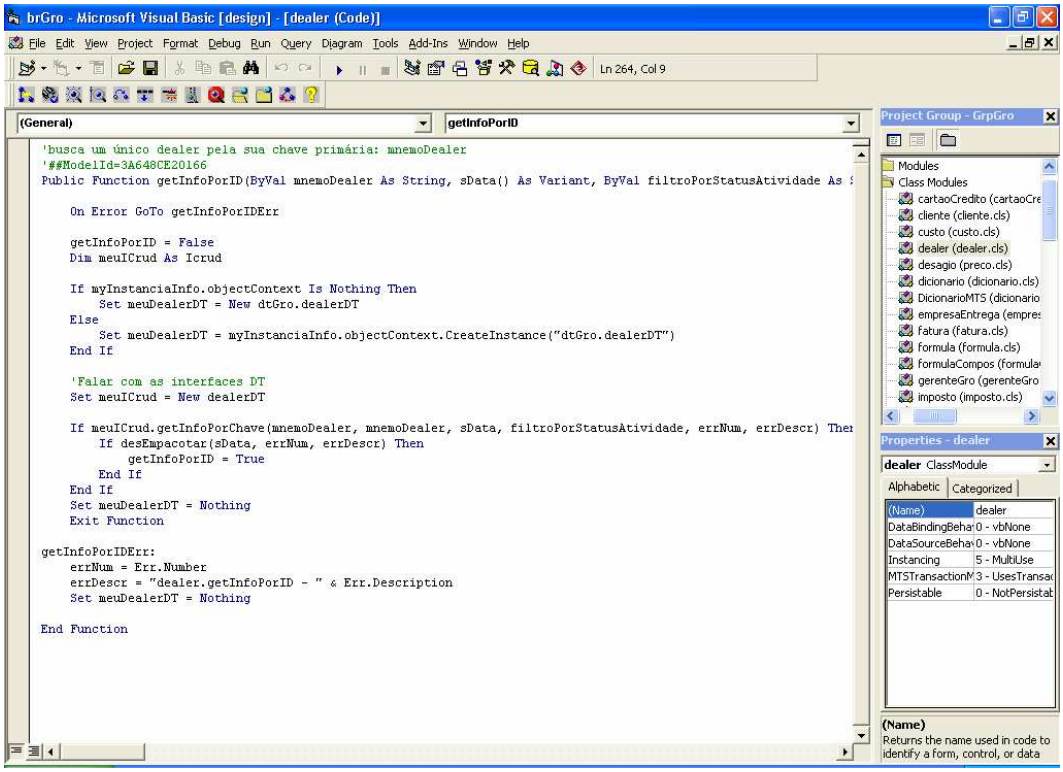

Figura 16 - Implementação de método de consulta para a classe *Dealer*

Pode-se verificar pelas figuras 15 e 16 que o código dos métodos é muito parecido, mesmo tendo sido tirados de duas classes diferentes. A razão é que todas as classes persistentes desta aplicação implementam a mesma solução padronizada para recuperar as informações de uma determinada instância de uma classe. Isto mostra que é possível configurar um código pré-definido utilizando-se uma parametrização, que neste caso foi o tipo de associação (associação simples, sem agregação) e o nome da classe.

Em uma consulta, quando duas classes têm uma associação de agregação entre si, normalmente é conveniente que a classe Todo selecione também as informações da classe Parte. Neste caso o código pré-configurado do método de consulta é diferente (prevê a existência da Parte) e a configuração requer os seguintes parâmetros: tipo da associação (composição ou agregação), nome da classe Todo e nome da classe Parte, conforme exemplo da figura 18.

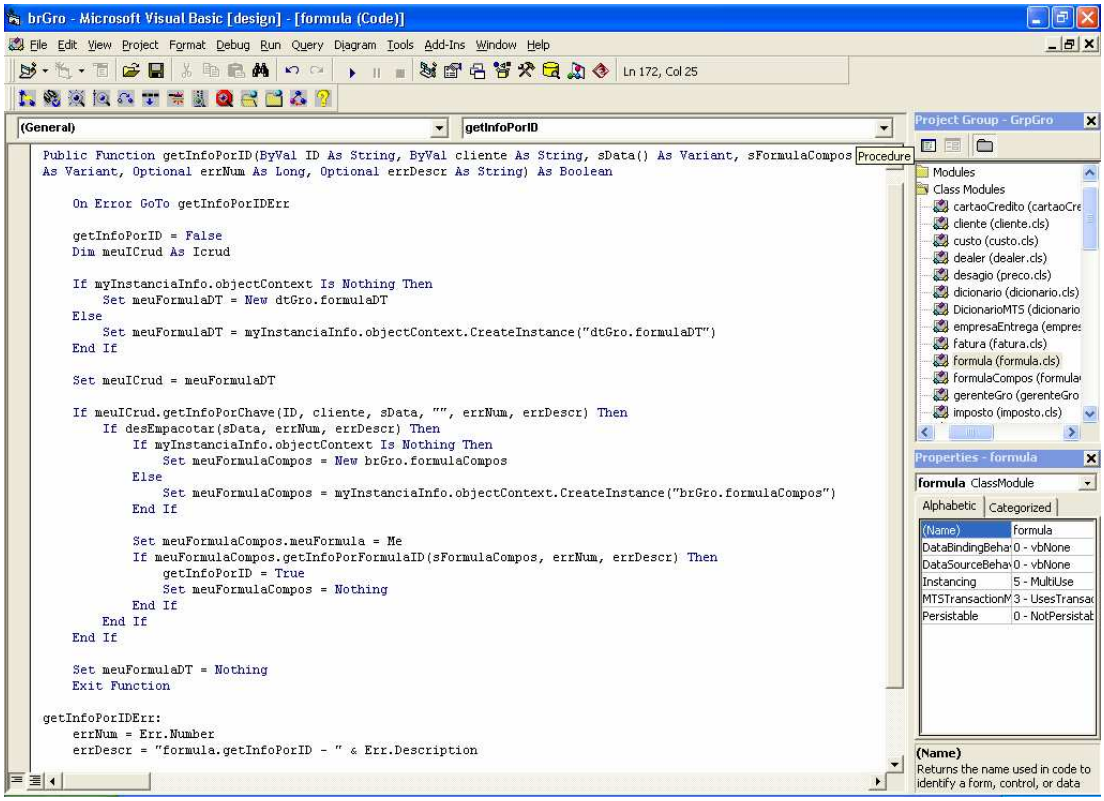

Figura 17 - Método de consulta em uma associação tipo agregação

Da mesma forma, classes que tenham um relacionamento do tipo herança também precisam de uma implementação específica dos métodos de persistência.

5.4 Utilização das soluções - um exemplo prático

Uma ferramenta CASE MDA pode ser configurada para gerar aplicações comerciais a partir de um conjunto de decisões de *design*. Como exemplo de solução que não fornece implementação (soluções de caixa branca), a ferramenta deve gerar *software* baseado no padrão camadas (*layers*) (BUSCHMANN *et al*, 1996) em uma configuração de três camadas, usando o padrão DAO (ALUR; CRUPI; MALKS, 2002) para a persistência de dados. A separação das áreas de interesse das camadas deve ser feita da seguinte forma:

- Camada superior: entidades de negócio;
- Camada intermediária: classes DAO;
- Camada inferior: acesso aos dados persistentes.

Classes DAO devem implementar uma interface CRUD (BRANDON, 2002) e o padrão Controlador de Casos de Uso (*Use-Case Controller*) (AGUIAR; SOUZA; PINTO, 2001) (EVANS, 2004) deve ser utilizado para mapear a especificação de requisitos para a implementação.

5.4.1 Utilização das soluções de caixa branca

Neste tópico é analisado um sistema de vendas varejo *online*, construído segundo a arquitetura (solução de Caixa Branca) descrita em 5.4. No caso de uso *Manutenção de Cliente* há uma consulta de cliente, que seleciona o cliente por seu ID e valida a senha do distribuidor (há dois objetos de negócio interagindo para permitir esta consulta). A representação convencional deste comportamento é:

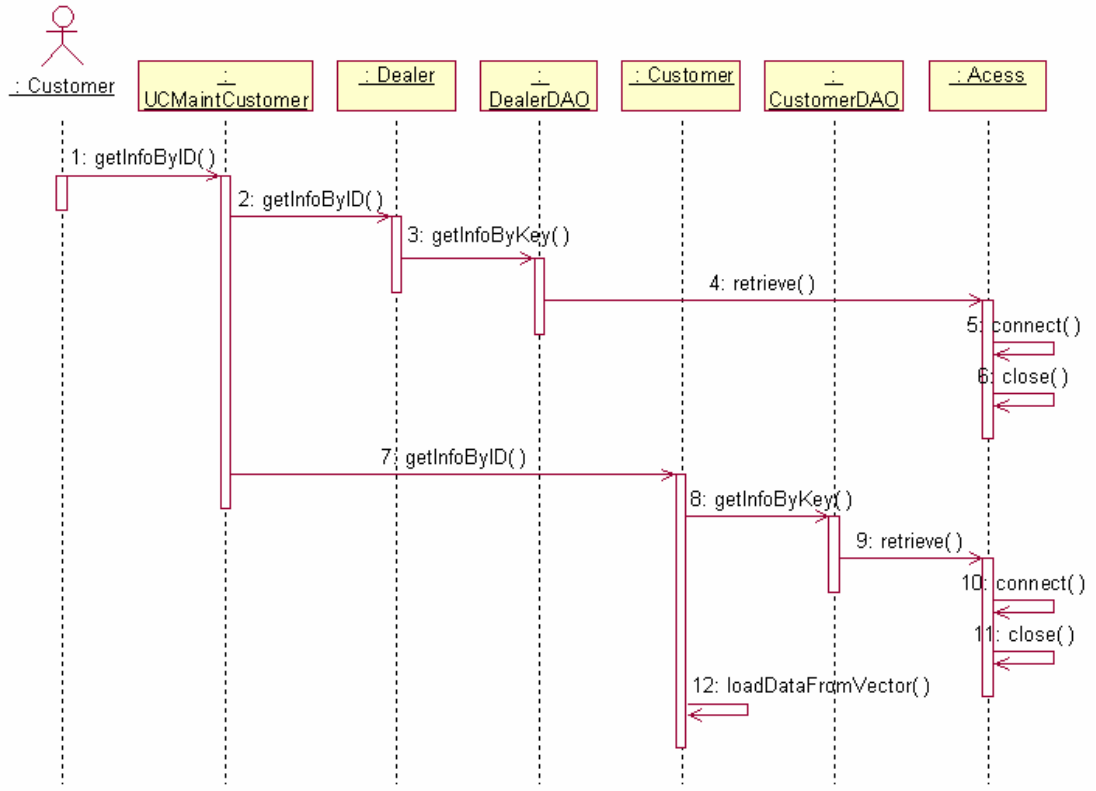

Figura 18 - Diagrama de seqüência para uma consulta no sistema de vendas varejo proposto (BELIX *et al*, 2005)

O diagrama de seqüência da figura 18 representa as mensagens necessárias para esta consulta dentro do contexto de arquitetura estipulado. As classes DAO e ACCESS são conseqüência das decisões adotadas.

Supondo agora uma ferramenta CASE MDA com transformações pré-definidas para automaticamente aplicar o mencionado conjunto de soluções de caixa branca. Neste mesmo exemplo, a ferramenta seria preparada para entender que cada classe persistente deve ter uma classe DAO correspondente e responsável por quaisquer requisição relacionada aos dados persistentes. As classes DAO, por sua vez, herdariam a interface CRUD e usariam uma classe (a classe ACCESS) para a execução das sentenças SQL.

