Universidade Federal de Pernambuco
Centro de Informática

Graduação em Ciência da Computação

Automatizando Refatoramentos Arquiteturais em UML-RT utilizando Transformação de Modelos

Eliaquim Lima Sá Neto

TRABALHO DE GRADUAÇÃO

Recife, 24 de janeiro de 2008

Universidade Federal de Pernambuco
Centro de Informática

Eliaquim Lima Sá Neto

Automatizando Refatoramentos Arquiteturais em UML-RT utilizando Transformação de Modelos

Monografia apresentada ao Centro de Informática da Universidade Federal de Pernambuco, como requisito parcial para obtenção do Grau de Bacharel em Ciência da Computação.

Orientador: Augusto Cesar Alves Sampaio
Co-orientador: Rodrigo Teixeira Ramos

Recife, 24 de janeiro de 2008

Aos meus pais, Josias e Kiria.

Agradecimentos

"Se pude ver mais longe do que outros, foi porque
me apoiei sobre os ombros de gigantes."
(Isaac Newton)

Em primeiro lugar, gostaria de agradecer a Deus, por estar sempre comigo, me dando forças, bençãos, amor e perseverança, e por me proporcionar a oportunidade de estar cumprindo mais essa etapa da minha vida. Agradecê-Lo também por ter colocado em minha vida todas as pessoas que cito abaixo.

Aos meus pais, Josias e Kiria, pelo amor incondicional que têm por mim, pelo exemplo que representam, e por sempre valorizarem minha educação, apoiando nos momentos mais difíceis e comemorando nos mais alegres.

Aos meus irmãos, Kilma e Lucas, por serem companheiros de todas as horas, e pelos momentos de brincadeiras e alegria que me proporcionam.

À minha namorada Daniela, pelo carinho, amor e atenção dedicados a mim, e pela paciência comigo durante essa fase conturbada de conclusão de curso. Por ter me ajudado a manter a calma e tranquilidade nos momentos em que eu não acreditava que iria conseguir terminar este trabalho.

A toda minha familia por sempre entender, durante esses quatro anos, quando tive que passar noites inteiras no CIn, quando tive que me ausentar de momentos familiares para estudar ou fazer projetos, até mesmo (e principalmente) nos fins de semana.

Aos amigos que fiz na faculdade, todos da equipe *Commit Solutions*, membros da comissão de formatura, dentre outros, por compartilharem comigo momentos de muita harmonia, união e companheirismo. Tenham certeza de que vocês são amigos que quero levar para vida toda.

Ao meu co-orientador Rodrigo Teixeira, por acreditar no meu potencial, por ter me ajudado em todos os momentos que precisei, e por ter lido cuidadosamente cada parte deste trabalho.

Ao meu orientador professor Augusto Sampaio pelo seu incentivo constante, por suas orientações e ajuda no desenvolvimento do trabalho.

Aos colegas de trabalho, aos professores e funcionários do Centro de Informática da UFPE. Agradecimento especial aos professores Nelson Rosa, Liliane Salgado, Kátia Guimarães, Silvio Melo e Marcília Campos, que construíram também laços de amizade, e por terem me ajudado no decorrer do curso, seja em monitorias ou em iniciação científica.

Resumo

Nos últimos anos, a indústria e a academia têm voltado grande parte de suas atenções para o desenvolvimento dirigido a modelos (MDD). MDD é um método que tem como principal objetivo aumentar a produtividade do desenvolvimento através da ênfase na atividade de modelagem. Em geral, neste método, desenvolvedores de *software* focam seu trabalho, inicialmente, em modelos de alto nível, independentes de qualquer tecnologia, com as regras de negócio do sistema. A evolução destes modelos é realizada através de transformações que os reestruturam, melhoram sua qualidade ou acrescentam informações a eles.

Por se tratar de uma área relativamente nova, a maioria dessas transformações não é ainda, de fato, automática. Este problema se acentua principalmente em alguns domínios específicos, que possuem uma menor comunidade que os suportam.

Este trabalho é uma contribuição no que diz respeito à automação de refatoramentos entre modelos. As transformações aqui abordadas fazem parte de um subconjunto de leis de transformações definidas em [14] para uma linguagem de descrição arquitetural chamada UML-RT [18], e foram implementadas utilizando a linguagem de meta-modelagem Kermeta [17].

O uso destas transformações é ainda exemplificado, neste trabalho, através de sua aplicação no desenvolvimento do clássico sistema Produtor-Consumidor, mostrando assim como o desenvolvedor poderá utilizar estas transformações para aumentar sua produtividade durante a modelagem, e evolução do modelo.

