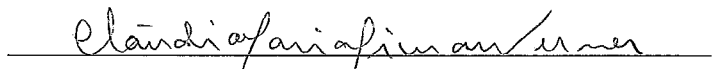
ODYSSEY-MDA: UMA ABORDAGEM PARA A TRANSFORMAÇÃO DE MODELOS

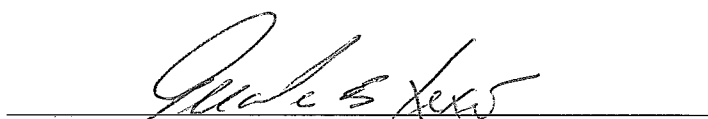
Natanael Elias Nascimento Maia

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

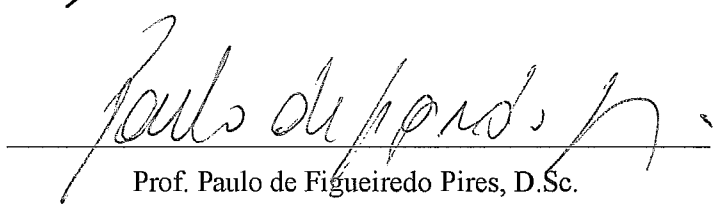
Aprovada por:



Prof. Cláudia Maria Lima Werner, D.Sc.



Prof. Geraldo Bonorino Xexéo, D.Sc.



Prof. Paulo de Figueiredo Pires, D.Sc.

RIO DE JANEIRO, RJ – BRASIL

MARÇO DE 2006

MAIA, NATANAEL ELIAS NASCIMENTO

Odyssey-MDA: Uma Abordagem Para a Transformação de Modelos [Rio de Janeiro] 2006

XIII, 105 p., 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2006)

Dissertação - Universidade Federal do Rio de Janeiro, COPPE

1. Transformação de modelos
2. Arquitetura orientada por modelos

I. COPPE/UFRJ II. Título (série)

Aos meus pais.

Agradecimentos

A Deus.

Aos meus pais e toda minha família, por terem sempre acreditado em mim, pelo carinho e pelo apoio que nunca me faltaram. Agradeço especialmente à minha mãe, Inês, por ter encontrado forças para me incentivar a prosseguir minha luta, mesmo nos momentos mais difíceis pelos quais passamos. Sua força foi muito importante. Te amo!

À professora Cláudia Werner, minha orientadora, por ter me aceito como seu orientando e membro deste excelente time, além da paciência e compreensão dos momentos difíceis pelos quais passei. Muito obrigado por compartilhar parte de seu conhecimento e suas idéias comigo, me mostrando o caminho da pesquisa.

À amiga Ana Paula Blois, que me orientou extra-oficialmente ao longo de todo este trabalho, por sua valiosa contribuição, incentivo e apoio na revisão deste texto. Você é um exemplo de força de vontade e dedicação.

Ao amigo Leonardo Murta, que muito me ajudou através de suas idéias, críticas, apoio e solidariedade no convívio do Grupo de Reuso e em todas as conferências que participamos.

Aos professores Paulo Pires e Geraldo Xexéo por aceitarem participar desta banca, agregando valor inestimável a este trabalho.

À professora e amiga Káthia Oliveira, por ter despertado em mim o interesse pela Engenharia de *Software* após suas cativantes aulas.

A todos os amigos do projeto *Odyssey* e do Laboratório de Engenharia de *Software* que tornaram minha passagem pelo Rio de Janeiro bastante agradável. Em especial, a atenção dos amigos Regiane, Aline, Luiz Gustavo, Marco Lopes, Alexandre, Mangan, Isabella, Beto, Cristine, Hamilton, Rafael, Artur e Sômulo.

Ao meu grande amigo Alex Marin, por toda a amizade e companheirismo nestes anos de luta, por agüentar meus momentos de mau humor e pelos momentos de diversão. Obrigado pela parceria!

À minha grande amiga Fabiana Actis, que apesar da distância, esteve sempre presente com seu carinho, apoio, compreensão e motivação. A sua companhia é muito importante em minha vida e me faz uma pessoa cada dia melhor. Te adoro!

Aos amigos Thiago Freitas, Leonardo Marques, Luiz Fernando, Karla Monreal, Gilsara Abreu, Maurício Dobke, Daniela Pascual, Thayse Dias, à toda a família Jensen, em especial às irmãs Verônica e Angélica, e claro, à família Campos, Mônica, Daniela, Maria Claret, Antônio Carlos e Júnior. Todos vocês são parte importantíssima de minha vida. Muito obrigado!

A todos os meus amigos, que direta ou indiretamente, me ajudaram e me apoiaram a trilhar este caminho.

À CAPES, pelo apoio financeiro ao desenvolvimento deste trabalho.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

ODYSSEY-MDA: UMA ABORDAGEM PARA A TRANSFORMAÇÃO DE MODELOS

Natanael Elias Nascimento Maia

Março / 2006

Orientadora: Cláudia Maria Lima Werner

Programa: Engenharia de Sistemas e Computação

O aumento da complexidade dos sistemas e o conseqüente aumento no custo necessário para desenvolver *software* fazem com que a busca por alternativas que possam reduzir este esforço no desenvolvimento se torne cada vez mais importante. Diversos avanços e técnicas surgiram com este objetivo, através do aumento do nível de abstração dos artefatos desenvolvidos. Uma dessas iniciativas é a Arquitetura Orientada por Modelos que permite a modelagem e a aplicação de transformações sobre os modelos, visando a obtenção do *software* de forma automatizada.

Este trabalho propõe uma abordagem prática e extensível para a definição de transformações de modelos, que permite a sua utilização em diversos ambientes de desenvolvimento de *software*. Estas transformações são definidas através de uma sintaxe *XML* e um conjunto de tratadores implementados em linguagem *Java*, e podem ser posteriormente executadas de maneira bidirecional, através da máquina de transformações implementada para apoiar o uso da abordagem proposta. É apresentado também, um exemplo de transformação de modelos para a plataforma *Enterprise JavaBeans*.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

ODYSSEY-MDA: AN APPROACH TO MODEL TRANSFORMATION

Natanael Elias Nascimento Maia

March / 2006

Advisor: Cláudia Maria Lima Werner

Department: Computer and Systems Engineering

The increase in system complexity and the consequent increase on necessary cost of software development make the search for alternatives that can reduce development effort extremely important. Several advances and techniques have appeared with this objective, by raising the level of abstraction of the developed artifacts. One of these initiatives is the Model Driven Architecture that allows the modeling and the application of transformations onto models aiming at obtaining software in an automatized way.

This work proposes a practical and extensible approach for the definition of model transformations that allows its use in various software development environments. These transformations are defined through XML syntax and a set of handlers implemented in Java language. They can be further executed in a bidirectional way, through the transformation machine implemented to support the use of the proposed approach. It is also presented an example of model transformation for the Enterprise JavaBeans platform.

Índice:

Índice de Figuras

Índice de Tabelas

Capítulo 1 – Introdução

1.1 – Motivação

Novas tecnologias surgem a todo o momento e as empresas, cada vez mais, necessitam de sistemas complexos para melhor conduzir e controlar seus negócios. Essa busca pela tecnologia e a sua adoção tem um custo associado e os investimentos podem ser grandes. Ao longo dos anos, o desenvolvimento de sistemas passou por diversas melhorias com o objetivo de reduzir os custos e aumentar a produtividade dos desenvolvedores.

Algumas das principais melhorias estão relacionadas ao aumento do nível de abstração necessário para projetar e implementar o *software*. Inicialmente, toda a codificação do *software* era realizada em linguagens muito próximas das linguagens de máquina, com pouca expressividade e de difícil manutenção. Com o aumento da complexidade dos sistemas desenvolvidos, essas linguagens foram se tornando insuficientes, o que resultou no surgimento de novas linguagens e paradigmas de programação com maior nível de abstração, como as linguagens orientadas a objeto (MELLOR *et al.*, 2004).

Através dessas linguagens, os desenvolvedores trabalham em níveis de abstração cada vez mais altos, e podem deixar algumas preocupações com relação às características de *hardware* das plataformas a cargo dos compiladores, responsáveis por traduzir automaticamente os programas escritos em alto nível, para representações em linguagens de máquina específicas para a plataforma de *hardware* desejada. De forma análoga, mais recentemente as preocupações relacionadas aos requisitos não-funcionais dos sistemas como, por exemplo, segurança, transação, distribuição ou persistência, foram deixando de ser desenvolvidas em cada projeto, para se tornarem serviços disponíveis e reutilizáveis em plataformas de *software*, conhecidos como *middlewares* (e.g.: *CORBA*, *EJB*) (SIEGEL, 2001).

Paralelamente, linguagens de modelagem foram surgindo (e.g.: *UML*), e tornaram possível a construção de modelos de projeto do *software* em desenvolvimento. Esses modelos permitiram a elaboração de representações dos requisitos do *software*, que podem ser visualizadas e manipuladas em tempo de análise e projeto do sistema, possibilitando uma melhor avaliação destes requisitos antes mesmo do código ser

implementado, quando possíveis mudanças envolvem menor esforço e custo (SOLEY, 2000).

Estes modelos surgiram, inicialmente, como artefatos de apoio ao processo de desenvolvimento, ajudando na comunicação dos desenvolvedores ou constituindo a documentação do *software*. No entanto, recentemente, surge a necessidade de mudar este paradigma no sentido de utilizar os modelos como artefatos centrais no desenvolvimento, a partir do qual o código é gerado. Este novo paradigma permite que os desenvolvedores concentrem seus esforços na elaboração de modelos que representam os conceitos e funcionalidades desejáveis no *software* projetado, ou seja, apenas os requisitos de negócio são modelados inicialmente. Estes modelos são então transformados, visando a aplicação dos conceitos providos pela plataforma tecnológica (*middleware*) escolhida. Este novo paradigma é conhecido como Arquitetura Orientada por Modelos, ou *MDA* (MELLOR *et al.*, 2004).

Neste cenário, o conceito chave para se obter os benefícios deste novo paradigma é a realização de transformações entre os modelos, o que possibilita uma maior automação no desenvolvimento e implementação do *software*, conseqüentemente, reduzindo esforço e custo.

Atualmente, existem várias abordagens para a realização de transformações entre modelos (OLDEVIK, 2004; JOUAULT *et al.*, 2005; KOZIKOWSKI, 2005; COMPUWARE, 2006; ECLIPSE, 2006; KENNEDY-CARTER, 2006), no entanto, a maioria dessas abordagens ainda apresenta deficiências com relação à interoperabilidade com ambientes de desenvolvimento, apoio à definição e execução de diversos tipos de transformações e a possibilidade de expansão de suas capacidades. Estas deficiências motivam a proposta deste trabalho.

1.2 – Objetivo

O objetivo deste trabalho é fornecer uma abordagem prática de apoio à definição e execução de transformações bidirecionais entre modelos de *software* e geração da implementação correspondente em uma linguagem de programação específica para uma plataforma. Tal abordagem não deve impor a utilização de um ambiente de desenvolvimento em particular, o que permite o seu uso em conjunto com ferramentas de modelagem disponíveis, além de ser extensível para possibilitar a definição de novas transformações não previstas inicialmente na abordagem.

Para que esse objetivo seja atingido, objetivos intermediários devem ser alcançados, como a adoção de uma linguagem para a definição de transformações que permita a definição de mapeamentos bidirecionais, e o desenvolvimento de uma máquina de transformações que implemente esta abordagem através de padrões existentes para o intercâmbio de modelos transformados com as ferramentas.

O ambiente de desenvolvimento de *software* adotado para ilustrar a utilização da máquina de transformações será o Ambiente *Odyssey* (ODYSSEY, 2006), que visa prover uma infra-estrutura de apoio à reutilização baseada em modelos de domínio. A reutilização de *software* no Ambiente *Odyssey* ocorre através dos processos de Engenharia de Domínio, que tem o objetivo de construir componentes reutilizáveis que solucionem problemas de domínios de conhecimento específicos (WERNER *et al.*, 2005), e de Engenharia da Aplicação, que tem o objetivo de construir aplicações em um determinado domínio, reutilizando os componentes genéricos já existentes (PRIETO-DIAZ *et al.*, 1991). O ambiente *Odyssey* utiliza extensões dos diagramas da *UML* para representar o conhecimento de um domínio, e permite que esses diagramas sejam reutilizados para facilitar a construção de aplicações.

1.3 – Organização

Este trabalho está organizado em 5 capítulos. No segundo capítulo, são ilustrados os principais conceitos sobre as transformações de modelos segundo a Arquitetura Orientada por Modelos. Nesse capítulo, também, são apresentadas algumas abordagens para a definição e execução de transformações sobre modelos e alguns critérios para a comparação destas abordagens, como independência de ferramenta *CASE*, a utilização de formatos e linguagens proprietárias, bidirecionalidade, entre outros.

No terceiro capítulo, é exibida a abordagem proposta, que consiste em um apoio à definição e execução de transformações sobre modelos, que atenda aos critérios definidos no segundo capítulo.

No quarto capítulo, é detalhado o protótipo que implementa a abordagem proposta. Para exemplificar o uso do protótipo, é utilizado um exemplo de transformação de modelos para a plataforma *J2EE*.

No quinto capítulo, são apresentadas as contribuições e limitações deste trabalho, assim como os trabalhos futuros.

Capítulo 2 – Transformações de Modelos

2.1 – Introdução

Uma vez que os sistemas requerem aspectos e serviços disponíveis em plataformas tecnológicas existentes no mercado, o desenvolvimento das funcionalidades de negócio passa a criar uma dependência entre tais funcionalidades e a plataforma na qual o sistema é desenvolvido.

Essa dependência, apesar de inevitável, deve ser minimizada para evitar que os sistemas se tornem obsoletos assim que a sua plataforma se torna ultrapassada, exigindo uma migração para uma versão mais recente da plataforma, ou até mesmo, uma substituição da tecnologia utilizada. Isto ocorre pelo fato de determinados domínios serem relativamente estáveis, o que não ocorre com as tecnologias existentes, que estão em constante evolução.

Para que esta característica de evolução tecnológica não resulte em um grande esforço de migração das funcionalidades para novas implementações em plataformas mais recentes, é necessária uma abordagem que apóie a realização dessas modificações de forma mais ágil, com pouco esforço de migração e re-codificação.

Este capítulo apresenta a teoria necessária para a definição destas abordagens de desenvolvimento, através da utilização da Arquitetura Orientada por Modelos (OMG, 2003b). Na Seção 2.2, a Arquitetura Orientada por Modelos e seus principais conceitos é definida. Na Seção 2.3, realizamos uma comparação entre as principais abordagens de transformação, e na Seção 2.4 concluímos este capítulo apresentado algumas considerações finais.

2.2 – Arquitetura Orientada por Modelos

A *OMG* (*Object Management Group*) definiu um conjunto de padronizações que possibilita a definição de abordagens de desenvolvimento orientado por modelos independentes de plataforma. Este conjunto de padronizações é conhecido como Arquitetura Orientada por Modelos – *MDA* (*Model Driven Architecture*) e tem como foco principal a separação entre a especificação das funcionalidades e a especificação da implementação em uma plataforma específica (OMG, 2003b). As abordagens baseadas em *MDA* permitem o desenvolvimento de *software* através da modelagem e aplicação

de **mapeamentos** automáticos entre os modelos e suas respectivas implementações. O objetivo da *OMG* com a *MDA* é permitir que os desenvolvedores se concentrem em determinar os requisitos de negócio da aplicação, não se preocupando com a **plataforma** em particular onde serão implementados.

Um dos principais benefícios da *MDA* é o apoio ao rápido desenvolvimento e entrega do *software*, através da realização de **transformações sobre os modelos** e **geração de código**, incentivando uma maior ênfase na modelagem do projeto.

A *MDA* é organizada na forma de um *framework*, que consiste em vários conceitos e padrões que podem ser utilizados pelas abordagens *MDA*. As próximas seções detalham os principais conceitos.

2.2.1 – Conceitos Básicos

A idéia principal da *MDA* está na utilização das linguagens de modelagem como linguagens de programação, e não apenas como linguagens de projeto. Com isso, os **modelos** de *software* deixam de ser apenas artefatos de documentação, para serem artefatos principais no processo de desenvolvimento do *software*.

Um **modelo** é uma representação simplificada de algum conceito, com o objetivo de observação, manipulação e entendimento sobre tal conceito (MELLOR *et al.*, 2004). No desenvolvimento de *software*, modelos são criados com o objetivo de diminuir a complexidade inerente ao desenvolvimento.

Antes da *MDA*, os modelos eram elaborados com o objetivo de facilitar a comunicação entre os desenvolvedores e como uma espécie de rascunho para o projeto de construção do *software* (MELLOR *et al.*, 2004). Com a *MDA*, os modelos agora necessitam de uma maior precisão para serem parte direta do processo de produção do *software*, isto é, estes modelos são artefatos de entrada para a construção do sistema através da execução de **transformações**.

Uma taxonomia para os diversos tipos de modelo utilizados no desenvolvimento com *MDA* é apresentada na Figura 2.1. Os modelos abstratos são apenas para a classificação da taxonomia, e os modelos concretos são aqueles que realmente são construídos no processo de desenvolvimento (FRANKEL, 2003).

Dos modelos concretos, todos, com exceção do **Modelo de Negócio**, são modelos de sistema. Cada um desses modelos representa o sistema a partir de um ponto de vista diferente. O **Modelo de Requisitos**, também conhecido como **Modelo Computacionalmente Independente** (*CIM*), é a representação mais abstrata do

sistema, já o **Modelo Independente de Plataforma** (*PIM*) é menos abstrato, pois apresenta parte da lógica do sistema e algumas considerações técnicas. O *PIM* representa um refinamento do Modelo de Requisitos. Um **Modelo Específico de Plataforma** (*PSM*) é um refinamento de um *PIM* e é descrito em termos específicos de uma plataforma onde o sistema é implementado. Os **Modelos Físicos** representam artefatos físicos utilizados no desenvolvimento ou execução do sistema como, por exemplo, arquivos ou nós computacionais.

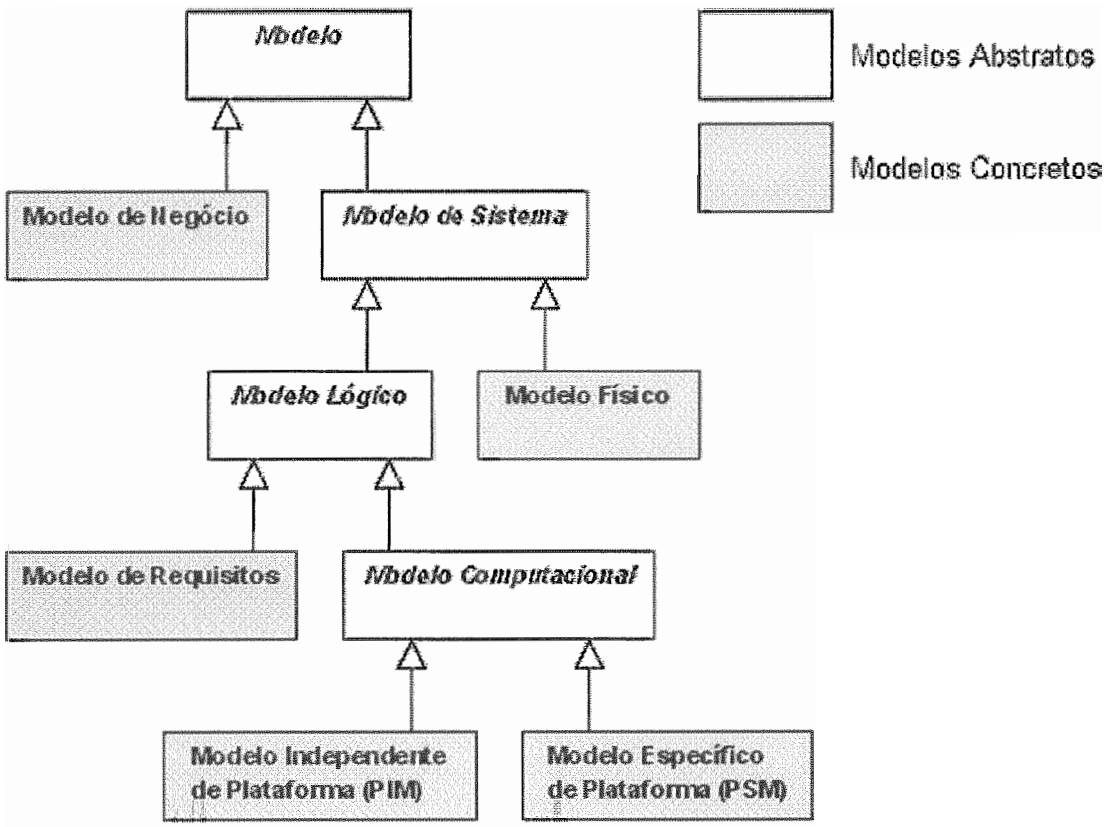


Figura 2.1: Taxonomia de modelos (FRANKEL, 2003).

O Modelo Computacionalmente Independente (*Computation Independent Model – CIM*) é uma visão do sistema a partir de uma perspectiva independente de seus detalhes computacionais, isto é, este modelo não revela a estrutura pela qual o sistema é construído. Um exemplo de *CIM* é um modelo de requisitos, especificado através de um vocabulário familiar aos interessados no sistema. Este modelo é particularmente importante para facilitar a comunicação entre o Engenheiro de Domínio, conhecedor dos conceitos e requisitos, e os Engenheiros de *Software*, que são responsáveis pelo projeto e construção do sistema.

O **Modelo Independente de Plataforma** (*Platform Independent Model – PIM*) é uma visão do sistema a partir de uma perspectiva independente de plataforma. A

independência de plataforma é uma qualidade que um modelo apresenta quando é elaborado de forma independente das características de um tipo de plataforma em particular.

Uma **plataforma** pode ser definida como um conjunto de subsistemas e tecnologias que provêem serviços coerentes através de interfaces e padrões de utilização. Os sistemas desenvolvidos para a plataforma utilizam estes serviços, sem que seja necessário o conhecimento dos detalhes inerentes à implementação dos mesmos (OMG, 2003b). Um exemplo de plataforma disponível atualmente no mercado é o *J2EE* (*Java 2 Enterprise Edition*) (SUN, 2006c) e o *.NET* (MICROSOFT, 2006).

A independência de plataforma é normalmente obtida em um certo grau, isto é, o modelo pode assumir a existência de certas características de um certo tipo geral de plataforma, por exemplo, plataformas para a implantação e execução de componentes distribuídos. Com isso, o modelo pode apresentar características específicas para essa classe de plataformas, sem ser específico para alguma plataforma em particular.

Uma forma de se atingir a independência de plataforma é modelar o sistema para uma máquina virtual tecnologicamente neutra[1], que disponibiliza serviços (e.g.: comunicação, transação, *logging*) definidos de forma independente de alguma implementação em particular. No entanto, ao considerarmos tal máquina virtual uma plataforma, o *PIM* se torna específico para essa plataforma, mas permanece independente das possíveis implementações desta máquina virtual.

O **Modelo Específico de Plataforma** (*Platform Specific Model – PSM*) é uma visão do sistema a partir de uma perspectiva específica para uma plataforma. O *PSM* combina a especificação contida no *PIM* com os conceitos presentes na plataforma escolhida para a implementação do sistema, revelando detalhes de como os recursos presentes na plataforma são utilizados. Pode existir ainda um outro modelo denominado **Modelo de Plataforma**, cujo objetivo é prover ao *PSM*, um conjunto de conceitos técnicos que representam as partes que compõem a plataforma, assim como os diferentes serviços providos pela mesma. Um exemplo é o *CORBA Component Model*

---

[1] O desenvolvimento de componentes para a plataforma *J2EE* segue esta filosofia. Em sua especificação, o *J2EE* define uma espécie de contrato que todas as implementações de servidores para esta plataforma devem seguir. Este contrato é conhecido como implementação de referência e permite que os componentes desenvolvidos para a plataforma *J2EE* sejam independentes das várias implementações de servidores de aplicação disponíveis no mercado.

(OMG, 2002a), que provê uma série de conceitos usados para especificar a utilização da plataforma de componentes *CORBA* por uma aplicação.

A especificação da transformação de um *PIM* em um *PSM* para uma plataforma em particular é realizada através da definição de **mapeamentos** (OMG, 2003b). De maneira geral, podemos utilizar mapeamentos que especificam transformações não apenas para a aplicação de conceitos de plataforma sobre um *PIM*, mas em qualquer transformação entre modelos.

O processo de utilização de uma abordagem *MDA* no desenvolvimento de *software* consiste basicamente na criação do *PIM* utilizando alguma **linguagem de modelagem**, a preparação de tal modelo utilizando algum **mecanismo de marcação**, a execução de **transformações** sobre o *PIM* para a obtenção do *PSM* e a **geração da implementação** do *software* na linguagem de programação da plataforma. Estes conceitos serão detalhados a seguir, nas próximas subseções.

2.2.2 – Linguagem de Modelagem

A *UML* (*Unified Modeling Language*) (OMG, 2005b) é considerada uma linguagem de modelagem padrão no desenvolvimento orientado a objetos que dá apoio a criação de modelos independentes de plataforma, separando a semântica existente nos modelos da implementação destes conceitos. Considere a necessidade de especificar um sistema de arquivos onde uma pasta contém uma coleção de vários registros e, ao apagar a pasta, todos os registros também devem ser removidos em conjunto. Através do conceito de *agregação*, presente na *UML*, é possível especificar tal necessidade de forma sucinta e declarativa, sem especificar como o comportamento deverá ser implementado. Informações independentes de plataforma, como *invariantes*[1], *pré-condições* e *pós-condições*[2], podem também ser expressas em *UML*.

Para apoiar a modelagem independente de plataforma, a *UML* permite também a sua extensão através de **perfis** (*UML Profiles*). Através dos perfis, é possível definir e usar construções adicionais além daquelas já pré-definidas na *UML*. Isto é feito através da definição de conjuntos de **estereótipos** e **valores etiquetados** (*tagged values*).

---

[1] Uma invariante é uma assertiva sobre o estado de um sistema que deve ser sempre verdadeiro, exceto durante a execução de alguma operação (FRANKEL, 2003).

[2] Pré-condições e pós-condições são assertivas sobre operações. Uma pré-condição para uma operação deve ser sempre verdadeira quando uma operação começa a sua execução. Já uma pós-condição deve ser verdadeira no término da execução de uma operação (FRANKEL, 2003).

Os perfis podem ser importantes para a definição de **marcações** a serem aplicadas sobre os *PIMs*, com o objetivo de prepará-lo para a execução de uma transformação. Por exemplo, caso seja necessário informar à transformação que um determinado valor de atributo não pode se repetir em diferentes instâncias de uma classe, o estereótipo *<<unique>>* poderia ser aplicado sobre tal atributo, com isso, a transformação poderia gerar a implementação que garante tal restrição, no *PSM* ou código-fonte. Um exemplo de perfil *UML* que pode ser utilizado para esta finalidade é o *UML Profile for Enterprise Distributed Object Computing* (*EDOC*) (OMG, 2004b).

Apesar deste apoio à extensão, a *UML* pode não ser suficiente para especificar todos os conceitos necessários na modelagem, especialmente no que diz respeito aos modelos específicos de plataforma. Neste caso, uma nova linguagem de modelagem pode ser definida, através da definição de um meta-modelo para essa linguagem, de forma apropriada para as necessidades dos engenheiros.

2.2.2.1 – *Meta Object Facility (MOF)*

O ***MOF*** (*Meta Object Facility*) é uma especificação da *OMG* que define uma linguagem abstrata para a descrição de modelos (OMG, 2002b). Contudo, vale ressaltar que o *MOF* não é uma gramática, mas uma linguagem usada para descrever uma estrutura de objetos, em outras palavras, através do *MOF*, é possível especificar formalmente uma linguagem de modelagem. O *MOF* é também um *framework* extensível, pois permite que novos padrões de metadados sejam adicionados (MATULA, 2003).

Exemplos de descrição de modelos através do *MOF* são: o meta-modelo da própria *UML* e o *CWM* (*Common Warehouse Metamodel*) (OMG, 2003a). Este último utilizado para a modelagem no domínio de *Datawarehouse*.

O *MOF* desempenha papel fundamental na *MDA*, pois, ao permitir que os mapeamentos das transformações sejam definidos em termos de construções *MOF*, é possível definir transformações entre modelos de diferentes meta-modelos. Por exemplo, a transformação de um modelo de classes, em um modelo relacional.

Outra possibilidade é a definição, através do *MOF*, de meta-modelos para as linguagens de programação. Assim, as transformações podem ser definidas com o objetivo de manter o sincronismo entre modelos *UML* e os modelos específicos para a linguagem de programação utilizada.

A arquitetura do *MOF* conta com quatro camadas (MATULA, 2003), ilustradas na Tabela 2.1.

Tabela 2.1: Arquitetura de metadados da *OMG* (MATULA, 2003).

Nível MOF	Termos Utilizados	Exemplos
M3	Meta-metamodelo	Modelo *MOF* (e.g: estrutura que define entidades como classes, atributos e operações).
M2	Metamodelo	Meta-modelos: *CWM* e *UML* (e.g: entidade classe composta de atributos e operações).
M1	Modelo, Metadado	Esquemas relacionais, instâncias do meta-modelo *CWM* e modelos *UML* (e.g: modelo de classes, onde a classe Pessoa apresenta os atributos nome, idade e sexo).
	Instância de Modelo	Dados e objetos (e.g: objeto João de idade 50 e sexo masculino).

2.2.2.2 – Outros Padrões

Assim como o *MOF*, outros dois padrões são importantes para uma utilização prática da *MDA* e, normalmente, são utilizados nas abordagens existentes. São eles o *XMI* (OMG, 2002c) e o *JMI* (DIRCZE, 2002).

A especificação ***XMI*** (*XML MetaData Interchange*) define um conjunto de regras que mapeiam os meta-modelos baseados no *MOF* (M2) e os modelos (M1) em documentos *XML* (OMG, 2002c). Esta especificação define regras de produção de Definição de Tipos de Documentos (*DTD*) para a geração de *DTD XML*, além de regras de produção de documentos *XML* para a geração de modelos num formato compatível com *XML*, para instâncias (M1) de qualquer linguagem (M2) baseada em *MOF*.

Através de *XMI*, é possível realizar a *interoperabilidade* de modelos entre diferentes ferramentas *CASE* ou de transformação. Por exemplo, um modelo *PIM* modelado numa ferramenta *CASE*, pode ser exportado em formato *XML* e para a importação em uma ferramenta de transformação que esteja em conformidade com a especificação *XMI* (OMG, 2002c).

A especificação ***JMI*** (*Java Metadata Interface*) permite a interoperabilidade em nível de *API* para a manipulação de modelos e meta-modelos. Através dessa especificação, é possível, por exemplo, a partir do meta-modelo da *UML*, gerar uma *API* para o acesso, navegação e manipulação dos elementos de modelos *UML*, de forma programática na linguagem *Java* (MATULA, 2003).

Assim como o *XMI*, a especificação *JMI* define regras padronizadas que permitem a geração de *APIs* para a manipulação de qualquer modelo (M1) descrito em qualquer linguagem (M2), desde que a linguagem seja baseada no *MOF*. Isto permite o desenvolvimento de ferramentas de modelagem de forma extensível, por meio de *plug-ins* escritos em *Java*, que podem ter acesso aos elementos do modelo, pela *API* gerada para o mesmo.

2.2.3 – Transformações de Modelos

A transformação de modelos pode ser definida como um processo que realiza a conversão de um modelo em outro modelo do mesmo sistema (OMG, 2003b).

O padrão de transformação de modelos da *MDA*, conforme ilustrado na Figura 2.2, combina o *PIM* com outras informações (e.g.: marcações) para produzir o modelo (ou modelos) específicos de plataforma (*PSM*).

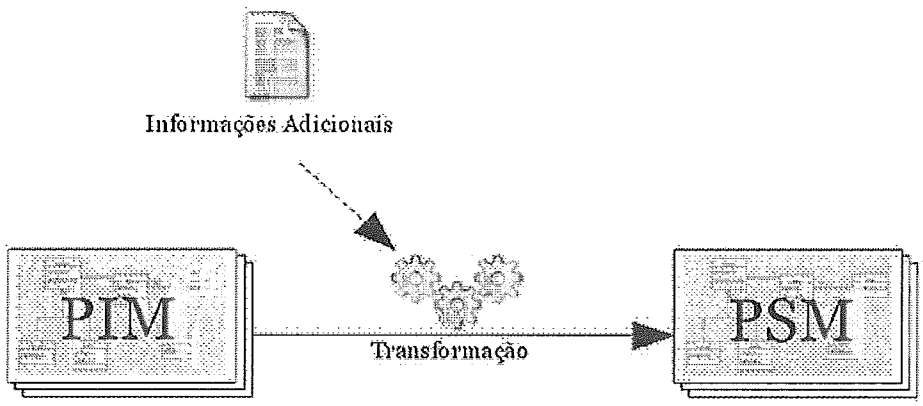


Figura 2.2: Transformação de modelos no *framework MDA*.

O *framework MDA* não define como as transformações devem ser realizadas, apenas indica, em idéias gerais, quatro métodos de transformações de modelo (OMG, 2003b). Esses métodos são explicados a seguir.

- *Transformação manual* – Praticamente idêntico à forma como o projeto de *software* vem sendo realizado até então, onde decisões são tomadas para criar um projeto que proporcione uma implementação em conformidade com os requisitos previamente definidos. A contribuição da *MDA*, neste caso, vem da distinção explicita de um modelo independente de plataforma, com o resultado da transformação que é um modelo específico de uma plataforma.
- *Transformação de PIM preparado com perfil* – Um modelo *PIM* é manipulado por um especialista, que realiza marcações utilizando um

perfil *UML* independente de plataforma. Posteriormente, o *PIM* é transformado para um *PSM* expresso por um segundo perfil *UML*, este específico de plataforma.