Apesar da arquitetura do software permanecer a mesma, ter uma ferramenta CASE MDA que automaticamente aplique este conjunto de soluções de caixa branca permitirá ao usuário - no PIM - especificar apenas o seguinte comportamento para o mesmo diagrama de seqüência:

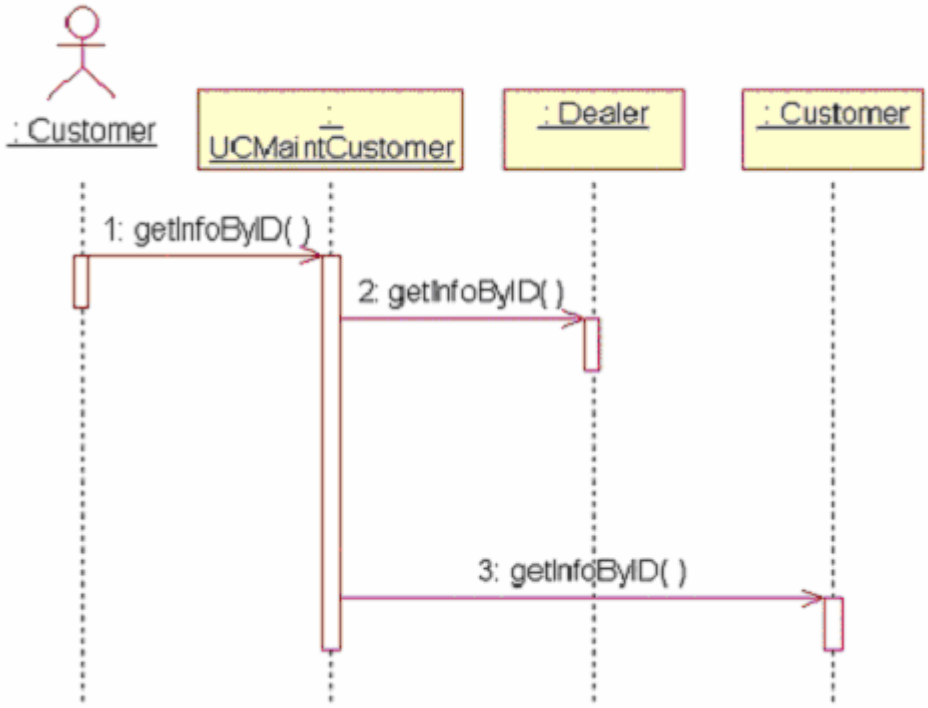

Figura 19 - Diagrama de seqüência da consulta, mas com a aplicação das soluções de caixa branca (BELIX *et al*, 2005)

Com o comportamento apresentado na figura 19, mas aplicando o conjunto de soluções de caixa branca, como definido em 5.4, a ferramenta CASE MDA é capaz de gerar um diagrama de seqüência PSM similar ao da figura 18.

Mas mesmo tal diagrama não contém toda a informação para uma ferramenta CASE gerar o código necessário. Apesar de ser representativo à cognição humana, um método chamado *obtenhaInformaçãoPorID* (*getInfoByID*) nada representa para um computador. Mesmo com a aplicação das soluções de caixa branca, ainda se faz necessário modelar o comportamento detalhado do método citado.

Outro possível caminho para resolver este problema é através da aplicação das soluções que realmente implementam código-fonte, ou seja, soluções de caixa preta e caixa cinza.

5.4.2 Aplicação das soluções de caixa preta e caixa cinza

Uma vez que a interface CRUD e o padrão DAO foram aplicados, o código-fonte dos métodos de persistência, parcialmente pré-definido, pode ser automaticamente configurado por uma ferramenta CASE para qualquer classe (as implementações são todas muito parecidas), em um caso de solução de caixa cinza.

Devem ser fornecidos parâmetros pela ferramenta CASE para configurar as variantes neste código parcialmente escrito (como tipo de associação e classes envolvidas). XVCL (*XML-Based Variant Configuration Language*), uma linguagem de uso geral para configurar variantes em artefatos texto, pode ser usada para gerar o código-fonte final (JARZABEK; ZHANG; SWE, 2002).

A classe ACCESS, responsável pelas conexões com o banco de dados e pela execução de sentenças SQL, não demandaria parametrização adicional e poderia ser automaticamente implementada, em um caso de solução de caixa preta.

Com esses dois últimos tipos de solução, uma ferramenta CASE MDA seria capaz de completamente implementar todos os métodos de persistência deste exemplo.

5.5 Considerações sobre modelagem e o valor das restrições

No exemplo apresentado neste capítulo, o desenvolvedor apenas modelaria o Caso de Uso até o ponto onde ele ainda pudesse decidir sobre o *design* da aplicação, sua estrutura e comportamento. É importante notar que, segundo a abordagem apresentada aqui, muitas das decisões de projeto são deixadas para a ferramenta. Não

há razão para modelar o que já está implementado/definido e que não requeira considerações adicionais. “Quando decidimos como modelar alguma coisa, determinar o correto nível de solução e detalhes é crítico para prover algo que seja benéfico para os usuários do um modelo. De uma maneira geral, é melhor modelar os artefatos que serão construídos e manipulados para produzir o resultado final. Modelar o interior de um *web server* ou de um *web browser* não ajudará os projetistas e arquitetos de uma aplicação *web*” (CONALLEN, 1999).

5.5.1 Restrições são úteis

Pode-se pensar em restrições fixas como algo que tornaria o design mais difícil, mas restrições e fronteiras são úteis – ajudam a focalizar a mente (COLWELL, 2005).

Uma ferramenta CASE MDA que contivesse um conjunto de transformações configurado para trabalhar segundo as decisões de *design* apresentadas neste capítulo não funcionaria para a geração de um sistema de processamento digital de sinais, o qual demanda a utilização de um padrão de arquitetura conhecido por *Pipes & Filters* (FRANÇOIS, 2003) (BUSCHMANN *et al*, 1996). Por outro lado, facilitaria enormemente a tarefa de modelagem de uma aplicação comercial, como a sugerida no exemplo estudado.

A ferramenta forneceria uma configuração padrão que poderia ser alterada pelo usuário. Seria útil se a ferramenta oferecesse a possibilidade de algum tipo de configuração para a geração de código, provavelmente através da escolha de conjuntos específicos de transformações. Neste caso é importante que a ferramenta tenha o conceito de árvore de decisão, já configurada, para apenas permitir escolhas consistentes. Não há razão para escolher J2EE com persistência gerenciada por container e *Hibernate* para o mesmo projeto.

5.6 Considerações finais do capítulo 5

Este capítulo apresenta uma taxonomia de soluções que podem ser aplicadas ao desenvolvimento MDA. A classificação abrange soluções que variam de um espectro que vai de nenhuma implementação até implementação completa de uma solução. O objetivo dessa taxonomia é entender como diferentes tipos de soluções agem no

desenvolvimento MDA, fornecendo resolução de problemas recorrentes e, alternativamente, auxiliando o desenvolvedor a conseguir implementação do software com baixo esforço de modelagem. Baixo esforço significa que parte do comportamento e estrutura do software pode ser automaticamente fornecida sem que haja necessidade de especificação por parte do usuário da ferramenta CASE MDA. O capítulo apresenta um exemplo para esclarecer estes pontos.

6 A APLICAÇÃO DAS SOLUÇÕES PRÉ-DEFINIDAS NO DESENVOLVIMENTO MDA

Este capítulo mostra como combinar soluções pré-definidas, transformações e diagramas UML para aumentar a quantidade de código gerado dentro do contexto MDA. O capítulo defende a formalização do relacionamento entre os diagramas UML como forma de verificar a consistência de um modelo de software e apresenta um exemplo, particular e incompleto, para demonstrar a abordagem MDA.

6.1 Abordagens *top-down* e *bottom-up*

Software é um complicado e complexo conjunto de decisões, conceitos e algoritmos, que fornece um mapeamento entre requisitos e recursos técnicos disponíveis. Este mapeamento pode ser visto a partir de duas direções diferentes: novos recursos técnicos são desenvolvidos para atender os requisitos de uma aplicação ou então recursos técnicos existentes são compostos para criar soluções que atendam estes requisitos (PIZKA; BAUER, 2004).

Apesar de ambas as direções serem válidas, elas implicam em uma forma de pensamento diferente. Na abordagem *top-down* o desenvolvimento de software inicia-se com os requisitos, os quais são progressivamente refinados até que a especificação fique suficientemente detalhada para o início da implementação. A abordagem *bottom-up*, por sua vez, origina-se nos dos recursos técnicos disponíveis e tenta atender os requisitos através da construção de serviços e componentes a partir do que já esteja implementado (PIZKA; BAUER, 2004).

Apesar do produto final ser sempre uma correspondência entre os requisitos e os recursos técnicos disponíveis, o caminho percorrido durante o desenvolvimento difere dependendo da abordagem escolhida, já que os passos são guiados por perspectivas diferentes (PIZKA; BAUER, 2004).

O MDD propõe um processo de desenvolvimento onde modelos de um sistema, em diferentes níveis de abstração, são transformados (refinados) até que o código-fonte seja finalmente gerado. Por essas características, alguns autores sugerem que o

MDD, de uma maneira geral, e o MDA em especial, sejam processos *top-down* (MORENO; VALLECILLO, 2004), (ESTUBLIER; VEGA, 2005).

No entanto, quando se fala em reuso, a abordagem *bottom-up* é requerida (MORENO; VALLECILLO, 2004). O reuso em si não é uma meta, mas é necessário para garantir menores custos de produção e manutenção de software, maior rapidez na entrega de sistemas e aumento da qualidade (ESTUBLIER; VEGA, 2005), (SOMMERVILLE; 2004).

Esta dissertação propõe a reutilização de soluções pré-definidas como forma de auxiliar o desenvolvedor a resolver pontos comuns e observáveis em diferentes aplicações. A intenção é conseguir uma maior produtividade no desenvolvimento MDA através da redução nos esforços de modelagem de software, tornando a abordagem MDA mais atrativa. Ou seja, esta proposta impõe um caráter simultaneamente *top-down* e *bottom-up* para o MDA:

- O desenvolvimento MDA é parcialmente *bottom-up* porque o objetivo da dissertação é propor a utilização de soluções pré-definidas - completas ou que requeiram complementação - como forma de atingir reuso de software e seus conseqüentes benefícios;
- É parcialmente *top-down* porque o desenvolvedor deve preencher o intervalo que separa os requisitos de um software dos recursos técnicos que os suportam. O desenvolvedor faz um refinamento sucessivo, utilizando modelos UML de diferentes graus de abstração, até atingir o nível de abstração dos recursos técnicos disponíveis.

6.2 Funcionalidades de uma ferramenta CASE

De uma maneira resumida, o desenvolvimento MDA é a transformação sucessiva de modelos que representam um mesmo sistema, o que coloca modelos e transformações na condição de artefatos-chave desta abordagem. Pode-se pensar então em uma ferramenta CASE MDA apenas como um processador de transformações, mas ela é mais do que isto.

Stephen Mellor (MELLOR, 2002) fornece uma lista de funcionalidades desejáveis em uma ferramentas CASE MDA:

- Construtores de modelos: fornecer IDE para a edição de modelos;
- Verificadores de modelos: interpretar modelos com valores reais, de tal forma que os usuários possam determinar se o comportamento do modelo é correto;
- Compiladores de modelos: compilar modelos para diversas plataformas;
- Depuradores de Modelos: executar código compilado, de tal forma que desenvolvedores possam ver o código compilado a partir de modelos em ação.
- Analisadores de modelos: encontrar estados inatingíveis;
- Testadores de modelos: gerar e executar casos de testes para os modelos.

Contudo, antes que uma ferramenta CASE possa executar as funções descritas, é necessário que esta verifique a consistência do modelo UML de um software, caso contrário a geração de código tornar-se inexeqüível.

6.3 Considerações sobre a geração automática de código a partir de modelos

Devido à semântica imprecisa e aos diferentes empregos dos diagramas UML dentro de um processo de desenvolvimento de software, não há uma noção universal de consistência (ENGELS *et al*, 2001). A solução deste problema é o estabelecimento de regras - em conformidade com a linguagem - que estabeleçam níveis de abstração e inter-relacionamento entre os modelos UML. Tal abordagem cria uma prescrição de uso da linguagem e permite que seja feita verificação de consistência vertical e horizontal, assegurando consistência antes e depois da transformação do modelo. Caso estas regras não sejam estabelecidas, a ferramenta não terá como fazer a verificação de consistência do modelo de um software.

É desejável que as ferramentas CASE MDA sejam lançadas com um conjunto inicial de transformações. Também é desejável que as ferramentas permitam a alteração ou criação de novas transformações, atendendo às necessidades específicas de um

usuário. A definição das transformações deve ser feita respeitando-se a prescrição de uso da UML, caso contrário a execução das transformações induzirá inconsistências no modelo gerado.

Também existe uma forte dependência entre as transformações e as soluções pré-definidas: ambas precisam ser desenvolvidas em paralelo, caso contrário as transformações podem modificar o modelo de uma forma incompatível com as soluções.

Por exemplo, soluções de caixa branca fornecem a descrição de uma solução, sem sua respectiva implementação. Conforme apresentado nesta dissertação, são utilizadas para estabelecer a arquitetura de um software. Um exemplo de solução de caixa branca é a utilização do padrão DAO para a persistência dos dados (ver figura 18). Esta solução é introduzida no modelo através de um conjunto particular de transformações PIM-PSM, o que denota o estreito relacionamento solução-transformação. Algumas dessas transformações, capazes de refinar o comportamento especificado na figura 19 para o especificado na figura 18 são:

- Toda classe persistente acarreta a geração de uma classe DAO;
- Toda classe DAO implementa uma interface CRUD padronizada;
- Toda a solicitação de serviços de persistência de uma classe é redirecionada para a classe DAO correspondente;
- Toda classe DAO utiliza a classe Acesso para fornecimento de conexões com o banco de dados e para a execução de sentenças SQL;

As soluções de caixa preta, por sua vez, fornecem implementação para elementos definidos nas transformações. No mesmo exemplo da figura 18, durante a transformação PSM-Código foi incluído o arquivo *Acesso.java*, para servir de implementação para a classe *Acesso*. Se as transformações definirem uma classe Acesso com métodos diferentes dos existentes em *Acesso.java*, não será possível gerar código consistente a partir do modelo UML.