Palavras-chave: refatoramentos arquiteturais, transformação de modelos, desenvolvimento baseado em modelos, metamodelagem, Kermeta, UML-RT.

Sumário

Lista de figuras

Lista de Tabelas

1. Introdução

O progresso obtido no processo de desenvolvimento de *software* durante os últimos vinte anos é facilmente percebido quando levamos em consideração a complexidade dos sistemas desenvolvidos atualmente. Apesar disso, a área de desenvolvimento de *software* ainda enfrenta uma série de desafios, como, por exemplo, a constante necessidade de adaptação do sistema às novas tecnologias e às mudanças em seus requisitos.

O impacto de tais desafios é ainda maior quando se é dada uma maior importância para a etapa de codificação do *software*, deixando de lado outros artefatos de requisitos e de análise e projeto do sistema, como diagramas de casos de uso, de classe e de interação. No dia-a-dia, desenvolvedores de *software* são tentados a fazer mudanças apenas no código-fonte da aplicação, pressionados pelos curtos prazos de entrega e pelas incessantes requisições de mudança. Da mesma maneira, esta pressão sobre os desenvolvedores faz com que artefatos de alto nível, como os de análise e projeto, sejam preteridos ou mesmo não atualizados.

Em uma visão de curto prazo, alterações apenas no código parecem resolver o problema das mudanças nos requisitos. Porém, quando olhada em uma visão de longo prazo, essa solução pode trazer sérios problemas que podem vir a comprometer toda a manutenção do *software*.

Visando mitigar os problemas da documentação e manutenção citados acima, bem como problemas de produtividade do desenvolvedor, portabilidade e interoperabilidade do sistema, observa-se cada vez mais o interesse da academia e da indústria na engenharia dirigida a modelos [8]. Este interesse tem-se voltado principalmente ao *framework* para desenvolvimento de *software* definido pela OMG (*Object Management Group*), chamado *Model Driven Architecture* (MDA). O maior destaque da MDA está na importância de modelos no processo de desenvolvimento de *software*. Na MDA, este processo é dirigido pela atividade de modelar o sistema de *software* [4].

Para resolver o problema da produtividade do desenvolvedor, a MDA faz com que o foco deste esteja no desenvolvimento do PIM (*Plataform Independent Model*), que consiste em uma especificação independente de qualquer tecnologia, da estrutura e do funcionamento do sistema [8]. Com isso, o desenvolvedor não tem mais que se preocupar com detalhes técnicos e pode ter sua atenção voltada para a elaboração de um modelo que melhor atende o negócio do sistema. Pelo fato de ser um modelo independente de qualquer tecnologia ou plataforma, o PIM possibilita que o *software* seja totalmente portável, sendo necessário apenas **transformações** do PIM em múltiplos PSM's (*Plataform Specific Model*), que são modelos específicos para cada plataforma [8]. Por conseguinte, tais PSM's podem ser sucessivamente transformados em modelos ainda mais específicos de plataforma, até serem transformados em código de programa.

Além de transformações entre os diversos níveis de abstração da plataforma (PIM e PSM), transformações em um mesmo nível também são necessárias para suportar amplamente o desenvolvimento dirigido a modelos. Desta maneira, fazem-se necessárias transformações de PIM para PIM, ou de PSM para PSM, a fim de

reestruturar ou melhorar a qualidade destes modelos. Tais transformações são conhecidas como **refatoramentos de modelo**, similarmente a refatoramentos de código.

Para que o processo de desenvolvimento de *software* seja dirigido pela atividade de modelagem, é essencial que as transformações entre modelos sejam automatizadas [8], concentrando o trabalho dos desenvolvedores apenas na elaboração destes. Por se tratar de uma área relativamente nova, a maioria dessas transformações não é ainda, de fato, automática, necessitando assim, de ferramentas que automatizem esse processo.

Este trabalho é uma contribuição no que diz respeito à automação de refatoramentos entre modelos. As transformações aqui abordadas fazem parte de um subconjunto de leis de transformações definidas em [14] para uma linguagem de descrição arquitetural chamada UML-RT [18], e foram implementadas utilizando a linguagem de meta-modelagem *Kermeta* [17].

Outra contribuição importante deste trabalho é a proposição de um metamodelo para a linguagem UML-RT. Sua construção foi necessária porque se trata de uma linguagem pertencente a uma ferramenta proprietária, cuja semântica é disposta implicitamente na ferramenta e em poucos documentos disponibilizados pela Rational [23]. O metamodelo aqui proposto foi projetado de forma a ser o mais abrangente possível, e os seus elementos representam as três principais visões arquiteturais da linguagem: diagrama de classes, diagrama de estrutura e diagrama de estados.