- *Transformação utilizando padrões e marcações* – Padrões podem ser utilizados para definir mapeamentos. Marcações correspondem a elementos desses padrões. Os elementos de um *PIM* são marcados e transformados de acordo com o padrão estabelecido no mapeamento, produzindo um *PSM*. Uma outra possibilidade é utilizar regras especificando que todos os elementos no *PIM*, que seguem um determinado padrão, serão transformados em instâncias de um outro padrão, no *PSM*.
- *Transformação automática* – Existem contextos onde um *PIM* pode conter toda a informação necessária para a implementação, ou seja, o *PIM* é completo em relação a sua classificação, estrutura, invariantes, pré e pós-condições. Nestes casos, o desenvolvedor pode especificar o comportamento diretamente no modelo, utilizando uma linguagem de ações. Essa linguagem faz com que o *PIM* seja computacionalmente completo e automaticamente transformado para o código-fonte do programa.

Com relação às plataformas, as transformações podem ser classificadas em três tipos básicos:

- *PIM–PIM* – São transformações realizadas sobre os modelos independentes de plataforma, sem aplicar conceitos de plataforma (e.g.: aplicação de padrões de projeto).
- *PIM–PSM* – Aplicam conceitos da plataforma a um modelo independente de plataforma ou tecnologia (e.g.: transformação de um modelo de classes em um modelo de componentes na arquitetura *EJB*).
- *PSM–PSM* – Realizam um refinamento no modelo específico de plataforma. Um exemplo é a aplicação dos padrões de projeto específicos para a plataforma *J2EE* (SUN, 2006a).

Com essas transformações, um modelo pode ser desenvolvido, refinado através de transformações *PIM-PIM*, convertido para uma plataforma (*PIM-PSM*) e refinado ainda mais, em direção à implementação do *software*. Em um outro momento, caso

exista a necessidade de gerar a implementação desde modelo em uma outra plataforma, o modelo inicial (*PIM*) pode sofrer outras transformações *PIM-PSM* para a nova plataforma, e assim por diante.

As transformações podem ser especificadas através da definição de mapeamentos entre os modelos transformados. O *framework MDA* define três categorias básicas de mapeamentos, são elas:

- *Mapeamentos de Tipos do Modelo* – Esta categoria inclui mapeamentos definidos de acordo com os tipos dos elementos dos modelos. Um mapeamento entre *PIM* e *PSM*, é descrito a partir dos tipos existentes no *PIM* e os tipos existentes no *PSM*. Numa transformação entre modelos de classes da *UML* e um modelo relacional, um possível mapeamento poderia ser definido entre o tipo classe e o tipo tabela relacional. Com isso, todas as classes seriam mapeadas para tabelas.
- *Mapeamentos de Instâncias do Modelo* – Nesta categoria, os elementos mapeados são identificados através de marcações que indicam como a transformação será realizada. Na transformação de um *PIM* para um *PSM*, o mapeamento pode definir uma marcação que representa um conceito do modelo *PSM*, e que poderá ser aplicada aos elementos do modelo *PIM*, para informar como o elemento marcado deve ser transformado. A diferença para a categoria anterior está no fato de elementos que são instâncias de um mesmo tipo poderem ser mapeados de forma diferente, de acordo as marcações realizadas.
- *Mapeamentos Combinados entre Tipos e Instâncias* – Esta categoria envolve mapeamentos que combinam as características dos dois tipos anteriores. Assim, um mapeamento entre tipos dos modelos, pode ser, de certa forma, configurado através de marcações que indicam características presentes no *PSM* que não podem ser definidos através dos tipos presentes no *PIM*.

Os mapeamentos de tipos do modelo apenas são capazes de expressar transformações em termos de regras que geram elementos de tipos do *PSM*, a partir de elementos que são de um determinado tipo do *PIM* (e.g.: todos os elementos do tipo classe da *UML* para elementos do tipo tabela em um meta-modelo relacional). Com isso, a transformação é sempre executada de forma determinística e guiada por informações independentes de plataforma (OMG, 2003b). No entanto, o engenheiro

pode ter a necessidade de utilizar as **marcações** para adicionar informações ao modelo, que serão utilizadas no processo de transformação, por exemplo, para especificar características não funcionais que não podem ser informadas através dos tipos existentes no *PIM*.

Uma outra distinção importante é a categorização com relação aos artefatos transformados. Existem transformações que manipulam exclusivamente modelos, conhecidas como **transformações modelo-modelo**. Já outras transformações podem gerar o código-fonte a partir de um modelo de entrada, definidas como **transformações modelo-texto**. As subseções a seguir detalham essas categorias de transformações.

2.2.3.1 – Transformações Modelo-Modelo

As transformações desta categoria realizam a transformação de um modelo de entrada, em um modelo de saída. Estes modelos podem ser instâncias do mesmo meta-modelo ou de meta-modelos diferentes. O tipo básico de transformação modelo-modelo é a transformação *PIM-PSM*, no entanto, antes de gerar o modelo *PSM*, diversas transformações no modelo *PIM* podem ser necessárias. Portanto, os modelos manipulados por essas transformações podem ser denominados modelo de entrada e saída, origem e destino ou ainda esquerda e direita para o caso de transformações bidirecionais.

A especificação de transformação entre modelos pode ser feita através de uma linguagem puramente **declarativa**, uma linguagem puramente **imperativa**, ou uma combinação entre esses dois tipos, formando uma linguagem **híbrida**.

Uma **linguagem declarativa** (lógica ou funcional) descreve relacionamentos entre variáveis através de funções ou regras de inferência, e um executor dessa linguagem (interpretador ou compilador) aplica algum algoritmo fixo sobre essas relações para produzir um resultado (FOLDOC, 2006). No caso da declaração de transformações, uma especificação declarativa pode conter informação suficiente para descrever completamente uma transformação, seja **unidirecional** ou **bidirecional** (GARDNER *et al.*, 2002).

Uma **linguagem imperativa** permite a especificação explicita do estado de um sistema computacional. Qualquer linguagem de manipulação de dados, por exemplo, *SQL*, pode ser considerada uma linguagem imperativa (FOLDOC, 2006). Uma implementação imperativa pode explicitamente construir os elementos no modelo de destino (GARDNER *et al.*, 2002).

2.2.3.2 – Transformações Modelo-Texto

O exemplo mais conhecido de transformações do tipo **modelo-texto** é a geração de código-fonte, que recebe um modelo de entrada (e.g.: um modelo de classes da *UML*), e realiza a geração dos arquivos de código-fonte para uma linguagem (e.g.: *Java*, C++). Transformações do tipo modelo-texto podem ser consideradas casos especiais de transformações do tipo modelo-modelo. Para isto, basta que seja definido um meta-modelo (M2) para expressar a linguagem de programação desejada. Assim, o código-fonte gerado pela transformação é representado através de um modelo que posteriormente é serializado em forma de arquivo texto.

2.3 – Comparação das Abordagens Existentes na Literatura

Dentre as abordagens que utilizam o *framework MDA* para apoiar o desenvolvimento de *software*, podemos identificar duas principais linhas de utilização (HAYWOOD, 2004). A primeira, conhecida como *elaboracionista*, trabalha em conformidade com os três primeiros métodos de transformação de modelos apresentados anteriormente (manual; *PIM* preparado com perfil; utilização de padrões e marcação). As ferramentas elaboracionistas dão grande importância à existência dos *PSMs*, a partir dos quais, parte do código-fonte do programa é gerado. Nesta categoria, estão inseridas ferramentas como o *OptimalJ* (COMPUWARE, 2006) e *ArcStyler* (INTERACTIVE-OBJECTS, 2006).

Em algumas abordagens da linha elaboracionista, os *PIMs* não apresentam a especificação comportamental da aplicação. Essa informação é introduzida através de refinamentos no *PSM* e no código-fonte gerado. Em outras palavras, a geração é apenas da estrutura estática do *software*. A Figura 2.3 ilustra um cenário elaboracionista de transformação *PIM-PSM* e geração de código-fonte.

A segunda linha de utilização da *MDA* é conhecida como traducionista e defende a idéia de que um sistema deve ser inteiramente especificado através do *PIM*. Também conhecida como *UML* Executável (MELLOR *et al.*, 2002), essa abordagem afirma que os *PSMs* e o código-fonte da aplicação devem ser 100% gerados automaticamente, ou seja, os desenvolvedores não precisam ter conhecimento da existência dos *PSMs*. Essa abordagem utiliza transformações automáticas para a obtenção do código-fonte. Exemplos de ferramentas que se encaixam nesta categoria são o *xUML* (KENNEDY-CARTER, 2006) e *Nucleus BridgePoint* (ACCELERATED-TECHNOLOGY, 2006).

são o *xUML* (KENNEDY-CARTER, 2006) e *Nucleus BridgePoint* (ACCELERATED-TECHNOLOGY, 2006).

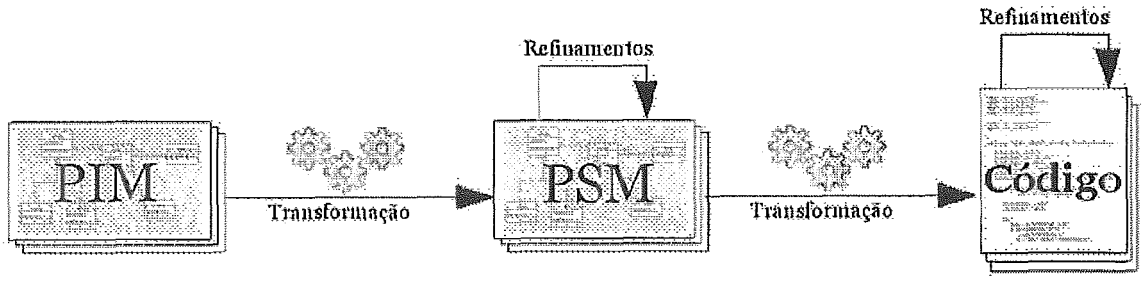


Figura 2.3: Cenário elaboracionista da *MDA*.

Como o *PSM* e o código-fonte não são manipulados pelos desenvolvedores, as abordagens traducionistas exigem que a especificação comportamental do sistema seja incluída no *PIM*. Neste caso, são utilizadas linguagens de ações semânticas como a *Object Action Language* (*OAL*) (ACCELERATED-TECHNOLOGY, 2006) ou a *Action Specification Language* (*ASL*) (KENNEDY-CARTER, 2006).

A Figura 2.4 ilustra um cenário de transformação em uma abordagem traducionista de *MDA*.

Figura 2.4: Cenário traducionista da *MDA*.

Um ponto importante a ser destacado em relação às abordagens traducionistas é o fato destas utilizarem linguagens de ações semânticas proprietárias. Isto causa uma dependência entre o *PIM* e o fornecedor da ferramenta *MDA* utilizada, o que reduz o potencial de reutilização desses modelos no futuro.

Já no caso das abordagens elaboracionistas, os refinamentos sucessivos realizados nos *PSMs* e no código-fonte criam a necessidade de sincronização entre diferentes representações. As alterações realizadas em *PSMs* ou código-fonte podem refletir em alterações no *PIM*, ou seja, um esforço em engenharia reversa pode ser empregado. Outro aspecto importante é que quanto maior o esforço em refinamentos, menor será o ganho de produtividade ao se migrar o *PIM* para a plataforma específica, revelando uma especificação (*PIM*) bastante superficial do sistema.

não é computacionalmente completo conforme requerido pelas ferramentas traducionistas. Outro fato a se destacar é que, para a realização de um intercâmbio dos *PIMs* entre ferramentas traducionistas, pode ser necessária a conversão das linguagens de ações semânticas utilizadas nos modelos.

As abordagens de transformação do tipo **modelo-modelo** podem ser classificadas em seis diferentes categorias (CZARNECKI *et al.*, 2003):

- *Manipulação direta:* Oferecem uma representação interna do modelo e uma *API* para manipulá-lo. São normalmente implementadas como um *framework* orientado a objetos, possivelmente provendo uma infra-estrutura mínima para organizar as transformações. Os usuários devem implementar toda a lógica das regras de transformação, e o escalonamento destas regras, utilizando uma linguagem de programação como *Java* e alguma *API* compatível com *MOF*, como *JMI*.
- *Relacionais:* São caracterizadas pelo uso de relações matemáticas. Os elementos de entrada e saída das transformações são definidos, de forma declarativa, por relações e restrições. A execução dessas relações é feita por programação lógica, onde as relações são implementadas por predicados lógicos. O uso de programação lógica permite o suporte à interpretação e execução das regras de maneira bidirecional.
- *Baseadas em transformações de grafos*: Utilizam grafos para representar modelos *UML* e, com isso, é possível transformá-los utilizando regras de transformação de grafos. Neste tipo de transformação, as regras são compostas de padrões (de grafos) que descrevem os lados (esquerdo e direito) da transformação. Na sua execução, ao se encontrar no modelo de entrada (esquerdo), o padrão definido na regra, este padrão é substituído pelo padrão da direita. Assim como as relacionais, as transformações desta categoria também são definidas de forma completamente declarativa.
- *Orientadas pela estrutura*: São caracterizadas pela divisão em duas fases distintas. A primeira fica encarregada de criar a estrutura hierárquica do modelo de destino. Já a segunda fase configura as propriedades e referências dos elementos no destino. O escalonamento das regras de

transformação fica a cargo do *framework* de execução, o usuário apenas define as regras, que podem conter trechos declarativos e imperativos.

- *Híbridas*: Basicamente são aquelas que combinam características de algumas das categorias anteriores.
- *Outras:* Outras duas categorias de abordagens de transformação modelo-modelo podem ser citadas, a primeira é o *framework* de transformações conhecido por *CWM* (*Common Warehouse Metamodel*) (OMG, 2003a) que fornece uma infra-estrutura para mapeamentos entre elementos de entrada e saída. No entanto, este *framework* não fornece uma linguagem concreta para a definição das transformações. Outras transformações utilizam *XSLT* (W3C, 1999) para transformar modelos. O *XSLT* é uma tecnologia padrão para a transformação de arquivos *XML* e pode ser utilizada para transformar modelos, uma vez que estes podem ser exportados em formato *XML*, através de *XMI*. No entanto, transformações deste tipo podem ser de difícil definição e manutenção, uma vez que as representações *XML* dos modelos são de difícil manipulação.

Já as abordagens de transformação do tipo **modelo-texto**, podem ser classificadas em três categorias:

- *Baseadas no padrão Visitor* (CZARNECKI *et al.*, 2003) – Consiste em prover um mecanismo cuja implementação é baseada no padrão de projeto conhecido como *Visitor* (GAMMA *et al.*, 1995) para navegar na representação interna do modelo, e escrever as informações coletadas em uma saída de texto. Transformações deste tipo podem fazer uso da *API JMI* disponível para a manipulação de modelos.
- *Baseadas em templates* (CZARNECKI *et al.*, 2003) – Um *template* normalmente consiste de um arquivo texto que contém uma espécie de meta-código que é substituído por informações contidas no modelo a ser gerado.
- *Híbridas* – Normalmente, as transformações puramente baseadas em *template* mantém a lógica de acesso às informações do modelo, definida diretamente no *template*. Isto requer a utilização de construções complexas, o que ocasiona uma dificuldade em manutenção destes

templates. Uma transformação híbrida realiza uma separação destas preocupações, onde um mecanismo baseado no padrão *Visitor* fica responsável por navegar no modelo e apenas coletar a informação. Essa coleção de informações é repassada a um mecanismo de *templates* que, em um último momento, faz a substituição das variáveis existentes nos *templates* para geração dos arquivos textuais. Com isso, os *templates* são mais simples, contendo apenas variáveis e com pouquíssima lógica de programação, acarretando uma maior facilidade em manutenção dos mesmos.

Na próxima subseção são apresentados alguns critérios para a comparação de algumas abordagens de transformação de modelos existentes na literatura.

2.3.1 – Critérios de Comparação Entre as Abordagens Existentes

As abordagens descritas neste capítulo são aqui comparadas segundo os seguintes critérios:

- *Independência de ferramenta CASE*: Indica se a abordagem pode ser utilizada por diferentes ferramentas *CASE*, ou se a mesma restringe o uso de uma ferramenta proprietária em particular. Este critério é importante para que a abordagem não obrigue os desenvolvedores a utilizarem alguma ferramenta que venha a se tornar obsoleta, inviabilizando a reutilização destes modelos em novas ferramentas.
- *Utilização de formatos e linguagens padronizadas*: Indica se os modelos e as transformações utilizam formatos, linguagens e padrões definidos e aceitos pela comunidade de desenvolvimento. Assim como o critério anterior, este critério pode indicar se a abordagem tem seu uso facilitado pela utilização de formatos padronizados. A utilização de linguagens proprietárias reduz o potencial de reutilização das transformações definidas, além de apresentarem uma curva de aprendizado maior.
- *Transformação modelo-modelo*: Indica se a abordagem apóia a definição de transformações do tipo modelo-modelo.
- *Transformação modelo-texto*: Indica se a abordagem apóia a definição de transformações do tipo modelo-texto (e.g.: geração de código).
- *Bidirecionalidade*: Indica se as transformações definidas na abordagem podem ser especificadas de forma bidirecional. A bidirecionalidade é

uma qualidade importante para as transformações, especialmente nas abordagens elaboracionistas, onde refinamentos tanto no *PSM* quanto no código-fonte são permitidos.

- *Possibilidade de extensão*: Indica se a abordagem permite a sua extensão para a definição de novas transformações de natureza não prevista pela abordagem.

Os critérios levantados acima foram obtidos através do estudo das contribuições e limitações existentes nas abordagens descritas a seguir, e em comparativos existentes na literatura. O fato de um critério não ser atendido por uma abordagem não indica que a mesma é inferior que as demais, pois tal critério pode ser irrelevante de acordo com o objetivo de tal abordagem, como é o caso do critério de bidirecionalidade em relação às abordagens traducionistas. Este conjunto de critérios, apesar de não formarem um conjunto completo nem necessariamente suficiente, servem como guia para a elaboração de uma abordagem que se proponha a ser abrangente, prática, e que, se possível, traga novas contribuições em relação às abordagens atualmente disponíveis na literatura, que é o caso da abordagem proposta neste trabalho.

2.3.2 – Abordagens Existentes na Literatura

A seguir, são apresentadas as abordagens de transformação comparadas neste capítulo.

2.3.2.1 – *Atlas Transformation Language (ATL)*

A linguagem de transformação *ATL* (*Atlas Transformation Language*) (JOUAULT *et al.*, 2005) é uma linguagem híbrida (composta por construções declarativas e imperativas) projetada para expressar transformações em conformidade com a arquitetura da *MDA*. Esta linguagem é descrita através de uma sintaxe abstrata definida como um meta-modelo *MOF*, uma sintaxe concreta textual e uma notação gráfica que permite a representação de visões parciais sobre transformações de modelos.

Uma transformação escrita em *ATL* é definida através de um conjunto de regras de transformação, cujo estilo recomendado é declarativo. Como pode ser difícil definir algumas regras de forma puramente declarativa, é permitido o uso de construções imperativas para estes casos.

Os modelos de entrada e saída (nível M1), assim como seus respectivos meta-modelos (nível M2), a transformação (nível M1) e o meta-modelo da linguagem *ATL*

(nível M2) são tratados de forma similar pela máquina de transformação da abordagem, e são importados na máquina de transformações através de *XMI*. Com isso, a abordagem é caracterizada como independente de ferramenta *CASE* para a elaboração dos modelos a serem transformados.

Apesar de existir uma notação gráfica para as transformações, ela é apenas utilizada para visualizar tal transformação. A definição é feita através da escrita de um arquivo texto, utilizando a sintaxe concreta da linguagem, a qual não é baseada em nenhum padrão existente.

A abordagem não provê apoio à definição de transformações modelo-texto e, não permite a definição de transformações bidirecionais. Suas transformações são unidirecionais por natureza. Na necessidade de transformações bidirecionais, o engenheiro deve definir duas transformações, uma para cada sentido (direta e reversa).

2.3.2.2 – *UML Model Transformation Tool (UMT)*

Nesta abordagem, os modelos *UML* são importados na ferramenta de transformação através de sua representação *XMI* (OLDEVIK, 2004). No entanto, os modelos são convertidos utilizando uma representação chamada *XMI Light*, que é uma simplificação do *XMI*.

As transformações são definidas através de um conjunto de transformadores que podem ser implementados de duas formas, por meio de transformações *XSL* (*XSLT*) (W3C, 1999) sobre os modelos representados em *XMI Light*, ou implementados diretamente em Linguagem *Java*.

Como os modelos são importados em formato *XMI*, qualquer ferramenta *CASE* que utilize esse padrão pode ser utilizada, o que caracteriza essa abordagem como independente de ferramenta *CASE*.

As transformações *XSLT*, apesar de seguirem um padrão definido pela *W3C*, e serem úteis na definição de transformações sobre arquivos *XML*, não são padrão para a definição de transformações sobre modelos. A utilização de *XSLT* para este fim implica em sérias limitações de escalabilidade e manutenibilidade (CZARNECKI *et al.*, 2003), devido à pobre legibilidade dos arquivos *XMI* e das transformações *XSLT*.

Os transformadores da *UMT* podem realizar transformações do tipo modelo-modelo e modelo-texto. No entanto, estes transformadores são implementados sempre de maneira unidirecional.

A extensão da abordagem é possível e pode ser realizada através da definição de novos transformadores *XSLT* ou *Java*.

2.3.2.3 – *UMLX*

A *UMLX* (ECLIPSE, 2006) é uma linguagem gráfica para a definição de transformações entre modelos *UML*. Nesta abordagem, os modelos transformados devem estar representados em *XML*, seguindo um formato definido por um esquema *XML* (*Schema*). Para definir as transformações, é necessário utilizar modelos de classes da *UML* para representar estes esquemas, e uma extensão do modelo de classes da *UML* para definir as transformações entre estes esquemas.

Cada transformação é totalmente definida de forma declarativa e graficamente através de uma ferramenta de meta-modelagem conhecida como *GEF* (*Graphical Editing Framework*). A abordagem não provê a apoio à execução das transformações definidas. Após ser definida graficamente, a definição da transformação pode ser convertida para a sintaxe textual da abordagem *ATL*, e executada naquela abordagem.

Apesar da abordagem requerer a utilização de uma ferramenta de modelagem própria (*GEF*) para a definição das transformações, ela não restringe o uso de alguma ferramenta CASE em particular para a elaboração de modelos. Qualquer ferramenta *CASE* que permita a exportação de modelos em formato *XMI* pode ser utilizada. Por outro lado, ela é dependente de um outro ambiente de execução, como a máquina de transformações da *ATL*.

Com relação à utilização de linguagens padronizadas, a abordagem utiliza uma notação gráfica que, de certo modo, é considerada proprietária por não ser possível utilizá-la em outra ferramenta diferente da *GEF*.

Não é possível definir transformações do tipo modelo-texto através da *UMLX*, e as transformações somente podem ser definidas de maneira unidirecional. Além disso, também não é possível estender a linguagem de acordo com futuras necessidades de transformação.

2.3.2.4 – *AndroMDA*

O *AndroMDA* é um gerador de código que recebe um modelo *UML* como entrada, em formato *XMI*, e gera um código-fonte como saída (KOZIKOWSKI, 2005). A geração de código é realizada através de uma série de *templates* configuráveis

específicos para plataformas e linguagens de programação. Estes conjuntos de *templates* são denominados cartuchos (*cartridge*).

A ferramenta *AndroMDA* pode ser integrada com alguns ambientes de modelagem, ou ainda importar, através de *XMI*, modelos armazenados em arquivos *XML*, representando, assim, uma abordagem independente de ferramenta *CASE*.

Os cartuchos pré-definidos na abordagem são utilizados para gerar código-fonte na linguagem *Java* para a plataforma *J2EE*. No entanto, a extensão da *AndroMDA* é possível com a criação de novos cartuchos que agregam os novos *templates* desejados, implementados em *VTL* (*Velocity Template Language*) (APACHE, 2005).

A abordagem não permite a definição de transformações modelo-modelo, além de não permitir definição de transformações bidirecionais. Apenas a execução direta (geração de código) é permitida.

2.3.2.5 – *xUML*

O processo de utilização da abordagem *xUML* (MELLOR *et al.*, 2002) envolve a criação de modelos *UML* executáveis independentes de plataforma (*PIM*), com a especificação comportamental do sistema definida através da linguagem de ações semânticas conhecida como *Action Specification Language* (*ASL*) (KENNEDY-CARTER, 2006).

Os modelos *UML* executáveis são desenvolvidos exclusivamente na ferramenta *xUML*, e combinados para gerar o *software* através de mapeamentos *PIM-PSM* definidos em *ASL*. Com isso, os modelos podem ser diretamente visualizados, executados, depurados e testados.

Como a abordagem restringe o seu uso sobre os modelos desenvolvidos na ferramenta *xUML*, assim, esta não pode ser considerada uma abordagem independente de ferramenta *CASE*.

A *xUML* apóia a geração completa da implementação do *software* a partir dos modelos *UML*, desenvolvidos previamente de forma precisa (MELLOR *et al.*, 2002). Assim, não há necessidade da definição de transformações bidirecionais, já que qualquer alteração no *software* deve ser feita no *PIM*.

2.3.2.6 – *OptimalJ*

O *OptimalJ* é uma ferramenta comercial de modelagem, que segue o *framework MDA* (COMPUWARE, 2006). O foco principal do *OptimalJ* é a aplicação da

plataforma *J2EE* nos modelos independentes de plataforma, utilizando várias tecnologias relacionadas.

As transformações são codificadas através de padrões e podem ser de dois tipos. O primeiro é utilizado para a definição de transformações do tipo modelo-modelo, conhecidos como padrões tecnológicos, e convertem *PIMs* em *PSMs*. O segundo tipo, padrões de implementação, provê apoio às transformações modelo-texto, e geram código-fonte a partir dos *PSMs*.

Os padrões tecnológicos, denominados copiadores incrementais (*incremental copiers*), são definidos de forma declarativa e baseados em regras. Estes copiadores podem ser executados diversas vezes sobre os modelos *PSMs* existentes, e não sobrescrevem as informações inseridas manualmente nos *PSMs*.

Já os padrões implementacionais são definidos através de *templates*, descritos através da *Template Pattern Language* (*TPL*).

Além de gerar modelos específicos para a plataforma *J2EE*, o *OptimalJ* permite a transformação do *PIM* em um modelo específico de banco de dados relacional, para prover a persistência dos objetos presentes no modelo *PSM* (*J2EE*).

Como toda a modelagem deve ser realizada dentro da própria ferramenta, a abordagem não pode ser categorizada como independente de ferramenta *CASE*.

Os copiadores e *templates* não utilizam formatos padronizados e também não são definidos de forma bidirecional. No entanto, é possível estender a ferramenta pela implementação de novos copiadores em *Java* e novos *templates*.

2.3.3 – Quadro Comparativo

A Tabela 2.2 classifica as abordagens de acordo com os critérios previamente apresentados. Os campos preenchidos com "sim" indicam que os critérios em questão são atendidos totalmente pela abordagem. Os campos preenchidos com "não" indicam que os critérios não são atendidos pela abordagem. Os campos preenchidos com "?" indicam que não foi possível inferir, através da literatura encontrada, se o critério é atendido totalmente pela abordagem.

Além das abordagens descritas neste capítulo, várias outras abordagens e ferramentas estão disponíveis na literatura. Uma lista de ferramentas *MDA* pode ser encontrada em (MODELBASED.NET, 2006). Um maior detalhamento com relação à comparação e classificação com outras abordagens *MDA* e linguagens de transformação

de modelos podem ser encontradas em (CZARNECKI *et al.*, 2003), (HEYSE *et al.*, 2005) e (GARDNER *et al.*, 2002).

Tabela 2.2: Quadro comparativo das abordagens.

Critérios	Abordagens					
	ATL	UMT	UMLX	AndroMDA	xUML	OptimalJ
Independência de ferramenta *CASE*	Sim	Sim	Sim	Sim	Não	
Formatos e linguagens padronizadas	Não	Não	Não	Não	Não	
Transformações modelo-modelo	Sim	Sim	Sim	Não	Sim	
Transformações modelo-texto	Não	Sim	Não	Sim	Sim	Sim
Bidirecionalidade	Não	Não	Não	Não	Não	Não
Possibilidade de extensão	?	Sim	Não	Sim	?	Sim

2.4 – Considerações Finais

Com o aumento da complexidade no desenvolvimento dos sistemas, surge a necessidade de uma abordagem que permita uma rápida adoção das tecnologias existentes, que possibilite aos desenvolvedores um foco maior nos detalhes conceituais e funcionais do *software*.

A *MDA* surge como um *framework* que provê às abordagens de desenvolvimento um apoio na execução de transformações sobre modelos independentes de plataforma, para a obtenção de modelos específicos e a implementação do *software* em uma plataforma em particular, como *J2EE*.

Conforme foi apresentado neste capítulo, tais transformações podem ser de diferentes tipos, como modelo-modelo ou modelo-texto, além de permitirem a sua definição bi-direcional ou unidirecional.

Algumas abordagens que utilizam o *framework MDA* foram avaliadas segundo os critérios de: (a) independência de ferramenta *CASE*; (b) utilização de formatos padronizados para a definição das transformações; (c) possibilidade de definição de transformações modelo-modelo; (d) possibilidade de definição de transformações modelo-texto; (e) definição bidirecional; (f) possibilidade de extensão.

Após a análise da Tabela 2.2, podemos chegar a algumas conclusões. Estas conclusões não podem ser consideradas sempre verdadeiras, visto que a tabela não é completa, mas pode servir como orientação para o projeto de uma abordagem mais abrangente e prática, a ser proposta. As conclusões são:

- Nenhuma das abordagens avaliadas utiliza um formato padrão para a definição das transformações, dificultando o aprendizado e a aplicação

das abordagens na prática. Existe um esforço da *OMG* para essa padronização, que se encontra em fase de finalização (OMG, 2005a). No entanto, a sua adoção depende da finalização de outras especificações que ainda estão indisponíveis.

- Nenhuma das abordagens avaliadas neste capítulo suporta a definição de transformações bidirecionais.

Este capítulo apresentou uma visão geral sobre a teoria e os conceitos relacionados à Arquitetura Orientada por Modelos (*MDA*), e uma avaliação de algumas das principais abordagens existentes na literatura. A partir desta avaliação, concluímos que nenhuma das abordagens avaliadas atende completamente aos critérios propostos, o que motiva a definição uma nova abordagem, apresentada no próximo capítulo.

Capítulo 3 – *Odyssey-MDA*: A Abordagem Proposta

3.1 – Introdução

No Capítulo 2, foram apresentados os principais conceitos sobre Arquitetura Orientada por Modelos (*MDA*) e o impacto de sua utilização no desenvolvimento de *software*. Além disso, foram apresentadas uma classificação dos diversos tipos de transformação possíveis, e uma categorização das abordagens que utilizam o *framework MDA*, suas principais características e deficiências.

Neste capítulo, apresentamos uma abordagem prática para a definição e execução de transformações, levando em consideração os critérios estabelecidos no Capítulo 2, que são: (a) independência de ferramenta *CASE* ou ambiente de desenvolvimento; (b) utilização de formatos e linguagens padronizados ou de fácil aprendizado; (c) apoio à definição e execução de transformações modelo-modelo; (d) apoio à obtenção da implementação através da geração de código (transformações modelo-texto); (e) possibilidade de extensão das transformações pré-definidas.

A abordagem proposta neste capítulo abrange os cenários de transformação do tipo modelo-modelo e as transformações do tipo modelo-texto. As próximas seções detalham esta abordagem e estão organizadas da seguinte forma: a Seção 3.2 apresenta uma visão geral da abordagem e suas principais características; a Seção 3.3 descreve como são definidas as transformações entre modelos; a Seção 3.4 discute as etapas de execução das transformações previamente definidas; a Seção 3.5 apresenta as transformações do tipo Modelo-Texto; a Seção 3.6 mostra como esta abordagem pode ser estendida. A Seção 3.7 encerra o capítulo, apresentando algumas considerações finais.

3.2 – Visão Geral da Abordagem Proposta

Uma visão geral da abordagem proposta é exibida na Figura 3.1. A abordagem foi concebida para permitir a definição de transformações de forma independente de ferramenta *CASE* – parte (a). O Projetista pode definir transformações sobre modelos *UML*. Essa definição de transformações será detalhada na Seção 3.3.

Após elaborar os modelos na sua ferramenta *CASE* – parte (b), o Engenheiro de *Software* pode aplicar as transformações previamente definidas através de uma máquina

de transformações – parte (c) – e gerar a implementação dos modelos na plataforma desejada. Para executar transformações sobre modelos, pode ser necessária a preparação prévia do modelo a ser transformado, através de marcações. Essa marcação e a execução das transformações será detalhada na Seção 3.4. A parte (d) ilustra a geração de código através de transformações do tipo modelo-texto, detalhada Seção 3.5.

Para que a abordagem seja extensível – parte (e) – optamos pela definição de uma infra-estrutura, detalhada na Seção 3.6, que permite a definição de novas transformações, assim como a implementação de novos mecanismos de transformação de elementos (*plug-ins*) e de geração de código para novas linguanges de programação.

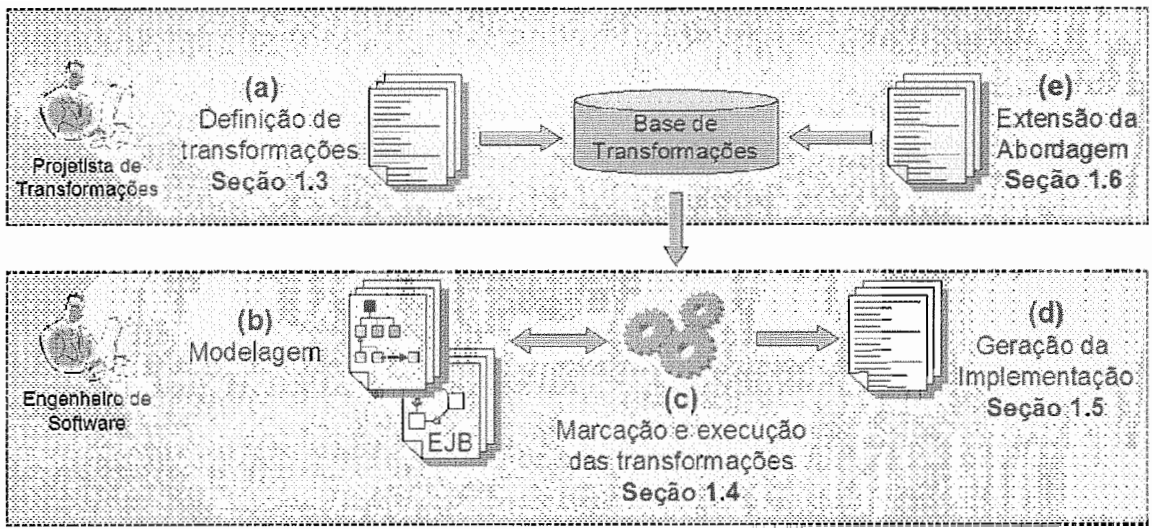


Figura 3.1: Abordagem proposta.