A implementação de uma solução de caixa preta não depende necessariamente da inclusão de código-fonte, componentes ou *webservices*: r*ecordsets* desconectados, não requerem a inclusão de código-fonte adicional, porque este recurso está

disponível como API das linguagens Java (a partir do JDK 1.5, com o *Cachedrowset*) e Visual Basic (a partir do VB5, com o uso do ADO).

Pelo exposto neste tópico, três diferentes pontos precisam ser resolvidos para o emprego da abordagem MDA:

- Regras para inter-relacionamento de modelos UML;
- Elaboração das soluções pré-definidas;
- Definição das transformações.

6.3.1 Regras para inter-relacionamento de modelos UML

Por uma questão de simplificação, nesse tópico somente são tratados os diagramas de Casos de Uso, diagrama de classes e diagramas de seqüência. Não são tratados os diagramas de atividades, porém as ações (*action boxes*) podem ser utilizadas dentro de diagramas de seqüência, como sugerido por (LARMAN, 2005).

O estabelecimento de um conjunto de regras de níveis de abstração permite o sucessivo refinamento de modelos de um software e auxilia a evitar os problemas de consistência vertical. Um conjunto particular e simplificado de regras para este caso é:

- Em um nível mais alto, requisitos funcionais devem ser descritos com a técnica de Casos de Uso;
- Em um nível intermediário as entidades encontradas nos casos de uso devem ser descritas em um diagrama de classes. Os caminhos das especificações de casos de uso devem ser descritos com diagramas de seqüência;
- Os métodos das classes podem ser parcial ou totalmente descritos com diagramas de seqüência;
    - Ação (action box) pode ser utilizada em diagramas de seqüência para manipular o valor de algum atributo da classe referenciada no diagrama.

Um conjunto de regras de relacionamento entre modelos de um mesmo nível de abstração permite a descrição multi-visão de aspectos de um software e auxilia a

evitar os problemas de consistência horizontal. Um conjunto particular e simplificado de regras para este caso é:

- Diagramas de seqüência somente podem trocar mensagens entre classes que tenham algum tipo de associação entre si, estabelecida através do diagrama de classes;
- Os métodos utilizados nos diagramas de seqüência devem estar presentes nas classes envolvidas;

Uma vez que estas regras de consistência entre modelos são definidas - como feito de forma simplificada nesta dissertação - um caminho possível para a consistência automática é a abordagem de tradução: converte-se a UML ou partes dela para uma linguagem bem definida e então verificação de consistência é feita por tradução (ASTESIANO; REGGIO, 2002).

Um exemplo desta abordagem de tradução pode ser encontrado em (LONG *et al*, 2005), onde os autores estudam os conceitos presentes nos diagramas de classe e seqüência (os dois únicos envolvidos no artigo em questão), traduzem estes conceitos para uma linguagem chamada rCOS (*Relational Calculus of Object Systems*), definem as condições de consistência entre diagrama de classes e seqüência e, por fim, criam um algoritmo para executar a verificação desta consistência.

Não é objetivo desse trabalho fornecer uma abordagem de verificação de consistência para modelos UML, mas sim mostrar que é necessário o estabelecimento de regras de relacionamento entre modelos para se alcançar um modelo de software consistente.

6.3.2 Elaboração das soluções pré-definidas

Este trabalho apresenta um conjunto resumido e particular de soluções, apenas para demonstrar a viabilidade da utilização das mesmas na abordagem MDA. As soluções são:

- O software deve utilizar o padrão Camadas e ser gerado em duas camadas - UI (*User Interface)* e *Negocio* (caixa branca);

- O software utiliza o padrão Controlador de Caso de Uso (AGUIAR; SOUZA; PINTO, 2001) (EVANS, 2004) (caixa branca);
- O software utiliza *Recordset* Desconectado para conexão com o banco de dados, para o transporte dos dados e para o controle multiacesso dos dados (caixa preta);
- As classes com esteriótipo <<Entidade>> recebem métodos padronizados de acesso ao banco de dados. Há implementação específica quando a classe estiver em uma associação tipo Todo-Parte, desempenhando o papel do Todo (caixa cinza);
- Uma classe pré-existente fornece conexões com o banco de dados. A classe é implementada por *Acesso.java* e utiliza *ArgumentosConn.Java* como parâmetro de conexão (caixa preta);
- Uma classe pré-existente fornece números aleatórios para serem usados como chave (ID) das classes persistentes. A classe é implementada por *Funcoes.java* (caixa preta);
- Existe a necessidade de se fazer a transposição das informações entre o *recordset* desconectado e os objetos e vice-versa. Uma classe fornece tal funcionalidade e é implementada por *Empacotador.java* (caixa preta);
- O software deve fazer uso de uma classe para tratamento de erros. A classe recebe o nome de MDAExceptions e é implementada por *MDAExceptions.Java* (caixa preta).

6.3.3 Definição das transformações

É objetivo desse trabalho fornecer um conjunto particular e resumido de regras de transformação, suficiente apenas para exemplificar a abordagem MDA. Este conjunto visa à criação dos objetos de negócio (*server-sided*), não sendo aplicáveis à criação da interface gráfica do usuário e nem ao banco de dados. Por uma questão de facilidade de leitura, as regras de transformação são apresentadas informalmente. Há transformações PIM-PIM (refinamento), PIM-PSM e PIM-Código.

As transformações PIM-PIM não embutem no modelo detalhes tecnológicos, apesar do PIM poder atender itens como persistência ou controle transacional. As particulares transformações PIM-PIM apresentadas aqui incluem no modelo o padrão Controlador de Casos de Uso e alguns métodos padronizados de persistência. É necessário que estes elementos sejam incluídos no PIM porque são utilizados na modelagem do sistema. Por exemplo, métodos de persistência podem ser utilizados nos diagramas de seqüência que descrevam outros métodos. Este particular exemplo ocorre na descrição do comportamento das classes controladoras de Casos de Uso.

Algumas das regras de transformações PIM-PIM (refinamento) utilizadas são:

- Todo Caso de Uso no diagrama de Casos de Uso acarreta a criação de uma classe não persistente no Diagrama de Classes. Tal classe utiliza o esteriótipo <<CCCU>>, que significa Classe Controladora de Caso de Uso;
- Para cada classe com esteriótipo <<CCCU>>, será criada uma classe não persistente com esteriótipo <<CIU>> (Controlador de Interface do Usuário);

  As classes <<CIU>> servem para mudar o meio de transporte das informações para fora da interface do sistema. Neste exemplo as classes <<CIU>> convertem o meio de transporte para classes ValueObj. Poderia também converter para formato XML, fornecendo uma interface do tipo *webservice*;
- Toda classe com esteriótipo <<Entidade>> é considerada persistente e recebe métodos de persistência (*getInfoPorID* e *getInfo*). Apenas os métodos de seleção são necessários porque o *recordset* desconectado cuida automaticamente de todas as alterações, exclusões ou inclusões de registros, bastando um simples comando (no caso do Java é `acceptChanges`) para atualizar o banco de dados;
- Toda classe com esteriótipo <<Entidade>> recebe uma variável do tipo *String* e de nome sQry (de *selection query*). Esta é utilizada para configurar a consulta genérica *getInfo*.
- Toda a classe com esteriótipo <<Entidade>> tem um identificador único, numérico, na forma de um atributo ID;

As regras de transformações PIM-PSM incorporam detalhes tecnológicos no modelo do software, tornando-o dependente da plataforma escolhida. Algumas das transformações propostas neste exemplo particular são:

- Toda a classe com esteriótipo <<Entidade>> recebe um método padronizado chamado `empacotar` (serve para fazer a transposição das informações entre *recordset* desconetado e os objetos);
- Para cada classe com esteriótipo <<Entidade>> há uma classe sem métodos, com os mesmos atributos, mesmo nome, mas com sufixo Value. Por exemplo a classe Cliente gera a classe ClienteValue. Esta classe Value é utilizada para comunicar as informações do sistema para além da camada UI;
- Uma transformação que crie um pacote chamado *Utilidades* no modelo UML para abrigar as classes Funções (para fornecimento de números aleatórios utilizados como ID dos objetos) e Empacotador;
- Uma transformação que inclua as classes Funções e Empacotador no pacote *Utilidades*;
- Uma transformação que crie os pacotes *Negócio e UI (User Interface)*, criando um relacionamento de dependência de *UI* para *Negocio*, como forma de implementar as duas camadas previstas na solução de caixa branca;
- Uma transformação que crie um relacionamento de dependência dos pacotes *Negocio* e *UI* e para o pacote *Utilidades*;
- Criar em *Negocio* uma classe Acesso, com o método `forneceConexao`;
- Criar em *Negocio* uma classe ArgumentosConn com os parâmetros para conexão com o banco de dados;

As transformações PSM-Código servem para transformar o modelo do software em código-fonte. Algumas das regras de transformações PSM-Código utilizadas são:

- A classe Funções deve ser substituída pelo arquivo Funcoes.Java;
- A classe Acesso deve ser substituída pelo arquivo Acesso.Java;

- A classe MDAExceptions deve ser substituída pelo arquivo MDAExceptions.Java;
- Para cada classe expressa no modelo, deve ser criado um arquivo Java com o mesmo nome. Os atributos da classe devem ser codificados neste arquivo;
- Respeitar a dependência entre pacotes para criar os `imports` das classes Java;
- Toda classe <<Entidade>> com método *empacotar* deve receber o texto padronizado que implemente o método em questão.
- Toda classe <<Entidade>> com método *getInfo* deve receber o texto padronizado que implemente o método em questão. O texto precisa ser configurado, recebendo o nome da classe como parâmetro (ver apêndice A, item A.1);
- Toda classe <<Entidade>> com método *getInfoPorID* deve receber o texto padronizado que implemente o método em questão. O texto precisa ser configurado, recebendo o nome da classe como parâmetro (ver apêndice A, item A.1);
    - Caso a classe desempenhe o papel de Todo em um relacionamento Todo-Parte, a implementação do método é diferente, mas para qual também existe texto padronizado. Neste caso a configuração do texto exige o nome da classe Todo e o nome da classe Parte

Como observação, a geração dos atributos das classes Java pode ser realizada com o uso da macro XSLT apresentada na tabela 3. A configuração dos métodos padronizados e incompletos *getInfoPorId* e *getInfo* pode ser feita com uso de XVCL.

6.4 Um exemplo de geração de código

Partindo-se da sugerida prescrição de uso para a UML e de definições de transformações e soluções pré-definidas, é possível desenvolver um software exemplo que atenda a especificação de requisitos constante no apêndice C.

Considerando-se a utilização das transformações para a criação das camadas UI e Negócio, bem como o pacote Utilidades, a organização lógica do software passa a ser a exibida pela figura 20:

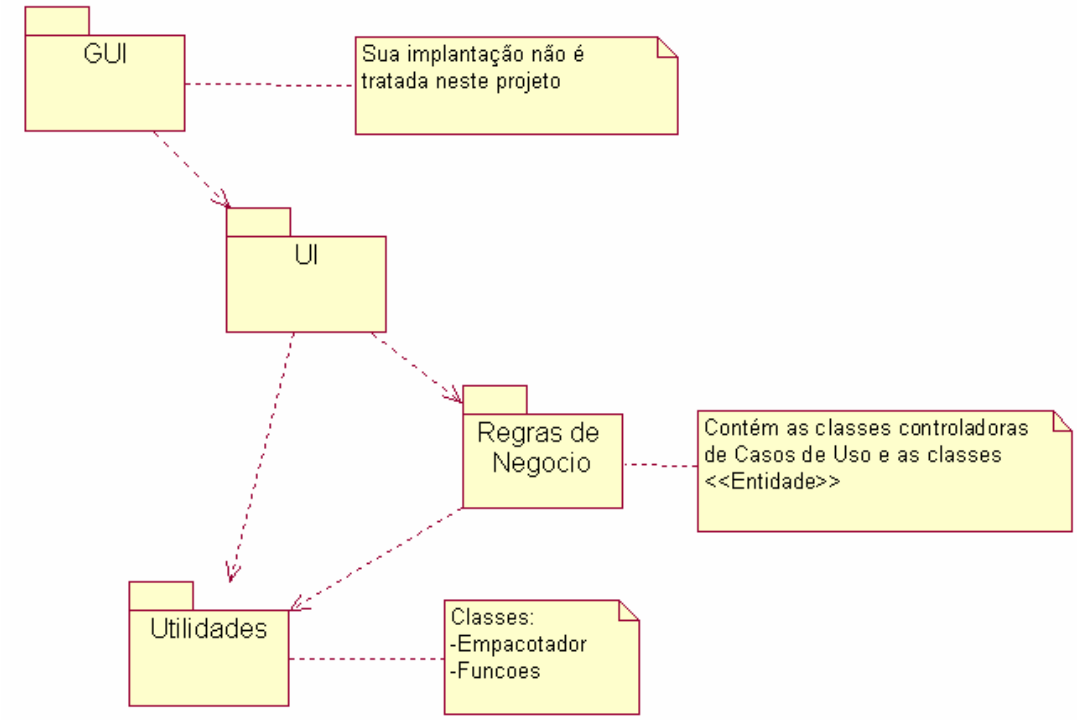

Figura 20 - Organização lógica do software exemplo

O diagrama de classes do software, antes da aplicação das transformações PIM-PIM é o mostrado na figura 21:

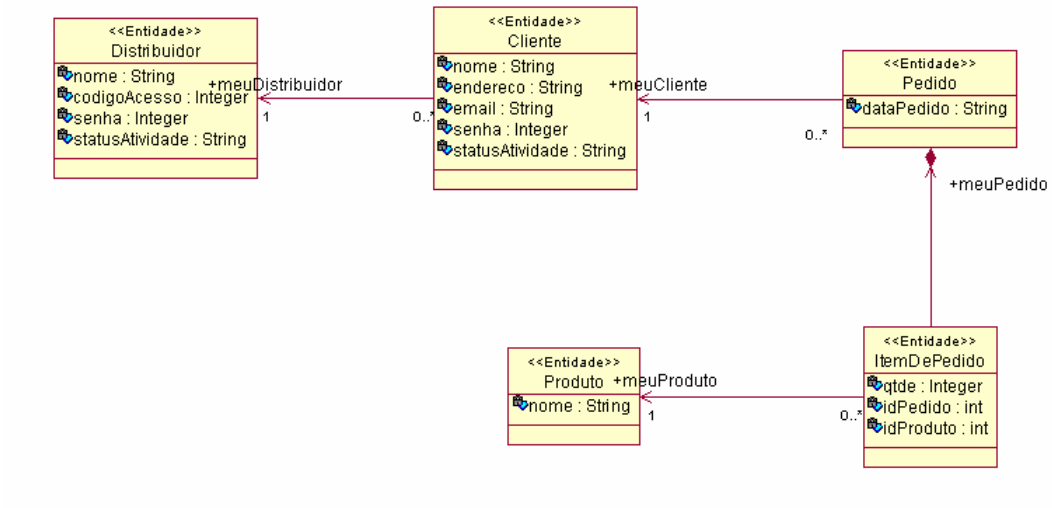

Figura 21 -Diagrama de classes antes da transformação PIM-PIM

Após a aplicação das transformações PIM-PIM descritas neste capítulo, o mesmo diagrama de classes passa a exibir os métodos de persistência fornecidos pelas soluções de caixa cinza. Tais métodos são representados no modelo com o uso das transformações PIM-PIM. Vale lembrar que ainda não há dependência de questões tecnológicas.

O diagrama de classes refinado pelas transformações PIM-PIM é exibido na figura 22:

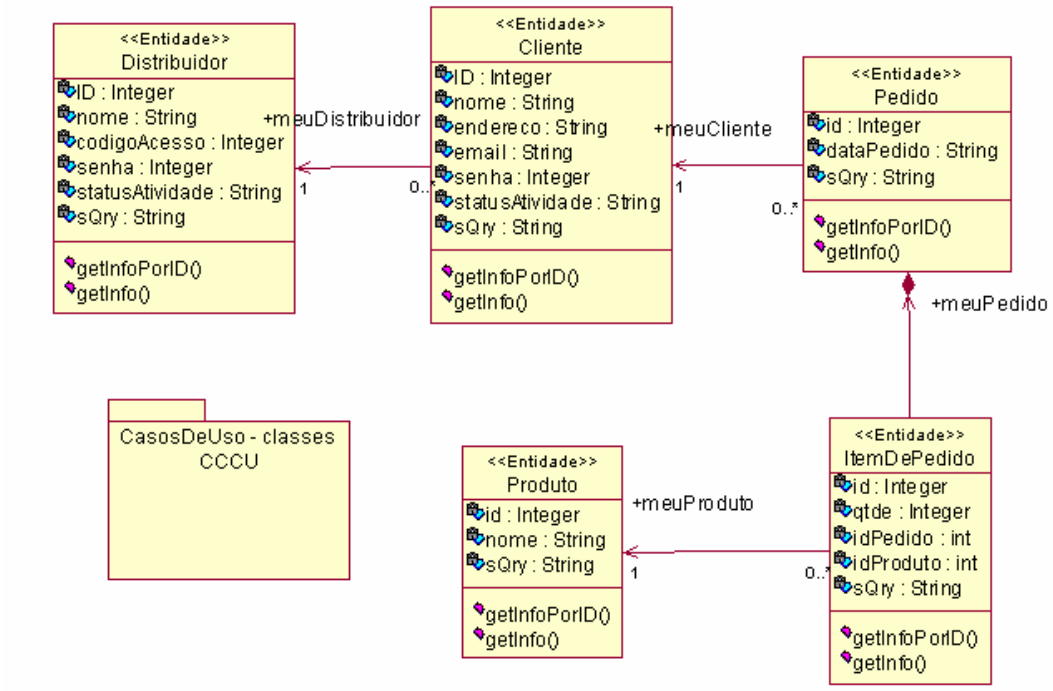

Figura 22 - Diagrama de classes refinado pela transformação PIM-PIM

O Pacote *Casos de Uso – Classes CCCU* é automaticamente aplicado pelas transformações e contém as classes controladoras para suportar os casos de uso do sistema.

Mais refinamento é incorporado com as transformações PIM-PSM, agora com a adição de detalhes tecnológicos. As transformações PIM-PSM incluem mais um método no diagrama de classes e criam as classes *ValueObject*, apenas com atributos, no pacote UI. O diagrama de classes PSM é apresentado na figura 23:

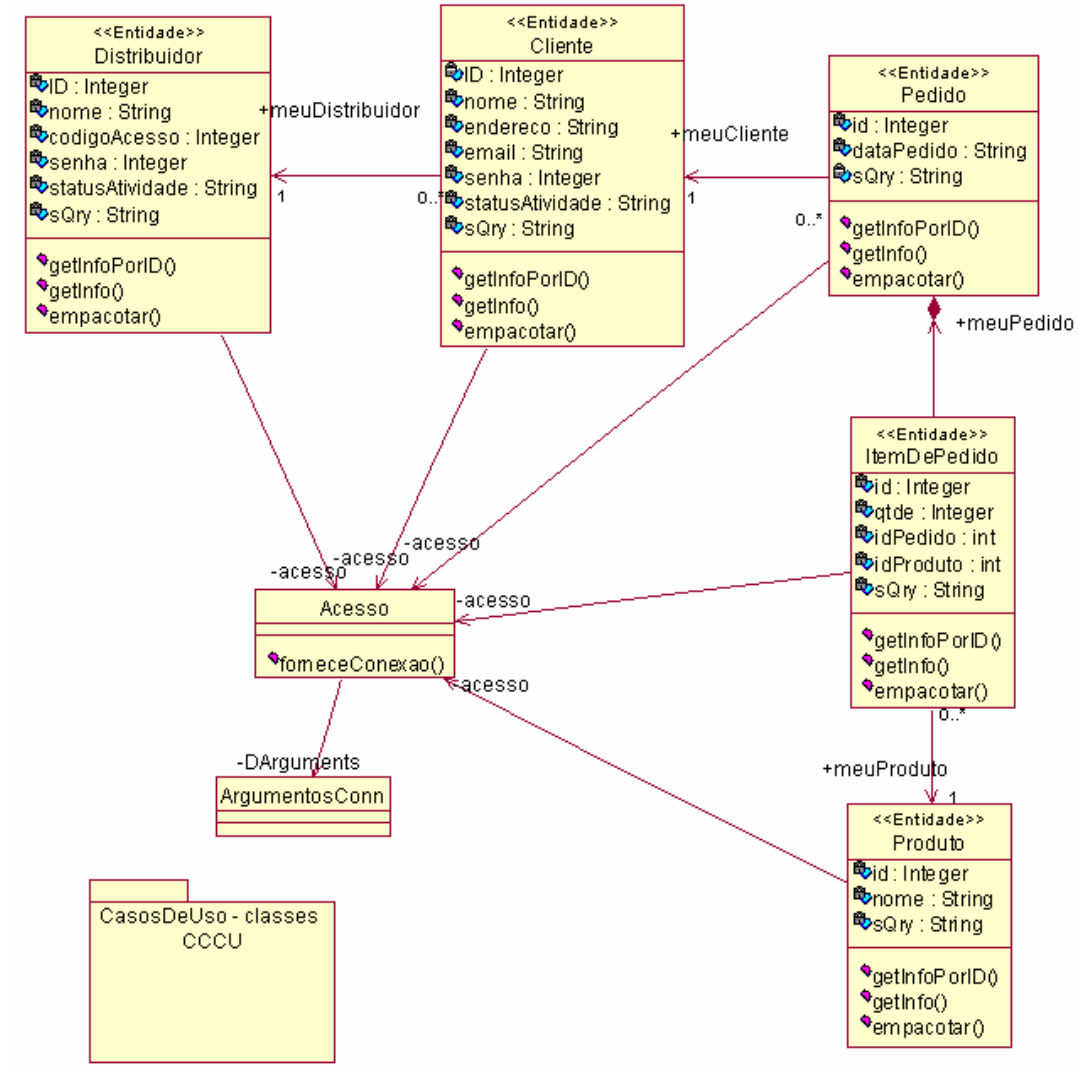

Figura 23 - Diagrama de classes PSM

As classes Acesso e ArgumentosConn são soluções de caixa preta que foram incluídas no modelo por conta das transformações PIM-PSM. Transformações PSM-Código as incluirão no código-fonte do software.

Aplicando-se as transformações PSM-Código, tem-se a criação de um arquivo texto para cada classe. Nestes arquivos há os atributos das classes e seus métodos. Os métodos padronizados *getInfo*, *getInfoPorID* e *empacotar* recebem implementação por serem exemplos de solução de caixa cinza e preta. Nesta última rodada de transformações, a classe Cliente é gerada como apresentado na tabela 6:

Tabela 6 -Código-fonte da classe Cliente

```
package mda.negocio;

import java.sql.Connection;
import java.sql.SQLException;
import java.sql.Statement;
import mda.utilidades.Empacotador;
import com.sun.rowset.CachedRowSetImpl;

public class Cliente
{
	private Integer id;
	private String nome;
	private String endereco;
	private String email;
	private Integer senha;
	private String statusAtividade;
	private String sQry;
	private Connection con;

	private Statement stmt;
	private Acesso acesso;

	public Distribuidor meuDistribuidor;

	public Integer getId() {
		return id;
	}

	public void setId(Integer id) {
		this.id = id;
	}

	public String getNome() {
		return nome;
	}

	public void setNome(String nome) {
		this.nome = nome;
	}

	public String getEndereco() {
		return endereco;
	}

	public void setEndereco(String endereco) {
		this.endereco = endereco;
```

```
}

public String getEmail() {
      return email;
}

public void setEmail(String email) {
      this.email = email;
}

public Integer getSenha() {
      return senha;
}

public void setSenha(Integer senha) {
      this.senha = senha;
}

public String getStatusAtividade() {
      return statusAtividade;
}

public void setStatusAtividade(String statusAtividade) {
      this.statusAtividade = statusAtividade;
}


/**
 * Comment=> Construtor padrão sem argumentos
 */
public Cliente()
{
      id = new Integer(0);
      nome="";
      endereco="";
      email="";
      senha = new Integer(0);
      statusAtividade="";
      sQry="";
      acesso = new Acesso();
      meuDistribuidor = new Distribuidor();
}

public void getInfoPorID(
            Integer pId,
            CachedRowSetImpl pCrs)
```

```
                throws MDAExceptions, SQLException {

        //Comment=> Monta a query para busca
        sQry = "SELECT * FROM TB_cliente ";
        sQry = sQry + " WHERE ID = " + pId.intValue();

        con = acesso.forneceConexao();
        stmt = con.createStatement();
        pCrs.populate(stmt.executeQuery(sQry));
        Empacotador.desempacotar(this, pCrs, 1);

        if (id.intValue() == 0) {
           throw new MDAExceptions("Cliente não encontrado!");
        }
        ;
        con.close();
    }

    public void getInfo(
                String pParametros,
                CachedRowSetImpl pCrs)
                throws SQLException {

        //Comment=> Monta a query para busca
        sQry = "SELECT * FROM TB_Cliente ";
        if (pParametros.length() != 0) {
              sQry = sQry + " WHERE " + pParametros;
        }
        ;
        con = acesso.forneceConexao();
        stmt = con.createStatement();
        pCrs.populate(stmt.executeQuery(sQry));
        con.close();
    }

    public void empacotar(CachedRowSetImpl recipiente)
    {
        Empacotador.empacotar(recipiente, this);
    }
}
```

6.4.1 Descrição de comportamento

Não é necessário descrever todo o comportamento de um software quando se usam as soluções pré-definidas. Este ponto foi exemplificado no capítulo 5 pelas figuras 18 e 19. Para o problema proposto (ver apêndice C), as soluções implementam todo o comportamento necessário para as classes <<Entidade>>, o que facilita o trabalho de modelagem dos métodos das classes controladoras de casos de uso.