O conjunto de transformações implementado neste trabalho foca principalmente em mudanças na declaração de elementos de projeto de UML-RT, como classes, cápsulas e protocolos. Cada lei de transformação possui dois sentidos de aplicação e garante que o comportamento dos elementos envolvidos é mantido após sua aplicação. Como exemplo de uma transformação aqui abordada, podemos citar a transformação responsável por associar uma cápsula (componente de software em UML-RT) a um protocolo de comunicação, que é utilizado para estabelecer como a cápsula pode interagir com outras cápsulas do sistema.

No capítulo 2 são introduzidos alguns conceitos básicos sobre o desenvolvimento baseado em modelos, necessários para o bom entendimento desta monografia. No decorrer do capítulo são apresentadas as definições de *modelo*, *metamodelo* e *transformações de modelo*, destacando suas principais características. Ao final deste capítulo a Arquitetura Dirigida a Modelos (MDA) proposta pela OMG é descrita em detalhes a fim de exemplificar um contexto de aplicação destes conceitos.

O capítulo 3 é dedicado exclusivamente à linguagens de transformação de modelos. Neste capítulo, são descritos vários tipos de linguagens de transformação, citando-se desde um padrão definido pela OMG para linguagens de transformação, o *QVT*, até linguagens específicas de modelo, como *Kermeta e ATL*, passando também por linguagens não específicas, como, *JMI* e *XSLT*. Uma maior atenção é dada a *Kermeta*, tendo em vista que foi a linguagem escolhida para a implementação deste trabalho.

No capítulo 4, mostramos o metamodelo elaborado neste trabalho para a linguagem de descrição arquitetural UML-RT. Este metamodelo serve como base semântica para os modelos manipulados pelas transformações implementadas neste trabalho, representando as três principais visões arquiteturais de UML-RT: definição

estática dos elementos de projeto (classes e cápsulas), definição comportamental (diagramas de estado), e a configuração de arquitetura em tempo de execução (diagramas com estruturas das cápsulas).

No capítulo 5, falamos sobre as transformações abordadas neste trabalho e como estas foram implementadas. Explicamos detalhadamente quais foram as tecnologias utilizadas, tais como *EMF, XMI*, e como o conjunto de transformações foi projetado utilizando o padrão *Command*. Falamos também, neste capítulo, das limitações presentes na implementação, e de como estas transformações são aplicadas a arquivos de modelos no formato XMI.

Finalmente, no capítulo 6, damos um exemplo de como as transformações aqui implementadas podem ser utilizadas em situações do cotidiano. Partindo de um modelo inicial, mostramos a aplicação e o funcionamento das transformações, aplicando-as sucessivamente até construírmos o modelo final de um sistema *Produtor-Consumidor*, amplamente conhecido na literatura da área.

O capítulo 7 contém as conclusões deste trabalho, assim como possibilidades de trabalhos futuros, trabalhos relacionados e as considerações finais.

2. Desenvolvimento baseado em modelos

Neste capítulo serão abordados os temas que representam o embasamento teórico necessário para o bom entendimento desta monografia. Iniciaremos com as definições de modelo, metamodelo e transformações de modelo. No final, apresentaremos MDA (*Model Driven Architecture*), seus padrões e suas características.

2.1 Modelo

Nesta seção falaremos sobre modelos, fornecendo algumas definições, critérios de classificação, taxonomia e linguagens.

2.1.1 Definições

A atividade de modelagem consiste na criação de uma representação, normalmente simplificada, de um objeto do mundo real para algum propósito cognitivo [1]. Esse objeto, dependendo do domínio que esteja sendo abordado, pode variar muito em sua forma, características e comportamento. Em alguns casos, ele não representa uma entidade física, mas sim um processo ou um sistema complexo.

O objeto construído na atividade de modelagem recebe o nome de *modelo*. Além de ser uma representação mais simples e barata do que a realidade, o modelo possui um alto teor de abstração já que, na maioria das vezes, não pode representar todos os aspectos do mundo real. Tal abstração permite que possamos lidar com o mundo de maneira simplificada, evitando sua complexidade, perigo e irreversibilidade [1].

De acordo com [2], a palavra *modelo* pode ter as seguintes definições:

1. Desenho ou imagem que representa o que se pretende reproduzir, desenhando, pintando ou esculpindo.
2. Tudo o que serve para ser imitado
3. Representação, em pequena escala, de um objeto que se pretende executar em ponto grande.
4. Pessoa exemplar
5. Pessoa contratada para desfilar ou exibir roupas confeccionadas por uma grande casa de modas ou por uma *griffe*.