Algumas características gerais da abordagem foram definidas com base nos critérios definidos no Capítulo 2 e serão apresentadas nas próximas seções.

3.3 – Apoio à Definição de Transformações

Conforme foi apresentado no capítulo anterior, uma transformação pode ser definida como um processo que realiza a conversão de um modelo em um outro modelo do mesmo sistema (OMG, 2003b).

Para ilustrar a utilização da abordagem proposta na definição de transformações, é preciso apresentar um exemplo de transformação, denominado *EJBComponents*, cujo objetivo é, a partir de um modelo independente de plataforma (*PIM*), gerar um modelo de componentes específico para a plataforma *Enterprise JavaBeans* (*PSM*). A execução reversa desta transformação também será possível e irá recuperar o modelo independente de plataforma original, a partir do modelo específico para *EJB*. Esta transformação é apresentada nas subseções a seguir.

3.3.1 – Exemplo de Transformação: Plataforma *Enterprise JavaBeans*

A especificação *Java 2 Platform, Enterprise Edition* (*J2EE*), definida pela *Sun Microsystems*, consiste em uma abordagem para o desenvolvimento de aplicações corporativas (SUN, 2006b). Essa especificação define uma arquitetura, conhecida como *Enterprise JavaBeans* (*EJB*) (SUN, 2003), para o desenvolvimento e implantação de sistemas corporativos baseados em componentes distribuídos. Esta pode ser considerada uma arquitetura cliente/servidor, com os componentes *EJB* posicionados em servidores, possivelmente distribuídos, e acessados por diferentes aplicações cliente.

A arquitetura *EJB* possuí um modelo de componentes que fornece um conjunto de convenções, regras e padrões para a implementação e implantação de componentes distribuídos, visando a reutilização dos componentes desenvolvidos em diferentes aplicações nessa plataforma (SUN, 2003). Ainda nessa especificação, são definidos os tipos de componentes permitidos e os relacionamentos entre estes componentes e a infra-estrutura onde os mesmos serão implantados, constituindo, assim, um contrato que todo componente deve obedecer.

Os componentes *EJB* são conhecidos como *enterprise beans*, são componentes localizados no lado do servidor e encapsulam a lógica de negócio da aplicação. A utilização dos *enterprise beans* possibilita o desenvolvimento de aplicações escaláveis[1], pois sua execução pode estar distribuída por diferentes servidores de forma transparente, permitindo ainda, controle transacional e acesso concorrente. Os tipos básicos de *enterprise beans* são:

- *Entity Bean*: Representa um objeto de negócio que persistido através de algum mecanismo de armazenamento como um banco de dados relacional. Neste caso, cada *entity bean* possuí uma tabela relacional associada, onde cada instância do componente corresponde a uma linha dessa tabela. Para uma aplicação desenvolvida no domínio de hotelaria, possíveis conceitos implementados através de *entity beans* são clientes, hospedes, reserva, etc.
- *Session Bean*: Representa a sessão de um cliente dentro do servidor de aplicação. Para ter acesso à aplicação disponível no servidor, o cliente invoca as operações dos *session beans*. Um possível *session bean* para

[1] Escalabilidade pode ser definida como a capacidade de um sistema se adaptar ao aumento da demanda por suas funcionalidades (FOLDOC, 2006).

uma aplicação de hotelaria poderia ser um controlador responsável por realizar reservas de quartos aos clientes.

- *Message-Driven Beans*: É um *enterprise bean* que permite o processamento assíncrono de mensagens. Estes componentes não fazem parte do foco principal deste trabalho. Maior detalhamento sobre estes componentes pode ser encontrado em (SUN, 2003)

A implementação dos *enterprise beans* envolve a codificação das regras e padrões definidos no modelo de componentes *EJB*. Alguns arquivos precisam ser codificados seguindo os padrões estabelecidos para que o componente possa ser implantado no servidor de aplicação *J2EE*. A Tabela 3.1 mostra os arquivos necessários para a implementação dos *enterprise beans*.

Do ponto de vista prático, o projetista de componentes *EJB* necessita de algum tipo de apoio automatizado para a realização da modelagem de um sistema para essa plataforma. Sem este apoio, essa tarefa pode se tornar tão repetitiva e susceptível a erros, que a equipe de desenvolvimento, normalmente, deixa a aplicação dos conceitos da plataforma para ser realizada apenas em tempo de codificação dos componentes, ou seja, as interfaces locais e remotas apenas são criadas em código-fonte. Isso pode causar inúmeros problemas relacionados à codificação de aspectos que não estão representados em artefatos de projeto.

Sendo assim, a definição de transformações, com o objetivo de aplicar tais conceitos e padrões a partir de modelos independentes, pode facilitar a especificação dos componentes na plataforma. A definição de transformações, também, facilita a obtenção da implementação (código) dos mesmos componentes, poupando os desenvolvedores da realização de tarefas repetitivas como a implementação dos vários arquivos que podem ser gerados pela transformação.

Com isso, o Engenheiro de *Software* não precisa se preocupar em representar todos os aspectos da plataforma *EJB* em seus modelos de projeto. Ele deve apenas se preocupar em modelar a aplicação, aplicar marcações sobre os elementos para representar entidades ou serviços de negócio, e executar a transformação *EJBComponents* sobre tal modelo, para a geração dos *enterprise beans*, suas interfaces locais, interfaces remotas e classes de chave primária.

Tabela 3.1: Arquivos necessários para implementação de um *enterprise bean*.

Arquivo	Descrição
Classe *Entity Bean*	É a implementação lógica do componente *entity bean*, contendo todas as suas funcionalidades. Normalmente, essas funcionalidades permitem a alteração e consulta dos campos existentes no componente. De acordo com a especificação, essa classe deve implementar algumas interfaces (remotas ou locais) e, assim, prover a implementação dos métodos definidos em tais interfaces.
Classe *Session Bean*	É a implementação lógica do componente *session bean*, contendo todos os serviços de negócio. De acordo com a especificação, essa classe deve implementar algumas interfaces (remotas ou locais) e, assim, prover a implementação dos métodos definidos em tais interfaces.
Classe de chave primária	Contém o subconjunto dos campos existentes no *entity bean* que identifica uma instância em particular do componente. A implementação dessa classe é opcional, uma vez que o Engenheiro pode utilizar uma classe padrão da linguagem *Java*, como a classe `String`.
Interface de Componente (remota)	Permite à aplicação cliente, em execução remota, realizar chamadas aos métodos de negócio do componente.
Interface *Home* (remota)	Permite à aplicação cliente, em execução remota, realizar chamadas aos métodos responsáveis pelo ciclo de vida do componente.
Interface de Componente (local)	Permite à aplicação cliente, em execução na mesma máquina do componente, realizar chamadas aos métodos de negócio deste componente.
Interface *Home* (local)	Permite à aplicação cliente, em execução na mesma máquina do componente, realizar chamadas aos métodos responsáveis pelo ciclo de vida deste componente.
Descritor de implantação	Contém as propriedades que podem ser editadas no momento de montagem ou implantação do componente no servidor de aplicação.

A transformação *EJBComponents* recebe um modelo de entrada (*PIM*) (Figura 3.2), e transforma todas as classes que representam as entidades de negócio, marcadas com estereótipo <<*entity*>>, em componentes e*ntity bean* (Figura 3.3). Todas as classes que representam os serviços providos pela aplicação, marcados com estereótipo <<*service*>>, são transformadas em componentes s*ession bean* (Figura 3.4).

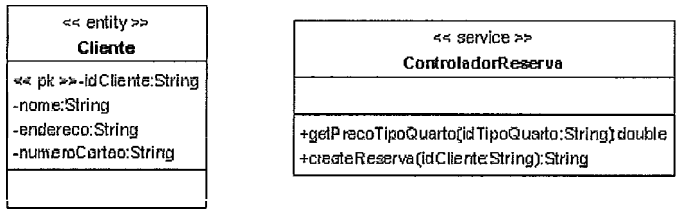


Figura 3.2: Classes *PIM* de entrada da transformação *EJBComponents*.

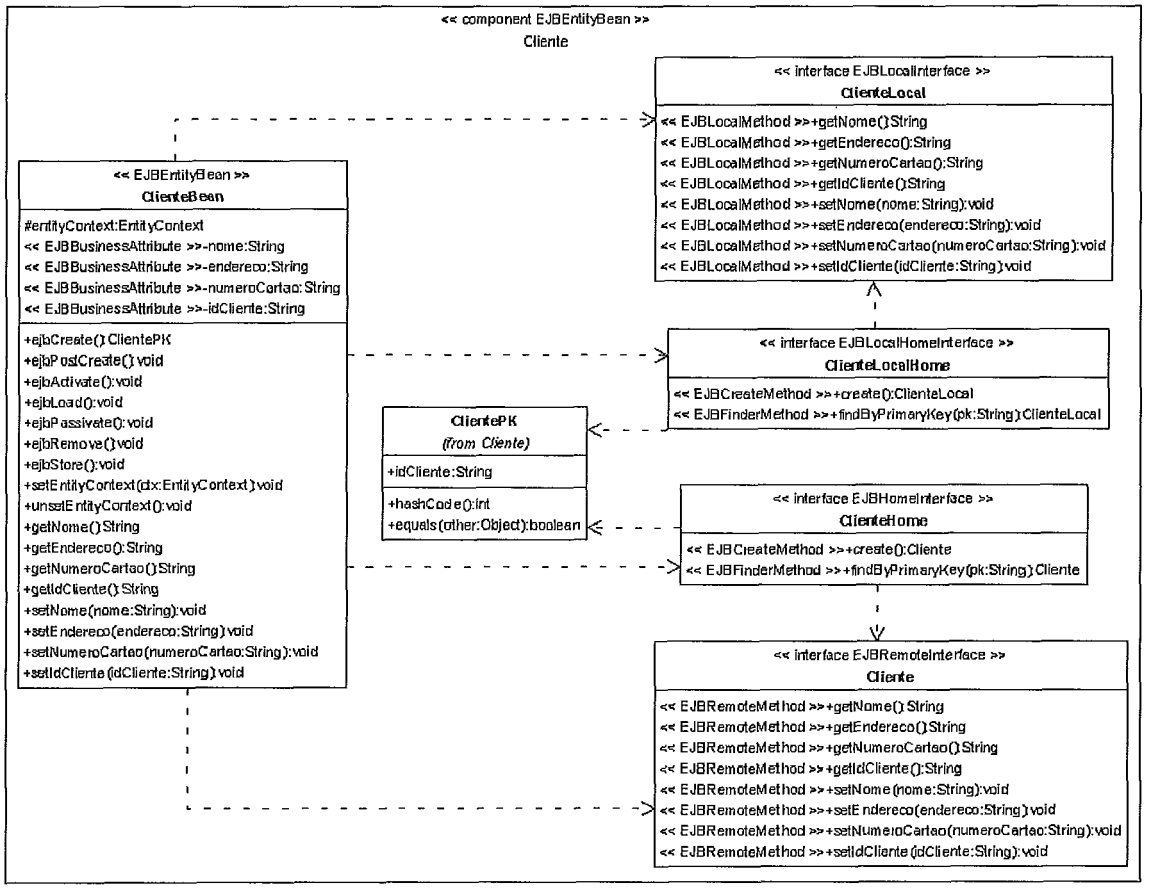


Figura 3.3: Componente *entity bean* gerado pela transformação *EJBComponents*.

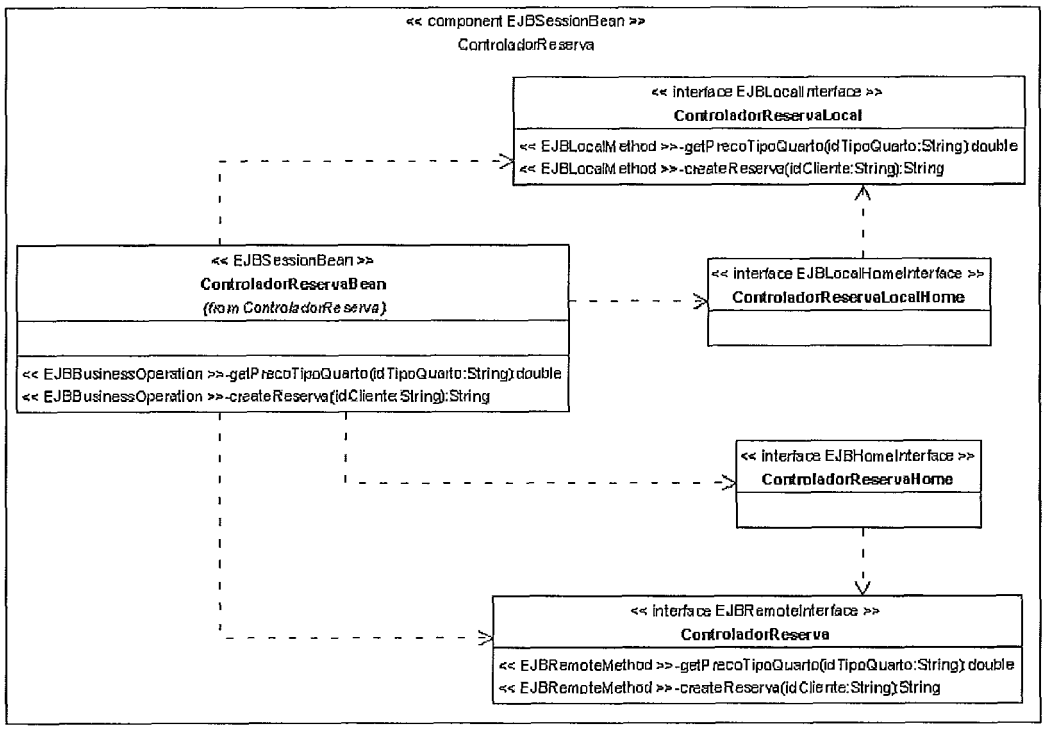


Figura 3.4: Componente *session bean* gerado pela transformação *EJBComponents*.

3.3.2 – Transformações Modelo-Modelo

Em um cenário típico de desenvolvimento, modelos de alto nível de abstração (e.g.: modelo de casos de uso) são refinados sucessivamente até o início da fase de implementação, onde os conceitos e funcionalidades representados nos modelos são codificados na linguagem de programação da plataforma escolhida. Estes refinamentos, na grande maioria dos casos, são caracterizados pela realização de tarefas repetitivas que podem resultar na inserção de inconsistências nos modelos. Justamente por isso, estes refinamentos podem ser automatizados através de transformações.

Dentre as várias linguagens e formatos para a especificação e definição de transformações, podemos citar o *framework* de transformações presente na especificação *CWM* (*Common Warehouse Metamodel*) (OMG, 2003a). Essa especificação provê apoio à definição de transformações de modelos de dados baseados no meta-modelo *CWM*. A Figura 3.5 apresenta, de forma simplificada, parte deste meta-modelo que trata de transformações. A partir deste meta-modelo, é possível definir transformações, pela definição de elementos do tipo *TransformationMap*, o qual é composto por mapeamentos do tipo *ClassifierMap*, *FeatureMap* e *ClassifierFeatureMap*. Os mapeamentos *ClassifierMap* mapeiam elementos Classifier, *FeatureMap* mapeiam *Features*, e *ClassifierFeatureMaps* podem mapear tanto *Classifier* para *Feature* ou vice-versa.

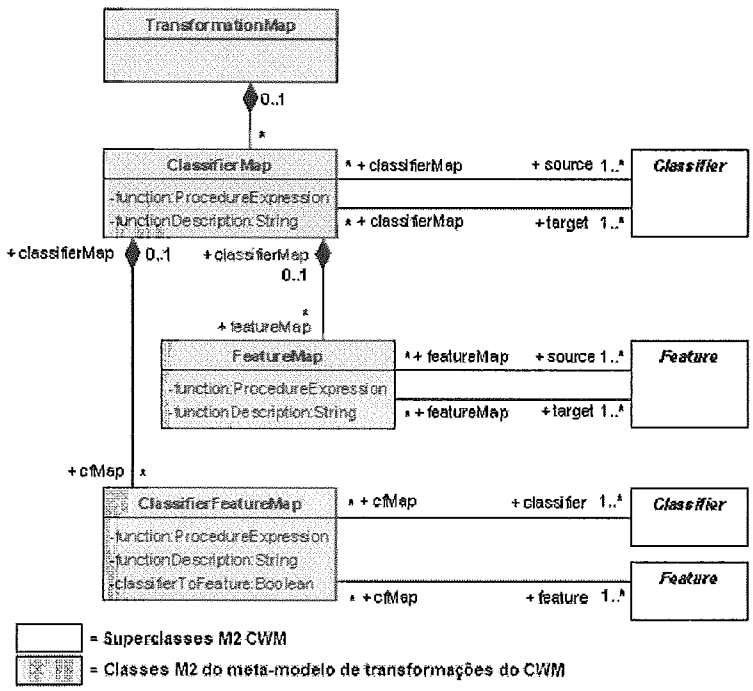


Figura 3.5: Fragmento do meta-modelo de transformações do *CWM*. Adaptado de (FRANKEL, 2003).

Outro aspecto é o fato dos mapeamentos definidos no meta-modelo de transformações do *CWM* permitirem apenas a definição da ligação entre os elementos dos modelos de origem e destino da transformação. Para a execução da transformação entre tais elementos, o meta-modelo prevê a utilização de expressões com este objetivo (*ProcedureExpression*), no entanto, o meta-modelo não provê uma linguagem concreta que permita a implementação dessas expressões.

Por estes motivos, optamos por adaptar o meta-modelo de transformações do *CWM* de modo a permitir a definição de mapeamentos entre os *Classifiers* e *Features* presentes no meta-modelo da *UML*, e no futuro, com o lançamento do *MOF* versão 2.0, permitir a definição de mapeamentos entre quaisquer elementos cujo meta-modelo seja definido através do *MOF*. A Figura 3.6 apresenta o modelo de transformações proposto neste trabalho. Os mapeamentos propostos (*ClassifierMap*, *FeatureMap* e *ClassifierFeatureMap*) são similares aos existentes no *CWM*, no entanto, os elementos mapeados são instâncias (classes *M2*) do meta-modelo da *UML*.

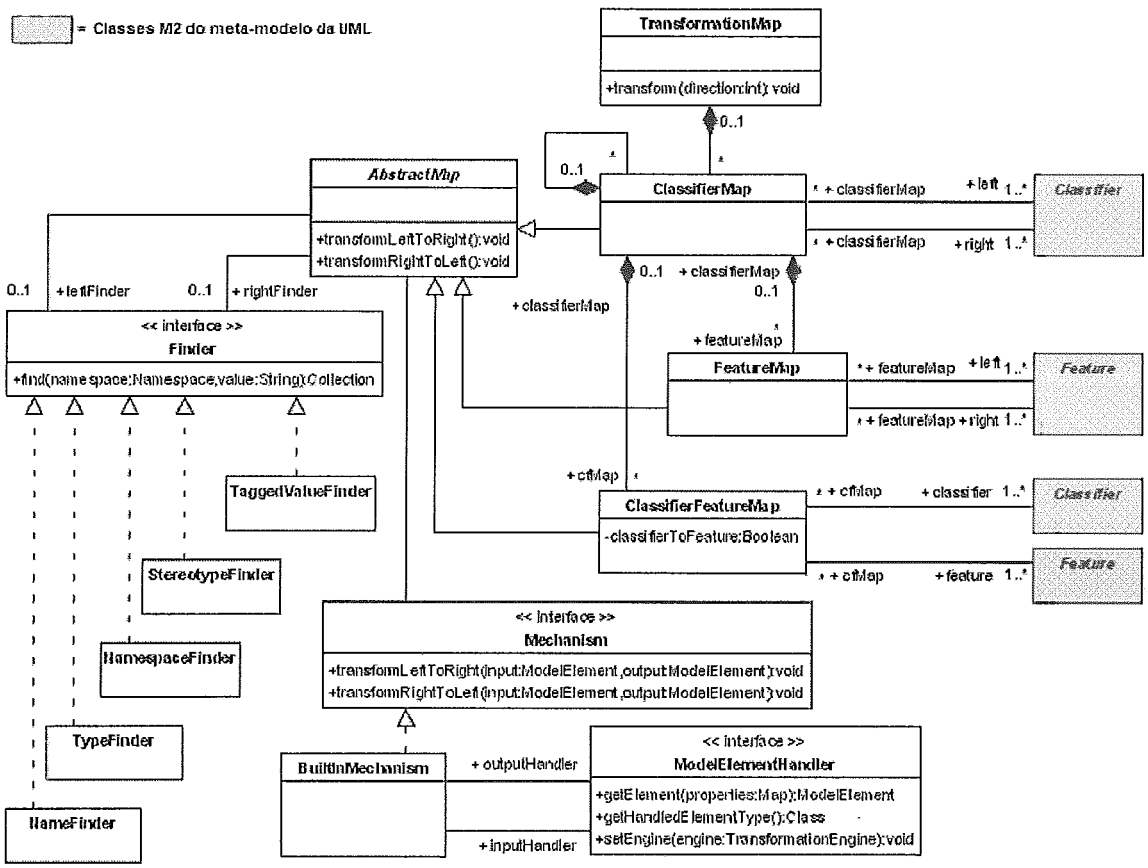


Figura 3.6: Modelo de transformações da abordagem proposta.

Como a abordagem proposta permite a definição de transformações bidirecionais, os mapeamentos também devem ser definidos de maneira bidirecional. Para tanto, optamos por utilizar os termos *esquerda* e *direita* para designar os elementos

participantes dos mapeamentos. No momento da execução da transformação, a direção escolhida (*esquerda → direita* ou *direita → esquerda*) indicará os elementos de entrada e de saída.

Os mapeamentos propostos nesta abordagem podem ser divididos em duas partes, uma declarativa e outra imperativa. A parte declarativa indica os elementos participantes do mapeamento, especificando, se necessário, os critérios de busca para selecionar os elementos a serem mapeados. Já a parte imperativa é responsável por configurar os mecanismos que implementam as transformações. De acordo com a terminologia apresentada no Capítulo 2, a abordagem dos mapeamentos apresentados nesta seção pode ser considerada híbrida, ao combinar construções declarativas e imperativas.

A adaptação do modelo de transformações *CWM* permitiu, também, a definição de uma infra-estrutura genérica de mecanismos que implementam as transformações entre os elementos dos modelos, representando, assim, uma base de mecanismos pré-definidos, onde o projetista das transformações pode selecionar e configurar os mecanismos desejados para cada transformação. Esta infra-estrutura será detalhada na Seção 3.3.2.2.

De acordo com a classificação dos mapeamentos apresentada no Capítulo 2, o meta-modelo de transformações do *CWM* permite, originalmente, apenas a definição de mapeamentos de tipos do modelo (*Model Type Mappings*). Para propor um modelo de transformações que, na prática, permita a definição de mapeamentos combinados entre tipos e instâncias (*Combined Type and Instance Mappings*), é necessária a utilização de uma instrumentação para selecionar as instâncias (elementos do modelo) que serão transformadas. Essa instrumentação é composta basicamente de:

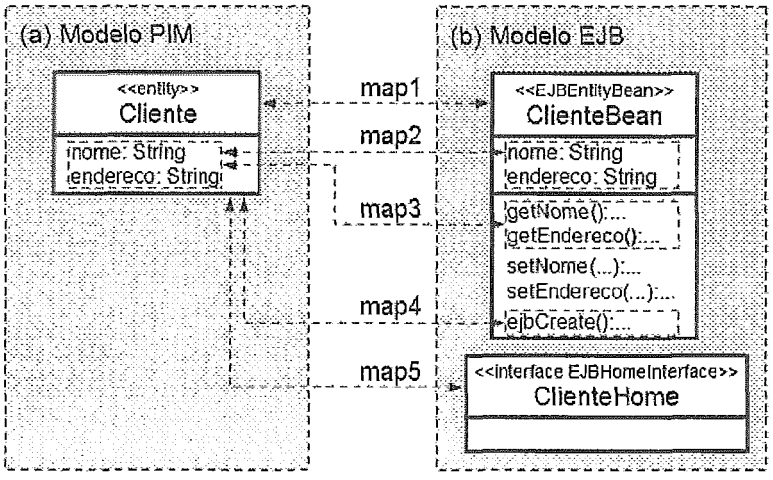
- Marcações – São informações que podem ser aplicadas a elementos de um modelo, com o propósito de guiar a aplicação de um mapeamento em particular. Por exemplo, o estereótipo *<<entity>>* é uma marcação que pode ser utilizada sobre um elemento do tipo Classe para indicar que este elemento representa um conceito do tipo entidade, que na transformação *EJBComponents* será mapeado para um componente *entity bean*. A Seção 3.4.1 detalha como essa marcação pode ser realizada.
- Critérios de busca – Permitem a seleção de elementos do modelo de origem que serão mapeados em elementos do modelo de destino. Elementos podem ser selecionados por vários critérios, como por

exemplo, as marcações aplicadas, o tipo do elemento, um padrão que ele apresenta, etc. A Seção 3.3.2.1 apresenta um detalhamento dos critérios de busca utilizados nesta abordagem.

A especificação do modelo de transformações desta abordagem é feita através da linguagem *XML* (W3C, 2004), de acordo com o esquema *XML* apresentado no Apêndice A. A utilização de *XML* evita a necessidade de criar uma nova sintaxe proprietária para a especificação das transformações, além de permitir a utilização de bibliotecas disponíveis para a manipulação e interpretação deste formato, na implementação da máquina de transformações desta abordagem.

Na especificação *XML* da transformação, o projetista da transformação indica os mapeamentos necessários, os mecanismos utilizados, os critérios de busca de elementos para a instanciação dos mapeamentos, e os relacionamentos que devem ser gerados no modelo de saída.

A Figura 3.7 ilustra alguns dos mapeamentos definidos na transformação *EJBComponents* e parte de sua especificação *XML* (a especificação completa pode ser consultada no Apêndice B). Um maior detalhamento sobre essa especificação será apresentado nas próximas subseções.




```
<transformation-map name="EJBComponents" ...>
    <classifier-map id="map1" name="Entity-Class-PIM - EntiyBean-Class-EJB" ...>
        <feature-map id="map2" name="Attribute-PIM - Attribute-EJB" ...> ...
        </feature-map>
        <feature-map id="map3" name="Attribute-PIM - GetterMethod-EJB" ...> ...
        </feature-map>
        <classifier-feature-map id="map4" name="Entity-Class-PIM - ejbCreate-Method-EJB" ...>...
        </classifier-feature-map>
    </classifier-map>
    <classifier-map id="map5" name="Entity-Class-PIM - EntityBean-HomeInterface-EJB" ...>...
    </classifier-map>
    ...
</transformation-map>
```

Figura 3.7: Parte da especificação *XML* da transformação *EJBComponents*.

3.3.2.1 – Critérios de Busca de Elementos

Ao definir cada mapeamento de uma transformação, o projetista pode utilizar alguns critérios de busca para a seleção dos elementos que serão mapeados. A máquina de transformações utiliza estes critérios, no momento de execução da transformação, para selecionar os elementos do modelo de entrada e instanciar tais mapeamentos sobre estes elementos. Os critérios de busca possíveis são:

- Tipo – permite a seleção dos elementos através do tipo destes elementos. Por exemplo, no caso da *UML*, é possível selecionar todos os elementos que são instâncias do tipo *Interface*.
- Nome – permite a seleção dos elementos através do nome. Ex. selecionar todos os elementos do modelo que possuam o nome "create".
- Espaço de Nomes (*namespace*) – permite a seleção de elementos que pertençam a um determinado espaço de nomes. Ex. selecionar todos os elementos cujo espaço de nomes seja "br.ufrj.cos.lens.odyssey".
- Estereótipo – permite a seleção de elementos que possuam um determinado estereótipo associado. Ex. selecionar todos os elementos que possuam o estereótipo "entity".
- Etiqueta (*tagged value*) – permite a seleção de elementos que possuam uma determinada etiqueta e, possivelmente, um valor associado a essa etiqueta. Ex. selecionar todos os elementos cuja etiqueta "persistent" tenha o valor "true".

Os critérios de busca através de estereótipos e etiquetas permitem a definição de mapeamentos com base no perfil *UML* que pode ser utilizado no modelo de entrada. No entanto, esta abordagem não obriga o Projetista de Transformações a definir um perfil *UML* para especificar uma transformação, ele pode simplesmente definir um grupo de estereótipos e etiquetas que satisfaz o seu objetivo.

Ao especificar um mapeamento, o projetista da transformação pode utilizar uma combinação dos critérios de busca para realizar a seleção com base no seu objetivo. Além disso, ao utilizar um critério, ele deve informar o lado da transformação (modelo da esquerda ou modelo da direita) no qual a máquina de transformações, no momento da execução, deverá selecionar os elementos.

Para o exemplo da Figura 3.7, a definição do mapeamento entre as classes *PIM* e as classes *entity bean* (*map1*) necessita de critérios para, no momento de sua execução,

selecionar todas as classes com estereótipo <<entity>> presentes no modelo *PIM* e, numa possível execução reversa, fazer a seleção das classes *entity bean* com estereótipo *<<EJBEntityBean>>* no modelo *EJB*. Estes critérios são apresentados na Tabela 3.2.

Tabela 3.2: Exemplo de utilização de critérios de busca.

Lado da Transformação	Critério	Valor
Esquerda (*PIM*)	Tipo	Classe
Esquerda (*PIM*)	Estereótipo	<<entity>>
Direita (*EJB*)	Tipo	Classe
Direita (*EJB*)	Estereótipo	<<EJBEntityBean>>

A Figura 3.8 apresenta a especificação *XML* dos critérios de busca da Tabela 3.2.

```
<classifier-map id="map1" name="Entity-Class-PIM - EntiyBean-Class-EJB" ...>
    <finder side="left" criteria="type" value="Class" />
    <finder side="left" criteria="stereotype" value="entity" />
    <finder side="right" criteria="type" value="Class" />
    <finder side="right" criteria="stereotype" value="EJBEntityBean" />
    ...
</classifier-map>
```

Figura 3.8: Especificação *XML* dos critérios de busca necessários ao mapeamento *Entity-Class-PIM* ↔ *EntiyBean-Class-EJB*.

Neste exemplo, o mapeamento é bidirecional, pois define critérios a serem utilizados em ambas as direções de execução. No entanto, a abordagem permite que o projetista defina mapeamentos de forma unidirecional, ao especificar critérios apenas para um dos lados da transformação. A especificação de uma transformação bidirecional pode ser composta de mapeamentos bidirecionais e unidirecionais, mas no momento de sua execução, a máquina de transformações executa, além dos mapeamentos bidirecionais, apenas os mecanismos cuja direção especificada seja a mesma da execução desejada.

Está em andamento um esforço, liderado pela *OMG*, para definir uma especificação de um mecanismo capaz de selecionar elementos de modelos através de consultas (*queries*) (OMG, 2004a). Apesar de várias propostas (GARDNER *et al.*, 2002), ainda não se sabe, até o momento de conclusão deste trabalho, como esse mecanismo será definido. Possivelmente será recomendada a utilização de *OCL* e *UML Action Semantics*. Futuramente, novos critérios de busca poderão ser implementados visando à adoção dessa nova padronização, conforme apresentado na Seção 3.6.3.

3.3.2.2 – Mecanismos de Transformação

Conforme visto anteriormente, os mapeamentos apenas realizam a ligação entre os elementos dos modelos de entrada e saída. A realização da transformação entre estes elementos é feita por mecanismos associados a cada mapeamento. A abordagem permite a utilização de uma infra-estrutura de mecanismos, denominados *built-ins*, que são montados dinamicamente pela máquina de transformações, de acordo com o tipo dos elementos relacionados pelo mapeamento. Caso esses *built-ins* não sejam suficientes, o projetista pode implementar novos mecanismos e incorporá-los a abordagem. Estes novos mecanismos são os *plug-ins* e serão discutidos na seção sobre extensibilidade da abordagem (Seção 3.6).

A especificação do uso de um *built-in* informa que a máquina de transformações deve instanciar um mecanismo composto de dois tratadores específicos para cada tipo dos elementos relacionados pelo mapeamento (tipos da esquerda e direita). Os tratadores são responsáveis por manipular (criar e configurar) os elementos transformados. Os tratadores disponíveis para a utilização nos *built-ins* são:

- *Component* – transforma componentes da *UML*.
- *Class* – transforma classes da *UML*.
- *Interface* – transforma interfaces da *UML*.
- *Attribute* – transforma atributos da *UML*.
- *Operation* – transforma operações da *UML*.

Na definição da transformação *EJBComponents*, existem vários mapeamentos entre elementos de diferentes tipos. Por exemplo, o mapeamento *map5* da Figura 3.7 relaciona classes e interfaces. A sua especificação, portanto, deve informar os tipos dos elementos mapeados. Neste caso, para o modelo da esquerda (*PIM*), é o tipo *Class*, e no lado da direita (*EJB*), deve ser o tipo *Interface*. Assim, ao executar a transformação definida por este mapeamento, a máquina de transformações instancia um *built-in* do tipo *Class↔Interface* que será responsável pela execução da transformação entre os elementos destes tipos. A especificação dos tratadores é feita através das propriedades de configuração, apresentada na próxima subseção.

3.3.2.2.1 – Propriedades de Configuração

Como os tratadores utilizados pelos *built-ins* são genéricos, a abordagem provê uma forma de configurar tais tratadores de modo a controlar o comportamento da

transformação. Para transformar, por exemplo, um elemento do tipo Classe em um elemento do tipo Interface, pode ser necessária a configuração destes elementos, com base nas características definidas no meta-modelo da *UML*. Esta configuração pode ser informada ao *built-in* através de algumas propriedades. A Tabela 3.3 apresenta as propriedades que podem ser utilizadas, suas descrições baseadas nas características presentes no meta-modelo da *UML* (OMG, 2004c), valores possíveis e para quais tratadores elas se aplicam.