Como a responsabilidade das classes UI está claramente determinada (ver item 6.3.3), soluções pré-definidas de caixa-cinza poderiam ter sido criadas de antemão para implementar métodos padronizados nestas classes. Para outros métodos que não podem ser previstos, o comportamento precisa ser modelado.

É importante notar que não é possível conceber de antemão soluções para todas as eventuais necessidades de um software. Isto equivale a dizer que o desenvolvimento seria completamente *bottom-up*. Conforme apresentado nesta dissertação, o desenvolvimento MDA é uma combinação das abordagens *top-down* e *bottom-up*.

O uso de soluções pré-definidas auxilia o desenvolvedor MDA a resolver problemas recorrentes. Além disso, a dissertação também defende que estas soluções sejam combinadas com a modelagem de software, permitindo atender eventuais necessidades de uma aplicação.

A consulta ao catálogo, por exemplo, não depende do distribuidor e é de acesso livre. Seu comportamento é muito simplificado, conforme exposto na figura 24:

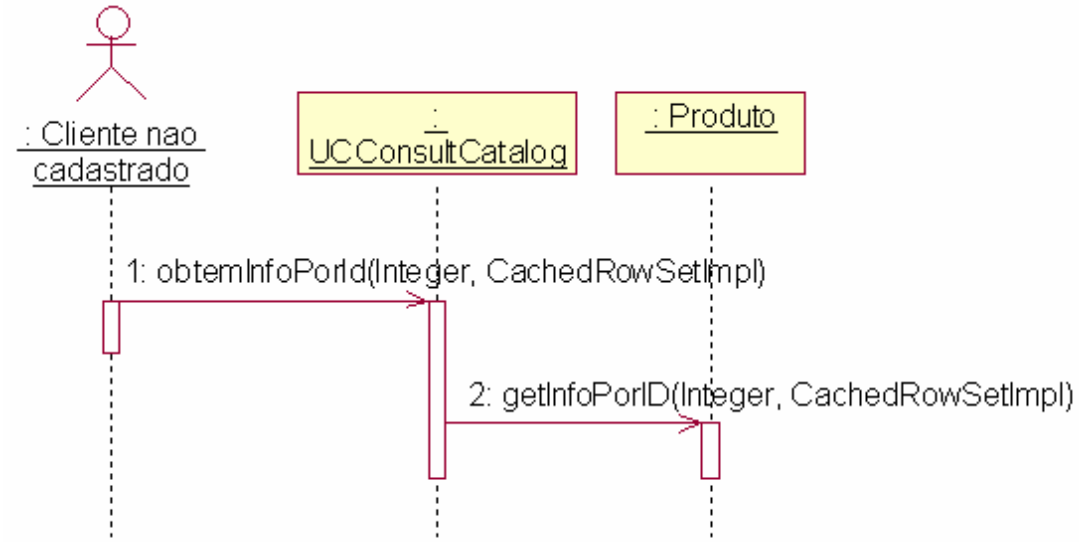

Figura 24 - Comportamento da consulta por ID ao catálogo

Tal comportamento pode ser traduzido pelo código da tabela 7, a seguir:

Tabela 7- Código-fonte da consulta por ID da classe controladora de Caso de Uso da consulta ao catálogo

```
public void obtemInfoPorId(
      Integer pIdProduto,
      CachedRowSetImpl pCrs)
      throws MDAExceptions, SQLException {

      oProduto = new Produto();
      oProduto.getInfoPorID(pIdProduto, pCrs);
}
```

Entretanto, a consulta ao cliente é mais sofisticada: depende da verificação da identidade do distribuidor, logo não é possível padronizar os métodos de consulta de todas as classes controladoras de caso de uso.

Algumas soluções acabam por atender uma grande gama de diferentes sistemas em diferentes domínios. (CZARNECKI; EISENECKER, 2000), divide os domínios em dois tipos, a saber:

- Domínios verticais: domínios de sistemas
- Domínios horizontais: domínios de partes de sistemas

Segundo Czarnecki, a aplicação de engenharia de domínio a um domínio vertical, como bancário, securitário ou de saúde, resulta em software reutilizável produzindo, por exemplo, um *framework* de sistema que cubra todo um domínio vertical. Por outro lado, a aplicação da engenharia de domínio a um domínio horizontal resulta em partes reutilizáveis de um sistema.

Soluções para persistência de dados, um ponto bastante explorado nesta dissertação, é um destes casos de partes largamente reutilizadas em sistemas comerciais. Sua aplicação automática, conforme apresentado nos capítulos 5 e 6, é muito vantajosa para o desenvolvedor.

6.5 Considerações finais do capítulo 6

Para que seja possível obter a geração automática de código a partir de modelos UML, primeiramente é preciso a definição de uma prescrição de uso para a linguagem, evitando assim os problemas de consistência horizontal e vertical e permitindo a verificação do modelo pela ferramenta CASE. A seguir é necessário o desenvolvimento das transformações e das soluções pré-definidas a serem aplicadas. Como as soluções e as transformações são fortemente dependentes umas das outras, elas devem ser desenvolvidas conjuntamente.

As soluções pré-definidas atendem necessidades recorrentes identificadas no desenvolvimento de software e são frutos da experiência, da observação atenta dos problemas e da seleção das melhores maneiras de resolvê-los.

As soluções apresentadas neste capítulo foram elaboradas para atender as necessidades de softwares comerciais e que possam ser especificados com a técnica de Casos de Uso. Apesar de o exemplo ser um sistema de vendas varejo, outros softwares que se adequem à mesma situação, como sistemas hospitalares ou de hotelaria, também se beneficiariam das mesmas transformações e soluções pré-definidas descritas neste capítulo.

7 CONSIDERAÇÕES FINAIS

A principal contribuição deste trabalho foi oferecer um estudo da abordagem MDA e uma visão prática de sua utilização. A sugestão de implementação do MDA exposta nesta dissertação leva em conta a possibilidade de encarar o MDA como um misto de abordagem *top-down* com *bottom-up*, de tal forma que o reuso seja possível e incentivado como forma de solução de problemas recorrentes aos domínios envolvidos (verticais ou horizontais) e como forma de aumentar produtividade no desenvolvimento de sistemas, tornando o MDA uma abordagem mais atraente para os desenvolvedores.

Ao propor o uso de soluções pré-definidas, a dissertação apresenta a abordagem MDA como sendo composta de outros artefatos que não apenas modelos e transformações. O desenvolvedor continua atuando sobre as transformações e sobre modelos UML (ou de outras linguagens derivadas do MOF), porém outros tipos de artefatos são trazidos para o contexto MDA: padrões, *frameworks*, componentes compilados, código-fonte (completa ou parcialmente escrito), programas XVCL (para resolver pontos de variabilidade em código-fonte), *web services* e tudo o mais que possa ser reutilizado para aumentar a produtividade no desenvolvimento MDA.

7.1 Trabalhos futuros

O trabalho mostra exemplos de aplicação de soluções para domínios horizontais, mostrando a capacidade do MDA em resolver questões tecnológicas com um mínimo de intervenção por parte do desenvolvedor. Para que o MDA seja ainda mais atraente, é necessária pesquisa adicional que proporcione a mesma capacidade de reutilização para domínios verticais. Os desafios são diferentes: enquanto a combinação da ferramenta CASE com as transformações é capaz de fornecer soluções que auxiliem a resolver a típica complexidade de implementação de um software OO, de uma forma total ou parcialmente independente do usuário, a resolução de questões referentes a domínios verticais necessita intervenção do usuário, que precisa configurar ou complementar uma solução pré-definida para atender suas necessidades específicas.

Há questões em aberto, como o problema de intercambiar *profiles* UML, transformações e toda a sorte de artefatos necessários ao desenvolvimento MDA, conforme proposto neste trabalho. Pesquisa adicional também é necessária para gerência de configuração, pois é preciso garantir que todos estes diferentes artefatos, com o passar do tempo, mantenham-se coerentes entre si para produzir um resultado final consistente.

Efetivamente a manutenção do software merece atenção: como o desenvolvimento é orientado por modelos, alterações no sistema acarretam alterações no modelo. Porém, não parece razoável que todo o código-fonte de um sistema precise ser gerado novamente a partir de uma alteração qualquer, mas apenas as porções de código-fonte afetadas.

Apêndice A - Exemplos de código-fonte de soluções de caixa cinza

A.1 Métodos de persistência (caixa cinza)

Na solução particular fornecida nesta dissertação, os métodos *getInfoPorId* e *getInfo* são aplicados às classes com esteriótipo <<Entidade>>. O Código precisa ser configurado com o fornecimento no nome da classe, em um caso de solução de caixa cinza.

O código da tabela 8 é aplicado a todas as classes <<Entidade>> Na transformação de PSM para código, o código é implementado com a substituição de *XXXXXXX* pelo nome da classe em questão.

Tabela 8 – Código-fonte base do método getInfoPorID

```
public void getInfoPorID(
            Integer pId,
            CachedRowSetImpl pCrs)
            throws MDAExceptions, SQLException {

      //Comment=> Monta a query para busca
      sQry = "SELECT * FROM TB_XXXXXXX ";
      sQry = sQry + " WHERE ID = " + pId.intValue();

      //Comment=> Recupera a conexão
      con = acesso.forneceConexao();

      //Comment=> Cria um Statement para manipular dados do DB
      // através da conexão
      stmt = con.createStatement();

      //Comment=> Executa a query e popula o CachedRowSet
      pCrs.populate(stmt.executeQuery(sQry));

      Empacotador.desempacotar(this, pCrs, 1);

      If (id.intValue() == 0) {
            throw new MDAExceptions("XXXXXXX não encontrado!");
      }
```

```
      ;

      // Comment=> Fecha o statement e a conexão
      con.close();

      }
```

O código-fonte base do método *getInfo*, uma consulta genérica, está na tabela 9. Da mesma forma, *XXXXXXX* deve ser substituído pelo nome da classe.

Tabela 9 – Código-fonte base do método getInfo

```
public void getInfo(
            String pParametros,
            CachedRowSetImpl pCrs)
            throws SQLException {

      //Comment=> Monta a query para busca
      sQry = "SELECT * FROM TB_XXXXXXX ";
      If (pParametros.length() != 0) {
            sQry = sQry + " WHERE " + pParametros;
      }
      ;

      //Comment=> Recupera a conexão
      con = acesso.forneceConexao();

      //Comment=> Cria um Statement para manipular os dados do DB
      // através da conexão
      stmt = con.createStatement();

      //Comment=> Executa a query e popula o CachedRowSet
      pCrs.populate(stmt.executeQuery(sQry));

      // Comment=> Fecha o statement e a conexão
      con.close();

}
```

Quando a classe desempenha o papel de Todo em uma associação tipo Todo-Parte, o método *getInfoPorID* é diferente. A razão é que, nesta particular implementação, a classe Todo também seleciona as informações de suas partes, conforme tabela 10:

Tabela 10 – Código-fonte base do método getInfoPorID para classes Todo

```
public void getInfoPorID(
            Integer pId,
            CachedRowSetImpl pCrsPedido,
            CachedRowSetImpl pCrsItensPedido)
            throws MDAExceptions, SQLException {

      sQry = "SELECT * FROM TB_XXXXXTODO ";
      sQry = sQry + " WHERE ID = " + pId.intValue();
      con = acesso.forneceConexao();
      stmt = con.createStatement();

      pCrsPedido.populate(stmt.executeQuery(sQry));
      Empacotador.desempacotar(this, pCrsPedido, 1);
      if (id.intValue() == 0) {
            throw new MDAExceptions("XXXXXTODO não encontrado!");
      }
      ;
      con.close();

      //Agora recupera as PARTES
      //do TODO em questão.
      sQry = " IDXXXXXTODO = " + pId.intValue();

      oXXXXXPARTE = new XXXXXPARTE();
      oXXXXXPARTE.getInfo(sQry, pCrsItensPedido);

}
```

Uma ferramenta de transformação pode fazer a configuração final do código apresentado nas tabelas 8,9 e 10. No pior dos casos, precisa levar em conta se uma classe está em uma associação Todo-Parte (para selecionar o código-fonte base do método *getInfoPorID*) e fornecer os nomes da classe Todo e da classe Parte. Logo,

as tabelas 8,9 e 10 são exemplos de código-fonte que - mediante pequena configuração - implementam soluções de caixa-cinza.