Ao analisar as definições listadas acima, descartando aquelas que não dizem respeito ao nosso contexto, podemos perceber a presença das características principais dos modelos, mencionadas no início. Vemos que, através do seu poder de abstração, o modelo é sempre diferente da entidade que representa no que diz respeito à detalhes que são deixados de lados e que, assim como podemos extrair um modelo de um objeto real, o processo também pode ser aplicado inversamente. A definição 3 deixa claro que um modelo pode ser usado como um exemplo para produzir ou construir algo.

É comum encontrar, na maioria das definições de modelo, referência à "algo que exista na realidade". Precisamos, então, de uma palavra que represente

tal expressão. Como este trabalho está focado no contexto de desenvolvimento de *software,* a palavra que melhor pode ser aplicada é *sistema*.

Um sistema pode ser dito como um conjunto de elementos que interagem entre si [3]. Na maioria das situações, a palavra sistema se refere a um sistema de *software*, mas se estivermos falando de um modelo de negócios, o próprio negócio seria o sistema. [4].

O princípio da substitutabilidade afirma que: *Um modelo M é dito como uma representação de um sistema S para um dado conjunto de questões Q se, para cada questão desse conjunto Q, o modelo M proverá a mesma resposta que o sistema S proveria respondendo a essa mesma questão* [3].

Figura 2.1. Modelo como representação de um sistema

A Figura 2.1 ilustra o princípio da substitutabilidade, mostrando que existe um relacionamento entre modelo e sistema chamado *repOf*, significando que um modelo é uma representação de um sistema. A Figura também deixa claro que, ao aplicar uma pergunta ao modelo, a resposta obtida será a mesma que o sistema forneceria caso a mesma pergunta fosse feita a ele.

Como exemplo de aplicação clara desse princípio podemos assumir o sistema S como sendo o planeta Terra, o modelo M como sendo um globo terrestre e o conjunto Q contendo a seguinte pergunta: "É possível viajar de Recife até Fernando de Noronha de carro?". É lógico que a resposta dessa pergunta, considerando o planeta Terra seria negativa, tendo em vista que Fernando de Noronha é uma ilha. Portanto, para essa pergunta, o globo terrestre confirma o princípio já que serve como modelo para que tal resposta também seja obtida.

2.1.2 Critérios

Para distinguir modelos de outros artefatos, precisamos de critérios. De acordo com [5], qualquer candidato a modelo deve estar de acordo com três critérios, caso contrário, não é um modelo:

- **Critério de mapeamento:** Existe um objeto ou fenômeno original que é mapeado no modelo.
- **Critério de redução:** Nem todas as propriedades do original são mapeadas no modelo, sendo ele de certa maneira reduzido. Por outro lado, o modelo deve refletir pelo menos alguma propriedade do original.
- **Critério de pragmatismo:** O modelo pode substituir o original para algum propósito, ou seja, o modelo é útil.

O critério de mapeamento não implica na existência física do objeto original, ele pode ser planejado ou até fictício. Como exemplo, podemos citar a estimativa de custo de um projeto de *software*, que normalmente é feita no início do processo de desenvolvimento, quando não se tem ainda o *software* em

funcionamento. Nesse caso, o modelo de custo é construído com base no planejamento do *software*, fazendo assim com que uma estimativa possa ser elaborada antes mesmo que o projeto esteja pronto. Nessa situação o modelo pode ser visto como uma projeção do que vem pela frente, uma especulação do futuro.

O critério de redução aparenta descrever uma fraqueza dos modelos, já que existe algo faltando no modelo que está presente no original [6]. Mas esta característica pode ser vista como um ponto positivo dos modelos: normalmente, o modelo pode ser manipulado enquanto que o original não pode.

Como efeito da redução, várias características do original - os atributos "dispensados" - não são encontradas no modelo. Por exemplo, o nome de uma pessoa não é visível em sua fotografia. Por outro lado, características que não vêm do original são adicionadas - atributos abundantes: o tamanho da fotografia não diz nada a respeito da pessoa fotografada [6].

O critério de pragmatismo representa a razão pela qual decidimos utilizar modelos. Como não somos capazes de manipular ou usar o objeto original, construímos o modelo e fazemos uso dele.

2.1.3 Taxonomia

Modelos podem ser classificados em relação à diversos aspectos. Quando de sua representação do objeto do mundo real, podemos classificá-lo como *descritivo* ou *prescritivo*. No primeiro caso, um modelo é dito *descritivo* quando reflete o objeto original, como por exemplo uma fotografia. Já os modelos *prescritivos*, representam modelos que são construídos para especificar algo a ser criado posteriormente. Modelos de *software* são exemplos claros de modelos *prescritivos,* tendo em vista que, normalmente, são construídos antes da elaboração do *software* propriamente dito, e têm como principal objetivo a sua especificação.