Tabela 3.3: Propriedades possíveis disponíveis para utilização em *built-ins*.

Propriedade	Descrição	Valores possíveis	
`left_type`	Informa qual tratador deve ser utilizado para o lado esquerdo da transformação.	Nomes dos tratadores	Todos
`right_type`	Informa qual tratador deve ser utilizado para o lado direito da transformação.	Nomes dos tratadores	Todos
`Name`	Informa nome do elemento	Nome do elemento	Todos
`stereotype`	Informa um estereótipo para o elemento. Esta propriedade pode ser utilizada mais de uma vez para um mesmo tratador, indicando a utilização de mais de um estereótipo no elemento transformado.	Nome do estereótipo	Todos
`visibility`	Especifica se o elemento pode ser visto ou referenciado por outros elementos do modelo.	`package`, `public`, `protected`, `private`	Todos
`is_specification`	Especifica se o elemento faz parte da especificação do elemento no qual ele está contido.	`true`, `false`	*Class*, *Interface*, *Operation*.
`is_root`	Para Classe ou Interface, indica (*true*) se o elemento não pode ser uma generalização de outro elemento. Para uma Operação, indica (*true*) que a classe dona da operação não pode herdar uma outra declaração dessa operação.	`true`, `false`	*Class*, *Interface*, *Operation*.
`is_leaf`	Para Classe ou Interface, indica (*true*) que o elemento não pode ter descendentes. Para uma Operação, indica (*true*) que a implementação da operação não pode ser sobrescrito, por um descendente.	`true`, `false`	*Class*, *Interface*, *Operation*.
`is_active`	Especifica se um objeto dessa classe mantém (*true*) sua própria *thread* de controle.	`true`, `false`	*Class*

Tabela 3.3: Propriedades possíveis disponíveis para utilização em *built-ins* (Cont.).

`is_query`	Especifica se a execução da operação mantém (*true*) o estado do sistema inalterado.	`true, false`	*Operation*
`owner_scope`	Especifica se a *Feature* aparece em cada instância do *Classifier* (*instance*) ou se existe apenas uma instância da *Feature* para todo o *Classifier* (*classifier*).	`classifier, instance`	*Attribute*, *Operation*
`changeable`	Especifica em quais condições o valor do atributo pode ser modificado após o objeto ter sido inicializado. O valor *frozen* indica que o atributo é uma constante, *addonly* indica que o valor do atributo pode ser apenas inicializado uma única vez, e *changeable* indica que ele é uma variável.	`frozen, addonly, changeable`	*Attribute*
`concurrency`	Especifica a semântica de chamadas concorrentes à mesma instância passiva de um objeto, cuja propriedade is_*active* seja *false*.	`concurrent, guarded, sequential`	*Component*, *Class*, *Interface*.
`namespace`	Especifica o espaço de nomes no qual o elemento está inserido.	Nomes de pacotes ou elementos *Classifier*.	*Class*

A Figura 3.9 mostra como é feita essa especificação das propriedades, em *XML*.

```
<classifier-map id="map5" name="Entity-Class-PIM – EntityBean-HomeInterface-EJB" ...>
    <property name="left_type" value="Class" />
    <property name="right_type" value="Interface" />
    <property name="stereotype" value="EJBHomeInterface" direction="forward" />
    ...
</classifier-map>
```

Figura 3.9: Especificação *XML* dos tipos relacionados pelo mapeamento *Entity-Class-PIM* ↔ *EntityBean-HomeInterface-EJB*.

Além do nome da propriedade e do valor associado, pode ser especificada também a direção na qual a configuração se aplica. No exemplo da Figura 3.9, a especificação do estereótipo *<<EJBHomeInterface>>* apenas se aplica à execução direta (*forward*) da transformação.

Com as propriedades apresentadas na Tabela 3.3, o projetista das transformações consegue apenas configurar valores simples para as propriedades. Entretanto, algumas configurações necessitam de valores complexos, para isso, a abordagem também provê uma forma de configurações compostas, onde uma propriedade pode conter outras propriedades.

Um exemplo do uso deste tipo de configuração é a especificação dos parâmetros das operações. Na transformação *EJBComponents*, o mapeamento *Entity-Class-PIM* ↔ *equalsMethod-EJB* é responsável por criar um método chamado de *equals* na classe que encapsula a chave primária do *entity bean* (*ClientePK*). A assinatura deste método deve possuir um parâmetro do tipo *Object*, e outro parâmetro, de retorno, do tipo primitivo *boolean*. Para configurar essa assinatura, é utilizada a propriedade *parameters* que pode conter sub-propriedades. Cada sub-propriedade representa um parâmetro e o nome de cada uma dessas sub-propriedades representa o nome do parâmetro correspondente. Essa configuração é apresentada na Figura 3.10.

```
<feature-map name="PKAttribute-PIM ↔ findByPrimaryKeyMethod-EJB" ...>
    ...
    <property name="parameters" direction="forward">
        <property name="other">
            <property name="direction_kind" value="in" />
            <property name="classifier_type" value="java.lang.Object" />
        </property>
        <property name="return">
            <property name="direction_kind" value="return" />
            <property name="data_type" value="boolean" />
        </property>
    </property>
    ...
</classifier-map>
```

Figura 3.10: Exemplo de configuração dos parâmetros de uma operação. Para configurar cada parâmetro, podem ser utilizadas as sub-propriedades definidas na Tabela 3.4.

Outro caso onde é necessária a utilização de propriedades compostas é na configuração de etiquetas (*tagged values*). Por exemplo, para especificar a necessidade da aplicação de uma etiqueta "isPersistent" com valor "true" sobre o elemento em transformação, basta configurar o tratador do elemento conforme mostra a Figura 3.11.

```
<property name="tagged_values" direction="forward">
    <property name="isPersistent" value="true" />
</property>
```

Figura 3.11: Exemplo de configuração de etiquetas.

No caso de um mapeamento bidirecional, pode ser necessário indicar se a configuração é aplicável apenas na execução em uma das duas direções, para isso, é utilizado o atributo *direction*, como pode ser visto também na Figura 3.11, onde a etiqueta especificada apenas deve ser criada no modelo da direita.

Tabela 3.4: Sub-propriedades para configuração dos parâmetros das operações.

Sub-propriedade	Descrição	Valores possíveis
`direction_kind`	Especifica o tipo da direção requerida pelo parâmetro. O valor *in* indica que o parâmetro é apenas de entrada, isto é, o seu conteúdo (no escopo da chamada da operação) não pode ser alterado. O valor *out* informa que a direção do parâmetro é de saída, permitindo que a operação passe informações ao chamador. Para que o parâmetro seja de entrada, e seu conteúdo no escopo de origem possa ser alterado, é utilizado o valor *inout*. Já para especificar o retorno da operação, o valor *return* deve ser utilizado.	`in, out, inout, return`
`data_type`	Especifica o tipo primitivo do parâmetro.	Nome de qualquer tipo primitivo, ou `void`.
`class_type`	Especifica uma classe para ser utilizada como tipo do parâmetro. Caso a classe informada não exista no modelo, uma nova classe com o nome informado é criada.	Nome de uma classe pertencente ao *namespace* atual, ou o nome completo (*namespace* + nome da classe).
`interface_type`	Especifica uma interface para ser utilizada como tipo do parâmetro. Caso a interface informada não exista no modelo, uma nova interface, com o nome informado, é criada.	Nome de uma interface pertencente ao *namespace* atual, ou o nome completo (*namespace* + nome da interface).
`classifier_type`	Especifica um *Classifier* para ser utilizado como tipo do parâmetro.	Nome de um *Classifier* pertencente ao *namespace* atual, ou o nome completo (*namespace* + nome do *Classifier*).

3.3.2.2.2 – Uso de Expressões Regulares para Transformações Textuais

Existem casos especiais onde é necessária a definição de transformações complexas envolvendo o nome dos elementos que apenas serão conhecidos em tempo de execução da transformação. Na transformação *EJBComponents*, este tipo de transformação textual ocorre, por exemplo, ao transformar uma classe com nome "Cliente" em uma outra classe com nome "ClienteBean". Outro caso seria a partir de um atributo "endereco", criar uma operação "getEndereco". Para que seja possível definir esta transformação textual, optamos por utilizar substituições com o uso de

expressões regulares[1], assim como é permitido na linguagem de programação *Perl* (WALL *et al.*, 1996).

Para definir uma transformação textual, é utilizada a propriedade *name_transformation*, que contém três sub-propriedades. São elas:

- *input* – o texto de entrada.
- *regexp* – a expressão regular que captura um padrão no texto da entrada (*input*).
- *subst* – o texto que deve substituir o padrão encontrado pela expressão regular definida em *regexp*.

Com essas substituições, é possível definir as transformações textuais e suas respectivas transformações reversas. A Tabela 3.5 ilustra alguns exemplos de utilização das expressões regulares. A coluna Entrada contém o texto de entrada, a coluna Saída contém o resultado da transformação.

Cada expressão regular descreve um padrão que seleciona uma parte do texto que deve ser substituído pelo padrão descrito na coluna Substituição. Na transformação definida na linha 1 da Tabela 3.5, a expressão regular "(.*)" seleciona todos os caracteres da entrada. Os parênteses indicam que estes caracteres serão utilizados no padrão da substituição, no lugar do "$1". Com isso, a substituição "find$1" adiciona o prefixo "find" à entrada. A transformação reversa dessa substituição é definida na linha 2, onde o prefixo "find" é substituído por uma *string* vazia, ou seja, é removido.

Ainda na Tabela 3.5, a transformação definida na linha 3 adiciona o sufixo "Bean" ao texto de entrada "Cliente" e a transformação da linha 4 faz a remoção de tal sufixo (transformação reversa). A transformação da linha 5 adiciona um prefixo e um sufixo ao texto de entrada ("Cliente" → "findClienteById"), enquanto que a transformação da linha 6 realiza a substituição reversa, retirando o prefixo e o sufixo. Com a transformação da linha 6, é possível remover uma seqüência qualquer de caracteres dentro de outra seqüência. Já na transformação da linha 8, o nome de um atributo é transformado no nome de uma operação de acesso, adicionando o prefixo "get" ao nome do atributo, e transformando o primeiro caractere do nome do atributo em caixa alta, através da substrituição "get\u$1". A transformação da linha 9 faz a

[1] Uma Expressão Regular descreve ou avalia conjuntos de *strings* de acordo com certas regras de sintaxe. Expressões Regulares são utilizadas por muitos editores de texto e utilitários para procurar e manipular trechos de texto baseados em certos padrões (FRIEDL, 2002).

transformação reversa, removendo o prefixo "get" e alterando o caractere inicial do nome do atributo para caixa baixa ("\l$1").

Além dos exemplos citados, o projetista ainda pode definir outras substituições. O estudo detalhado das expressões regulares está além do escopo deste trabalho e pode ser encontrado em (WALL *et al.*, 1996).

Tabela 3.5: Exemplos de transformações textuais através de expressões regulares.

	Entrada	Saída esperada	Expressão regular	Substituição	Forma reduzida
1	Cliente	findCliente	(.*)	find$1	find#INPUT#
	O "$1" é substituído pelo padrão encontrado pela expressão regular contida dentro dos parênteses. Com isso, o texto da entrada é inserido no lugar do "$1", transformando "Cliente" em "findCliente".				
2	findCliente	Cliente	^find		-
	O "^" indica que a expressão deve estar no prefixo, ou seja, a expressão "^find" seleciona o prefixo "find". A ausência de valor na substituição indica que o prefixo será substituído por uma cadeia de caracteres vazia, isto é, será removido.				
3	Cliente	ClienteBean	(.*)	$1Bean	#INPUT#Bean
	O "$1" é substituído pelo padrão encontrado pela expressão regular contida dentro dos parênteses. Com isso, o texto da entrada é inserido no lugar do "$1", transformando "Cliente" em "ClienteBean".				
4	ClienteBean	Cliente	Bean$		
	O "$" indica que a expressão deve estar no sufixo, ou seja, a expressão "Bean$" seleciona o sufixo "Bean". A ausência de valor na substituição indica que o sufixo será substituído por uma cadeia de caracteres vazia, isto é, será removido.				
5	Cliente	findClienteById	(.*)	find$1ById	find#INPUT#ById
	O "$1" é substituído pelo padrão encontrado pela expressão regular contida dentro dos parênteses. Com isso, o texto da entrada é inserido no lugar do "$1", transformando "Cliente" em "findClienteById".				
6	findClienteById	Cliente	^find(.*)ById$	$1	-
	A expressão regular "^find(.*)ById$" seleciona os caracteres presentes entre o prefixo "find" e o sufixo "ById". A substituição é realizada para remover esses prefixos e sufixo, mantendo apenas os caracteres entre eles ("$1").				
7	VLixoalor	Valor	Lixo		-
	Todas as ocorrências de "Lixo" são substituídas por uma cadeia de caracteres vazia, ou seja, são removidos.				
8	atributo	getAtributo	(.*)	get\u$1	-
	A substituição adiciona o prefixo "get" ao texto da entrada. A expressão "\u" indica um pré-processamento necessário para alterar o primeiro caractere selecionado para caixa alta.				
9	getAtributo	atributo	^get(.*)	\l$1	-
	A substituição remove o prefixo "get" do texto da entrada. A expressão "\l" indica um pré-processamento necessário para alterar o primeiro caractere selecionado para caixa baixa.				

3.3.2.3 – Manipulação de Relacionamentos

Com o mapeamentos e mecanismos apresentados até aqui, apenas os elementos que são subtipos de *Classifier* e *Feature* (*MOF*) são transformados. No entanto, deve ser possível manipular, também, os relacionamentos entre os elementos *Classifier* dos modelos. Por exemplo, na geração do modelo de classes para a plataforma *EJB*, alguns relacionamentos, como as dependências entre a classe que implementa o *Entity Bean* e suas interfaces locais e remotas, também devem ser gerados. Para este caso, é necessário definir os relacionamentos que serão gerados pela transformação no modelo de destino, de acordo com a direção escolhida no momento de execução.

Para especificar a geração de relacionamentos entre os elementos, a abordagem permite que o projetista identifique os mapeamentos destes elementos participantes, e defina o relacionamento com base nessa identificação. Os tipos de relacionamento possíveis são:

- Generalização (*Generalization*)
- Associação (*Association*)
- Dependência (*Dependency*)
- Ligação (*Binding*)
- Abstração (*Abstraction*)
- Uso (*Usage*)
- Permissão (*Permission*)

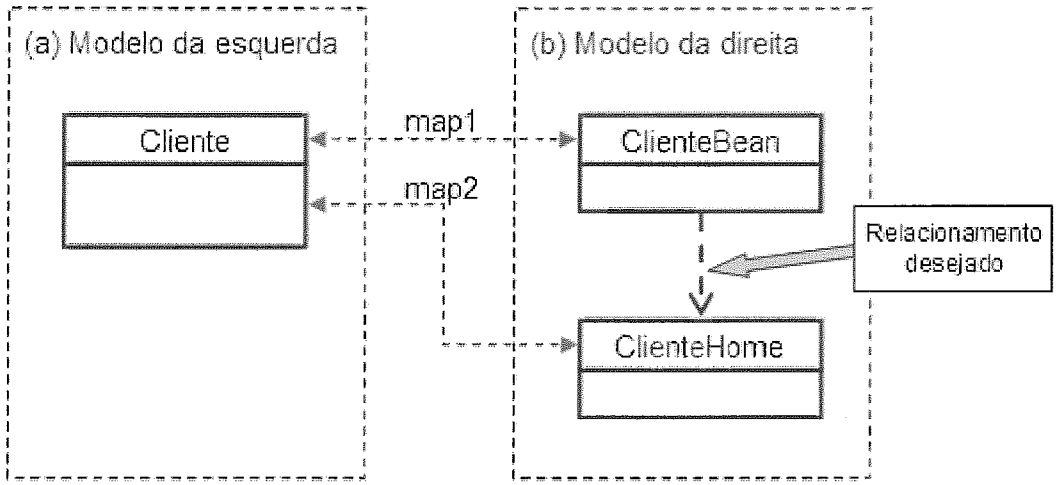
Ao definir a geração de um relacionamento, o projetista deve configurar os participantes do relacionamento através das propriedades listadas na Tabela 3.6.

A abordagem permite a utilização dos principais relacionamentos presentes na *UML*. No entanto, caso outro meta-modelo venha a ser utilizado, a abordagem poderá ser estendida para permitir a utilização de outros tipos de relacionamento, conforme será visto na Seção 3.6.

A Figura 3.12 ilustra um exemplo onde é necessário gerar, no modelo da esquerda (*EJB*), um relacionamento do tipo *Dependency* entre todos os elementos gerados a partir do mapeamento *map1* e todos os elementos de *map2*. A especificação *XML* deste relacionamento contém a referência para os mapeamentos participantes, assim como as propriedades que configuram o comportamento da geração do relacionamento.

Tabela 3.6: Propriedades utilizadas na configuração dos relacionamentos.

Relacionamentos	Propriedade	Descrição	Valores possíveis
Generalização	`child`	Na especificação de generalizações, essa propriedade informa se os elementos do mapeamento indicado fazem o papel de sub-elementos.	`yes` (o valor `No` não é necessário pois é indicado pela ausência de tal propriedade)
Generalização	`parent`	Na especificação de generalizações, essa propriedade indica os elementos que serão os super-elementos.	`yes` (o valor `No` não é necessário pois é indicado pela ausência de tal propriedade)
Dependência, Ligação, Abstração, Uso e Permissão	`client`	Na especificação de dependências (e suas variantes), informa se os elementos do mapeamento indicado fazem o papel de cliente.	`yes` (o valor `No` não é necessário pois é indicado pela ausência de tal propriedade)
Dependência, Ligação, Abstração, Uso e Permissão	`supplier`	Na especificação de dependências (e suas variantes), informa se os elementos do mapeamento indicado fazem o papel de fornecedor (*supplier*).	`yes` (o valor `No` não é necessário pois é indicado pela ausência de tal propriedade)
Associação	`is_navegable`	Especifica se uma associação deve ser navegável para algum dos sentidos.	`true, false`
Associação	`ordering`	Especifica se o conjunto de ligações da associação deve ser ordenada ou não.	`unordered, ordered`
Associação	`aggregation`	Especifica se o final da associação é uma agregação simples ou agregação do tipo composição.	`aggregate, composite`
Associação	`multiplicity`	Indica a multiplicidade de um final da associação.	`1, *, 0..1, 0..*, 1..*`, ou ainda qualquer outra multiplicidade
Associação	`changeability`	Especifica em quais condições o valor do da associação pode ser modificado após o objeto ter sido inicializado. O valor *frozen* indica que o mesmo é uma constante, *addonly* indica que o valor pode ser inicializado apenas uma única vez, e *changeable* indica que ele é uma variável.	`changeable, frozen, addOnly`

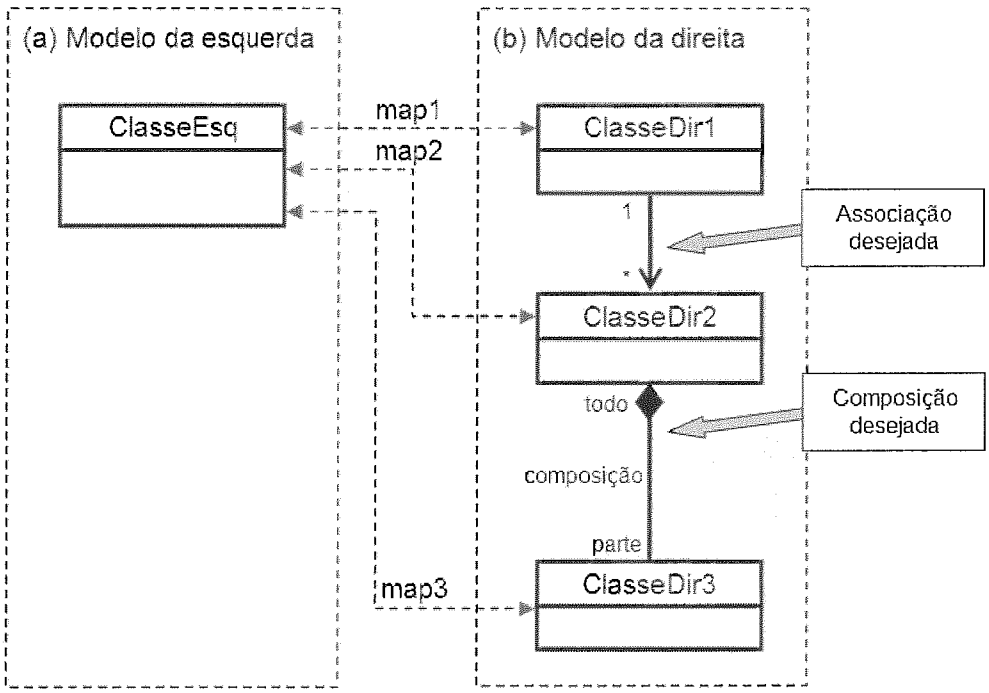



```
<classifier-map id="map1" name="map 1">...</classifier-map>
<classifier-map id="map2" name="map 2">...</classifier-map>
...
<relationship type="Dependency" direction="forward">
    <map id="map1">
        <property name="client" value="yes" />
    </map>
    <map id="map2">
        <property name="supplier" value="yes" />
    </map>
</relationship>
```

Figura 3.12: Exemplo de geração de relacionamento.

Para que essa geração seja possível, ambos os mapeamentos participantes do relacionamento devem possuir os mesmos critérios de busca de elementos do modelo de origem. Essa restrição garante que a máquina de transformação realize a geração do relacionamento apenas entre os elementos que foram criados a partir do mesmo elemento de origem. Por exemplo, caso o modelo da esquerda (origem), ilustrado na Figura 3.12, possua uma classe *Fornecedor*, que também seja mapeada por *map1* e *map2*, a restrição garante a geração da dependência, no modelo da direita, apenas entre as classes *FornecedorBean* e *FornecedorHome*, não interferindo nos elementos e relacionamentos gerados a partir da classe *Cliente*.

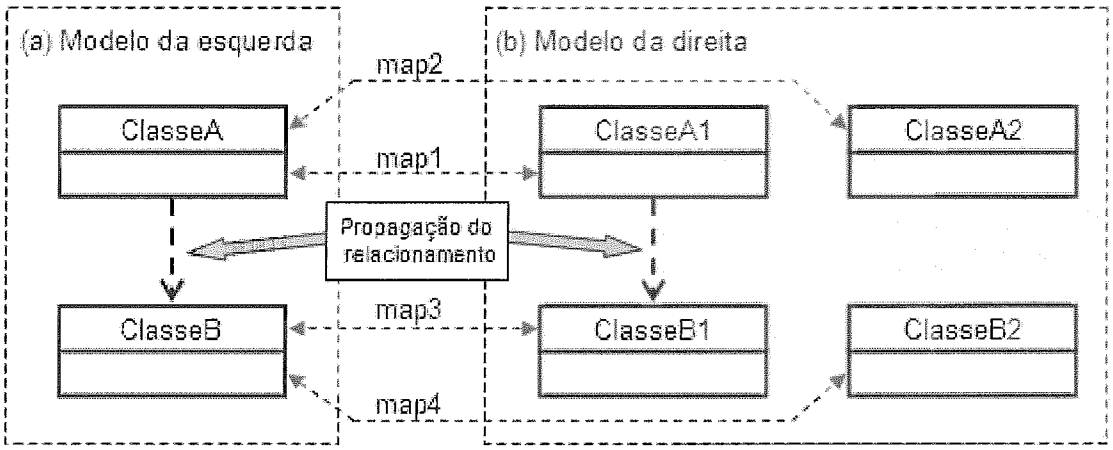
Na Figura 3.13, são ilustrados mais dois exemplos de geração de relacionamentos, uma associação e uma composição. A especificação *XML* detalha como devem ser informadas as propriedades destes relacionamentos, como multiplicidade, navegabilidade e papéis.




```
<classifier-map id="map1" name="map 1">...</classifier-map>
<classifier-map id="map2" name="map 2">...</classifier-map>
<classifier-map id="map3" name="map 3">...</classifier-map>
...
<relationship type="Association" direction="forward">
    <map id="map1">
        <property name="multiplicity" value="1">
    </map>
    <map id="map2">
        <property name="is_navegable" value="true" />
        <property name="multiplicity" value="*" />
    </map>
</relationship>
<relationship type="Association" name="composição" direction="forward">
    <map id="map2">
        <property name="end_name" value="todo" />
        <property name="aggregation" value="composite" />
    </map>
    <map id="map3">
        <property name="end_name" value="parte" />
    </map>
</relationship>
```

Figura 3.13: Exemplo de geração de associação e composição.

Outra situação que deve ser levada em consideração é a necessidade de especificar a propagação de relacionamentos existentes no modelo de origem, para o modelo de destino. A Figura 3.14 ilustra esta situação, onde os relacionamentos entre os elementos do modelo de origem devem ser propagados apenas para os elementos gerados pelos mapeamentos *map1* e *map3*. Esta definição é realizada através do atributo *preserve-relationships* presente na especificação dos mapeamentos *map1* e *map3*.




```
<classifier-map name="map1" preserve-relationships="true">...</classifier-map>
<classifier-map name="map2">...</classifier-map>
<classifier-map name="map3" preserve-relationships="true">...</classifier-map>
<classifier-map name="map4">...</classifier-map>
```

Figura 3.14: Exemplo de propagação de relacionamentos.

3.4 – Apoio à Execução de Transformações Modelo-Modelo

Após as transformações terem sido definidas, o Engenheiro pode então executá-las sobre os modelos do sistema em desenvolvimento. Neste contexto, um aspecto importante é a interoperabilidade entre a ferramenta *CASE* utilizada na modelagem e a ferramenta de execução de transformações. Dois cenários de interoperabilidade podem ser identificados. No primeiro, a ferramenta *CASE* se comunica diretamente com a ferramenta de transformação de modelos de forma transparente ao Engenheiro, isto é, as funcionalidades de transformação são chamadas diretamente da ferramenta *CASE* sem a necessidade prévia de exportação dos modelos.

No segundo cenário, a ferramenta *CASE* utilizada na modelagem não possuí uma interface de comunicação com a ferramenta de transformação. Ainda assim, o Engenheiro pode exportar o seu modelo em formato padronizado, através de *XMI*, e importar tal modelo na ferramenta de transformação, que possui uma interface gráfica própria. Após realizar as transformações, o modelo resultante pode ser exportado, também através de *XMI*, e importado de volta à ferramenta *CASE* original.

3.4.1 – Marcação de Modelos

Conforme visto anteriormente, algumas transformações entre modelos necessitam de marcações prévias para guiar sua execução. Essa marcação consiste basicamente na aplicação de estereótipos, etiquetas e restrições *OCL* sobre elementos do modelo. Considerando os dois cenários citados anteriormente, a utilização de uma

ferramenta *CASE* para realizar tal marcação é possível, porém pode não ser tão eficiente. Isso ocorre pois algumas ferramentas não provêem apoio adequado a essa etapa, obrigando o engenheiro a selecionar cada item do modelo de forma separada e aplicar tais marcações. Além de pouco produtivo, essa aplicação mantém a marcação vinculada ao modelo, o que dificulta a aplicação de outras marcações visando a realização de outras transformações.

Uma outra possibilidade é utilizar a ferramenta de transformações para realizar tais marcações. Com isso, o projetista poderia selecionar um perfil *UML* a ser aplicado sobre o modelo, e para cada marcação definida no perfil, ele poderia marcar vários elementos do modelo de uma só vez. Por exemplo, através da ferramenta, o engenheiro pode selecionar todos os elementos do modelo que devem receber o estereótipo *<<entity>>*. Da mesma forma, a ferramenta possibilita ainda a remoção de um perfil para a aplicação de outro, para que outra transformação seja executada.

A execução de uma transformação de modelos, na abordagem proposta neste trabalho, é realizada em vários estágios. Inicialmente, o engenheiro realiza a configuração da transformação, depois a máquina de transformação instancia os mapeamentos, e o engenheiro, então, pode escolher os mapeamentos desejados. A máquina de transformação executa os mecanismos dos mapeamentos e os relacionamentos são gerados. Estes estágios são detalhados a seguir.

3.4.1.1 – Configuração Inicial

A configuração inicial é necessária para que o engenheiro possa controlar o comportamento da execução da transformação. Neste estágio, ele deve fazer as seguintes escolhas:

a) Escolha do modelo de origem da transformação.
b) Escolha do modelo de destino da transformação. Caso o modelo de destino não exista, o engenheiro deve escolher um nome para o modelo que será criado pela transformação. O engenheiro pode também selecionar o mesmo modelo selecionado como origem.
c) Escolha da direção da transformação (direta ou reversa).

3.4.1.2 – Instanciação dos Mapeamentos

Após a configuração, a execução da transformação dá início à instanciação dos mapeamentos. Neste estágio, a máquina de transformação avalia a especificação da

transformação e, para cada mapeamento definido, a máquina realiza a busca dos elementos no modelo de origem que satisfazem aos critérios definidos nestes mapeamentos. Na transformação *EJBComponents* por exemplo, o mapeamento *ClassifierMap* responsável por mapear classes entidade (*PIM*) em classes de implementação do *entity bean* (*PSM*) é instanciado para cada classe marcada com estereótipo <<*entity*>> no modelo de entrada.

Em seguida, caso o mapeamento avaliado possua sub-mapeamentos, os mesmos também são instanciados. No exemplo anterior, o mapeamento *ClassifierMap* possui, entre outros sub-mapeamentos, um *FeatureMap* responsável por transferir atributos da classe de origem para a classe de destino. Assim, a máquina instancia um *FeatureMap* para cada atributo encontrado na classe de origem. O mesmo ocorre com os mapeamentos do tipo *ClassifierFeatureMap*.

3.4.1.3 – Pré-Visualização dos Mapeamentos

Após a máquina de transformações instanciar todos os mapeamentos de acordo com os critérios de busca, o engenheiro pode então avaliar essa lista de mapeamentos e, neste estágio, ele pode marcar quais mapeamentos serão efetuados posteriormente. A transformação *EJBComponents*, por exemplo, ao gerar as interfaces definidas na plataforma *EJB*, instancia os mapeamentos para todas as interfaces possíveis, locais e remotas. No entanto, as interfaces locais nem sempre são necessárias em todos os componentes *entity bean*. Assim, o engenheiro pode desmarcar os mapeamentos que são responsáveis pelas interfaces indesejadas.

É importante mencionar que essa estratégia pode fazer com que a execução da transformação se torne um processo menos automatizado. Neste caso, o projetista da transformação deve rever a especificação dos mapeamentos e, se possível, definir critérios de busca mais precisos. No exemplo das interfaces locais, uma etiqueta chamada "localExecution" poderia ser aplicada às classes *PIM*, no momento de sua marcação, e assim, critérios de busca para essa marcação poderiam ser utilizados nos mapeamentos responsáveis pela criação das interfaces locais dos componentes *EJB*.

3.4.1.4 – Execução dos Mecanismos

Após a confirmação dos mapeamentos desejados, a máquina de transformações percorre todos os mapeamentos, e executa cada um dos mecanismos associados. Nesta

etapa, de acordo com a configuração de cada tratador instanciado nos mecanismos, os elementos são criados no modelo de destino.

A criação de elementos no modelo de destino é feita de acordo com as configurações definidas nas propriedades dos mapeamentos. Com isso, ao executar o mecanismo do mapeamento entre a classe entidade (*PIM*) e a classe de implementação do *entity bean* (*PSM*), as configurações de tal mapeamento indicam como a classe de destino deverá ser criada. Por exemplo, uma classe de origem chamada "Cliente" (*PIM*) é transformada, de acordo com a configuração, para uma classe chamada "ClienteBean". Caso já exista no modelo de destino, um elemento do mesmo tipo (classe), mesmo nome ("ClienteBean") e localizado no mesmo espaço de nomes (*namespace*), isto indica que um mapeamento anterior já criou tal elemento ou, que o modelo de destino (um modelo pré-existente) já possui tal elemento. Neste caso, a execução do mecanismo não afeta as informações não configuradas através das propriedades. Este tratamento é importante para manter um certo sincronismo entre os modelos de origem e destino.

Consideramos agora um cenário onde a transformação *EJBComponents* tenha sido executada, criando um novo modelo de componentes *EJB*, desvinculado do modelo *PIM* inicial. Após a execução dessa transformação, o engenheiro realiza uma alteração incremental no modelo de destino, adicionando alguns atributos e operações em algumas classes. Paralelamente, alterações foram feitas no modelo *PIM* original e com isso, os modelos perderam o sincronismo.

Nessa situação, o engenheiro pode executar novamente a transformação direta (*PIM* → *PSM*), selecionando as versões atuais dos modelos de origem e destino, para atualizar o modelo *EJB* de acordo com as alterações realizadas no modelo *PIM*. Uma execução reversa poderia trazer as informações adicionadas no modelo *EJB* para o modelo *PIM*.

3.4.1.5 – Tratamento dos Relacionamentos

Após a máquina de transformação ter executado todos os mecanismos associados aos mapeamentos, é possível, agora, gerar os relacionamentos esperados no modelo de destino. Esta etapa é dividida em duas fases:

a) Geração de novos relacionamentos no modelo de saída – Conforme apresentado na Seção 3.3.2.3, a especificação da transformação pode definir alguns relacionamentos entre os elementos gerados pela execução

dos mecanismos dos mapeamentos. Nesta fase, a máquina avalia estas definições e cria os relacionamentos entre os elementos que foram gerados e transformados na etapa anterior.

b) Propagação dos relacionamentos existentes no modelo de entrada – Nesta fase, a máquina de transformação transfere os relacionamentos existentes entre elementos do modelo de entrada para os elementos do modelo de saída. Essa transferência é feita com base nos mapeamentos entre tais elementos. Caso exista, no modelo de origem, um relacionamento entre dois elementos e, esses elementos são elementos de origem em mapeamentos configurados com o atributo *preserve-relationships* contendo o valor *true*, o elemento gerado a partir do primeiro elemento é relacionado, através de uma cópia do relacionamento original, com o elemento gerado a partir do segundo elemento. Por exemplo, caso exista uma ligação entre duas entidades no modelo *PIM*, Cliente e Fornecedor, e o mapeamento, que gera as classes *entity bean* equivalentes no modelo *EJB*, esteja configurado para preservar seus relacionamentos, as classes *ClienteBean* e *FornecedorBean*, no modelo *EJB* recebem um relacionamento de ligação.