A.2 Fornecimento de conexões com o banco de dados (caixa preta)

Fornecimento de conexões com o banco de dados é um caso que pode ser facilmente resolvido sem que o desenvolvedor precise se preocupar em modelar um componente específico para isto. O código a seguir mostra a classe utilizada neste projeto para fornecer conexões (tabela 11) e a classe com os parâmetros para a conexão (tabela 12). Estes são exemplos de código-fonte que implementam soluções de caixa preta.

Tabela 11 - Classe para fornecimento de conexões DB

```
package mda.negocio;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class Acesso
{
   private ArgumentosConn DArguments = null;
   private Statement Stmt = null;
   private Connection con;

   public Acesso()
   {
      DArguments = new ArgumentosConn();
   }

   public Acesso(ArgumentosConn pDArgumetos)
   {
      DArguments = new
      ArgumentosConn(pDArgumetos.getDbDriver(),
      pDArgumetos.getDbLocation() ,pDArgumetos.getDbUrl());
   }

   public Connection forneceConexao() throws SQLException
   {
```

```
      try
      {
      Class.forName(this.DArguments.getDbDriver()).newInstance();
      con=DriverManager.getConnection(this.DArguments.getDbUrl());
      return con;
      }
      catch(SQLException e)
      {
            throw new SQLException(e.toString());
      }
      catch(Exception e1)
      {
            throw new SQLException(e1.toString());
      }
      }
}
```

Tabela 12 - Classe com parâmetros para a conexão DB

```
package mda.negocio;

public class ArgumentosConn
{
   private String DbDriver;
   private String DbLocation;
   private String DbUrl;

   public ArgumentosConn()
   {
      DbDriver ="sun.jdbc.odbc.JdbcOdbcDriver";
      DbLocation="conexaoTeste";
      DbUrl="jdbc:odbc:" + DbLocation;
   }

   public ArgumentosConn(String pDbDriver,
                         String pDbLocation,
                         String pDbUrl)
   {
      DbDriver = pDbDriver;
      DbLocation= pDbLocation;
      DbUrl = pDbUrl;
   }
```

```
    public String getDbDriver() {
       return DbDriver;
    }

    public void setDbDriver(String dbDriver) {
       DbDriver = dbDriver;
    }

    public String getDbLocation() {
       return DbLocation;
    }

    public void setDbLocation(String dbLocation) {
       DbLocation = dbLocation;
    }

    public String getDbUrl() {
       return DbUrl;
    }

    public void setDbUrl(String dbUrl) {
       DbUrl = dbUrl;
    }
}
```

Apêndice B - Representação em XMI de um modelo UML

A representação XMI foi gerada a partir de um modelo UML de uma única classe Cliente, representada na figura 25:

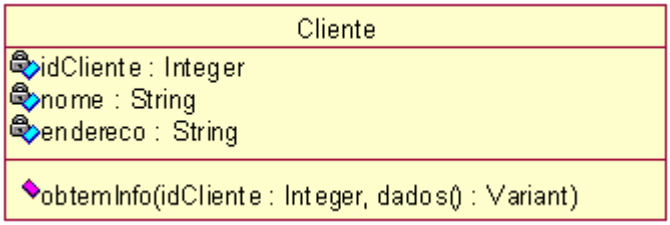

Figura 25 - Classe Cliente com três atributos e um método

O código XMI gerado está na tabela 13:

Tabela 13 - Representação XMI gerada a partir do modelo da figura 25

```
<?xml version = '1.0' encoding = 'ISO-8859-1'?>
<!-- <!DOCTYPE XMI SYSTEM 'UMLX13-11.dtd' > -->
<XMI xmi.version = '1.1'
xmlns:UML='href://org.omg/UML/1.3' timestamp = 'Sat Oct 22 14:31:30
2005' >
 <XMI.header>
  <XMI.documentation>
   <XMI.exporter>Unisys.JCR.2</XMI.exporter>
   <XMI.exporterVersion>1.3.6</XMI.exporterVersion>
  </XMI.documentation>
  <XMI.metamodel xmi.name = 'UML' xmi.version = '1.3'/>
 </XMI.header>
<XMI.content>
<!-- =================== classeCliente    [Model] ============= -->
<UML:Model xmi.id = 'G.0'
  name = 'classeCliente' visibility = 'public' isSpecification =
'false'
  isRoot = 'false' isLeaf = 'false' isAbstract = 'false' >
  <UML:Namespace.ownedElement>
    <!-- ====== classeCliente::Cliente    [Class] ========== -->
    <UML:Class xmi.id = 'S.294.1431.16.1'
      name = 'Cliente' visibility = 'public' isSpecification =
'false'
      isRoot = 'true' isLeaf = 'true' isAbstract = 'false'
```

```
isActive = 'false'
namespace = 'G.0' >
<UML:Classifier.feature>
  <!-- == classeCliente::Cliente.idCliente    [Attribute] = -->
  <UML:Attribute xmi.id = 'S.294.1431.16.2'
    name = 'idCliente' visibility = 'private'
           isSpecification = 'false'
    ownerScope = 'instance'
    changeability = 'changeable' targetScope = 'instance'
    type = 'G.1' >
    <UML:StructuralFeature.multiplicity>
      <UML:Multiplicity >
        <UML:Multiplicity.range>
          <UML:MultiplicityRange xmi.id = 'id.2951631.1'
            lower = '1' upper = '1' />
        </UML:Multiplicity.range>
      </UML:Multiplicity>
    </UML:StructuralFeature.multiplicity>
    <UML:Attribute.initialValue>
      <UML:Expression
        language = '' body = '' />
   </UML:Attribute.initialValue>
  </UML:Attribute>
  <!-- == classeCliente::Cliente.nome    [Attribute] == -->
  <UML:Attribute xmi.id = 'S.294.1431.16.3'
    name = 'nome' visibility = 'private'
           isSpecification = 'false'
    ownerScope = 'instance'
    changeability = 'changeable' targetScope = 'instance'
    type = 'G.2' >
    <UML:StructuralFeature.multiplicity>
      <UML:Multiplicity >
        <UML:Multiplicity.range>
          <UML:MultiplicityRange xmi.id = 'id.2951631.2'
            lower = '1' upper = '1' />
        </UML:Multiplicity.range>
      </UML:Multiplicity>
    </UML:StructuralFeature.multiplicity>
    <UML:Attribute.initialValue>
      <UML:Expression
        language = '' body = '' />
    </UML:Attribute.initialValue>
  </UML:Attribute>
  <!-- = classeCliente::Cliente.endereco    [Attribute] == -->
  <UML:Attribute xmi.id = 'S.294.1431.16.4'
    name = 'endereco' visibility = 'private'
           isSpecification = 'false'
```

```
  ownerScope = 'instance'
  changeability = 'changeable' targetScope = 'instance'
  type = 'G.2' >
  <UML:StructuralFeature.multiplicity>
    <UML:Multiplicity >
      <UML:Multiplicity.range>
        <UML:MultiplicityRange xmi.id = 'id.2951631.3'
          lower = '1' upper = '1' />
      </UML:Multiplicity.range>
    </UML:Multiplicity>
  </UML:StructuralFeature.multiplicity>
  <UML:Attribute.initialValue>
    <UML:Expression
      language = '' body = '' />
  </UML:Attribute.initialValue>
</UML:Attribute>
<!-- = classeCliente::Cliente::obtemInfo    [Operation] = -->
<UML:Operation xmi.id = 'S.294.1431.16.5'
  name = 'obtemInfo' visibility = 'public'
         isSpecification = 'false'
  ownerScope = 'instance'
  isQuery = 'false'
  concurrency = 'sequential'
  isRoot = 'false'
  isLeaf = 'false'
  isAbstract = 'false' specification = '' >
  <UML:BehavioralFeature.parameter>
    <UML:Parameter xmi.id = 'XX.22.1431.16.3'
      name = 'idCliente'
      visibility = 'public'
      isSpecification = 'false'
      kind = 'inout'
      type = 'G.1' >
      <UML:Parameter.defaultValue>
        <UML:Expression
          language = '' body = '' />
      </UML:Parameter.defaultValue>
    </UML:Parameter>
    <UML:Parameter xmi.id = 'XX.22.1431.16.4'
      name = 'dados()'
      visibility = 'public'
      isSpecification = 'false'
      kind = 'inout'
      type = 'G.3' >
      <UML:Parameter.defaultValue>
        <UML:Expression
          language = '' body = '' />
      </UML:Parameter.defaultValue>
```

```
            </UML:Parameter>
          </UML:BehavioralFeature.parameter>
        </UML:Operation>
      </UML:Classifier.feature>
    </UML:Class>
    <!-- ===== Integer    [DataType] ==================== -->
    <UML:DataType xmi.id = 'G.1'
      name = 'Integer' visibility = 'public'
             isSpecification = 'false'
      isRoot = 'false' isLeaf = 'false' isAbstract = 'false' />
    <!-- ==== String    [DataType] ==================== -->
    <UML:DataType xmi.id = 'G.2'
      name = 'String' visibility = 'public' isSpecification = 'false'
      isRoot = 'false' isLeaf = 'false' isAbstract = 'false' />
    <!-- =========== Variant    [DataType] ========= -->
    <UML:DataType xmi.id = 'G.3'
      name = 'Variant' visibility = 'public'
           isSpecification = 'false'
      isRoot = 'false' isLeaf = 'false' isAbstract = 'false' />
  </UML:Namespace.ownedElement>
</UML:Model>
<UML:TaggedValue xmi.id = 'XX.22.1431.16.2'
  tag = 'persistence' value = 'transient'
  modelElement = 'S.294.1431.16.1' />
<UML:Diagram xmi.id = 'S.294.1431.16.8'
  name = 'Main' toolName = 'Rational Rose 98'
  diagramType = 'ClassDiagram' style = ''
  owner = 'G.0' >
  <UML:Diagram.element>
    <UML:DiagramElement xmi.id = 'XX.22.1431.16.5'
      geometry = ' 512, 352, 386, 336,'
      style = 'Font.Blue= 0,
               Font.Green= 0,
               Font.Red= 0,
               Font.FaceName=Arial,
               Font.Size= 10,
               Font.Bold=0,
               Font.Italic=0,
               Font.Strikethrough=0,
               Font.Underline=0,
               LineColor.Blue= 51,
               LineColor.Green= 0,
               LineColor.Red= 153,
               FillColor.Blue= 204,
               FillColor.Green= 255,
               FillColor.Red=255,
               FillColor.Transparent=1,
               AutomaticResize=1,
               ShowAllAttributes=1,
               ShowAllOperations=1,
               ShowOperationSignature=0,
```

```
                SuppressAttributes=0,
                SuppressOperations=0,'
        subject = 'S.294.1431.16.1' />
    </UML:Diagram.element>
</UML:Diagram>
</XMI.content>
</XMI>
```

Apêndice C - Especificação do software exemplo

Para demonstrar o papel das soluções de caixa branca, preta e cinza em um software, essa dissertação apresenta uma implementação como exemplo. É um sistema de vendas varejo, com diferentes atores com diferentes níveis de acesso.

C.1 Visão do sistema

A EmpresaX (uma empresa fictícia) é uma indústria que atua no ramo de suprimentos para escritórios. A empresa não atua no mercado varejo; vende apenas no atacado para uma rede de distribuidores.

A EmpresaX decide criar um sistema de vendas varejo que possa ser utilizado pelos seus distribuidores e que funcione como um atrativo, visando aumentar a sua rede de distribuição O sistema tem apenas uma única base de dados mas é multi-distribuidor.

Cada distribuidor pode utilizar as funcionalidades do sistema a partir de seus próprios *websites*, via *webservices*, criando nos clientes a sensação de operarem o sistema do distribuidor.

A EmpresaX respeita a confidencialidade das carteiras de clientes dos seus distribuidores. Os clientes são cadastrados via um distribuidor, de tal forma que outros distribuidores não possam acessar o mesmo cliente. Se um cliente fizer compras em dois distribuidores distintos, ele terá dois cadastros distintos apesar do sistema ser único.

Para que essa confidencialidade possa ser implementada, cada transação disparada por um cliente do distribuidor precisa conter o ID e código de acesso do distribuidor. Esta autorização é transparente para o usuário: o *website* do distribuidor fornece estes parâmetros.

Há casos onde não é necessário identificar o distribuidor. A consulta ao catálogo de produtos da EmpresaX é livre e não precisa de identificação do distribuidor.

Há também o caso de transações exclusivas dos distribuidores, como a consulta de clientes por nome. Nesta situação o distribuidor precisa fornecer seu ID e uma senha. Cabe ao sistema validar o distribuidor, permitindo ou não o acesso.

Do ponto de vista do cliente, a operação com o *site* do distribuidor é *B2C* (*business to customer*). O próprio cliente faz seu cadastramento no *site* do distribuidor, de tal forma que o cliente não percebe tratar-se do sistema da EmpresaX. Os clientes acessam a funcionalidade de cadastramento através do *website* de algum dos distribuidores da EmpresaX, fornecem seus dados e recebem um ID e senha de cliente.