Quando um arquiteto desenha uma casa antiga, e depois adiciona ao seu desenho algumas modificações como sugestão, o modelo inicialmente é descritivo, e depois prescritivo. Nesse caso, o modelo é chamado de *transiente* [6].

2.1.4 Linguagem

Para a descrição de um modelo, é necessária uma linguagem. Esta linguagem pode ser UML, português, inglês, uma linguagem de programação, dentre outras. Para permitir que transformações possam ser aplicadas a modelos, que é o objetivo deste trabalho, essa linguagem precisa ser uma linguagem bem-definida. Ela é dita bem-definida porque possui uma forma bem definida e um significado que pode ser interpretado por computadores [4].

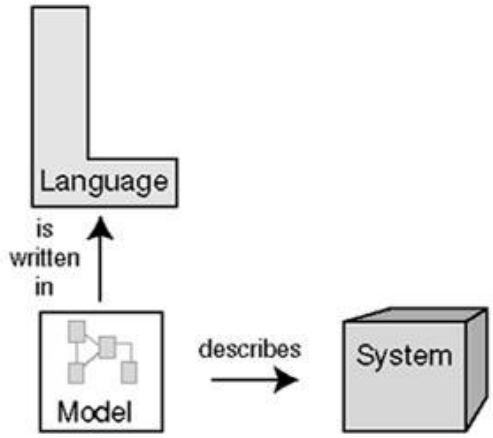

Figura 2.2. Modelo, sistema e linguagem

A análise da Figura 2.2 resume o que foi dito anteriormente: um modelo descreve um sistema e é escrito em uma linguagem. No contexto deste trabalho, esse sistema representa um *software* e a linguagem precisa ser bem-definida, para que possa ser interpretada por um computador.

2.2 Metamodelo

Para o bom entendimento de um modelo, precisamos saber o que representam as entidades presentes nele e qual o significado de cada relacionamento entre elas. Todo modelo deve estar associado a um *metamodelo*, caso contrário não terá significado algum. Um metamodelo é justamente a definição precisa dos elementos, relacionamentos e regras necessários para a criação e entendimento de modelos semânticos [7].

A analogia com o paradigma de orientação a objetos é válida quando da tentativa de mostrar a relação entre um modelo e seu metamodelo associado. O modelo está para o metamodelo assim como o objeto está para a classe, ou seja, o modelo é uma *instância* do metamodelo. Também na orientação a objetos se aplica o princípio de que para o entendimento de um objeto é necessário conhecer a classe da qual ele é uma instância.

A metamodelagem faz parte do domínio de definição de linguagens, portanto, um metamodelo define uma parte de uma linguagem específica de domínio (DSL). Essa parte depende de como o metamodelo será utilizado, podendo ser a sintaxe da linguagem, sua semântica, visões, diagramas, dentre outros [3]. A Figura 2.3 mostra um exemplo de DSL definida por um metamodelo.

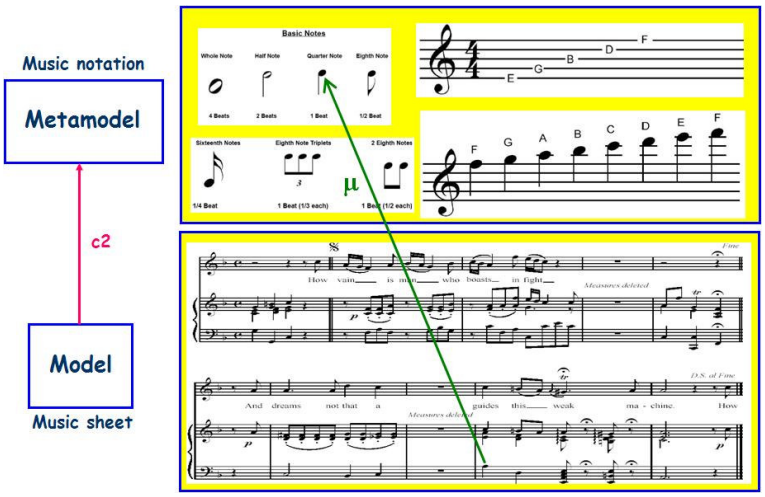

Figura 2.3. Exemplo de DSL – Linguagem de notação musical

Nesse exemplo, temos a notação musical representando o metamodelo, onde são definidos cada elemento, seus nomes, significados e representações gráficas. Os elementos definidos no metamodelo são utilizados na partitura musical, que representa o modelo neste caso. Podemos ver que existe uma relação clara entre modelo e metamodelo, e o conhecimento da teoria definida no metamodelo, permite que o músico possa ler a partitura e tocá-la no seu instrumento.