3.5 – Transformações Modelo-Texto

Além das transformações entre modelos, uma outra necessidade é prover ao engenheiro uma forma de obter a implementação desses modelos na plataforma desejada, isto é, transformar os modelos em código-fonte. Segundo a classificação das abordagens apresentadas no Capítulo 2, este tipo de transformação tem grande importância no processo de desenvolvimento, podendo aumentar a qualidade do código desenvolvido, ajudar na consistência e previsibilidade do código gerado, aumentar a produtividade dos programadores, possibilitar um aumento no nível de abstração dos modelos e apoiar a reutilização do código gerado (HEYSE *et al.*, 2005).

De maneira geral, algumas abordagens existentes atendem ao requisito de interoperabilidade através da utilização de *XMI*. No entanto, o requisito de genericidade não é totalmente atendido, apesar das abordagens preverem esse tipo de extensão. Já a geração não destrutiva ou *round-trip* dificilmente são encontradas nas abordagens de geração independentes de algum ambiente de modelagem. Essas características são normalmente encontradas em ferramentas *CASE* como o *Poseidon* (GENTLEWARE,

2005) ou o *Together* (BORLAND, 2006). Assim, optamos por desenvolver uma abordagem simplificada de geração de código, que funciona de forma independente da máquina de transformação de modelos e permite a geração de código por meio do mecanismo de *templates*.

Ainda segundo a classificação do Capítulo 2, a abordagem de transformações do tipo modelo-texto proposta neste trabalho segue uma linha híbrida, apresentando uma máquina genérica que suporta a inclusão de mecanismos de geração de texto baseados no padrão *Visitor* (GAMMA *et al.*, 1995), que navegam no modelo através das interfaces *JMI*, para a extração de informações úteis no preenchimento de *templates* (CZARNECKI & HELSEN, 2003).

A infra-estrutura provida pela abordagem disponibiliza uma máquina de geração de texto genérica, onde mecanismos de geração de texto específicos podem ser executados. Inicialmente, apenas o mecanismo responsável por gerar código *Java* a partir de modelos de classes *UML* foi desenvolvido, mas novos mecanismos e *templates* podem ser adicionados à abordagem, conforme será apresentado na Seção 3.6.

As transformações do tipo modelo-texto desta abordagem são executadas em duas etapas:

a) Inicialmente, o mecanismo de geração de texto desejado é escolhido, e o modelo de entrada é passado para este mecanismo, cuja execução é responsável por navegar e coletar as informações presentes no modelo que serão utilizadas posteriormente para preencher os *templates*. Para a geração de código-fonte *Java* a partir de uma classe *UML*, o mecanismo pode percorrer todos os atributos e operações e coletar informações como nome, visibilidade, tipo dos atributos, tipos dos parâmetros das operações. Estas informações são armazenadas numa estrutura de dados como uma tabela (*Map*).

b) Após a coleta das informações, essa tabela é passada juntamente com o *template* específico, para uma máquina genérica responsável pela leitura da tabela e a escrita dessas informações em campos definidos no *template*. Com isso, um arquivo texto é gerado com as informações presentes no modelo.

3.6 – Possibilidade de Extensão

Com a evolução das tecnologias existentes, novas plataformas se tornam disponíveis e, por isso, novas transformações serão necessárias. Como não é possível prever a forma e o objetivo dessas transformações, esta abordagem foi desenvolvida de modo a permitir a sua extensão em praticamente todos os aspectos apresentados anteriormente. Esta seção detalha como essa extensão pode ser realizada.

A forma mais simples de extensão é justamente a definição de novas transformações baseadas nas regras e padrões exigidos por uma nova plataforma ou tecnologia. Esta definição já foi discutida na Seção 3.3. No entanto, dependendo do objetivo dessas novas transformações, os tratadores existentes nesta abordagem podem não ser suficientes para tratar novos tipos de elementos não previstos inicialmente. Para contornar tal situação, esta abordagem permite ao projetista de transformações a definição de novos tratadores, mecanismos de transformação de elementos, buscadores, e geradores de relacionamentos. No que diz respeito às transformações modelo-texto, a máquina de geração de texto proposta permite a sua extensão para a inclusão de novos mecanismos e *templates*. Caso essa máquina não seja suficiente, a abordagem *Odyssey-MDA* permite ainda a sua utilização em conjunto com outras abordagens existentes para geração de código ou engenharia reversa, desde que essas abordagens possibilitem a interoperabilidade de modelos através de *XMI*.

3.6.1 – Tratadores

Novos tratadores são necessários para que seja possível a transformação de elementos não previstos inicialmente na abordagem. Com a inclusão de novos meta-modelos, os tratadores para cada um dos elementos presentes nesses novos meta-modelos devem ser implementados e disponibilizados na abordagem.

A implementação de um tratador deve realizar a interface ilustrada na Figura 3.15.

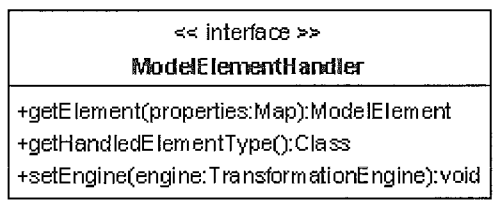


Figura 3.15: Interface dos tratadores.

A operação *getHandledElementType* retorna o tipo de elemento que o tratador reconhece. Por exemplo, caso o projetista tenha necessidade de implementar um tratador para casos de uso, essa operação deve retornar a classe do meta-modelo da *UML* da qual os elementos tratados são instâncias. Neste caso, a superclasse *UseCase*.

Conforme visto anteriormente, os tratadores são utilizados para se montar um mecanismo em tempo de execução. Ao executar sua transformação, o mecanismo chama a operação *getElement* do tratador, passando as configurações através das propriedades. Com isso, essa operação consulta, por meio de *callback*, a máquina de transformações, para verificar a existência de um elemento pré-existente com o mesmo nome informado pelas propriedades. Caso exista, esse elemento pré-existente é configurado pelo tratador e retornado, caso contrário, um novo elemento é instanciado, configurado e retornado. Toda essa manipulação dos elementos é realizada através das interfaces *JMI* disponíveis na infra-estrutura.

3.6.2 – Mecanismos

A inclusão de novos tratadores pode não ser suficiente para permitir a realização das transformações desejadas pelo projetista. Isso ocorre pois os tratadores apenas configuram os elementos transformados. A lógica da transformação dos elementos está inserida na instância do mecanismo *built-in* montada em tempo de execução, e não nos tratadores. Caso o projetista deseje algum mecanismo que implemente alguma lógica ou funcionalidade diferente, ele deverá implementar este mecanismo e incorporá-lo à máquina de transformações. Estes novos mecanismos são denominados *plug-ins* e podem ser de três tipos básicos:

a) *Plug-in* de transformação de elementos – Segue a mesma interface do mecanismo *built-in* montado em tempo de execução, ou seja, implementa a interface *Mechanism* que define operações para a transformação de elementos em ambos os sentidos, conforme ilustrado na Figura 3.16.

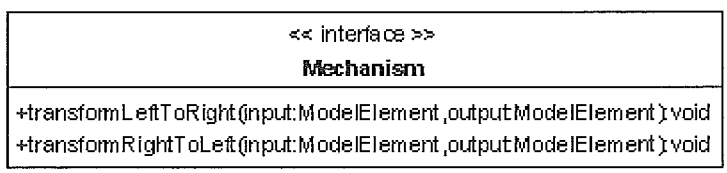


Figura 3.16: Interface dos *plug-ins* de transformação de elementos.

b) *Plug-in* de pré-execução – Caso o projetista necessite de alguma preparação prévia, ele pode implementar um *plug-in* de pré-execução.

Este tipo de mecanismo é executado antes dos mecanismos associados aos mapeamentos terem suas execuções iniciadas.

c) *Plug-in* de pós-execução – Funciona de maneira similar ao tipo de pré-execução, mas é executado após todos os mecanismos terem terminado suas execuções.

Ambos os *plug-ins* de pré e pós-execução, devem ser implementados, realizando a interface *ExecutionCode*, ilustrada na Figura 3.17. Esta interface define apenas uma operação, que cada *plug-in* deve implementar para que a máquina de transformações possa chamar a execução, passando como parâmetro a definição do mapeamento da transformação.

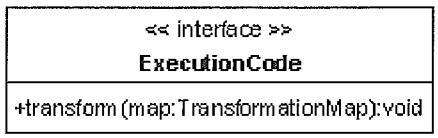


Figura 3.17: Interface dos *plug-ins* de pré e pós execução.

3.6.3 – Buscadores

Novos buscadores podem ser implementados e incorporados à esta abordagem, para possibilitar a utilização de critérios de busca diferentes dos critérios apresentados na Seção 3.3.2.1. Outros buscadores podem, também, possibilitar a utilização de cláusulas *OCL*, e no futuro permitir uma maior compatibilidade com o *QVT* (OMG, 2005a). A implementação desses buscadores se dá pela realização da interface *Finder* apresentada na Figura 3.18. Essa interface define apenas a operação *find* que recebe como parâmetro o espaço de nomes (*namespace*) onde os elementos serão pesquisados, de acordo com o critério informado no parâmetro *value*. A implementação deve pesquisar os elementos e retornar uma coleção com os elementos.

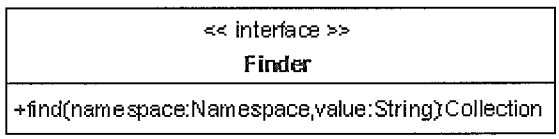


Figura 3.18: Interface dos buscadores.

3.6.4 – Geradores de Relacionamentos

Com a inclusão de novos meta-modelos, assim como novos tratadores devem ser implementados para a manipulação dos novos tipos de elementos definidos nesses novos meta-modelos, novos tipos de relacionamento também poderão estar presentes e

assim, geradores desses relacionamentos podem ser desenvolvidos pelo projetista e, incorporados à esta abordagem. A Figura 3.19 ilustra a interface que os geradores de relacionamento devem realizar. Esta interface define uma operação chamada *generate*, que é chamada pela máquina de transformação, a qual informa os participantes de tal relacionamento e as propriedades de configuração do mesmo.

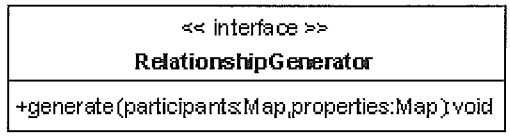


Figura 3.19: Interface dos geradores de relacionamento.

3.6.5 – Mecanismos de Geração de Texto e *Templates*

Conforme foi apresentado anteriormente, a máquina de geração de código proposta nesta abordagem apenas realiza a geração de código *Java* a partir de modelos de classes da *UML*. Entretanto, caso o projetista necessite de outros mecanismos como, por exemplo, geração de código para a linguagem C++, ou até mesmo a geração de algum tipo de documento de configuração, como descritores de implantação de componentes *enterprise beans* (em linguagem *XML*), ele pode desenvolver esses novos mecanismos seguindo a interface *CodeGenEngine* ilustrada na Figura 3.20.

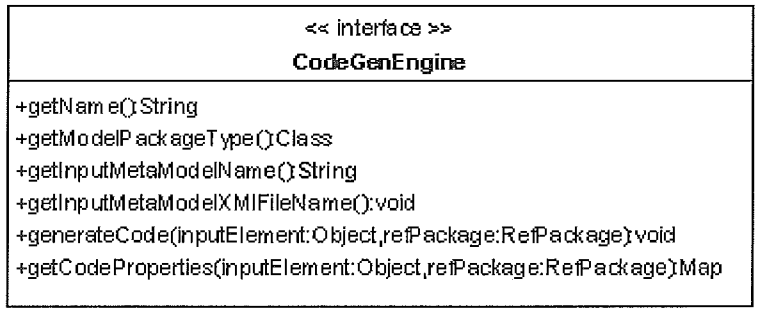


Figura 3.20: Interface dos mecanismos de geração de texto.

Cada mecanismo de geração de texto deve implementar, entre outras operações, a operação *generateCode,* responsável por gerar o código a partir do elemento (parâmetro *inputElement*) presente no modelo (parâmetro *model*). A operação *getCodeProperties* deve retornar uma coleção do tipo *Map*, contendo as propriedades que serão utilizadas para preencher o *template* específico para o tipo do elemento informado.

3.7 – Considerações Finais

Este capítulo descreveu uma abordagem prática para a definição e execução de transformações entre modelos, e geração de código, que atende a alguns critérios levantados no capítulo anterior, como:

- *Independência de ambiente de desenvolvimento*: Esta abordagem não restringe o seu uso a algum ambiente em particular. O projetista de transformações e o Engenheiro de *Software*, podem utilizar a sua ferramenta *CASE* ou ambiente de modelagem de sua preferência, desde que a interoperabilidade seja mantida com a utilização de *XMI* para o intercâmbio de modelos.
- *Apoio à definição e execução de transformações do tipo modelo-modelo*: É possível definir transformações do tipo modelo-modelo através da definição de mapeamentos entre os elementos do modelo, e os mecanismos associados a estes mapeamentos.
- *Apoio à definição e execução de transformações do tipo modelo-texto*: Além das transformações modelo-modelo, a abordagem permite a definição de transformações do tipo modelo-texto, para a geração da implementação dos modelos na linguagem de programação utilizada na plataforma escolhida.
- *Apoio à definição e execução de transformações bidirecionais entre modelos*: A abordagem permite a definição de transformações modelo-modelo que podem ser executadas de forma bidirecional e possibilita a manutenção do sincronismo entre os modelos.
- *Possibilidade de Extensão*: Vários aspectos existentes na abordagem podem ser estendidos, visando a sua utilização na definição de novas transformações e novas plataformas.

Com relação ao critério da utilização de formatos e linguagens padronizados, optamos por utilizar uma sintaxe *XML* para a escrita das definições das transformações. No futuro, espera-se realizar um trabalho para alinhar esta proposta com o padrão definido pela *OMG* com o *QVT* (OMG, 2005a).

Ao executar transformações do tipo modelo-modelo, poderão ocorrer conflitos relacionados à remoção ou nomenclatura dos elementos pré-existentes no modelo de destino. Uma solução para estes conflitos não faz parte do escopo inicial deste trabalho,

mas uma extensão da abordagem para a adoção desta característica poderá ser realizada no futuro.

Para demonstrar a utilização da abordagem, foi apresentado um exemplo de transformação denominada *EJBComponents* cujo objetivo é transformar um modelo de classes da *UML*, independente de plataforma, em um modelo de componentes na plataforma *EJB*.

No próximo capítulo, é descrito um protótipo que implementa a abordagem proposta, discutindo sua utilização, de acordo com as características apresentadas até aqui.

Capítulo 4 – Protótipo Implementado

4.1 – Introdução

No capítulo anterior, foi apresentada a proposta da abordagem de transformação de modelos e geração de código, denominada *Odyssey-MDA*, cujo propósito é auxiliar o Engenheiro de *Software* na definição e execução de transformações sobre modelos, visando a transformação de modelos independentes de plataforma em modelos específicos para uma plataforma escolhida e a posterior obtenção da implementação destes modelos. Para apoiar a utilização da abordagem na prática, apresentamos, neste capítulo, a implementação de um protótipo da máquina de transformações e geração de código *Odyssey-MDA* (MAIA *et al.*, 2005).

Considerando que a utilização de transformações para a evolução de modelos de componentes representa apenas parte do processo de desenvolvimento de aplicações, é necessária a integração do protótipo desenvolvido com outros ambientes de desenvolvimento de *software* ou ferramentas *CASE*. Neste sentido, a implementação do protótipo foi realizada visando a sua utilização tanto no contexto do ambiente de desenvolvimento *Odyssey* (ODYSSEY, 2006), ou de forma *stand-alone*, para possibilitar a utilização da abordagem em conjunto com outras ferramentas *CASE* disponíveis no mercado (e.g.: Poseidon for *UML*, Together) (GENTLEWARE, 2005; BORLAND, 2006).

Este capítulo está dividido da seguinte forma: a Seção 4.2 apresenta o contexto de utilização da ferramenta *Odyssey-MDA*, de forma *stand-alone* e no ambiente *Odyssey*. A Seção 4.3 apresenta a implementação da ferramenta *Odyssey-MDA*, descrevendo suas principais características. A Seção 4.4 descreve a utilização da ferramenta, e a Seção 4.5 conclui este capítulo com algumas considerações finais.

4.2 – Contexto de Utilização

Nesta Seção apresentamos a utilização *stand-alone*, onde a ferramenta pode ser utilizada de forma independente de alguma ferramenta *CASE*, e apresentamos também a utilização dentro ambiente *Odyssey*, onde o protótipo implementado pode ser instalado e ser utilizado para transformar os modelos e gerar código.

4.2.1 – *Stand-alone*

Conforme apresentado no capítulo anterior, a abordagem apresentada pode ser utilizada de forma independente de alguma ferramenta *CASE* em particular. Neste caso, a implementação discutida neste capítulo também leva em consideração o contexto de utilização *stand-alone*, onde a ferramenta implementada possui uma interface gráfica própria que permite a importação e exportação de modelos através de *XMI* e a execução das transformações disponíveis.

4.2.2 – Ambiente *Odyssey*

O *Odyssey* (ODYSSEY, 2006) é um ambiente de desenvolvimento de *software* que oferece ferramentas para apoiar a construção de aplicações através da reutilização. Para isso, o ambiente contempla as atividades de desenvolvimento *para* reutilização, através da Engenharia de Domínio (ED), e desenvolvimento *com* reutilização, através da Engenharia de Aplicação (EA). O ambiente oferece os processos de reutilização conhecidos como *Odyssey-ED* (BRAGA, 2000) e *CDB-Arch-DE* (BLOIS *et al.*, 2004) para apoiar a Engenharia de Domínio, e o processo *Odyssey-EA* (MILLER, 2000) para apoiar a Engenharia de Aplicação.

O enfoque principal do ambiente *Odyssey* está na especificação e construção de modelos de domínio que possam ser reutilizados posteriormente pela equipe de desenvolvimento. O conhecimento contido nos modelos de domínio abrange desde elementos conceituais até artefatos de projeto (como, por exemplo, classes e diagramas de seqüência). Isso possibilita que o processo de Engenharia de Aplicação reutilize esses elementos durante o desenvolvimento de novos projetos, buscando reduzir o tempo e o esforço empregado.

O ambiente *Odyssey* é composto por um conjunto de ferramentas que procuram apoiar as etapas definidas pelos processos de reutilização. Dentre elas, podemos citar a ferramenta de documentação de artefatos (MURTA, 1999); especificação e instanciação de arquiteturas específicas de domínios (XAVIER, 2001); camada de mediação e navegação inteligente (BRAGA, 2000); ferramentas para a modelagem e acompanhamento de processos (MURTA *et al.*, 2002); ferramenta de notificação de críticas em modelos *UML* (DANTAS, 2001); apoio à engenharia reversa (VERONESE & NETTO, 2001); suporte a padrões no projeto de *software* (DANTAS *et al.*, 2002); ferramenta para controle de versões de modelos (OLIVEIRA *et al.*, 2004); apoio à

edição concorrente de modelos (LOPES *et al.*, 2004), entre outras. Todo o ambiente foi implementado utilizando-se como base a linguagem *Java* (SUN, 2005).

Assim como as demais ferramentas, o protótipo da ferramenta *Odyssey-MDA* foi implementado de forma a ser incorporado ao ambiente *Odyssey* através do mecanismo de carga dinâmica de componentes presente no ambiente (MURTA *et al.*, 2004). Este mecanismo permite a instalação dos componentes sob demanda. A lista de componentes disponíveis para instalação é carregada em formato *XML* (W3C, 2004) a partir de um servidor específico. Com base nessa lista, o usuário pode selecionar os componentes que deseja instalar no ambiente, e os mesmos são transferidos do servidor para o ambiente do usuário, através do protocolo *HTTP* (W3C, 2005).

4.3 – Detalhes de Implementação

A ferramenta *Odyssey-MDA*, conforme pode ser visto na Figura 4.1, é dividida em dois módulos principais e um repositório de modelos, os quais são detalhados nas próximas subseções.

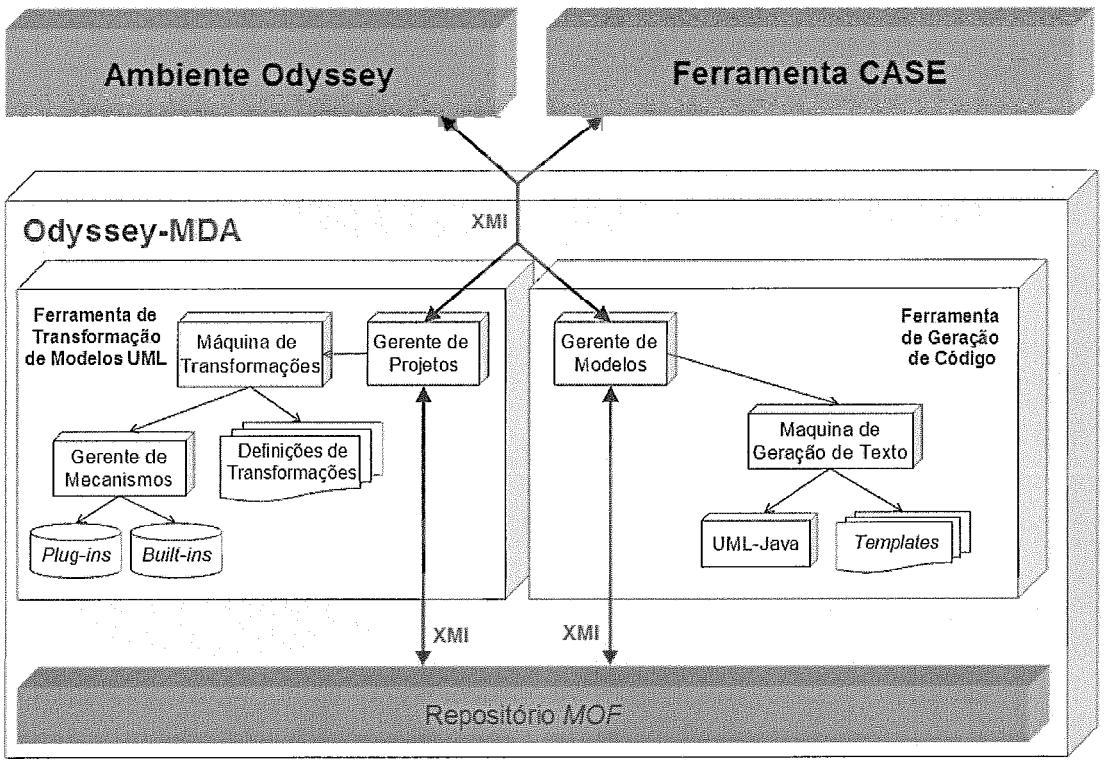


Figura 4.1: Estrutura usada na implementação do protótipo.

A interoperabilidade de modelos entre os ambientes de desenvolvimento de *software* interessados na abordagem e a ferramenta *Odyssey-MDA* é possível através do padrão *XMI* (*XML Metadata Interchange*) (OMG, 2003c). A especificação *XMI* define

um conjunto de regras que mapeiam os meta-modelos *MOF* e os modelos em documentos *XML* que podem ser reconhecidos pelos ambientes.

Para que a ferramenta *Odyssey-MDA* seja capaz de realizar transformações sobre modelos desenvolvidos em outras ferramentas *CASE*, foi implementada uma interface gráfica onde o usuário pode criar um projeto de transformação de modelos, importar seus modelos através do padrão *XMI*, executar transformações, e exportar os modelos de volta para a ferramenta *CASE* de origem. No caso da geração de código, foi implementada uma interface em linha de comando, onde o usuário pode chamar a geração de código, informando o arquivo *XMI* que contém o modelo e o mecanismo de geração de código desejado (e.g.: *UML* para *Java*).

Para realizar as transformações e geração de código sobre os modelos desenvolvidos no Ambiente *Odyssey*, foi necessária a implementação de uma um mecanismo que permita ao *Odyssey* solicitar as funcionalidades da ferramenta *Odyssey-MDA*. Esse mecanismo é uma implementação da interface *Tool* (MURTA *et al.*, 2004) (Figura 4.2) que informa ao *Odyssey* uma lista dos menus com as opções de transformações de modelos e geração de código. Com isso, a transferência dos modelos, através de *XMI*, entre o ambiente de desenvolvimento e a ferramenta *Odyssey-MDA* é feita de forma transparente ao usuário.

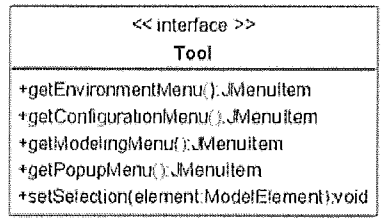


Figura 4.2: Interface *Tool*.

4.3.1 – Repositório *MOF*

O repositório *MOF* utilizado nessa implementação é conhecido como *MDR* (*Metadata Repository*) (MATULA, 2003). O *MDR* é um módulo da *IDE NetBeans* (NETBEANS, 2005) capaz de armazenar qualquer meta-modelo *MOF* (OMG, 2002b) e instâncias desse meta-modelo. Na ferramenta *Odyssey-MDA*, o repositório *MDR* é o componente utilizado para armazenar e gerenciar o meta-modelo da *UML* e os modelos *UML* que serão transformados.

O *MDR* tem sua implementação baseada em *JMI* (*Java Metadata Interface*) (DIRCZE, 2002), que possibilita a geração de interfaces em linguagem *Java* (*APIs*), a partir de meta-modelos *MOF*, e a utilização dessas *APIs* para manipular os elementos

armazenados no repositório. Utilizando *JMI*, é possível implementar os mecanismos que transformam os elementos da *UML* (i.e. classe, atributo, operação, etc.), que são instâncias do meta-modelo *MOF*, de forma programática, através da *API* do meta-modelo da *UML*. A vantagem da utilização de *JMI* neste trabalho, é que a ferramenta implementada passa a ser extensível, possibilitando ao usuário, desenvolver novos mecanismos de transformação (*plug-ins*), apenas conhecendo a *API* de manipulação dos elementos que ele deseja transformar.

4.3.2 – Ferramenta de Transformações de Modelos *UML*

No Capítulo 2, foram identificados dois principais tipos de transformações, transformações do tipo modelo-modelo e transformações modelo-texto. Na ferramenta *Odyssey-MDA*, o módulo responsável pela realização de transformações do tipo modelo-modelo é composto de três componentes:

- Gerente de Projetos: responsável por criar, salvar e carregar os projetos de transformação. Em um projeto de transformação, o usuário pode importar, exportar e transformar modelos. É possível salvar um projeto com seus modelos para uma execução posterior das transformações. O Gerente de Projetos também é responsável por realizar essa importação e exportação dos modelos, através de *XMI*, para o repositório *MOF*.
- Máquina de Transformações: responsável por ler e validar os arquivos *XML* das definições das transformações; analisar o modelo de entrada da transformação e instanciar os mapeamentos necessários; solicitar a execução de cada um dos mecanismos associados aos mapeamentos; e gerar os relacionamentos no modelo de saída.
- Gerente de Mecanismos: instancia os mecanismos (*built-ins* ou *plug-ins*) solicitados pela Máquina de Transformações.

Devido à complexidade da implementação desses componentes, apresentamos na Figura 4.3 um diagrama de classes simplificado, contendo apenas as classes mais importantes da ferramenta de transformações de modelos.

A classe *MDAFacade* implementa o padrão *Facade* (GAMMA *et al.*, 1995), que torna possível o acesso às funcionalidades em um único ponto, chamado de Fachada. Através dos métodos presentes na Fachada, os ambientes interessados podem chamar as operações definidas no Gerente de Projetos (*ProjectManager*), tais como criar, salvar e abrir projetos de transformações; importar modelos para o projeto de transformação e

exportar os modelos existentes no projeto. Ainda através da Fachada, os ambientes também podem obter da Máquina de Transformações (*TransformationEngine*), a lista de transformações disponíveis (agrupadas por plataformas ou independentes de plataforma) e iniciar essas transformações sobre os modelos.

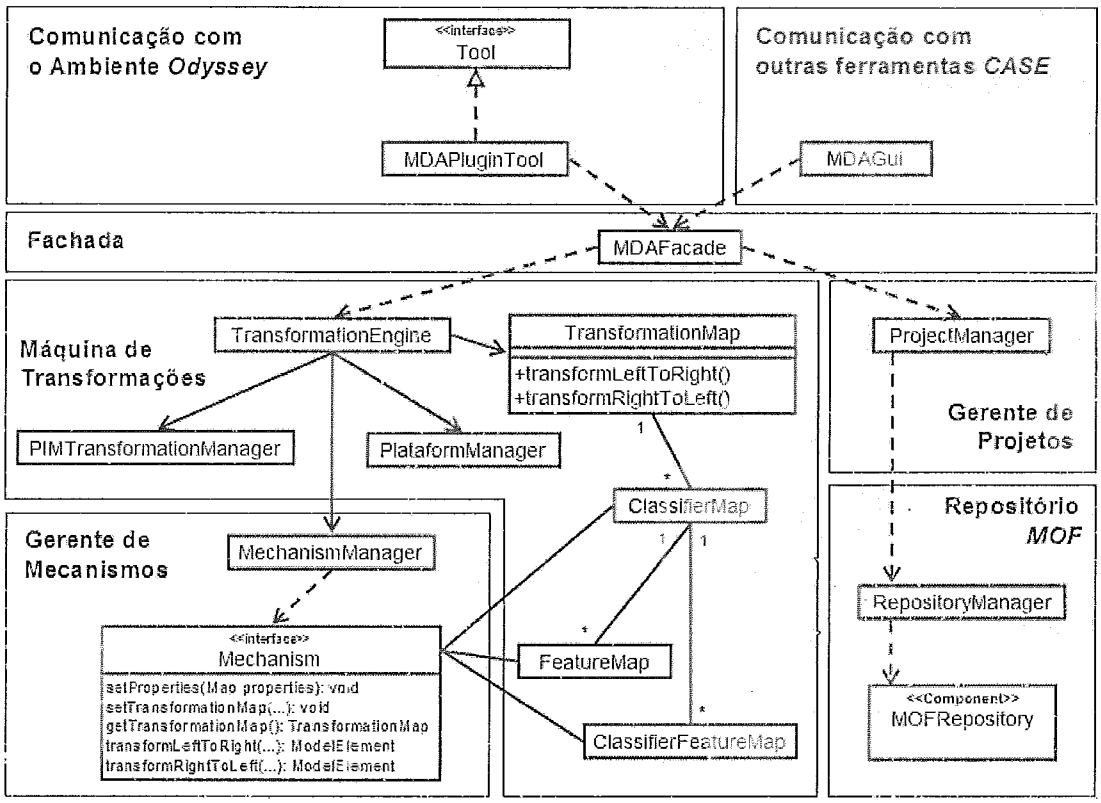


Figura 4.3: Diagrama de classes da ferramenta de transformação de modelos

A classe *MDAPluginTool* implementa a interface *Tool* que permite a comunicação entre o Ambiente *Odyssey* e a ferramenta *Odyssey-MDA* de forma transparente. Já a classe *MDAGui* implementa a interface gráfica para a utilização *stand-alone*. Essa interface gráfica permite a chamada das funcionalidades presentes na fachada.

A Máquina de Transformações, representada pela classe *TransformationEngine*, utiliza os Gerentes de Transformações (*PIMTransformationManager* e *PlataformManager*) para apresentar uma lista das transformações disponíveis ao usuário. Após a escolha da transformação desejada, a máquina instancia o mapeamento da transformação (*TransformationMap*) e consulta o Gerente de Mecanismos (*MechanismManager*) para obter mecanismos para cada mapeamento, com os tratadores específicos para cada elemento mapeado.

A ferramenta *Odyssey-MDA* provê uma infra-estrutura de tratadores genéricos, denominados *built-ins*, os quais são utilizados pelos mecanismos instanciados em tempo

de execução, para realizam transformações simples sobre os elementos de modelos *UML*. A abordagem proposta no capítulo anterior ilustra os *built-ins* implementados na ferramenta.

A realização de uma transformação se dá pela execução de cada mecanismo associado aos mapeamentos. Caso os tratadores presentes na ferramenta não sejam suficientes para a definição de uma transformação desejada, o projetista da transformação poderá implementar novos tratadores e incorporar à ferramenta. Para isso, ele deve implementar a interface *ModelElementHandler* apresentada no capítulo anterior. Caso seja necessária a implementação de uma transformação não convencional, isto é, cuja lógica não seja possível de se implementar através de um mecanismos e tratadores, o projetista pode implementar seu próprio mecanismo, através da implementação da interface *Mechanism* apresentada no capítulo anterior. Estes mecanismos definidos pelo Projetista são denominados *plug-ins*.

4.3.3 – Ferramenta de Geração de Código

O segundo módulo, denominado Ferramenta de Geração de código, realiza as transformações do tipo modelo-texto e é composto de dois componentes, denominados Gerente de Modelos e Máquina de Geração de Texto. A Figura 4.4 apresenta um diagrama de classes contendo as classes mais importantes da ferramenta.

Assim como a ferramenta de transformação de modelos, a ferramenta de geração de código também possui uma classe que centraliza o acesso às funcionalidades através do padrão *Facade*. Essa classe, *CodeGenFacade*, define duas operações. A primeira (*getEngineKeys*), retorna a lista de mecanismos de geração de código (*engines*) disponíveis na ferramenta, já a segunda operação (*generateCode*), inicia o processo de geração de código. No momento da chamada da operação *generateCode*, é informado o mecanismo de geração desejado (e.g.: *UML-Java*), o nome do arquivo *XMI* do modelo e o local onde o código deve ser gerado. A Fachada, então, se comunica com o Gerente de Modelos (*ModelManager*), que é o responsável por realizar a importação e exportação dos modelos, através de *XMI*, para o repositório *MOF*.