C.2 Atores do sistema

Os principais atores do sistema – aqueles que constam da porção implementada do sistema – são:

- Gerente

  Cadastra os produtos e os distribuidores da EmpresaX.

- Distribuidor

  Comercializa os produtos da EmpresaX entre os seus clientes. Os distribuidores atuam na *internet* utilizando as funcionalidades disponíveis no sistema da EmpresaX, mas o fazem através de seus próprios *websites*.

- Cliente cadastrado

  Compra os produtos da EmpresaX através do *website* do seu distribuidor. O próprio cliente faz a manutenção de seu cadastro no distribuidor.

- Cliente não cadastrado

  O único acesso permitido é o cadastramento de cliente. O próprio cliente faz o seu cadastramento.

C.3 Diagrama de casos de uso

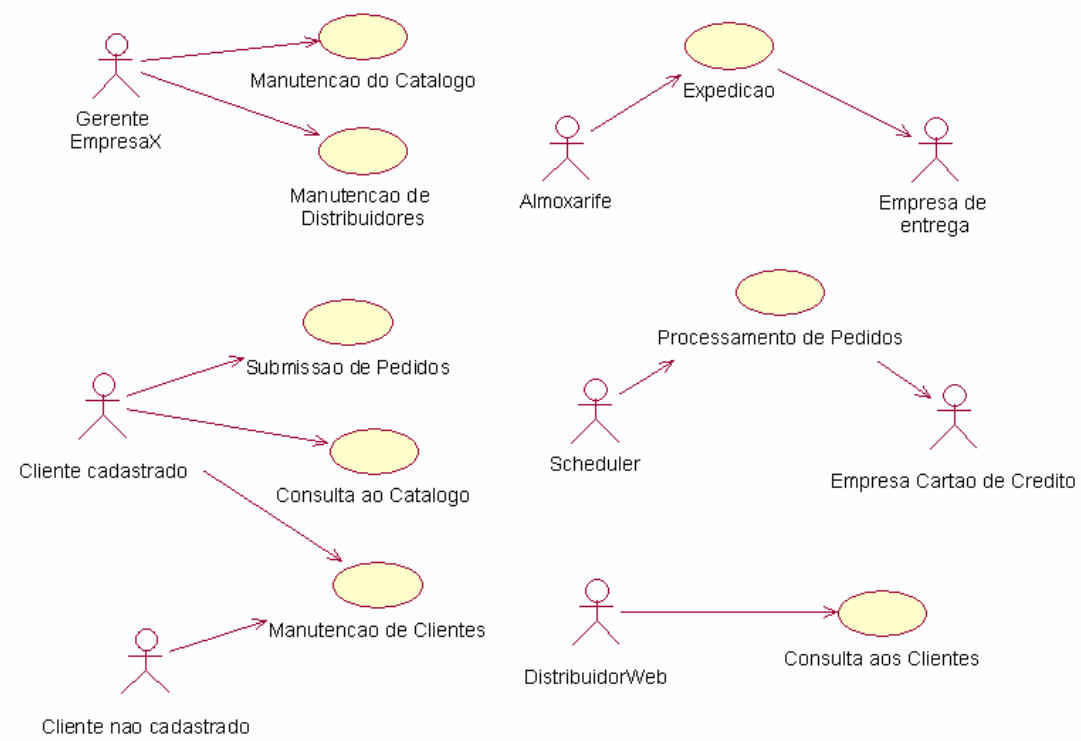

Figura 26 -Diagrama de Casos de Uso da aplicação exemplo

C.4 Especificação de casos de uso

Apenas alguns Casos de Uso são implementados na aplicação-exemplo desenvolvida para esta dissertação. A seguir estão as especificações dos Casos de Uso Manutenção de Clientes, Consulta aos Clientes e Submissão de Pedidos.

Caso de Uso: Manutenção de Clientes

Descrição

Este caso de uso refere-se à criação, alteração ou exclusão de um usuário válido no sistema.

Atores

Cliente cadastrado;

Cliente não cadastrado

Fluxo de eventos

Caminho Primário – *Inserção de Cliente*

- O usuário acessa o *site* de algum dos distribuidores da EmpresaX

  A EmpresaX disponibiliza seu sistema para todos os seus Distribuidores. Como o sistema é segmentado por distribuidor, o usuário sempre acessa o sistema através do *site* de algum dos distribuidores cadastrados.

- O usuário acessa o cadastramento de clientes
- O usuário cadastra seus dados básicos e conta de e-mail, mas não cadastra ID.

  O cliente cadastra seus dados, mas o sistema é quem gera o ID on-line para o cliente.

- O usuário seleciona a inclusão após cadastrar seus dados
- O sistema verifica se o distribuidor é válido no sistema

  O *site* do distribuidor deve fornecer o ID e o código de acesso do distribuidor para validação.O sistema somente aceita a inclusão de um cliente caso o distribuidor escolhido pelo cliente seja válido.

- Após o cadastramento, um ID de cliente será gerado e exibido para o cliente

  Ao término do cadastramento o sistema fornece o ID e a senha do cliente.

Caminho Alternativo 1 – *Consulta de Cliente*

- Cliente acessa o *website* do seu distribuidor e solicita consulta de Cliente
- Cliente fornece o seu ID e senha
- O sistema checa se o Distribuidor é válido

  O sistema verifica se o ID e código de acesso de distribuidor são válidos no sistema

- O sistema checa se o ID e senha do cliente são válidos
- O sistema recupera os dados cadastrais do cliente

O sistema procura o ID do cliente e verifica se ele pertence ao distribuidor em questão

- Os dados cadastrais são exibidos para o cliente

Precondições

- Conta de e-mail

  Cliente da EmpresaX precisa ter uma conta de e-mail

Pós-Condições

- Cliente cadastrado / alterado / excluído

Caso de Uso: Consulta aos Clientes

Descrição

Este caso de uso permite a consulta dos clientes de um determinado Distribuidor.

Atores

Distribuidor.

Fluxo de eventos

Caminho Primário – *Consulta de Cliente por ID*

- O Distribuidor escolhe a consulta por ID

  O distribuidor fornece seu próprio ID e senha, juntamente com um ID de cliente.

- O sistema verifica os dados do distribuidor

  O sistema verifica se a senha e ID do distribuidor são válidos e se o distribuidor está ativo no sistema

- O sistema recupera as informações do cliente

  O sistema procura pelo cliente com o ID fornecido e verifica se ele está atrelado ao distribuidor em questão.

- Os dados cadastrais do cliente são exibidos para o distribuidor

Caminho Alternativo 1 – *Consulta de cliente por nome*

- Distribuidor fornece um nome (ou parte de um nome) de cliente

  O distribuidor fornece seu próprio ID e senha, juntamente com uma parte de um nome de cliente.

- O sistema verifica os dados do distribuidor

  O sistema verifica se a senha e ID do distribuidor são válidos e se o distribuidor está ativo no sistema

- O sistema recupera as informações do cliente

  O sistema recupera todos os clientes do distribuidor em questão que se encaixem no nome fornecido

- Uma lista de cliente é exibida para o distribuidor

Precondições

- Distribuidor válido

  O distribuidor precisa existir e estar ativo no sistema

Pós-Condições

- Consulta realizada

Caso de Uso: Submissão de Pedidos

Descrição

Este caso de uso refere-se à criação, exclusão, alteração ou consulta de pedidos.

Atores

Cliente cadastrado.

Fluxo de eventos

Caminho Primário – *Abertura de pedido*

- O cliente cadastrado acessa a submissão de pedidos
- O cliente inclui itens de pedido a partir de uma lista de produtos disponíveis
- O cliente salva o pedido

- O sistema atribui um ID ao pedido e o exibe para o cliente

  O sistema cria o pedido e inclui todos os itens de pedido selecionados

Caminho Alternativo 1 –*Consulta de pedidos por ID*

- O usuário acessa a submissão de pedidos
- O cliente fornece um ID de pedido e seleciona a busca por ID
- O sistema verifica se o pedido existe para o cliente em questão
- O sistema recupera o pedido e todos os itens de pedido relacionados
- O sistema exibe as informações de pedido e itens de pedido para o cliente

Caminho Alternativo 1 –*Consulta de pedidos por ID*

- O usuário acessa a submissão de pedidos
- O sistema busca todos os IDs de pedidos existentes para o cliente em questão

  O sistema busca apenas os pedidos, sem considerar os itens de cada pedido
- O sistema exibe todos os IDs de pedidos existentes para o cliente em questão

Precondições

- Cliente cadastrado e distribuidor válidos no sistema

  Tanto o cliente quanto o distribuidor precisam ser válidos e estarem ativos no sistema

Pós-Condições

- Pedido cadastrado / consultado

Lista de Referências

AGUIAR, A; SOUZA; A, PINTO, A. Use-Case Controller. In: Sixth European Conference on Pattern Languages of Programs (EuroPLoP 2001). **Proceedings…** Irsee, Alemanha. Julho de 2001.

ALMEIDA, J; VAN SINDEREN, M; PIRES, L. The role of the RM-ODP Computational Viewpoint Concepts in the MDA approach. In: 1st European Workshop on Model-Driven Architecture with Emphasis on Industrial Applications (MDA-IA 2004). **Proceedings…** Holanda.. p. 43-51. Março 2004

ALMEIDA, J; DIJKMAN, R; VAN SINDEREN, M; PIRES, L. On the Notion of Abstract Platform in MDA Development. In: 8th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2004). **Proceedings...** Monterey, Estados Unidos. 20-24 Setembro 2004. p. 253-263.

ALUR, D; CRUPI, J; MALKS, D. Padrões da camada de integração. In:_____. **Core J2EE Patterns:** As melhores práticas e estratégias de design. Tradução Altair Dias Caldas de Moraes; Claúdio Belleza Dias; Guilherme Dias Caldas de Moraes. Rio de Janeiro: Campus, 2002. p. 344-372.

ASTESIANO, E; REGGIO G. An Attempt at Analysing the Consistency Problems in the UML from a Classical Algebraic Viewpoint. In: 16th International Workshop on Algebraic Development Techniques. **Proceedings…**Frauenchiemsee, Alemanha. Setembro 24-27, 2002. p. 56-81.

ATKINSON, C; KÜHNE, T; Model-Driven Development: A Metamodel Foundation. **IEEE SOFTWARE**, vol20, issue5, p. 36-41, Set/Out 2003.

BARESI, L; PEZZÈ, M. Formal Interpreters for Diagram Notations. **ACM Transactions on Software Engineering and Methodology (TOSEM)**. vol.14, Issue 1, p. 42-84, Janeiro 2005.

BELIX, J; FERNANDES, S; MELNIKOFF, S; SPINA, E. Effort Reducing in Software Modeling on MDA Approach. **WSEAS TRANSACTIONS ON COMPUTER**. vol 4, Issue 6, p. 621-626. Junho/2005.

BETTIN, J; VÖLTER, M. Patterns for Model-Driven Software-Development. Versão 1.5, 08/junho/2005. Disponível em <http://hillside.net/europlop/Papers/wwd/D3.pdf>. Acessado em 20/setembro/2005.

BETTIN, J. Model-Driven Software Development. MDA Journal. Abril, 2004. Disponível em <http://www.bptrends.com/publicationfiles/04-04%20COL%20MDSD%20Frankel%20-%20Bettin%20-%20Cook.pdf>. Acessado em 30/agosto/2004.

_____. Model-Driven Software Development: An Emerging Paradigm for Industrialized Software Asset Development. Junho 2004. Disponível em < http://www.softmetaware.com/mdsd-and-isad.pdf>. Acessado em 10/dezembro/2004.

BILLIG, A; BUSSE S; LEICHER, A; SÜß J. Plataform Independent Model Transformation Based on Triple. In: 5th ACM/IFIP/USENIX International Conference on Middleware. **Proceedings…** Toronto, Canadá, Outubro 18-22, 2004, p. 493-511.

BONIFATI, A; CERI S. Comparative analysis of five XML query languages. **ACM SIGMOD**. vol 29, issue 1, p 68-79, Março 2000.

BOOCH, G; RUMBAUGH, J; JACOBSON, I. **The Unified Modeling Language User Guide** – Second Edition**.** NJ – EUA. Addison-Wesley - Object Technology Series. 2005. 475p.

BRANDON, D. CRUD Matrices for Detailed Object Oriented Design. **Journal of Computing Sciences in Colleges**. vol18, Issue 2, p. 306-322. Dezembro, 2002.

BROWN, A. An introduction to Model Driven Architecture Part I: MDA and today's systems. IBM / The Rational Edge. 12/janeiro/2004. Disponível em < http://www-

128.ibm.com/developerworks/rational/library/content/RationalEdge/feb04/3100.pdf> Acessado em 21/07/2005.

BROWN, A; CONALLEN, J. An introduction to Model Driven Architecture Part II: Lessons from the design and use of an MDA toolkit. IBM / The Rational Edge. 15/abril/2005. Disponível em <http://www-128.ibm.com/developerworks/rational/library/apr05/brown/brown.pdf> Acessado em 21/07/2005.