É importante ressaltar que, assim como os modelos, os metamodelos também podem ser descritos através de diagramas ou de linguagens. Na descrição de metamodelos, também desejamos utilizar uma linguagem bem-definida, para que possa ser interpretado por um computador. Para o escopo deste trabalho, utilizamos a linguagem *Kermeta*, que será explicada com maior detalhes no capítulo 3.

O metamodelo entra como um novo elemento para fazer parte da Figura 2.1, como mostrado na Figura 2.4 abaixo.

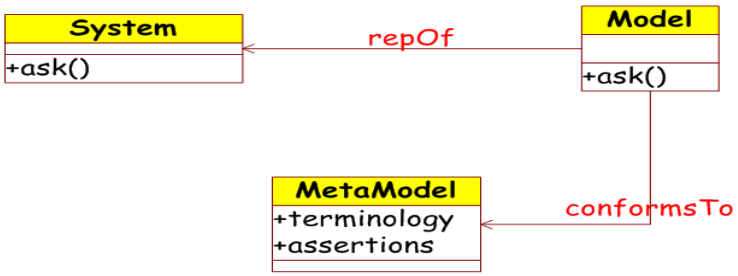

Figura 2.4. Relacionamento entre sistema, modelo e metamodelo

Resumindo tudo que foi dito sobre modelo e metamodelo, a Figura 2.4 nos mostra que o modelo é uma representação de um sistema e está de acordo com um metamodelo. Este último define as terminologias e regras necessárias para a construção de modelos semânticos.

Um metamodelo também precisa de uma linguagem para ser descrito. Para definir um metamodelo temos a figura do metametamodelo, sendo esse o último

grau de abstração em modelagem, já que ele serve como linguagem para se auto-descrever. A Figura 2.5 nos mostra um exemplo de modelo em UML, que possui uma relação com o seu metamodelo, e este último possui uma relação com o metametamodelo. Nesse caso, o metametamodelo está definido em MOF (*Meta-Object Facility*).

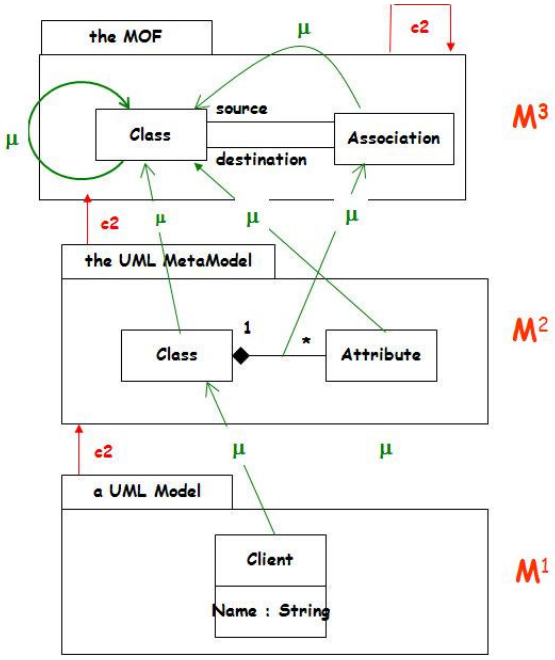

Figura 2.5. Exemplo de modelo, metamodelo e metametamodelo

No exemplo da Figura 2.5 vemos no modelo UML a classe Cliente que possui um atributo nome. Essa classe é mapeada na entidade Classe pertencente ao metamodelo de UML. O atributo de Cliente é representado através de uma composição entre uma Classe e um Atributo no metamodelo. Finalmente, as entidades Classe e Atributo do metamodelo são mapeadas na entidade Classe do metametamodelo, e a composição é representada através da entidade Associação. Esta última possui relacionamentos com a entidade Classe, que representa os dois lados da associação. Podemos também ver como o metametamodelo pode servir para descrever ele mesmo. Uma Associação também é uma Classe, e uma Classe também é uma Classe.

2.3 Transformação de Modelo

Até esse ponto foram discutidas noções estáticas sobre a representação de um sistema (modelo) e sobre a definição da semântica destas representações (metamodelo). Nenhuma informação foi dada sobre como estes modelos são alterados, sobre como eles evoluem. Tais alterações são as atividades que caracterizam o desenvolvimento do sistema dirigido pela atividade de modelagem.

Esta evolução é suportada por ferramentas de transformação de modelos, que tomam um modelo como origem e registram alterações sobre os seus elementos em um modelo de destino.