A Máquina de Geração de Texto é responsável por controlar o acesso aos mecanismos (*engines*). Cada mecanismo é específico para um tipo de geração (e.g.: *UML-Java*), e é implementado através de um conjunto de tratadores (*handlers*) que navegam pelo modelo *UML* através da *API JMI* dos elementos, coletando as propriedades que serão utilizadas para preencher *templates* de código-fonte. A

implementação dos *handlers* segue o padrão de projeto conhecido como *Visitor* (GAMMA *et al.*, 1995), o qual permite a definição de novos tratadores e inclusão na infra-estrutura de geração de código, sem que sejam necessárias mudanças estruturais. Para a manipulação dos *templates*, a Máquina de Geração de Texto utiliza o mecanismo conhecido como *Velocity* (APACHE, 2005), o qual permite a junção das informações contidas nos *templates* com as propriedades coletadas pelos *handlers*.

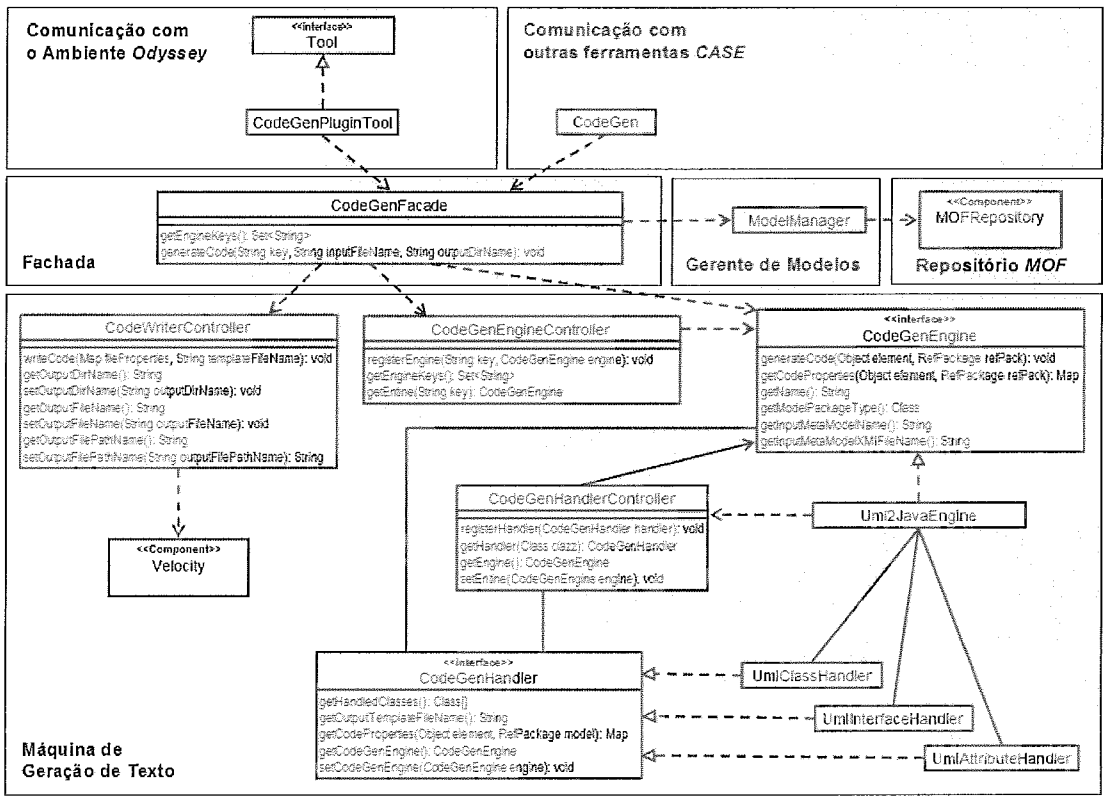


Figura 4.4: Diagrama de classes da ferramenta de geração de código

A decisão de se implementar essa geração através de *handlers* e *templates* se deve à facilidade em se implementar a navegação em modelos através da *API JMI* disponível, e com isso, permitir que novos mecanismos de geração para outras linguagens sejam implementados pelo usuário, e incorporados a essa ferramenta.

4.4 – A Utilização do Protótipo

Nas subseções seguintes, são detalhadas as atividades de definição de transformações entre modelos, execução das transformações entre modelos, definição de *templates* e mecanismos de geração de código, finalizando com a utilização do protótipo para executar a transformação definida e realização da geração de código. Para ilustrar o funcionamento do protótipo, utilizaremos a transformação *EJBComponents* apresentada no Capítulo 3.

4.4.1 – Definição de transformações modelo-modelo

Para a realização de uma transformação sobre um modelo, é preciso definir os mapeamentos entre os elementos do modelo a ser transformado, denominado modelo de entrada, e os elementos desejados no modelo resultante da transformação, denominado modelo de saída. A especificação de uma transformação deve ser elaborada em formato *XML*, e deve conter o conjunto de mapeamentos que compõem essa transformação. A Figura 4.5 mostra um trecho da especificação *EJBComponents*. Na linha 4 da especificação, pode ser vista a definição do mapeamento *ClassifierMap* entre os componentes do modelo *PIM* e os componentes do modelo *EJB*. Na linha 18, temos a especificação do mapeamento *ClassifierMap* entre a classe que representa a implementação do componente *PIM* e a classe que implementa o *EntityBean* no componente *EJB* resultante do modelo de saída. Aninhado neste último mapeamento, o sub-mapeamento *FeatureMap* da linha 39 realiza a cópia dos atributos entre as classes relacionadas no *ClassifierMap* da linha 15.

Para fazer a seleção dos elementos no modelo de entrada que serão mapeados em elementos no modelo de saída, são necessários alguns critérios de busca para a seleção de elementos, os buscadores (*finders*). Para cada mapeamento, podemos ter buscadores que selecionam elementos em cada lado da transformação, no lado esquerdo ou direito, a depender da direção escolhida no momento da execução da transformação. No caso de uma execução direta (direita → esquerda), os buscadores das linhas 7 e 8 da transformação *EJBComponents* fazem a seleção de todos os elementos do modelo de entrada (direita), do tipo *Component* (linha 7) que estejam marcados pelo estereótipo *<<entity>>* (linha 8). Já no caso de execução reversa (esquerda → direita), os buscadores das linhas 9 e 10 selecionam, no modelo de entrada (esquerda), os elementos do tipo *Component* marcados por estereótipos *<<EntityBeanComponent>>*.

Ao definir mapeamentos, o projetista da transformação deve informar os tipos dos elementos mapeados, para que no momento da execução da transformação, a máquina de transformações possa instanciar um mecanismo com os tratadores específicos para cada elemento mapeado. Por exemplo, o *FeatureMap*, definido na linha 39 da Figura 4.5, é um mapeamento entre atributo e atributo, conforme configurado pelas propriedades das linhas 44 e 45. Assim, no momento de sua execução, um mecanismo com dois *built-ins Attribute* é instanciado para transformar atributos do modelo de entrada em atributos do modelo de saída.

```
1  <transformation-mapping name="EJBComponents"
2      implementation-package="br.ufrj.cos.lens.odyssey.tools.mda.transformations.builtin">
3
4      <classifier-map name="Entity-Component-PIM - EntityBean-Component-EJB" ... >
5
6          <!-- FINDERS -->
7          <finder side="left" criteria="type" value="Component" />
8          <finder side="left" criteria="stereotype" value="entity" />
9          <finder side="right" criteria="type" value="Component" />
10         <finder side="right" criteria="stereotype" value="EntityBeanComponent" />
11
12         <property name="left_type" value="Component" />
13         <property name="right_type" value="Component " />
14
15         <!-- FORWARD PROPERTIES -->
16         <property direction="forward" name="stereotype" value="EntityBeanComponent" />
17
18         <classifier-map name="Entity-Class-PIM - EntiyBean-Class-EJB" id="entityBeanClass"
19             preserve-relationships="yes">
20             <!-- FINDERS -->
21             <finder side="left" criteria="stereotype" value="entity" />
22             <finder side="right" criteria="stereotype" value="EJBEntityBean" />
23
24             <property name="left_type" value="Class" />
25             <property name="right_type" value="Class" />
26
27             <!-- FORWARD PROPERTIES -->
28             <property direction="forward" name="name" value="#SOURCE_NAME#Bean" />
29             <property direction="forward" name="stereotype" value="EJBEntityBean" />
30
31             <!-- REVERSE PROPERTIES -->
32             <property direction="reverse" name="name_transformation">
33                 <property name="input" value="#SOURCE_NAME#" />
34                 <property name="regexp" value="(.*)Bean$" />
35                 <property name="subst" value="$1" />
36             </property>
37             <property direction="reverse" name="stereotype" value="entity" />
38
39             <feature-map name="Attribute-PIM - Attribute-EJB" implementation="AttributeAttribute">
40                 <!-- FINDERS -->
41                 <finder side="left" criteria="type" value="Attribute" />
42                 <finder side="right" criteria="stereotype" value="EJBBusinessAttribute" />
43
44                 <property name="left_type" value="Attribute" />
45                 <property name="right_type" value="Attribute" />
46
47                 <!-- FORWARD PROPERTIES -->
48                 <property direction="forward" name="stereotype" value="EJBBusinessAttribute" />
49             </feature-map>
50             ...
51         </classifier-map>
52         ...
53     </classifier-map>
54 ...
55 </transformation-mapping>
```

Figura 4.5: Especificação *XML* da transformação *EJBComponents*.

Conforme visto no capitulo anterior, deve existir uma forma para configurar os elementos manipulados pelos mecanismos e assim, controlar o comportamento da transformação. Para realizar essa configuração, o projetista pode utilizar algumas propriedades pré-definidas. Ainda na transformação da Figura 4.5, as propriedades das linhas 28 e 29 configuram, respectivamente, o nome da classe *PSM*, adicionando o

sufixo "Bean" ao nome da classe (e.g.: Cliente → ClienteBean), e o estereótipo da classe *PSM* (*<<EJBEntityBean>>*).

Para configurar a transformação dos nomes dos elementos, o projetista da transformação pode utilizar substituições com expressões regulares. A propriedade definida nas linhas de 32 até 36, utiliza esse tipo de substituição para realizar a transformação reversa do nome da classe *PSM* e extração do sufixo "Bean" (ClienteBean → Cliente). Outro exemplo desse tipo de substituição é transformar o nome "atributo" em "getAtributo" (transformação direta), ou o nome "setAtributo" em "atributo" (transformação reversa).

Outras propriedades que podem ser configuradas são as características dos elementos, e que estão definidas no meta-modelo da *UML*, as quais foram apresentadas no capítulo anterior. Além disso, caso o projetista decida por implementar novos mecanismos (*plug-ins*), ele poderá definir também o conjunto de propriedades configuráveis desses *plug-ins*.

Uma vez que mapeamentos entre os elementos já foram definidos, resta definir os relacionamentos que serão gerados pela transformação. Para definir um relacionamento, devemos informar os elementos participantes que serão relacionados e o tipo do relacionamento, por exemplo, dependência, generalização ou associação. Os elementos relacionados devem ser identificados nos mapeamentos pelos quais eles foram gerados, como ilustrado na Figura 4.6. A figura, ainda, apresenta a definição da geração de uma dependência (*Dependency*) entre os elementos identificados como *entityBeanClass* e *entityRemoteInterface*.

```
<transformation-mapping name="EJBComponents"
    implementation-package="br.ufrj.cos.lens.odyssey.tools.mda.transformations.builtin">
    ...
        <classifier-map name="Entity-Class-PIM - EntiyBean-Class-EJB" id="entityBeanClass"
            preserve-relationships="yes">
        ...
        <classifier-map name="Entity-Class-PIM - EntiyBean-RemoteInterface-EJB" id="entityRemoteInterface">
        ...
        <relationship name="Dependency (entityBean -> entityRemoteInterface)" type="Dependency"
            direction="forward">
            <map id="entityRemoteInterface">
                <property name="supplier" value="yes">
            </map>
            <map id="entityBeanClass">
                <property name="client" value="yes">
            </map>
        </relationship>
    ...
</transformation-mapping>
```

Figura 4.6: Definição da geração de relacionamento entre elementos.

Caso seja necessária a configuração do relacionamento gerado, o projetista pode utilizar as propriedades de configuração, de forma similar à utilizada para configurar os mecanismos. Com isso, ele pode, por exemplo, informar que uma associação gerada é navegável ou não, ou informar que uma associação é uma agregação ou composição.

4.4.2 – Execução de transformações modelo-modelo

Agora que temos a transformação *EJBComponents* definida e incorporada à ferramenta *Odyssey-MDA*, podemos apresentar a mesma execução no contexto de utilização *stand-alone* e dentro do ambiente de reutilização *Odyssey*. A seguir, apresentaremos essa execução, destacando suas principais atividades.

4.4.2.1 – *Stand-alone*

Ao utilizar a ferramenta no contexto *stand-alone*, o usuário pode elaborar seu modelo na ferramenta *CASE* de sua preferência, desde que a ferramenta escolhida possibilite a exportação e importação de modelos em formato *XMI*. Uma ferramenta que, atualmente, apresenta este recurso é o *Poseidon for UML* (GENTLEWARE, 2005).

Para realizar a transformação *EJBComponents*, os elementos do modelo de entrada (*PIM*) que serão mapeados em elementos do modelo *EJB* devem ser marcados de acordo com os critérios definidos pelos buscadores no momento da definição da transformação. Essa marcação pode ser realizada na ferramenta *CASE* de origem ou através de um *plug-in* desenvolvido para a ferramenta *Odyssey-MDA*, denominado *ModelMarker* (Figura 4.7).

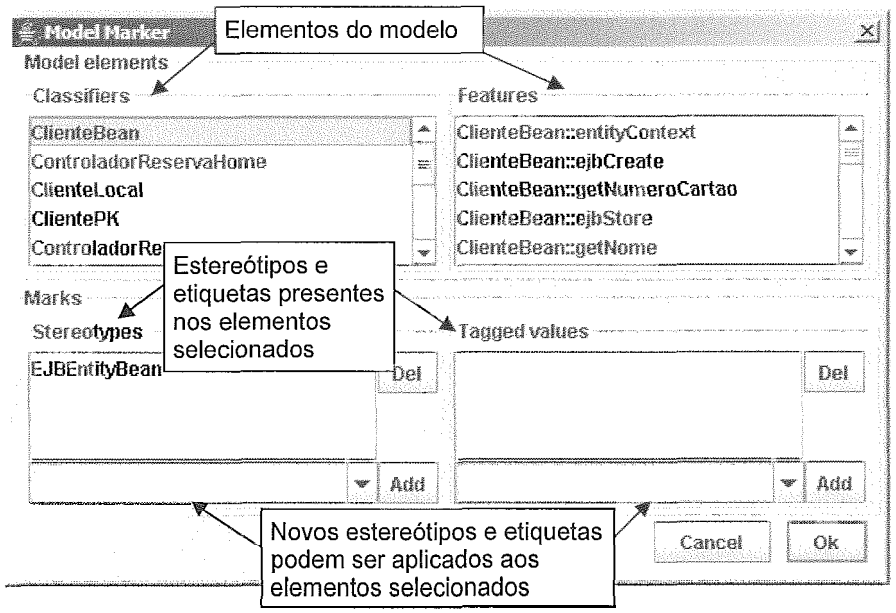


Figura 4.7: *Plug-in* desenvolvido para realização de marcações sobre modelos *UML*.

Através do *plug-in ModelMarker*, o usuário pode selecionar elementos do tipo *Classifier* (e.g.: classes e interfaces) e suas respectivas *Features* (e.g.: atributos e operações), para a aplicação dos estereótipos ou etiquetas esperados pelos *finders*. A vantagem da utilização deste *plug-in* em relação à marcação na ferramenta *CASE* está na facilidade proporcionada ao usuário, que pode selecionar múltiplos elementos para serem marcados de uma só vez, ao contrário de como ocorre na maioria das ferramentas *CASE*.

Após a marcação na ferramenta *CASE*, o modelo deve ser exportado através de *XMI* e importado na ferramenta *Odyssey-MDA*. A importação (Figura 4.8) é iniciada pelo menu *File* ▸ *Import model*, onde um dialogo é aberto e o usuário pode informar o arquivo *XMI* do modelo e, logo depois, um nome para o modelo importado.

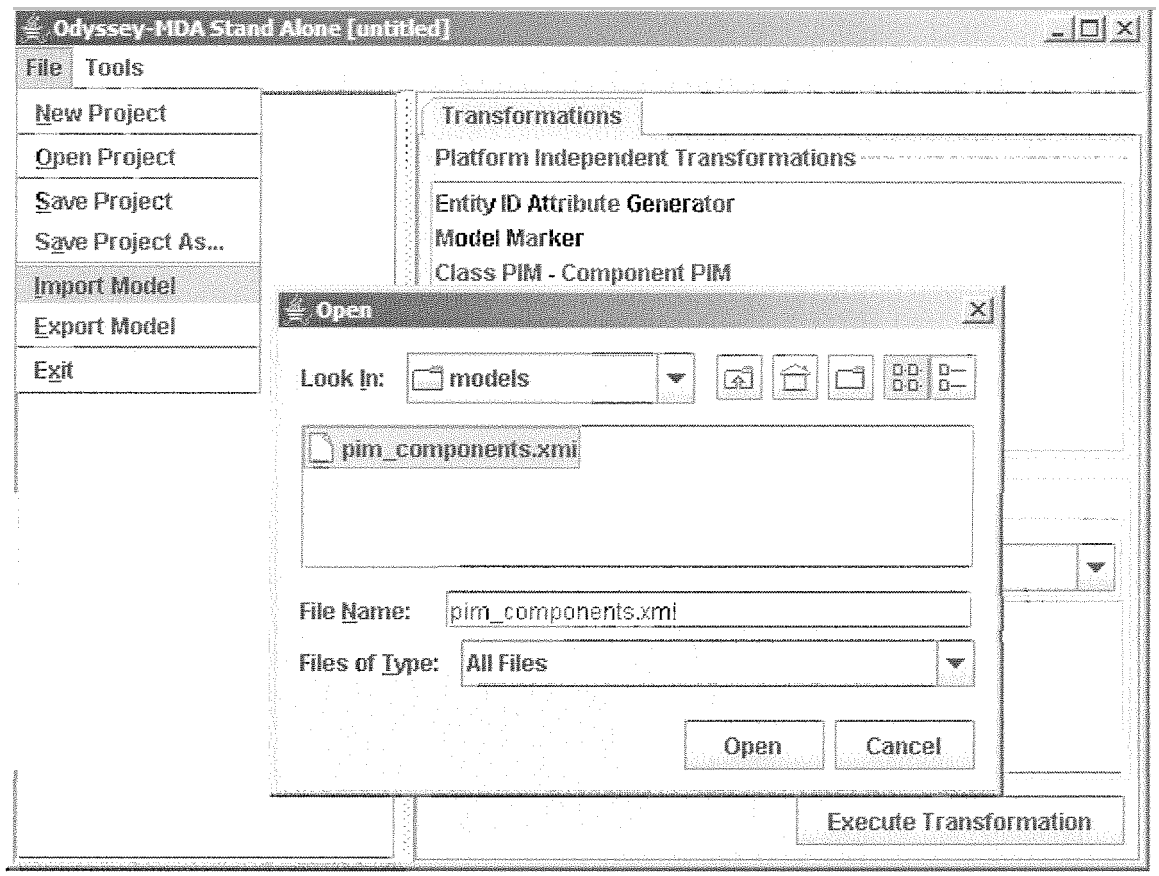


Figura 4.8: Importação de modelo *XMI* na ferramenta *Odyssey-MDA*.

O passo seguinte (Figura 4.9) consiste em selecionar a plataforma (*Enterprise JavaBeans*), a transformação desejada (*EJBComponents*) e pressionar o botão *Execute Transformation* para iniciar a execução da transformação. Com isso, o *wizard* de execução da transformação é aberto (Figura 4.10).

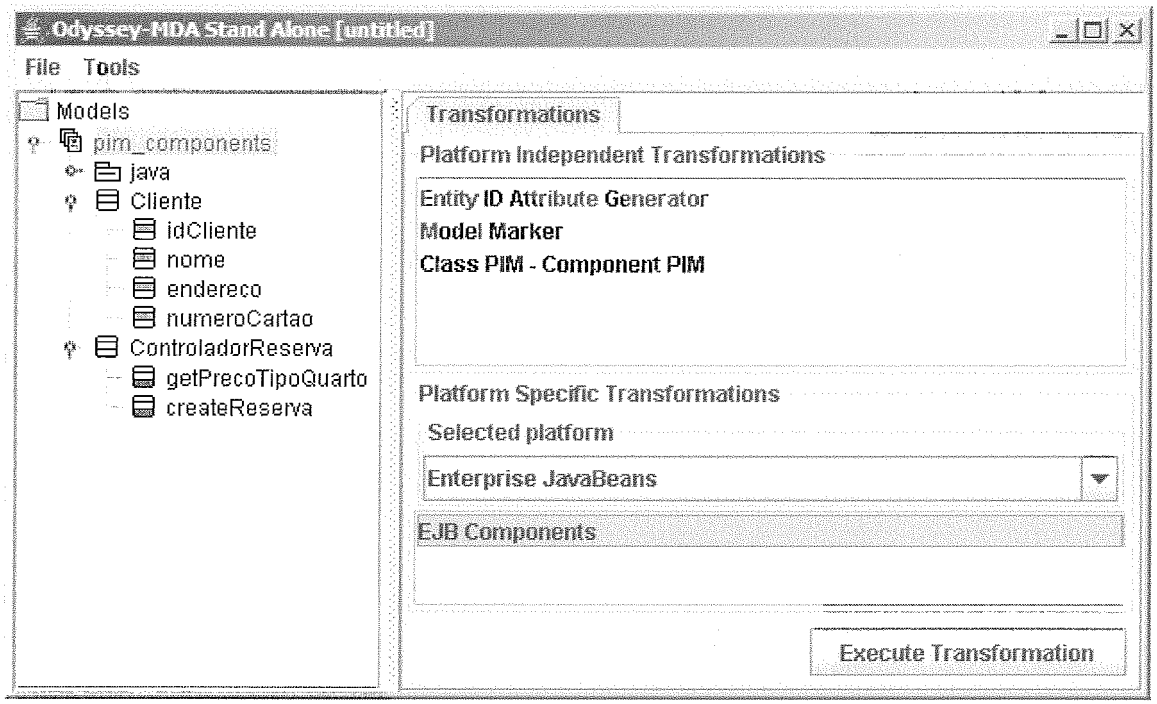


Figura 4.9: Seleção da transformação *EJBComponents*.

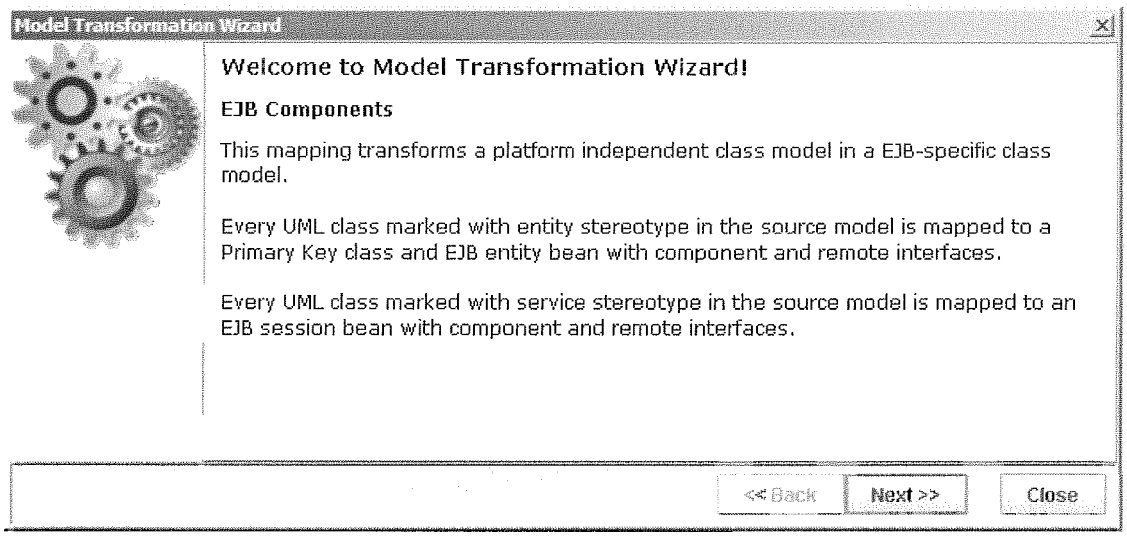


Figura 4.10: *Wizard* de execução da transformação *EJBComponents*.

Antes de executar a transformação, o usuário deve configurar o comportamento da transformação (Figura 4.11). Para isso, ele deve selecionar o modelo de entrada (origem) e o modelo de destino (saída). A ferramenta permite a seleção de um modelo existente como destino, e essa execução preservará os elementos (*Classifiers* e *Features*) presentes no destino, e que não foram alvo de transformação, o que caracteriza uma transformação *não-destrutiva*.

Caso o modelo de destino não exista, o usuário deve informar o nome do modelo e o mesmo será criado no momento da execução da transformação. O usuário, também, pode escolher o mesmo modelo como origem e destino, tendo a possibilidade de

informar o nome de um novo pacote onde os elementos devem ser criados pela transformação.

As outras configurações são a direção da transformação (direta ou reversa) e a possibilidade de pré-visualizar cada mapeamento a ser realizado. Essas configurações são apresentadas na Figura 4.11 abaixo.

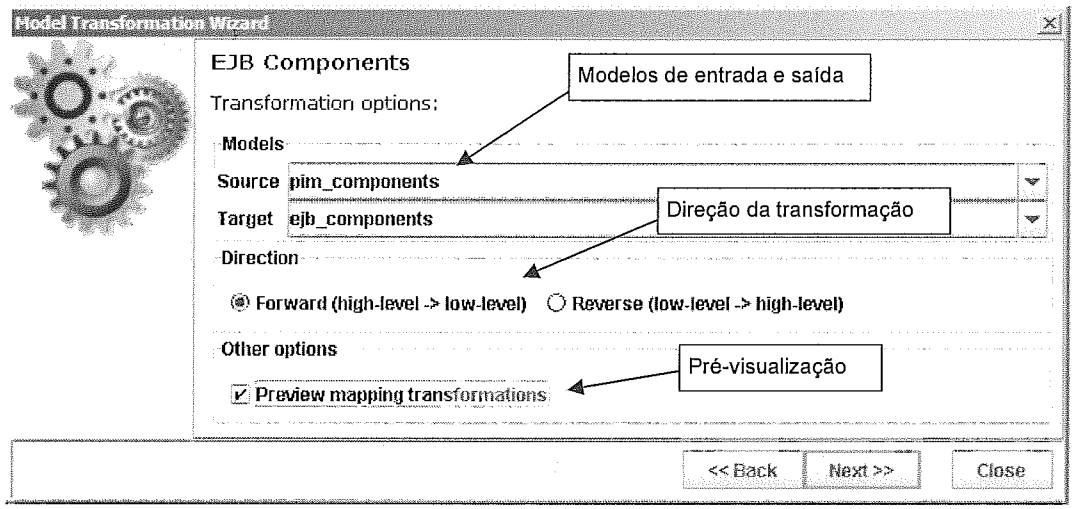


Figura 4.11: Configurações da execução da transformação *EJBComponents*.

Após a configuração, a execução da transformação se inicia e a máquina de transformação instancia cada mapeamento, seus mecanismos e faz a busca por elementos que satisfazem os critérios de busca definidos nos buscadores. O usuário pode pré-visualizar cada elemento do modelo de entrada e os mapeamentos instanciados para cada um desses elementos. A Figura 4.12 mostra a pré-visualização da execução direta da transformação *EJBComponents*. Todos os mapeamentos da classe Cliente podem ser visualizados na árvore de pré-visualização, onde o usuário pode desmarcar algum mapeamento indesejado.

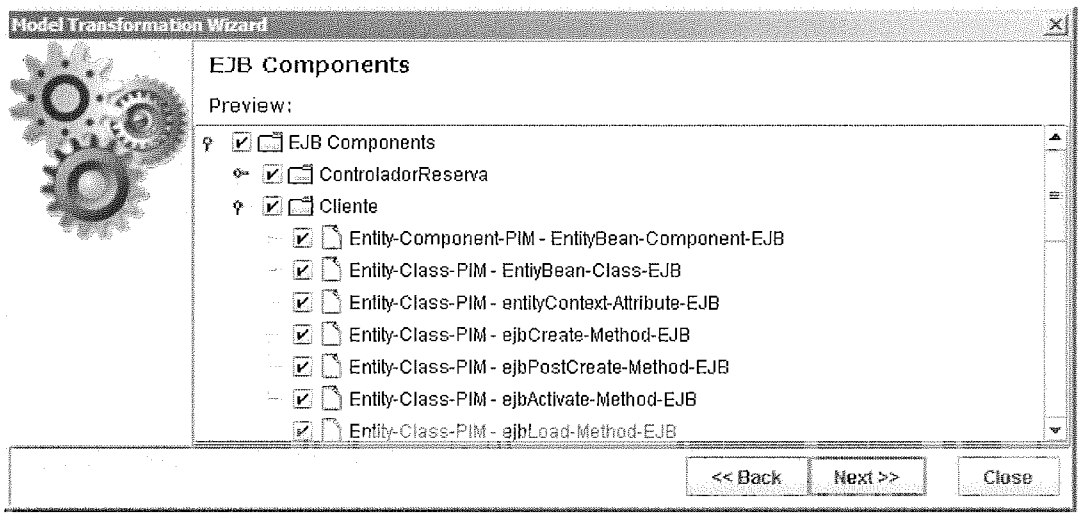


Figura 4.12: Pré-visualização da execução da transformação *EJBComponents*.

Após a pré-visualização, o usuário deve pressionar o botão *Finish* do *wizard* e cada um dos mecanismos associados aos mapeamentos, executam suas transformações, gerando elementos no modelo de saída, que pode ser visto na Figura 4.13.

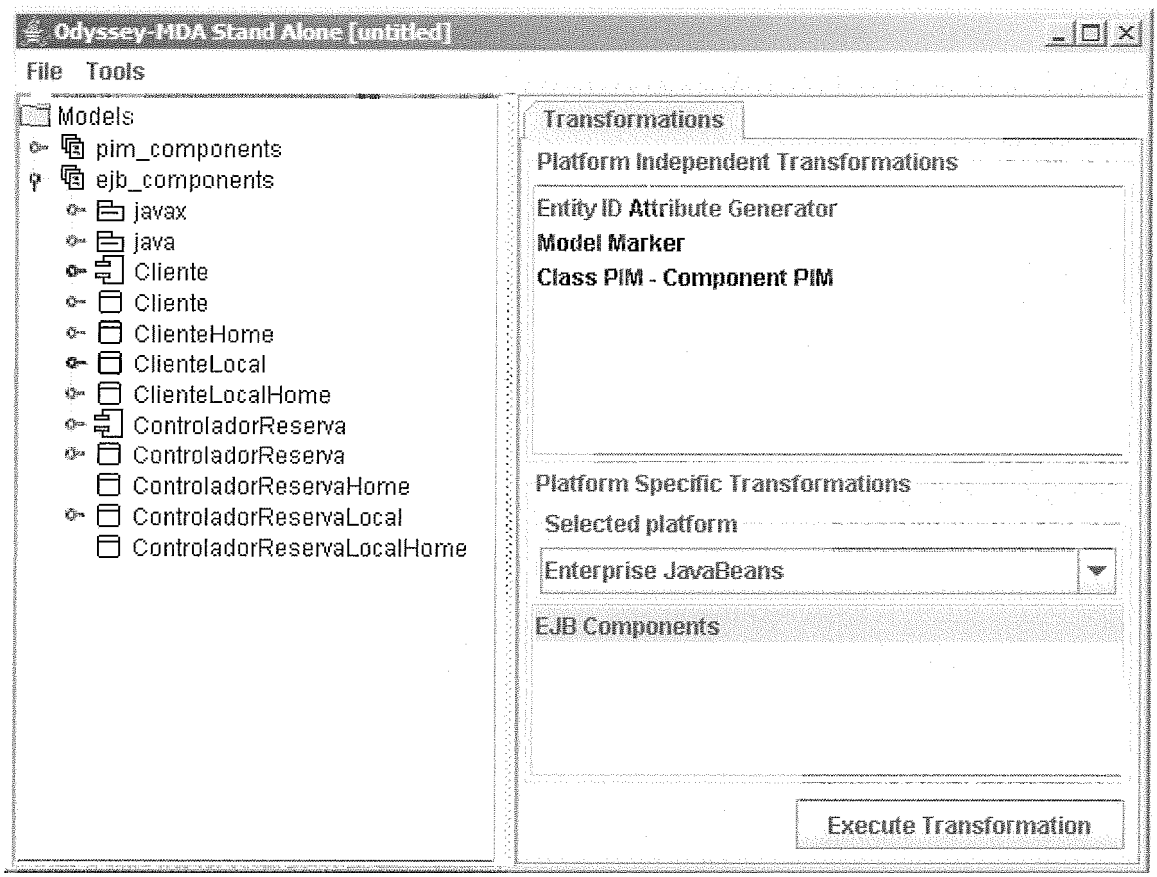


Figura 4.13: Modelo de saída da transformação *EJBComponents*.

Ao término da execução, o modelo de saída pode ser exportado através de *XMI*, no menu *File* ▸ *Export model*. Um dialogo é apresentado ao usuário, onde ele deverá escolher a localização e o nome do arquivo em que o modelo deverá ser salvo. Após a exportação, o modelo pode ser importado na ferramenta *CASE* de origem.

Caso o usuário realize mudanças no modelo *PSM* gerado pela transformação, e deseje atualizar o modelo *PIM* original com essas mudanças, ele pode realizar uma transformação reversa, escolhendo o modelo *PSM* como entrada, o modelo *PIM* original como saída e escolhendo a direção *reverse*. Com isso, caso o atributo *cep* seja adicionado à classe que implementa o *entity bean* Cliente (modelo *PSM*), a execução reversa atualiza a classe Cliente no modelo *PIM*, adicionando esse atributo. Este exemplo de execução reversa permite a manutenção da consistência entre os modelos de diferentes níveis de abstração (*PIM* e *PSM*) do sistema.