_____. An introduction to Model Driven Architecture Part III: How MDA Affects the Iterative Development Process. IBM / The Rational Edge. 15/maio/2005. Disponível em <http://www-128.ibm.com/developerworks/rational/library/may05/brown/> Acessado em 21/07/2005.

BURKS - the Brighton University Resource Kit for Students of Computer Science. University of Brighton. Agosto/2001. Disponível em < http://burks.brighton.ac.uk/burks/language/prolog/pms/node2.htm> Acessado em 01/01/2005.

BUSCHMANN F; MEUNIER R; ROHNERT H; SOMMERLAD P; STAL M. **Pattern-Oriented Software Architecture.** A System of Patterns. Chichester, UK, John Wiley & Sons Ltd., 1996. 467p.

CAPLAT, G; SOURROUILLE, J. Model Mapping Using Formalism Extensions. **IEEE Software**, vol22, no. 2, p. 44-51, Março/Abril 2005.

CLEMENTS P; NORTHROP L. **Software Product Lines : Practices and Patterns**. Addison-Wesley Professional, 1ª ed., Agosto/2001. 608p.

COLWELL B. Engineers, Programmers, and Black Boxes. **IEEE Computer Society**. **Computer.** vol38, nº 3, p. 8-11. Março 2005.

COENEN, F. Topics in Information Processing, 1999. Disponível em <http://www.csc.liv.ac.uk/~frans/OldLectures/2CS24/declarative.html>. Acessado em 01/01/2005.

CONALLEN J. Modeling Web Application Architectures with UML. **Communications of ACM**., vol42, nº 10, p 63-70. Outubro 1999.

CRUZ, C; RIBEIRO, U. **Metodologia Científica: Teoria e Prática**. 1ª Ed. Rio de Janeiro. Axcel Books do Brasil. 2003. 218p.

CZARNECKI, K; EISENECKER U. **Generative Programming: Methods, Tools, and Applications**. Addison-Wesley. 2000. 832p.

ENGELS, G; KUSTER, J. M.; GROENEWEGEN, L ;HECKEL, R. A Methodology for Specifying and Analyzing Consistency of Object-Oriented Behavioral Models. In: 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering. **Proceedings…**Viena, Austria. 2001, p.186-185.

ESTUBLIER, J; VEGA, G. Reuse and Variability in Large Software Applications. In: 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering. **Proceedings…**Lisboa, Portugal. . 2005, p. 316 – 325.

EVANS, G. Getting From Use Cases to Code Part 1: Use-Case Analysis. IBM – The Rational Edge. Julho 2004. Disponível em <http://www-106.ibm.com/developerworks/rational/library/content/RationalEdge/jul04/5383.pdf>. Acessado em 03/Fev/2005.

FERNANDES, S; BELIX, J; MELNIKOFF, S; SPINA, E. Confronting Antagonistic Views of Software Design. In: WSEAS International Conference on SYSTEMS SCIENCE and ENGINEERING (ICOSSE 2005). **Proceedings...** Rio de Janeiro, Brasil. Abril/2005, p. 72-76.

FUNG, K. **XSLT:** Interagindo com XML e HTML. Editora Ciência Moderna. Rio de Janeiro, Brasil. 2001. 409p.

FRANÇOIS, A. Software Architecture for Computer Vision: Beyond Pipes and Filters. Julho, 2003. Disponível em <http://iris.usc.edu/~afrancoi/pdf/sacv-tr.pdf>. Acessado em 06/02/2005.

GREENFIELD, J. Software Factories vs. MDA. [Depoimento ao *website* TheServerSide.com]. TheServerSide. 03/janeiro/2005. Disponível em: <http://www.theserverside.net/tss?service=direct/0/NewsThread/threadViewer.bookmark.link&sp=l30082>. Acesso em: 17/09/2005.

GREENFIELD, J; SHORT K; COOK, S; KENT, S. **Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools**. Wiley Publishing, Inc., 2004. 666p.

HAREL, D; RUMPE, B. Meaningful Modeling: What's the Semantics of "Semantics"? **COMPUTER**. IEEE Computer Society Press. vol37, Issue10, p.64-72, Outubro 2004.

HUDAK, P. Conception, Evolution, and Application of Functional Programming Languages. **ACM Computing Surveys**, vol21, nº 3, p. 359-411, Setembro 1989.

HUNTER, J. XML – Application Programming Interfaces (APIs). IBM DeveloperWorks. 12/outubro/2002. Disponível em <http://www.idevelopment.info/data/Programming/java/xml/XML_APIs.pdf> Acessado em 03/01/2005.

INTERNATIONAL ENGINEERING CONSORTIUM (IEC). **Specification and Description Language (SDL)** 1996. Disponível em <http://www.iec.org/online/tutorials/sdl/index.html>. Acessado em 20/janeiro/2006.

IBM-RATIONAL. **The Rational Unified Process (RUP)**. Versão 2003.06.13. 2003

JARZABEK, S; ZHANG, H; SWE, S. XVCL: A Tutorial. In: 14th International Conference on Software Engineering and Knowledge Engineering. **Proceedings…** Ischia, Itália, 15-19 Julho 2002. p.341-349.

JOHNSON, R. Frameworks = (Components + Patterns). **Communications of the ACM**. vol40, n. 10, p. 39-42, 1997.

KAY, M. **XSLT Referência do Programador**. 2ª Ed. Editora Alta Books. Rio de Janeiro, Brasil. 2002. 667p.

KENT, S; EVANS, A; RUMPE, B. UML Semantics FAQ. In: 13th European Conference on Object-Oriented Programming (ECOOP'99). **Proceedings…**vol1743. 14-18/junho. 1999. p 33-54.

KLEPPE, A; WARMER J; BAST W. **MDA Explained: The Model Driven Architecture Practice and Promise**. 2ª Ed. EUA. Addison-Wesley - Object Technology Series. 2003. 169p.

LARMAN, C. UML Interaction Diagrams. In:_____. **Applying UML and Patterns**: An Introduction to Object-Oriented Analysis and Design and Iterative Development. Pearson Education Inc., 3ªed. 2005 p.221-247.

LONG, Q; LIU, Z; LI, X; JIFENG, H. Consistent Code Generation From UML Models. In: The 2005 Australian Software Engineering Conference (ASWEC'05). **Proceedings...** Austrália, 29/março a 01/Abril/2005, p.23-30.

MARCONI, M; LAKATOS; E. **Metodologia Científica**. 4ª Ed. São Paulo. Atlas, 2004. 305p.

MCLAUGHLIN, B. All About JAXP. IBM DeveloperWorks. Novembro/2000. Disponível em < http://www.idevelopment.info/data/Programming/java/xml/All_About_JAXP.pdf> Acessado em 27/12/2004.

MEEK, B. The Static Semantics File. **ACM SIGPLAN Notices**. vol25, Issue 4, p. 33-42. Abril 1990.

MELLOR, S; Make Models Be Assets. **Communications of the ACM**, vol45, nº 11, p. 76-78, Novembro 2002.

MENDES, A. **Arquitetura de Software: Desenvolvimento Orientado para Arquitetura**. Editora Campus, 2002. 212p.

MORENO, N; VALLECILLO, A. What do we do with re-use in MDA? In: Second European Workshop on Model Driven Architecture (MDA) with an emphasis on

Methodologies and Transformations. **Proceedings…** Canterbury, Inglaterra. 7 e 8 de Setembro/2004.

OBJECT MANAGEMENT GROUP (OMG). **Model Driven Architecture (MDA) – Document number ormsc/2001-07-01**. Needham, Mass., EUA, Julho 2001. Disponível em <http://www.omg.org/docs/ormsc/01-06-01.pdf>. Acessado em 12/dezembro/2003.

_____. **Request For Proposal: MOF 2.0 Query / Views / Transformations**. Needham, Mass., EUA, Abril 2002. Disponível em < http://www.omg.org/docs/ad/02-04-10.pdf>. Acessado em 26/dezembro/2004.

_____. **XML Metadata Interchange (XMI) Specification**. Needham, Mass., EUA, maio 2003. Disponível em <http://www.omg.org/docs/formal/03-05-02.pdf >. Acessado em 10/dezembro/2004.

_____. **MDA Guide Version 1.0.1**. Needham, Mass., EUA, julho 2003. Disponível em <http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf>. Acessado em 08/dezembro/2004.

_____. **UML 2.0 OCL Specification**. Needham, Mass., EUA, Novembro 2003. Disponível em < http://www.omg.org/docs/ptc/03-10-14.pdf>. Acessado em 19/julho/2005.

_____. **UML 2.0 Infrastructure Specification**. Needham, Mass., EUA, Dezembro 2003. Disponível em < http://www.omg.org/docs/ptc/03-09-15.pdf>. Acessado em 09/dezembro/2004.

_____. **PIM and PSM for Software Radio Components**. Needham, Mass., EUA, Maio 2004. Disponível em <http://www.omg.org/cgi-bin/doc?dtc/2004-05-04>. Acessado em 21/setembro/2005.

_____. **UML 2.0 Superstructure Specification**. Needham, Mass., EUA, Outubro 2004. Disponível em <http://www.omg.org/cgi-bin/doc?ptc/04-10-02>. Acessado em 19/maio/2005.

PÉREZ-MARTINEZ, J. Heavyweight extensions to the UML metamodel to describe the C3 architectural style. **ACM SIGSOFT Software Engineering Notes**. vol28, Issue 3, pg 1-6. Maio 2003.

PIZKA, M; BAUER, A. A Brief Top-Down and Bottom-Up Philosophy on Software Evolution. In: 7th Int. Workshop on Principles of Software Evolution (IWPSE'04). **Proceedings…** Kyoto, Japan. Setembro 2004, p. 131-136.

PUTMAN, J. **Architecting with RM-ODP**. Prentice Hall PTR. 2001. 834p.

REED, P. Acrescentando Funcionalidade Adicional aos Recursos da Linguagem de Definição de Dados Rational Rose. In:_____. **Desenvolvendo Aplicativos com Visual Basic e UML.** Tradução Mário Moro Fecchio. Makron Books do Brasil Editora ltda, 2000. p. 379-387.

REGGIO, G; CERIOLI, M; ASTESIANO, E. Towards a Rigorous Semantics of UML Supporting its Multiview Approach. In: 3rd International Conference on the Unified Modeling Language (UML'2000) - Workshop on Dynamic Behavior in UML Models. **Proceedings…** York, Reino Unido. 2-3 Outubro. 2000. p.171-186.

RUMPE, B. A Note on Semantics with an Emphasis on UML. In: Second ECOOP Workshop on Precise Behavioral Semantics. **Proceedings…** Bruxelas, Bélgica. 24/Julho, 1998. p.177-197.

SENDALL, S; KOZACZYNSKI, W. Model transformation: the heart and soul of model-driven software development. **IEEE SOFTWARE**, vol20, Issue5, pg 42-45, Set/Out 2003.

SCHATTKOWSKI, T. UML 2.0 - overview and perspectives in SoC design. In: Design, Automation and Test in Europe (DATE05). **Proceedings…** Munique, Alemanha. Março, 2005, p.832 - 833

SIEGEL, J; OMG STAFF STRAGEGY GROUP. Developing in OMG's Model-Driven Architecture. **Object Management Group White Paper**. Revisão 2.6. Novembro, 2001. Disponível em <URL: http://www.omg.org/cgi-bin/doc?omg/2001-12-01>. Acessado em 12/12/2003.

SOMMERVILLE, I. **Engenharia de Software**. Tradução André Maurício de Andrade Ribeiro. Addison-Wesley, Inc. 6ª ed. 2004, p.260.

THOMAS, D; BARRY, B. Model Driven Development – The Case for Domain Oriented Programming. In: 18th annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'03). **Proceedings…** Anaheim, Estados Unidos. 26-30 Outubro 2003, p.2-7.

WORLD WIDE WEB CONSORTIUM (W3C). **XSL Transformations (W3C Recommendation) Version 1.0**. Novembro 1999. Disponível em <http://www.w3.org/TR/xslt>. Acessado em 29/setembro/2004.

_____. **XML Path Language (XPath) Version 1.0** Novembro 1999. Disponível em <http://www.w3.org/TR/xpath>. Acessado em 05/dezembro/2004.

WHITTLE, J; ARAÚJO J; TOVAL A; ALEMÁN, J. Rigorously Automating Transformations of UML Behavior Models. In: 3rd International Conference on the Unified Modeling Language (UML'2000) - Workshop on Dynamic Behavior in UML Models. **Proceedings…** York, Reino Unido. 2-3 Outubro. 2000.

ZHANG, Y; XU, B. A Survey of Semantic Description Frameworks for Programming Languages. **ACM SIGPLAN Notices**. vol39, Issue 3, pgs 14-30. March 2004.