No interior de uma ferramenta de automação de transformações são encontradas *definições de transformações*, que descrevem como um modelo deve ser transformado. Percebe-se que existe uma diferença entre o processo de transformação ou construção de um modelo a partir de outro, e a definição da transformação. Para executar o processo de transformação, uma ferramenta utiliza a definição de transformação de interesse sobre o modelo de entrada [4].

Para a realizacao da transformacao, é necessário que exista um mapeamento de elementos da linguagem "fonte" para elementos da linguagem "destino". Este conjunto de mapeamentos é definido como o conjunto de *regras de transformação*.

Portanto, quando se fala em transformações é necessário ter os seguintes conceitos bem estabelecidos:

- Uma *transformacao* é a geração automática de um modelo, dado um modelo entrada, de acordo com a definição da transformação.
- Uma *definição de transformação* é um conjunto de regras de transformação, que juntas descrevem como um modelo na linguagem origem pode ser transformado em um modelo na linguagem destino.
- Uma *regra de transformacao* é um mapeamento entre um elemento da linguagem origem e outro elemento da linguagem destino.

Vale ressaltar que, apesar da definição de transformação poder mencionar modelos em linguagens diferentes, transformações podem também ser definidas em modelos na mesma linguagem. Como exemplos de transformações entre linguagens diferentes podemos citar transformações de PIM para PSM, PSM para PIM ou PSM para código. Tais transformações representam, em geral, refinamentos ou abstrações do modelo na linguagem fonte.

Exemplos comuns de transformações que utilizam uma mesma linguagem fonte e destino são refatoramentos de modelos. Tais refatoramentos, assim como refatoramentos de programas, tem o intuito de reestruturar o modelo, melhorando sua qualidade sem alterar sua semântica. A Figura 2.6 abaixo representa uma das transformações abordadas neste trabalho.

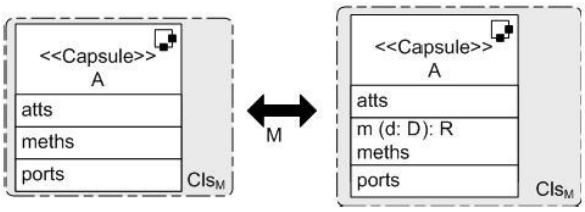

Figura 1.6. Transformação de Inserir Método.

A Figura 2.6 mostra um exemplo de como o modelo origem é transformado em um modelo destino, neste caso, com a inserção de um novo método na Cápsula **A**. Essa lei de transformação pode ser aplicada nos dois sentidos, assim como mostra a Figura, podendo ser entendida como uma inserção ou remoção de método, dependendo do referencial adotado. Para cada sentido de aplicação,

existem condições que devem ser satisfeitas para a realização da transformação. As transformações implementadas neste trabalho são explicadas detalhadamente no Capítulo 6 e no Apêndice.

2.4 Model Driven Architecture – MDA

Existem hoje vários métodos de desenvolvimento baseado em modelos, porém o que mais vem se destacando na academia e na indústria é o MDA (Arquitetura Dirigida a Modelos) [12]. Nessa arquitetura são definidos papéis para os modelos e transformações entre eles, de forma a melhorar a qualidade no desenvolvimento de *software*.

MDA se propõe a atingir alguns dos principais problemas do desenvolvimento de *software*, garantindo maior produtividade no processo, portabilidade e interoperabilidade do sistema final. É importante ressaltar que se trata de uma opção de desenvolvimento baseado em modelos, não sendo a única portanto.

Nas seções seguintes analisaremos como *softwares* vêm sendo desenvolvidos atualmente, e entraremos em maior detalhes nos principais problemas desse modelo de desenvolvimento tradicional. Será dada uma definição mais abrangente de MDA, mostrando seu ciclo de desenvolvimento, suas transformações e suas propostas para resolver os problemas acima citados.

2.4.1 Contexto atual do desenvolvimento de software

O progresso obtido no desenvolvimento de *software* não é tão fácil de se medir quando comparado ao progresso que o *hardware* vem passando nos últimos vinte anos. O avanço no desenvolvimento de *hardware* pode ser facilmente percebido quando utilizamos como métrica a velocidade dos processadores, que tem crescido exponencialmente nos últimos tempos [4]. Na análise do progresso do *software* não encontramos tal facilidade de medição, devido à escassez de parâmetros e/ou métricas, já que não podemos medir esse avanço levando em consideração apenas a velocidade ou custos de desenvolvimento.

No entanto, podemos ter uma pista de como o desenvolvimento de *software* tem evoluído, se observarmos quão complexos e grandes são os sistemas desenvolvidos atualmente. Um programa que não possui interface gráfica e não se comunica com outros sistemas talvez representasse um desafio para programadores há vinte anos. Hoje em dia, essa não é mais a realidade do *software*, já que, por mais simples que seja o sistema, normalmente ele é desenvolvido utilizando-se mais de uma tecnologia e com comunicação com outros sistemas.