4.4.2.2 – *Odyssey*

A principal diferença entre a utilização *stand-alone*, apresentada anteriormente, e a utilização no Ambiente *Odyssey*, está no fato da exportação e importação dos modelos através de *XMI* serem feitos de forma transparente ao usuário. Com isso, o engenheiro pode escolher as plataformas e transformações diretamente no ambiente de modelagem do *Odyssey*, através de sua árvore semântica de elementos, conforme pode ser visto na Figura 4.14.

Após a seleção da transformação desejada, o *wizard* de execução da transformação é aberto e o engenheiro pode realizar a configuração da execução desta transformação, conforme também é ilustrado na Figura 4.14.

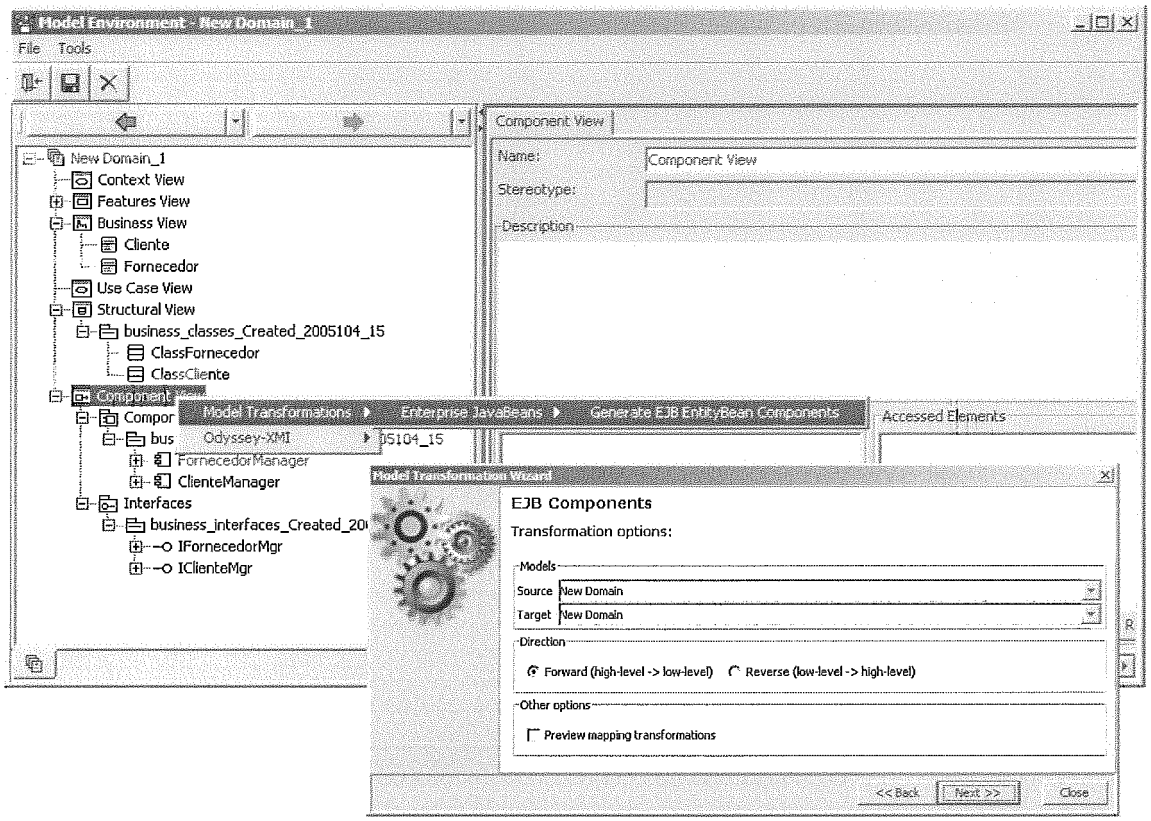


Figura 4.14: Ferramenta *Odyssey-MDA* no ambiente *Odyssey*.

4.4.3 – Geração de Código

Para possibilitar a geração de código, foi desenvolvido um mecanismo de geração para a linguagem *Java* a partir de modelos *UML*, e *templates* escritos em *Velocity Template Language* (*VTL*) (APACHE, 2005) que são utilizados nessa geração. A Figura 4.15 ilustra o *template* definido para gerar código *Java* a partir de uma classe *UML*.

```
/* Generated by java_class_template.vm */
#if($package_name)
package $package_name;
#end
public class $class_name #extends_txt($extends) #implements_txt($realization) {
#if($attributes.size()>0)
    /* Attributes */
#foreach( $attribute in $attributes )
    $!attribute.visibility $!attribute.scope $!attribute.changeability $attribute.type $attribute.name;
#end
#end
#if($associations.size()>0)
    /* Associations */
#foreach( $association in $associations )
    $!association.visibility $!association.scope $!association.changeability $association.type $association.role$!association.initialValue;
#end
#end
#if($constructors.size()>0)
    /* Constructors */
#foreach( $constructor in $constructors )
    $!constructor.visibility ${clazz_name}() {

    }
#end
#end
#if($methods.size()>0)
    /* Methods */
#foreach( $method in $methods )
    $!method.visibility $!method.scope $!method.type $!method.concurrency $method.return_type
${method.name}(#parameter_list($method.parameters)) {
        $!method.intermediate_code
    }
#end
#end
}
```

Figura 4.15: *Template* para geração de código para classes Java.

Assim como ocorre na ferramenta de transformação de modelos, a geração de código pode ser utilizada no Ambiente *Odyssey* de forma transparente. A Figura 4.16 ilustra essa utilização.

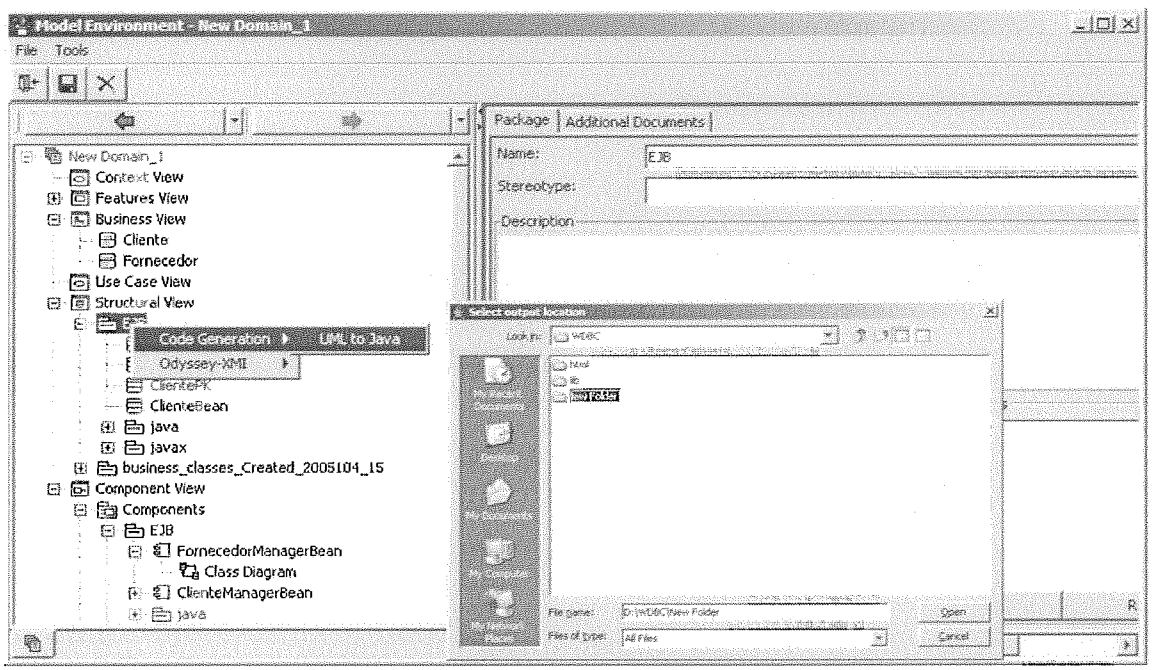


Figura 4.16: Geração de código *UML* para *Java* no Ambiente *Odyssey*.

A figura X ilustra parte do código-fonte gerado para a classe que implementa o *entity bean* Cliente.

```
/* Generated by java_class_template.vm */

package ejb;

public class ClienteBean {

    /* Attributes */

    private void idCliente;

    protected EntityContext entityContext;

    private String endereco;

    private String nome;

    /* Methods */

    public String getIdCliente() {

    }
    public ClientePK ejbCreate() {

    }
    public void ejbPostCreate() {

    }
    public void ejbActivate() {

    }
    public void ejbLoad() {

    }
    public void ejbPassivate() {

    }
    public void ejbRemove() {

    }
    public void ejbStore() {

    }
    public void setEntityContext() {

    }
    public void unsetEntityContext() {

    }
    public String getEndereco() {

    }
    public String getNome() {

    }
    public void setEndereco() {

    }
    public void setNome() {

    }
}
```

Figura 4.17: Código-fonte *Java* gerado para a classe Cliente.

4.5 – Considerações Finais

Apresentamos neste capítulo a implementação da proposta de uma abordagem de transformação de modelos e geração de código, denominada *Odyssey-MDA*, cujo objetivo é auxiliar o Engenheiro de *Software* na definição e execução de transformações sobre modelos. Com a ferramenta implementada é possível transformar modelos independentes de plataforma em modelos específicos para uma plataforma escolhida e a posterior obtenção de parte da implementação do *software* na linguagem de programação utilizada pela plataforma.

A ferramenta implementada permite a sua utilização através do Ambiente *Odyssey*, onde as transformações podem ser escolhidas e executadas diretamente de dentro do Ambiente, ou permite a sua utilização de forma independente de ferramenta *CASE*, pois apresenta uma interface *stand-alone* através da qual os modelos podem ser importados em *XMI*, transformados e exportados de volta para a ferramenta *CASE* de origem. Além disso, as transformações são executadas na ferramenta de forma não-destrutiva, ao preservar informações existentes no modelo de destino.

Para ilustrar a utilização da ferramenta, adotamos o exemplo de transformação *EJBComponents* definido no capitulo anterior, que recebe um modelo de componentes independente de plataforma (*PIM*), e gera um modelo de componentes específico para a plataforma *Enterprise JavaBeans*. Essa transformação foi definida e incorporada à ferramenta *Odyssey-MDA* e executada em ambas as direções (direta e reversa).

Discutimos também a extensão da ferramenta para a definição de novos mecanismos de transformação (os *plug-ins*), e em especial, a implementação de um *plug-in* responsável por apoiar o engenheiro na realização da marcação dos elementos dos modelos, aplicando estereótipos e etiquetas.

Capítulo 5 – Conclusões

5.1 – Contribuições

Conforme discutido neste trabalho, grande parte das abordagens de transformação de modelos trabalha de forma unidirecional, não permitindo a definição de transformações que possam ser executadas em ambas as direções, tanto no sentido direto (*PIM→PSM*) quanto no sentido reverso (*PSM→PIM*). Além disso, as abordagens apresentam certas limitações que motivaram a proposta deste trabalho, que consiste em uma abordagem para a transformação de modelos. Esta abordagem apresenta as seguintes características:

- Independência de ferramenta CASE ou ambiente de desenvolvimento, o que permite a sua utilização em conjunto com as ferramentas disponíveis no mercado, desde que tais ferramentas sejam capazes de exportar e importar seus modelos através de *XMI*. Esta característica impede que a abordagem não possa ser utilizada no caso de alguma ferramenta em particular ser descontinuada.
- A linguagem para a definição das transformações utiliza conceitos baseados no meta-modelo de transformações do *CWM* e, além disso, é utilizada uma linguagem *XML* de fácil utilização, conforme apresentado nos capítulos 3 e 4.
- As transformações do tipo modelo-modelo são especificadas através da definição de mapeamentos declarativos entre os elementos dos modelos e, a associação de mecanismos que implementam as transformações sobre tais elementos.
- As transformações modelo-modelo podem ser definidas de forma bidirecional a partir de uma única especificação. Com isso, as transformações podem ser reversíveis, possibilitando a sua execução tanto para a recuperação de informações independentes de plataforma, quanto para a manutenção do sincronismo entre modelos de diferentes níveis de abstração.

- A possibilidade de definição de transformações modelo-texto, com o objetivo de geração de arquivos texto, ou o código-fonte na linguagem de programação desejada, a partir dos modelos desenvolvidos.
- A abordagem permite sua extensão, visando a definição de novas transformações, mecanismos e geradores de texto, conforme apresentado na Seção 3.6.

Estas características permitem que a abordagem atenda praticamente todos os critérios definidos na Seção 2.3.1, como exibido na Tabela 5.1.

Tabela 5.1: Quadro comparativo das abordagens.

Critérios	Abordagens						
	ATL	UMT	UMLX	AndroMDA	xUML	OptimalJ	Odyssey-MDA
Independência de ferramenta *CASE*	Sim	Sim	Sim	Sim	Não	Não	**Sim**
Formatos e linguagens padronizadas	Não	Não	Não	Não	Não	Não	**Não**
Transformações modelo-modelo	Sim	Sim	Sim	Não	Sim	Sim	**Sim**
Transformações modelo-texto	Não	Sim	Não	Sim	Sim	Sim	**Sim**
Bidirecionalidade	Não	Não	Não	Não	Não	Não	**Sim**
Possibilidade de extensão	?	Sim	Não	Sim	?	Sim	**Sim**

O critério de utilização de linguagens ou formatos padronizados para a definição de transformações não foi atendido por esta abordagem, pois, ao longo da realização deste trabalho, a *OMG* ainda não havia finalizado o seu processo de elaboração da especificação *QVT*. No futuro, espera-se alinhar a abordagem *Odyssey-MDA* com a especificação *QVT*.

Para apoiar a utilização da abordagem, foi implementado um protótipo da máquina de transformações, denominada *Odyssey-MDA*, que permite:

- A utilização de forma desacoplada de qualquer ferramenta *CASE* ou ambiente de desenvolvimento, possuindo uma interface gráfica própria, com funcionalidades de importação e exportação de modelos através de *XMI*, além da execução das transformações sobre os modelos importados.

- A integração transparente com Ambiente *Odyssey*, onde as transformações (modelo-modelo e geração de código) podem ser escolhidas e executadas diretamente sobre o modelo em desenvolvimento dentro o ambiente.
- A utilização de um *plug-in* especialmente desenvolvido para apoiar o engenheiro na tarefa de marcação dos modelos que serão transformados.
- A ferramenta permite que o engenheiro possa visualizar previamente os mecanismos que serão executados pela maquina de transformações, permitindo que ele possa desabilitar a execução de algum mapeamento indesejado. Isso possibilita uma maior interatividade do engenheiro no processo de execução da transformação, de acordo com seus objetivos.
- A execução das transformações sobre modelos pode ser feita de maneira não-destrutiva, isto é, caso um modelo existente seja escolhido como saída para uma transformação, informações presentes no mesmo são atualizadas de maneira incremental. Mesmo que os elementos mapeados já possuam informações não presentes no modelo de entrada, no caso destas informações não serem manipuladas pela transformação, as mesmas são preservadas.

5.2 – Limitações e Trabalhos Futuros

Algumas limitações e trabalhos futuros puderam ser detectados durante esse trabalho. Essas limitações e possíveis trabalhos futuros são apresentados a seguir.

5.2.1 – Linguagem Não Padronizada

Com relação ao critério de utilização de linguagens e formatos padronizados, esta abordagem utiliza uma sintaxe *XML* que apesar de simples, foi proposta neste trabalho e não é padronizada. Ao longo da realização deste trabalho, a *OMG* vem trabalhando na especificação para um mecanismo de consultas (*queries*), visões sobre modelos (*views*) e transformações de modelos (*transformations*), conhecida como *QVT*. Esta especificação, apesar de estar em fase final, ainda depende do lançamento da nova versão do *MOF* (versão 2.0) para ser totalmente liberada. No futuro, espera-se pesquisar uma forma de adequar a proposta deste trabalho à especificação do *QVT*.

5.2.2 – Notação Gráfica

As definições das transformações são especificadas diretamente em arquivos XML. A utilização de uma notação gráfica exigiria a implementação de um ambiente de modelagem para a definição das transformações de forma visual, o que não é objetivo deste trabalho. No entanto, a especificação QVT irá fornecer uma linguagem gráfica concreta para as transformações, e um esforço para a utilização dessa notação poderá ser realizado no futuro.

5.2.3 – Restrição à Modelos *UML*

A ferramenta implementada é capaz apenas de manipular modelos *UML*. Apesar da abordagem ser perfeitamente aplicável a outros modelos baseados em meta-modelos descritos em *MOF*, por questões práticas, a implementação ficou restrita ao meta-modelo *UML*. Com o recente lançamento da versão 2.0 da *UML*, e com a futura versão 2.0 do *MOF*, a ferramenta poderá ser atualizada para suportar a execução de transformações definidas sobre quaisquer meta-modelos *MOF*.

5.2.4 – Resolução de Conflitos

Ao trabalhar com modelos em diferentes níveis de abstração para o mesmo sistema, a atualização destes modelos para remoção de elementos ou alteração de nomes podem causar conflitos que não são resolvidos pela ferramenta. Contudo, a ferramenta de transformação de modelos poderia incorporar novos mecanismos que sejam capazes de detectar os diferentes tipos de conflitos, resolver aqueles que podem ser tratados de forma automática, ou interagir com o engenheiro para que o mesmo solucione os casos mais complexos.

5.2.5 – Ferramenta de Transformação modelo-texto

O protótipo da máquina de geração de texto implementada não apóia a geração não-destrutiva de código-fonte. Alterações realizadas sobre o código-fonte gerado não são preservadas após uma nova geração de código. Melhorias poderão ser realizadas para que seja possível identificar os trechos de código que foram inseridos pelo usuário, preservando-os numa geração posterior.

Além disso, não é objetivo do protótipo a realização de engenharia reversa sobre código-fonte. Outra ferramenta já existente no Ambiente Odyssey, denominada *ARES* (VERONESE & NETTO, 2001), é capaz de realizar estas transformações, recuperando

modelos *UML* a partir de código-fonte *Java*. Um trabalho futuro poderá estudar uma forma de integrar a ferramenta *ARES* e a máquina de geração de texto da *Odyssey-MDA* para possibilitar a execução de geração de código *round-trip*, onde o sincronismo entre os modelos e o código-fonte correspondente é sempre mantido.

Existe ainda a possibilidade de utilizar outras ferramentas para a geração de código, como o *AndroMDA* (KOZIKOWSKI, 2005). Neste caso, a máquina de execução de transformações modelo-modelo seria utilizada para gerar modelos *PSM* no formato esperado pelo *AndroMDA*, onde o código-fonte seria gerado.

5.2.6 – Modelagem Comportamental

O foco maior nas transformações apoiadas pela Odyssey-MDA está na geração da estrutura estática dos modelos e do código-fonte. Apesar de ser possível inferir e gerar uma parte do comportamento dinâmico de certas operações, esta geração não foi contemplada neste trabalho. Futuramente, a especificação do comportamento dinâmico das operações contidas nos modelos poderá ser especificada de forma independente de plataforma, através de alguma linguagem de ações semânticas, e a abordagem poderá utilizar tal especificação para gerar uma maior porcentagem de código-fonte.

5.2.7 – Rastreabilidade Entre os Modelos

A abordagem não prevê o armazenamento ou a apresentação de um registro dos rastros entre os elementos que serviram como entrada para uma transformação e os elementos que foram gerados pela execução. Tal mecanismo poderá ser implementado no futuro e a abordagem poderá apresentar tal melhoria.

5.2.8 – Avaliação em Projetos Reais

A abordagem proposta nesse trabalho deve ser avaliada e verificada através da realização de estudos de caso e da sua utilização em projetos reais de desenvolvimento de *software*. Essa utilização possivelmente implicaria no surgimento de novos requisitos não previstos inicialmente, e que poderiam ser implementados através da extensão ou adaptação desta abordagem.

Referências Bibliográficas

ACCELERATED-TECHNOLOGY, 2006, "Nucleus BridgePoint". In: http://www.acceleratedtechnology.com/embedded/nuc_modeling.html, accessed in 20/01/2006.

APACHE, 2005, "Velocity Java-based Template Engine". In: http://jakarta.apache.org/velocity, accessed in 29/11/2005.

BLOIS, A.P., BECKER, K., WERNER, C.M.L., 2004, "Um Processo de Engenharia de Domínio com foco no Projeto Arquitetural Baseado em Componentes". In: *IV Workshop de Desenvolvimento Baseado em Componentes*, pp. 15-20, João Pessoa, Paraíba, Brasil, Setembro.

BORLAND, 2006, "Borland Together Tecnologies". In: http://www.borland.com/us/products/together/, accessed in 02/01/2006.

BRAGA, R.M.M., 2000, *Busca e Recuperação de Componentes em Ambientes de Reutilização de Software*, Tese de D.Sc., COPPE/UFRJ, Rio de Janeiro.

COMPUWARE, 2006, "OptimalJ - Model-driven Java development tool". In: http://www.compuware.com/products/optimalj/, accessed in 20/01/2006.

CZARNECKI, K., HELSEN, S., 2003, "Classification of Model Transformation Approaches". In: *Online Proceedings of the 2nd OOPSLA'03 Workshop on Generative Techniques in the Context of MDA*, Anaheim, California, October 2003.

DANTAS, A.R., 2001, *Oráculo: Um Sistema de Críticas para a UML*, Projeto Final de Curso, DCC/IM/UFRJ, Rio de Janeiro.

DANTAS, A.R., VERONESE, G., CORREA, A., *et al.*, 2002, "Suporte a Padrões no Projeto de Software". In: *Caderno de Ferramentas do XVI Simpósio Brasileiro de Engenharia de Software (SBES'2002)*, pp. 450-455, Gramado, Rio Grande do Sul, Outubro.

DIRCZE, R., 2002, "JSR 40: Java Metadata Interface (JMI) Specification - version 1.0", *Unisys Corporation and Sun Microsystems*, Java Community Process.

ECLIPSE, 2006, "UMLX: A graphical transformation language for MDA". In: http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/gmt-home/subprojects/UMLX/index.html, accessed in 10/02/2006.

FOLDOC, 2006, "Free Online Dictionary of Computing". In: http://foldoc.org/, accessed in 10/12/2005.

FRANKEL, D.S., 2003, *Model Driven Architecture: Applying MDA to Enterprise Computing*, Wiley Publishing, Inc.

FRIEDL, J., 2002, *Mastering Regular Expressions*, 2nd ed., O'Reilly & Associates, Inc.

GAMMA, E., HELM, R., JOHNSON, R., *et al.*, 1995, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed., Addison-Wesley.

GARDNER, T., GRIFFIN, C., KOEHLER, J., *et al.*, 2002, "A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard", *OMG Document: ad/03-08-02*.

GENTLEWARE, 2005, "Poseidon For UML". In: http://gentleware.com, accessed in 15/11/2005.

HAYWOOD, D., 2004, "MDA: Nice Idea, Shame About the..." In: http://www.theserverside.com/articles/article.tss?l=MDA_Haywood, accessed in 15/01/2006.

HEYSE, W., JONCKERS, V., WAGELAAR, D., 2005, "Apprenticeship Report - Generic Code Generation Approaches". In: http://wilma.vub.ac.be/~wheyse/stage/workdocument.pdf, accessed in 20/12/2005.

INTERACTIVE-OBJECTS, 2006, "ArcStyler Overview". In: http://www.interactive-objects.com/products/arcstyler-overview, accessed in 20/01/2006.

JOUAULT, F., KURTEV, I., 2005, "Transforming Models with ATL". In: *Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005*, Montego Bay, Jamaica, October, 2005.

KENNEDY-CARTER, 2006, "xUML - Executable UML". In: http://www.kc.com/xuml.php, accessed in 20/01/2006.

KOZIKOWSKI, J., 2005, "A Bird's Eye view of AndroMDA". In: http://www.andromda.org/contrib/birds-eye-view.html, accessed in 20/01/2006.

LOPES, M.A.M., MANGAN, M.A.S., WERNER, C.M.L., 2004, "MAIS: Uma Ferramenta de Percepção para apoiar a Edição Concorrente de Modelos de

Análise e Projeto". In: *XVIII Simpósio Brasileiro de Engenharia de Software, Sessão de Ferramentas*, pp. 61-66, Brasília, DF, Brasil, Outubro.

MAIA, N.E.N., BLOIS, A.P.B., WERNER, C.M., 2005, "Odyssey-MDA: Uma Ferramenta para Transformações de Modelos UML". In: *XIX Simpósio Brasileiro de Engenharia de Software, Seção de Ferramentas*, Uberlândia, MG, Brasil, Outubro 2005.

MATULA, M., 2003, "NetBeans Metadata Repository", *NetBeans Community*.

MELLOR, S.J., BALCER, M.J., 2002, *Executable Uml: A Foundation for Model-Driven Architecture*, Addison-Wesley Professional.

MELLOR, S.J., SCOTT, K., UHL, A., *et al.*, 2004, *MDA Distilled Principles of Model-Driven Architecture*, Addison-Wesley.

MICROSOFT, 2006, "Microsoft .NET Homepage". In: http://www.microsoft.com/net/default.mspx, accessed in 10/02/2006.

MILLER, N., 2000, *A Engenharia de Aplicações no Contexto da Reutilização Baseada em Modelos de Domínio*, Tese de M.Sc., COPPE/UFRJ, Rio de Janeiro.

MODELBASED.NET, 2006, "MDA Tools". In: http://www.modelbased.net/mda_tools.html, accessed in 20/01/2006.

MURTA, L.G.P., 1999, *FRAMEDOC: Um Framework para a Documentação de Componentes Reutilizáveis*, Projeto Final de Curso, DCC/IM/UFRJ.

MURTA, L.G.P., BARROS, M.O., WERNER, C.M.L., 2002, "Charon: Uma Ferramenta para a Modelagem, Simulação, Execução e Acompanhamento de Processos de Software". In: *XVI Simpósio Brasileiro de Engenharia de Software, Caderno de Ferramentas*, pp. 366-371, Gramado, RS, Outubro.

MURTA, L.G.P., VASCONCELOS, A.P.V., BLOIS, A.P.T.B., *et al.*, 2004, "Runtime Variability through Component Dynamic Loading". In: *XVIII Simpósio Brasileiro de Engenharia de Software, Seção de Ferramentas*, pp. 67-72, Brasília, DF, Brasil, Outubro.

NETBEANS, 2005, "Welcome to NetBeans". In: http://www.netbeans.org/, accessed in 15/11/2005.

ODYSSEY, 2006, "Odyssey SDE". In: http://reuse.cos.ufrj.br/odyssey/, accessed in 20/02/2006.

OLDEVIK, J., 2004, "UML Model Transformation Tool - Overview and user guide documentation". In: http://umt-qvt.sourceforge.net/docs/UMT_documentation_v08.pdf, accessed in 10/02/2006.

OLIVEIRA, H., MURTA, L.G.P., WERNER, C.M.L., 2004, "Odyssey-VCS: Um Sistema de Controle de Versões Para Modelos Baseados no MOF". In: *XVIII Simpósio Brasileiro de Engenharia de Software, Sessão de Ferramentas*, pp. 85-90, Brasília, DF, Brasil, Outubro.

OMG, 2002a, "CORBA Component Model, Version 3.0", *Object Management Group*, OMG document formal/2002-06-65.

OMG, 2002b, "Meta-Object Facility (MOF), Version 1.4", *Object Management Group*, OMG document formal/2002-04-03.

OMG, 2002c, "XML Metadata Interchange (XMI) Specification, Version 1.2", *Object Management Group*, OMG document formal/02-01-01.

OMG, 2003a, "Common Warehouse Metamodel (CWM) Specification, Version 1.1", *Object Management Group*, OMG document formal/2003-03-02.

OMG, 2003b, "MDA Guide Version 1.0.1", *Object Management Group*, OMG document omg/2003-06-01.

OMG, 2003c, "XML Metadata Interchange (XMI) Specification, version 1.1", *Object Management Group*.

OMG, 2004a, "Revised submission for MOF 2.0 Query/Views/Transformations RFP", *QVT-Merge Group*, OMG document ad/2004-10-04.

OMG, 2004b, "UML Profile for Enterprise Distributed Object Computing ". In: http://www.omg.org/technology/documents/formal/edoc.htm, accessed in 05/02/2006.

OMG, 2004c, "Unified Modeling Language (UML), Version 1.4.2." *Object Management Group*, OMG formal/04-07-02.

OMG, 2005a, "MOF Query / Views / Transformations - final adopted specification", *Object Management Group*, OMG document ptc/05-11-01.

OMG, 2005b, "Unified Modeling Language (UML), Version 2.0 Superstructure Specification." *Object Management Group*, OMG document formal/05-07-04.

PRIETO-DIAZ, R., ARANGO, G., 1991, "Domain Analysis Concepts and Research Directions". In: PRIETO-DIAZ, R., ARANGO, G. (eds), *Domain Analysis and Software Systems Modeling*, IEEE Computer Society Press.

SIEGEL, J., 2001, *Developing in OMG's Model-Driven Architecture*, Object Management Group, OMG document omg/01-12-01.

SOLEY, R., 2000, *Model Driven Architecture*, Object Management Group, OMG document omg/00-11-05.

SUN, 2003, "Enterprise JavaBeans Specification, Version 2.1", *Sun Microsystems*.

SUN, 2005, "Java Technology". In: http://java.sun.com, accessed in 05/10/2005.

SUN, 2006a, "Core J2EE Patterns: Patterns index page". In: http://java.sun.com/blueprints/corej2eepatterns/Patterns/index.html, accessed in 10/02/2006.

SUN, 2006b, "The J2EE 1.4 Tutorial". In: http://java.sun.com/j2ee/1.4/docs/tutorial/doc/index.html, accessed in 02/01/2006.

SUN, 2006c, "Java Technology". In: http://java.sun.com, accessed in 05/10/2005.

VERONESE, G.O., NETTO, F.J., 2001, *Uma Ferramenta de Apoio a Recuperação de Projetos no Ambiente Odyssey*, Projeto Final de Curso, DCC/IM/UFRJ.

W3C, 1999, "XSL Transformations (XSLT), Version 1.0". In: http://www.w3.org/TR/xslt, accessed in 10/02/2006.

W3C, 2004, *Extensible Markup Language (XML) 1.0 (Third Edition)*, World Wide Web Consortium.

W3C, 2005, "HTTP - Hypertext Transfer Protocol". In: http://www.w3.org/Protocols, accessed in 20/10/2005.

WALL, L., SCHWARTZ, R.L., CHRISTIANSEN, T., 1996, *Programming Perl*, Sebastopol, California, O'Reilly & Associates.

WERNER, C.M.L., BRAGA, R.M.M., 2005, "A Engenharia de Domínio e o Desenvolvimento

Baseado em Componentes". In: GIMENES, I.M.S., HUZITA, E.H.M. (eds), *Desenvolvimento Baseado em Componentes: Conceitos e Técnicas*, Rio de Janeiro, Ciência Moderna.

XAVIER, J.R., 2001, *Criação e Instanciação de Arquiteturas de Software Específicas de Domínio no Contexto de uma Infra-Estrutura de Reutilização*, Tese de M.Sc., COPPE/UFRJ, Rio de Janeiro.

Apêndice A – Esquema *XML* das Transformações

Neste Apêndice é exibido o esquema *XML* utilizado pelas especificações das transformações.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<xsd:schema
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://reuse.cos.ufrj.br/odyssey/tools/mda/mdaxmlmapping"
        xmlns="http://reuse.cos.ufrj.br/odyssey/tools/mda/mdaxmlmapping">

<xsd:element name="transformation-mapping">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="classifier-map" minOccurs="1" maxOccurs="unbounded"/>
            <xsd:element ref="relationship" minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd:string" use="required"/>
        <xsd:attribute name="description" type="xsd:string"/>
        <xsd:attribute name="implementation-package" type="xsd:string"/>
        <xsd:attribute name="pre-execution-code" type="xsd:string"/>
        <xsd:attribute name="post-execution-code" type="xsd:string"/>
    </xsd:complexType>
</xsd:element>

<xsd:element name="classifier-map">
  <xsd:complexType>
    <xsd:sequence>
        <xsd:element ref="finder" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element ref="property" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element ref="classifier-map" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element ref="feature-map" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element ref="classifier-feature-map" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="description" type="xsd:string"/>
    <xsd:attribute name="implementation" type="xsd:string"/>
    <xsd:attribute name="preserve-relationships" type="xsd:string" default="no"/>
    <xsd:attribute name="id" type="xsd:string"/>
  </xsd:complexType>
</xsd:element>

<xsd:element name="feature-map">
  <xsd:complexType>
    <xsd:sequence>
        <xsd:element ref="finder" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element ref="property" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="description" type="xsd:string"/>
    <xsd:attribute name="implementation" type="xsd:string" use="required"/>
    <xsd:attribute name="id" type="xsd:string"/>
  </xsd:complexType>
</xsd:element>

<xsd:element name="classifier-feature-map">
  <xsd:complexType>
    <xsd:sequence>
        <xsd:element ref="finder" minOccurs="0" maxOccurs="unbounded"/>
        <xsd:element ref="property" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="description" type="xsd:string"/>
    <xsd:attribute name="implementation" type="xsd:string" use="required"/>
    <xsd:attribute name="id" type="xsd:string"/>
  </xsd:complexType>
</xsd:element>
```

```
<xsd:element name="finder">
  <xsd:complexType>
    <xsd:attribute name="side" type="xsd:string" use="required"/>
    <xsd:attribute name="criteria" type="xsd:string" use="required"/>
    <xsd:attribute name="value" type="xsd:string"/>
  </xsd:complexType>
</xsd:element>

<xsd:element name="property">
  <xsd:complexType>
    <xsd:sequence>
        <xsd:element ref="property" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="value" type="xsd:string"/>
    <xsd:attribute name="direction" type="xsd:string"/>
  </xsd:complexType>
</xsd:element>

<xsd:element name="relationship">
  <xsd:complexType>
    <xsd:sequence>
        <xsd:element ref="element" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="type" type="xsd:string" use="required"/>
    <xsd:attribute name="direction" type="xsd:string"/>
    <xsd:attribute name="implementation" type="xsd:string"/>
  </xsd:complexType>
</xsd:element>

<xsd:element name="element">
  <xsd:complexType>
    <xsd:attribute name="id" type="xsd:string" use="required"/>
  </xsd:complexType>
</xsd:element>

</xsd:schema>
```

Apêndice B – Especificação *EJBComponents*

Neste Apêndice é exibida a especificação *XML* da transformação *EJB Components*, utilizada como exemplo neste trabalho.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<transformation-mapping
  xmlns="http://reuse.cos.ufrj.br/odyssey/tools/mda/mdaxmlmapping"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://reuse.cos.ufrj.br/odyssey/tools/mda/mdaxmlmapping/
mapping.xsd"
  name="EJBComponents"
  description="This mapping transforms a platform independent class model in a EJB-
specific component model. \n\nEvery UML class marked with entity stereotype in the
source model is mapped to an EJB Component, EJB Implementation, Primary Key class and
EJB entity bean with component and remote interfaces. \n\nEvery UML class marked with
service stereotype in the source model is mapped to an EJB Component, EJB
Implementation, component and remote interfaces."
  implementation-package="br.ufrj.cos.lens.odyssey.tools.mda.transformations.builtin"
>

<!--############################## ENTITY ##############################-->

<!-- ENTITY COMPONENT - ENTITY BEAN COMPONENT-->

  <classifier-map name="Entity-Component-PIM - EntityBean-Component-EJB"
description="...">
      <!-- FINDERS -->
      <finder side="left" criteria="stereotype" value="entity" />
      <finder side="right" criteria="stereotype" value="EJBEntityBean" />
      <!-- PROPERTIES -->
      <property name="left_type" value="Class" />
      <property name="right_type" value="Component" />
      <!-- FORWARD PROPERTIES -->
      <property name="stereotype" value="EntityBeanComponent" direction="forward" />

  <!-- ENTITY CLASS - ENTITY BEAN IMPLEMENTATION CLASS-->
      <classifier-map name="Entity-Class-PIM - EntiyBean-Class-EJB" description="..."
id="entityBean" preserve-relationships="yes">
          <!-- FINDERS -->
          <finder side="left" criteria="stereotype" value="entity" />
          <finder side="right" criteria="stereotype" value="EJBEntityBean" />
          <!-- PROPERTIES -->
          <property name="left_type" value="Class" />
          <property name="right_type" value="Class" />
          <!-- FORWARD PROPERTIES -->
          <property name="name" value="#SOURCE_NAME#Bean" direction="forward" />
          <property name="stereotype" value="EJBEntityBean" direction="forward" />
          <!-- REVERSE PROPERTIES -->
          <property name="name_transformation" direction="reverse">
                  <property name="input" value="#SOURCE_NAME#" />
                  <property name="regexp" value="(.*)Bean$" />
                  <property name="subst" value="$1" />
          </property>
          <property name="stereotype" value="entity" direction="reverse" />

  <!-- FEATURE MAPPING: copy business attributes -->
          <feature-map name="Attribute-PIM - Attribute-EJB" description="...">
              <!-- FINDERS -->
              <finder side="left" criteria="type" value="Attribute" />
              <finder side="right" criteria="stereotype" value="EJBBusinessAttribute"/>
              <!-- PROPERTIES -->
              <property name="left_type" value="Attribute" />
              <property name="right_type" value="Attribute" />
              <!-- FORWARD PROPERTIES -->
              <property name="stereotype" value="EJBBusinessAttribute"
direction="forward" />
```

```
        </feature-map>

  <!-- FEATURE MAPPING: getters for business atributes -->
        <feature-map name="Attribute-PIM - Getter-Method-EJB" description="...">
            <!-- FINDERS -->
            <finder side="left" criteria="type" value="Attribute" />
            <!-- PROPERTIES -->
            <property name="left_type" value="Attribute" />
            <property name="right_type" value="Operation" />
            <!-- FORWARD PROPERTIES -->
            <property name="name_transformation" direction="forward">
                <property name="input" value="#SOURCE_NAME#" />
                <property name="regexp" value="(.*)" />
                <property name="subst" value="get\u$1" />
            </property>
            <property name="visibility" value="public" direction="forward" />
            <property name="parameters" direction="forward">
                <property name="return">
                    <property name="direction_kind" value="return" />
                    <property name="classifier_type" value="#SOURCE_TYPE#" />
                </property>
            </property>
        </feature-map>

  <!-- FEATURE MAPPING: setters for business atributes  -->
        <feature-map name="Attribute-PIM - Setter-Method-EJB" description="...">
            <!-- FINDERS -->
            <finder side="left" criteria="type" value="Attribute" />
            <!-- PROPERTIES -->
            <property name="left_type" value="Attribute" />
            <property name="right_type" value="Operation" />
            <!-- FORWARD PROPERTIES -->
            <property name="name_transformation" direction="forward" >
                <property name="input" value="#SOURCE_NAME#" />
                <property name="regexp" value="(.*)" />
                <property name="subst" value="set\u$1" />
            </property>
            <property name="visibility" value="public" direction="forward" />

            <property name="parameters" direction="forward">
                <property name="#SOURCE_NAME#">
                    <property name="direction_kind" value="in" />
                    <property name="classifier_type" value="#SOURCE_TYPE#" />
                </property>
            </property>
        </feature-map>

  <!-- CLASSIFIER-FEATURE MAPPING: entityContext -->
        <classifier-feature-map name="Entity-Class-PIM - entityContext-Attribute-EJB"
description="...">

            <!-- FINDERS -->
            <finder side="left" criteria="stereotype" value="entity" />
            <!-- PROPERTIES -->
            <property name="left_type" value="Class" />
            <property name="right_type" value="Attribute" />
            <!-- FORWARD PROPERTIES -->
            <property name="name" value="entityContext" direction="forward" />
            <property name="classifier_type" value="javax.ejb.EntityContext"
direction="forward" />
            <property name="visibility" value="protected" direction="forward" />
            <property name="tagged_values" direction="forward">
                <property name="isTransformableToLeft" value="false" />
            </property>
        </classifier-feature-map>

  <!-- CLASSIFIER-FEATURE MAPPING: ejbCreate method -->
        <classifier-feature-map name="Entity-Class-PIM - ejbCreate-Method-EJB"
description="...">
            <!-- FINDERS -->
            <finder side="left" criteria="stereotype" value="entity" />
            <!-- PROPERTIES -->
            <property name="left_type" value="Class" />
            <property name="right_type" value="Operation" />
            <!-- FORWARD PROPERTIES -->
            <property name="name" value="ejbCreate" direction="forward" />
            <property name="visibility" value="public" direction="forward" />
```

```
            <property name="parameters" direction="forward">
                <property name="return">
                    <property name="direction_kind" value="return" />
                    <property name="uml_class_type" value="#SOURCE_NAME#PK"/>
                </property>
            </property>
            <property name="tagged_values" direction="forward">
                <property name="isTransformableToLeft" value="false" />
            </property>
        </classifier-feature-map>

  <!-- CLASSIFIER-FEATURE MAPPING: ejbPostCreate method -->
        <classifier-feature-map name="Entity-Class-PIM - ejbPostCreate-Method-EJB"
description="...">
            <!-- FINDERS -->
            <finder side="left" criteria="stereotype" value="entity" />
            <!-- PROPERTIES -->
            <property name="left_type" value="Class" />
            <property name="right_type" value="Operation" />
            <!-- FORWARD PROPERTIES -->
            <property name="name" value="ejbPostCreate" direction="forward" />
            <property name="visibility" value="public" direction="forward" />
            <property name="tagged_values" direction="forward">
                <property name="isTransformableToLeft" value="false" />
            </property>
        </classifier-feature-map>

  <!-- CLASSIFIER-FEATURE MAPPING: ejbActivate -->
        <classifier-feature-map name="Entity-Class-PIM - ejbActivate-Method-EJB"
description="...">
            <!-- FINDERS -->
            <finder side="left" criteria="stereotype" value="entity" />
            <!-- PROPERTIES -->
            <property name="left_type" value="Class" />
            <property name="right_type" value="Operation" />
            <!-- FORWARD PROPERTIES -->
            <property name="name" value="ejbActivate" direction="forward" />
            <property name="visibility" value="public" direction="forward" />
        </classifier-feature-map>

  <!-- CLASSIFIER-FEATURE MAPPING: ejbLoad -->
        <classifier-feature-map name="Entity-Class-PIM - ejbLoad-Method-EJB"
description="...">
            <!-- FINDERS -->
            <finder side="left" criteria="stereotype" value="entity" />
            <!-- PROPERTIES -->
            <property name="left_type" value="Class" />
            <property name="right_type" value="Operation" />
            <!-- FORWARD PROPERTIES -->
            <property name="name" value="ejbLoad" direction="forward" />
            <property name="visibility" value="public" direction="forward" />
        </classifier-feature-map>

  <!-- CLASSIFIER-FEATURE MAPPING: ejbPassivate -->
        <classifier-feature-map name="Entity-Class-PIM - ejbPassivate-Method-EJB"
description="...">
            <!-- FINDERS -->
            <finder side="left" criteria="stereotype" value="entity" />
            <!-- PROPERTIES -->
            <property name="left_type" value="Class" />
            <property name="right_type" value="Operation" />
            <!-- FORWARD PROPERTIES -->
            <property name="name" value="ejbPassivate" direction="forward" />
            <property name="visibility" value="public" direction="forward" />
        </classifier-feature-map>

  <!-- CLASSIFIER-FEATURE MAPPING: ejbRemove -->
        <classifier-feature-map name="Entity-Class-PIM - ejbRemove-Method-EJB"
description="...">
            <!-- FINDERS -->
            <finder side="left" criteria="stereotype" value="entity" />
            <!-- PROPERTIES -->
            <property name="left_type" value="Class" />
            <property name="right_type" value="Operation" />
            <!-- FORWARD PROPERTIES -->
            <property name="name" value="ejbRemove" direction="forward" />
            <property name="visibility" value="public" direction="forward" />
```

```
        </classifier-feature-map>

  <!-- CLASSIFIER-FEATURE MAPPING: ejbStore -->
        <classifier-feature-map name="Entity-Class-PIM - ejbStore-Method-EJB"
description="...">
            <!-- FINDERS -->
            <finder side="left" criteria="stereotype" value="entity" />
            <!-- PROPERTIES -->
            <property name="left_type" value="Class" />
            <property name="right_type" value="Operation" />
            <!-- FORWARD PROPERTIES -->
            <property name="name" value="ejbStore" direction="forward" />
            <property name="visibility" value="public" direction="forward" />
        </classifier-feature-map>

  <!-- CLASSIFIER-FEATURE MAPPING: setEntityContext -->
        <classifier-feature-map name="Entity-Class-PIM - setEntityContext-Method-EJB"
description="...">
            <!-- FINDERS -->
            <finder side="left" criteria="stereotype" value="entity" />
            <!-- PROPERTIES -->
            <property name="left_type" value="Class" />
            <property name="right_type" value="Operation" />
            <!-- FORWARD PROPERTIES -->
            <property name="name" value="setEntityContext" direction="forward" />
            <property name="visibility" value="public" direction="forward" />
            <property name="parameters" direction="forward">
                <property name="ctx">
                    <property name="direction_kind" value="in" />
                    <property name="classifier_type" value="javax.ejb.EntityContext"/>
                </property>
            </property>
        </classifier-feature-map>

  <!-- CLASSIFIER-FEATURE MAPPING: unsetEntityContext -->
        <classifier-feature-map name="Entity-Class-PIM - unsetEntityContext-Method-EJB"
description="...">
            <!-- FINDERS -->
            <finder side="left" criteria="stereotype" value="entity" />
            <!-- PROPERTIES -->
            <property name="left_type" value="Class" />
            <property name="right_type" value="Operation" />
            <!-- FORWARD PROPERTIES -->
            <property name="name" value="unsetEntityContext" direction="forward" />
            <property name="visibility" value="public" direction="forward" />
        </classifier-feature-map>

    </classifier-map>


  <!-- ENTITY - ENTITY BEAN PK -->

    <classifier-map name="Entity-Class-PIM - EntiyBeanPK-Class-EJB" description="..."
id="entityPK">
        <!-- FINDERS -->
        <finder side="left" criteria="stereotype" value="entity" />
        <!-- PROPERTIES -->
        <property name="left_type" value="Class" />
        <property name="right_type" value="Class" />
        <!-- FORWARD PROPERTIES -->
        <property name="name" value="#SOURCE_NAME#PK" direction="forward" />

  <!-- FEATURE MAPPING: EntityPK fields -->
        <feature-map name="Entity-Class-PIM - PrimareKey-Attribute-EJB"
description="...">
            <!-- FINDERS -->
            <finder side="left" criteria="stereotype" value="pk" />
            <!-- PROPERTIES -->
            <property name="left_type" value="Class" />
            <property name="right_type" value="Attribute" />
            <!-- FORWARD PROPERTIES -->
            <property name="visibility" value="public" direction="forward" />
        </feature-map>

  <!-- CLASSIFIER-FEATURE MAPPING: hashCode -->
        <classifier-feature-map name="Entity-Class-PIM - hashCode-Method-EJB"
description="...">
```

```
            <!-- FINDERS -->
            <finder side="left" criteria="stereotype" value="entity" />
            <!-- PROPERTIES -->
            <property name="left_type" value="Class" />
            <property name="right_type" value="Operation" />
            <!-- FORWARD PROPERTIES -->
            <property name="name" value="hashCode" direction="forward" />
            <property name="visibility" value="public" direction="forward" />
            <property name="parameters" direction="forward">
                <property name="return">
                    <property name="direction_kind" value="return" />
                    <property name="data_type" value="int" />
                </property>
            </property>
        </classifier-feature-map>

  <!-- CLASSIFIER-FEATURE MAPPING: equals -->
        <classifier-feature-map name="Entity-Class-PIM - equals-Method-EJB"
description="...">
            <!-- FINDERS -->
            <finder side="left" criteria="stereotype" value="entity" />
            <!-- PROPERTIES -->
            <property name="left_type" value="Class" />
            <property name="right_type" value="Operation" />
            <!-- FORWARD PROPERTIES -->
            <property name="name" value="equals" direction="forward" />
            <property name="visibility" value="public" direction="forward" />
            <property name="parameters" direction="forward">
                <property name="other">
                    <property name="direction_kind" value="in" />
                    <property name="classifier_type" value="java.lang.Object" />
                </property>
                <property name="return">
                    <property name="direction_kind" value="return" />
                    <property name="data_type" value="boolean" />
                </property>
            </property>
        </classifier-feature-map>

    </classifier-map>

  </classifier-map>

<!-- ENTITY - REMOTE COMPONENT INTERFACE -->

  <classifier-map name="Entity-Class-PIM - EntityBean-RemoteInterface-EJB"
description="..." id="entityRemoteInterface">
    <!-- FINDERS -->
    <finder side="left" criteria="stereotype" value="entity" />
    <!-- PROPERTIES -->
    <property name="left_type" value="Class" />
    <property name="right_type" value="Interface" />
    <!-- FORWARD PROPERTIES -->
    <property name="stereotype" value="EJBRemoteInterface" direction="forward" />

<!-- FEATURE MAPPING: getters for business attributes -->
    <feature-map name="Attribute-PIM - Getter-Method-EJB" description="...">
        <!-- FINDERS -->
        <finder side="left" criteria="type" value="Attribute" />
        <!-- PROPERTIES -->
        <property name="left_type" value="Attribute" />
        <property name="right_type" value="Operation" />
        <!-- FORWARD PROPERTIES -->
        <property name="name_transformation" direction="forward">
            <property name="input" value="#SOURCE_NAME#" />
            <property name="regexp" value="(.*)" />
            <property name="subst" value="get\u$1" />
        </property>
        <property name="stereotype" value="EJBRemoteMethod" direction="forward" />
        <property name="visibility" value="public" direction="forward" />
        <property name="parameters" direction="forward">
            <property name="return">
                <property name="direction_kind" value="return" />
                <property name="classifier_type" value="#SOURCE_TYPE#" />
            </property>
        </property>
    </feature-map>
```

```
<!-- FEATURE MAPPING: setters -->
      <feature-map name="Attribute-PIM - Setter-Method-EJB" description="...">
          <!-- FINDERS -->
          <finder side="left" criteria="type" value="Attribute" />
          <!-- PROPERTIES -->
          <property name="left_type" value="Attribute" />
          <property name="right_type" value="Operation" />
          <!-- FORWARD PROPERTIES -->
          <property name="name_transformation" direction="forward" >
              <property name="input" value="#SOURCE_NAME#" />
              <property name="regexp" value="(.*)" />
              <property name="subst" value="set\u$1" />
          </property>
          <property name="visibility" value="public" direction="forward" />
          <property name="stereotype" value="EJBRemoteMethod" direction="forward" />
          <property name="parameters" direction="forward">
              <property name="#SOURCE_NAME#">
                  <property name="direction_kind" value="in" />
                  <property name="classifier_type" value="#SOURCE_TYPE#" />
              </property>
          </property>
      </feature-map>

  </classifier-map>

<!-- ENTITY - REMOTE HOME INTERFACE -->

  <classifier-map name="Entity-Class-PIM - EntityBean-HomeInterface-EJB"
description="..." id="entityHomeInterface">
      <!-- FINDERS -->
      <finder side="left" criteria="stereotype" value="entity" />
      <!-- PROPERTIES -->
      <property name="left_type" value="Class" />
      <property name="right_type" value="Interface" />
      <!-- FORWARD PROPERTIES -->
      <property name="name" value="#SOURCE_NAME#Home" direction="forward" />
      <property name="stereotype" value="EJBHomeInterface" direction="forward" />

<!-- FEATURE MAPPING: findByPrimaryKey method -->
      <feature-map name="PKAttribute-PIM - findByPrimaryKey-Method-EJB"
description="...">
          <!-- FINDERS -->
          <finder side="left" criteria="stereotype" value="pk" />
          <!-- PROPERTIES -->
          <property name="left_type" value="Attribute" />
          <property name="right_type" value="Operation" />
          <!-- FORWARD PROPERTIES -->
          <property name="name" value="findByPrimaryKey" direction="forward" />
          <property name="stereotype" value="EJBFinderMethod" direction="forward" />
          <property name="visibility" value="public" direction="forward" />
          <property name="parameters" direction="forward">
              <property name="pk">
                  <property name="direction_kind" value="in" />
                  <property name="classifier_type" value="#SOURCE_TYPE#" />
              </property>
              <property name="return">
                  <property name="direction_kind" value="return" />
                  <property name="interface_type" value="#OWNER_SOURCE_NAME#" />
              </property>
          </property>
      </feature-map>

<!-- CLASSIFIER-FEATURE MAPPING: create method -->
      <classifier-feature-map name="Entity-Class-PIM - create-Method-EJB"
description="...">
          <!-- FINDERS -->
          <finder side="left" criteria="stereotype" value="entity" />
          <!-- PROPERTIES -->
          <property name="left_type" value="Class" />
          <property name="right_type" value="Operation" />
          <!-- FORWARD PROPERTIES -->
          <property name="name" value="create" direction="forward" />
          <property name="stereotype" value="EJBCreateMethod" direction="forward" />
          <property name="visibility" value="public" direction="forward" />

          <property name="parameters" direction="forward">
```

```
            <property name="return">
                <property name="direction_kind" value="return" />
                <property name="interface_type" value="#SOURCE_NAME#" />
            </property>
        </property>
    </classifier-feature-map>

  </classifier-map>

<!-- ENTITY - LOCAL COMPONENT INTERFACE -->

  <classifier-map name="Entity-Class-PIM - EntityBean-LocalInterface-EJB"
description="..." id="entityLocalInterface">
    <!-- FINDERS -->
    <finder side="left" criteria="stereotype" value="entity" />
    <!-- PROPERTIES -->
    <property name="left_type" value="Class" />
    <property name="right_type" value="Interface" />
    <!-- FORWARD PROPERTIES -->
    <property name="name" value="#SOURCE_NAME#Local" direction="forward" />
    <property name="stereotype" value="EJBLocalInterface" direction="forward" />

<!-- FEATURE MAPPING: getters for business attributes -->
    <feature-map name="Attribute-PIM - Getter-Method-EJB" description="..." >
        <!-- FINDERS -->
        <finder side="left" criteria="type" value="Attribute" />
        <!-- PROPERTIES -->
        <property name="left_type" value="Attribute" />
        <property name="right_type" value="Operation" />
        <!-- FORWARD PROPERTIES -->
        <property name="name_transformation" direction="forward">
            <property name="input" value="#SOURCE_NAME#" />
            <property name="regexp" value="(.*)" />
            <property name="subst" value="get\u$1" />
        </property>
        <property name="stereotype" value="EJBLocalMethod" direction="forward" />
        <property name="visibility" value="public" direction="forward" />
        <property name="parameters" direction="forward">
            <property name="return">
                <property name="direction_kind" value="return" />
                <property name="classifier_type" value="#SOURCE_TYPE#" />
            </property>
        </property>
    </feature-map>

<!-- FEATURE MAPPING: setters -->
    <feature-map name="Attribute-PIM - Setter-Method-EJB" description="...">
        <!-- FINDERS -->
        <finder side="left" criteria="type" value="Attribute" />
        <!-- PROPERTIES -->
        <property name="left_type" value="Attribute" />
        <property name="right_type" value="Operation" />
    <!-- FORWARD PROPERTIES -->
        <property name="name_transformation" direction="forward" >
            <property name="input" value="#SOURCE_NAME#" />
            <property name="regexp" value="(.*)" />
            <property name="subst" value="set\u$1" />
        </property>
        <property name="visibility" value="public" direction="forward" />
        <property name="stereotype" value="EJBLocalMethod" direction="forward" />
        <property name="parameters" direction="forward">
            <property name="#SOURCE_NAME#">
                <property name="direction_kind" value="in" />
                <property name="classifier_type" value="#SOURCE_TYPE#" />
            </property>
        </property>
    </feature-map>

  </classifier-map>

<!-- ENTITY - LOCAL HOME INTERFACE -->

  <classifier-map name="Entity-Class-PIM - EntityBean-LocalHomeInterface-EJB"
description="..." id="entityLocalHomeInterface">
    <!-- FINDERS -->
    <finder side="left" criteria="stereotype" value="entity" />
    <!-- PROPERTIES -->
```

```
      <property name="left_type" value="Class" />
      <property name="right_type" value="Interface" />
      <!-- FORWARD PROPERTIES -->
      <property name="name" value="#SOURCE_NAME#LocalHome" direction="forward" />
      <property name="stereotype" value="EJBLocalHomeInterface" direction="forward" />

<!-- FEATURE MAPPING: findByPrimaryKey method -->
      <feature-map name="PKAttribute-PIM - findByPrimaryKey-Method-EJB"
description="...">
          <!-- FINDERS -->
          <finder side="left" criteria="stereotype" value="PK" />
          <!-- PROPERTIES -->
          <property name="left_type" value="Attribute" />
          <property name="right_type" value="Operation" />
          <!-- FORWARD PROPERTIES -->
          <property name="name" value="findByPrimaryKey" direction="forward" />
          <property name="stereotype" value="EJBFinderMethod" direction="forward" />
          <property name="visibility" value="public" direction="forward" />
          <property name="parameters" direction="forward">
              <property name="pk">
                  <property name="direction_kind" value="in" />
                  <property name="classifier_type" value="#SOURCE_TYPE#" />
              </property>
              <property name="return">
                  <property name="direction_kind" value="return" />
                  <property name="interface_type" value="#OWNER_SOURCE_NAME#Local" />
              </property>
          </property>
      </feature-map>

<!-- CLASSIFIER-FEATURE MAPPING: create method -->
      <classifier-feature-map name="Entity-Class-PIM - create-Method-EJB"
description="...">
          <!-- FINDERS -->
          <finder side="left" criteria="stereotype" value="entity" />
          <!-- PROPERTIES -->
          <property name="left_type" value="Class" />
          <property name="right_type" value="Operation" />
          <!-- FORWARD PROPERTIES -->
          <property name="name" value="create" direction="forward" />
          <property name="stereotype" value="EJBCreateMethod" direction="forward" />
          <property name="visibility" value="public" direction="forward" />

          <property name="parameters" direction="forward">
              <property name="return">
                  <property name="direction_kind" value="return" />
                  <property name="interface_type" value="#SOURCE_NAME#Local" />
              </property>
          </property>
      </classifier-feature-map>

  </classifier-map>

<!--############################## SERVICE ##############################-->

<!-- SERVICE - SESSION BEAN -->

  <classifier-map name="Service-Component-PIM - SessionBean-Component-EJB"
description="...">
      <!-- FINDERS -->
      <finder side="left" criteria="stereotype" value="service" />
      <finder side="right" criteria="stereotype" value="EJBSessionBean" />
      <!-- PROPERTIES -->
      <property name="left_type" value="Class" />
      <property name="right_type" value="Component" />
      <!-- FORWARD PROPERTIES -->
      <property name="stereotype" value="SessionBeanComponent" direction="forward" />

  <!-- SERVICE CLASS - SERVICE BEAN IMPLEMENTATION CLASS -->
      <classifier-map name="Service-Class-PIM - SessionBean-Class-EJB" description="..."
id="sessionBean" preserve-relationships="yes">
          <!-- FINDERS -->
          <finder side="left" criteria="stereotype" value="service" />
          <finder side="right" criteria="stereotype" value="EJBSessionBean" />
          <!-- PROPERTIES -->
          <property name="left_type" value="Class" />
          <property name="right_type" value="Class" />
```

```
        <!-- FORWARD PROPERTIES -->
        <property name="name" value="#SOURCE_NAME#Bean" direction="forward" />
        <property name="stereotype" value="EJBSessionBean" direction="forward" />
        <!-- REVERSE PROPERTIES -->
        <property name="name_transformation" direction="reverse">
               <property name="input" value="#SOURCE_NAME#" />
               <property name="regexp" value="(.*)Bean$" />
               <property name="subst" value="$1" />
        </property>

  <!-- FEATURE MAPPING: copy business operations -->
        <feature-map name="Operation-PIM - Method-EJB" description="...">
            <!-- FINDERS -->
            <finder side="left" criteria="type" value="Operation" />
            <finder side="right" criteria="stereotype" value="EJBBusinessOperation"/>
            <!-- PROPERTIES -->
            <property name="left_type" value="Operation" />
            <property name="right_type" value="Operation" />
            <!-- FORWARD PROPERTIES -->
            <property name="stereotype" value="EJBBusinessOperation"
direction="forward" />
        </feature-map>

     </classifier-map>
  </classifier-map>

<!-- SERVICE - REMOTE COMPONENT INTERFACE -->

  <classifier-map name="Service-Class-PIM - SessionBean-RemoteInterface-EJB"
description="..." id="sessionRemoteInterface">
     <!-- FINDERS -->
     <finder side="left" criteria="stereotype" value="service" />
     <!-- PROPERTIES -->
     <property name="left_type" value="Class" />
     <property name="right_type" value="Interface" />
     <!-- FORWARD PROPERTIES -->
     <property name="stereotype" value="EJBRemoteInterface" direction="forward" />

<!-- FEATURE MAPPING: copy business operations -->
     <feature-map name="Operation-PIM - Method-EJB" description="...">
        <!-- FINDERS -->
        <finder side="left" criteria="type" value="Operation" />
        <!-- PROPERTIES -->
        <property name="left_type" value="Operation" />
        <property name="right_type" value="Operation" />
        <!-- FORWARD PROPERTIES -->
        <property name="stereotype" value="EJBRemoteMethod" direction="forward" />
     </feature-map>

  </classifier-map>

<!-- SERVICE - REMOTE HOME INTERFACE -->

  <classifier-map name="Service-Class-PIM - SessionBean-HomeInterface-EJB"
description="..." id="sessionHomeInterface">
     <!-- FINDERS -->
     <finder side="left" criteria="stereotype" value="service" />
     <!-- PROPERTIES -->
     <property name="left_type" value="Class" />
     <property name="right_type" value="Interface" />
     <!-- FORWARD PROPERTIES -->
     <property name="name" value="#SOURCE_NAME#Home" direction="forward" />
     <property name="stereotype" value="EJBHomeInterface" direction="forward" />

  </classifier-map>

<!-- SERVICE - LOCAL COMPONENT INTERFACE -->

  <classifier-map name="Service-Class-PIM - SessionBean-LocalInterface-EJB"
description="..." id="sessionLocalInterface">
     <!-- FINDERS -->
     <finder side="left" criteria="stereotype" value="service" />
     <!-- PROPERTIES -->
     <property name="left_type" value="Class" />
     <property name="right_type" value="Interface" />
     <!-- FORWARD PROPERTIES -->
     <property name="name" value="#SOURCE_NAME#Local" direction="forward" />
```

```
      <property name="stereotype" value="EJBLocalInterface" direction="forward" />

<!-- FEATURE MAPPING: copy business operations -->
      <feature-map name="Operation-PIM - Method-EJB" description="...">
          <!-- FINDERS -->
          <finder side="left" criteria="type" value="Operation" />
          <!-- PROPERTIES -->
          <property name="left_type" value="Operation" />
          <property name="right_type" value="Operation" />
          <!-- FORWARD PROPERTIES -->
          <property name="stereotype" value="EJBLocalMethod" direction="forward" />
      </feature-map>

  </classifier-map>

<!-- SERVICE - LOCAL HOME INTERFACE -->

  <classifier-map name="Service-Class-PIM - SessionBean-LocalHomeInterface-EJB"
description="..." id="sessionLocalHomeInterface">
      <!-- FINDERS -->
      <finder side="left" criteria="stereotype" value="service" />
      <!-- PROPERTIES -->
      <property name="left_type" value="Class" />
      <property name="right_type" value="Interface" />
      <!-- FORWARD PROPERTIES -->
      <property name="name" value="#SOURCE_NAME#LocalHome" direction="forward" />
      <property name="stereotype" value="EJBLocalHomeInterface" direction="forward" />

  </classifier-map>

<!--############################## RELATIONSHIPS ##############################-->

<!-- ENTITY REMOTE INTERFACES -->

  <relationship name="Dependency (entityBean - entityRemoteInterface)" type="Dependency"
direction="forward">
      <map id="entityRemoteInterface">
          <property name="supplier" value="yes" />
      </map>
      <map id="entityBean">
          <property name="client" value="yes" />
      </map>
  </relationship>

  <relationship name="Dependency (entityBean - entityHomeInterface)" type="Dependency"
direction="forward">
      <map id="entityHomeInterface">
          <property name="supplier" value="yes" />
      </map>
      <map id="entityBean">
          <property name="client" value="yes" />
      </map>
  </relationship>

  <relationship name="Dependency (entityHomeInterface - entityRemoteInterface)"
type="Dependency" direction="forward">
      <map id="entityRemoteInterface">
          <property name="supplier" value="yes" />
      </map>
      <map id="entityHomeInterface">
          <property name="client" value="yes" />
      </map>
  </relationship>

  <relationship name="Dependency (entityHomeInterface - entityPK)" type="Dependency"
direction="forward">
      <map id="entityPK">
          <property name="supplier" value="yes" />
      </map>
      <map id="entityHomeInterface">
          <property name="client" value="yes" />
      </map>
  </relationship>

<!-- ENTITY LOCAL INTERFACES -->
```

```
  <relationship name="Dependency (entityBean - entityLocalInterface)" type="Dependency"
direction="forward">
      <map id="entityLocalInterface">
          <property name="supplier" value="yes" />
      </map>
      <map id="entityBean">
          <property name="client" value="yes" />
      </map>
  </relationship>

  <relationship name="Dependency (entityBean - entityLocalHomeInterface)"
type="Dependency" direction="forward">
      <map id="entityLocalHomeInterface">
          <property name="supplier" value="yes" />
      </map>
      <map id="entityBean">
          <property name="client" value="yes" />
      </map>
  </relationship>

  <relationship name="Dependency (entityLocalHomeInterface - entityLocalInterface)"
type="Dependency" direction="forward">
      <map id="entityLocalInterface">
          <property name="supplier" value="yes" />
      </map>
      <map id="entityLocalHomeInterface">
          <property name="client" value="yes" />
      </map>
  </relationship>

  <relationship name="Dependency (entityLocalHomeInterface - entityPK)"
type="Dependency" direction="forward">
      <map id="entityPK">
          <property name="supplier" value="yes" />
      </map>
      <map id="entityLocalHomeInterface">
          <property name="client" value="yes" />
      </map>
  </relationship>

<!-- SESSION REMOTE INTERFACES -->

  <relationship name="Dependency (sessionBean - sessionRemoteInterface)"
type="Dependency" direction="forward">
      <map id="sessionRemoteInterface">
          <property name="supplier" value="yes" />
      </map>
      <map id="sessionBean">
          <property name="client" value="yes" />
      </map>
  </relationship>

  <relationship name="Dependency (sessionBean - sessionHomeInterface)" type="Dependency"
direction="forward">
      <map id="sessionHomeInterface">
          <property name="supplier" value="yes" />
      </map>
      <map id="sessionBean">
          <property name="client" value="yes" />
      </map>
  </relationship>

  <relationship name="Dependency (sessionHomeInterface - sessionRemoteInterface)"
type="Dependency" direction="forward">
      <map id="sessionRemoteInterface">
          <property name="supplier" value="yes" />
      </map>
      <map id="sessionHomeInterface">
          <property name="client" value="yes" />
      </map>
  </relationship>

<!-- SESSION LOCAL INTERFACES -->

  <relationship name="Dependency (sessionBean - sessionLocalInterface)"
type="Dependency" direction="forward">
      <map id="sessionLocalInterface">
```

```
            <property name="supplier" value="yes" />
        </map>
        <map id="sessionBean">
            <property name="client" value="yes" />
        </map>
    </relationship>

    <relationship name="Dependency (sessionBean - sessionLocalHomeInterface)"
type="Dependency" direction="forward">
        <map id="sessionLocalHomeInterface">
            <property name="supplier" value="yes" />
        </map>
        <map id="sessionBean">
            <property name="client" value="yes" />
        </map>
    </relationship>

    <relationship name="Dependency (sessionLocalHomeInterface - sessionLocalInterface)"
type="Dependency" direction="forward">
        <map id="sessionLocalInterface">
            <property name="supplier" value="yes" />
        </map>
        <map id="sessionLocalHomeInterface">
            <property name="client" value="yes" />
        </map>
    </relationship>

</transformation-mapping>
```