Apesar de ser possível construir sistemas mais complexos, a indústria de *software* ainda se depara com uma série de problemas. Para melhor explicar como MDA atinge tais problemas, faremos uma breve análise de cada um deles.

2.4.2 Principais problemas do desenvolvimento de software

- **Produtividade**

Durante as etapas de elicitação de requisitos, análise e projeto do sistema, normalmente, são produzidos vários documentos tais como diagramas de casos de uso, diagramas de interação, diagramas de atividades, etc. Todos esses artefatos geram um grande pilha de documentos, que na maioria das vezes é deixada de lado à medida que a codificação evolui.

A conexão entre o código e os diagramas começa a desaparecer à medida que o código-fonte progride. Ao invés de serem uma especificação exata do código, os diagramas se tornam apenas figuras que não têm muita relação com o sistema que está sendo desenvolvido [4].

O abandono dos diagramas de especificação acontece porque normalmente, quando alguma mudança precisa ser feita, ela é realizada no código, por ser mais rápido e mais prático. É bem verdade que programadores não gostam de ter que editar documentos e diagramas apenas para manter a compatibilidade entre o que está modelado e o que está codificado.

Entretanto, essa atitude pode trazer sérias conseqüências à manutenção do sistema. À medida que o *software* evolui, novas pessoas são integradas à equipe, e estas pessoas precisam entender do problema, saber como funciona o código-fonte para resolver possíveis *bugs* ou implementar novas funcionalidades. É nesse momento que a negligência em relação aos documentos mostra seu resultado.

- **Portabilidade**

A indústria de *software* se diferencia das outras pela constante evolução e surgimento de novas tecnologias [4]. As empresas precisam seguir essas novas tecnologias porque não podem ficar para trás na concorrência, e também porque elas representam novas formas de resolver antigos problemas.

Com essa constante mudança de tecnologias, o *software* precisa ser portado para novas tecnologias ou para novas versões delas. Existe a opção de não migrar para o que há de mais novo, e ficar com o legado. Nesse caso, existirá a necessidade de interoperar o sistema com outros desenvolvidos com tecnologias mais novas.

- **Interoperabilidade**

Sistemas de *software* raramente vivem isolados [4]. A maioria deles precisa se comunicar com outros sistemas, principalmente aqueles que possuem uma interface de acesso via *web*, que precisam interagir com algum sistema de *back-end*.

Além da comunicação com outros sistemas, também é necessário que as diversas tecnologias existentes conversem entre si e possam interagir. Essa necessidade de comunicação pode ser traduzida em necessidade de interoperabilidade.

- **Manutenção e Documentação**

Documentação tem sido um problema no desenvolvimento de *software* há muito tempo. Escrever documentos normalmente é considerada uma tarefa chata e entediante pelos programadores, que acreditam estar perdendo seu tempo quando têm que escrever ou atualizá-los.

Os programadores estão enganados quando pensam que atualizar e escrever documentos não são tarefas deles. Pelo contrário, os responsáveis por estar construindo o sistema, precisam colaborar para manter a compatibilidade entre documentos e código-fonte.

Algumas linguagens de programação auxiliam nesse quesito por possuirem geração automática de documentação do código-fonte. Mas essa medida resolve apenas parte do problema, já que documentos que representam abstrações maiores também precisam ser constantemente atualizados.

2.4.3 Definição

A *Model Driven Architecture* (MDA) é um *framework* para desenvolvimento de *software* criado pela *Object Management Group* (OMG). Ela define uma abordagem que separa a especificação de uma funcionalidade do sistema da especificação da implementação desta funcionalidade em uma plataforma específica [8].

A abordagem e os padrões suportados pela MDA permitem que o mesmo modelo que especifica as funcionalidades do sistema possa ser implementado em múltiplas plataformas, através de padrões de mapeamento, e permite que aplicações diferentes possam ser integradas relacionando seus modelos explicitamente.

Na próxima sub-seção explicamos o ciclo de desenvolvimento da MDA e depois mostramos como ela se propõe a resolver os problemas mencionandos anteriormente.

2.4.4 Ciclo de desenvolvimento da MDA

O ciclo de desenvolvimento proposto por MDA não é muito diferente do modelo tradicional de desenvolvimento. As mesmas fases são identificadas, e uma das principais diferenças está nos artefatos gerados durante o processo, que podem ser compreendidos por computadores. A Figura 2.7 ilustra tal ciclo de desenvolvimento.

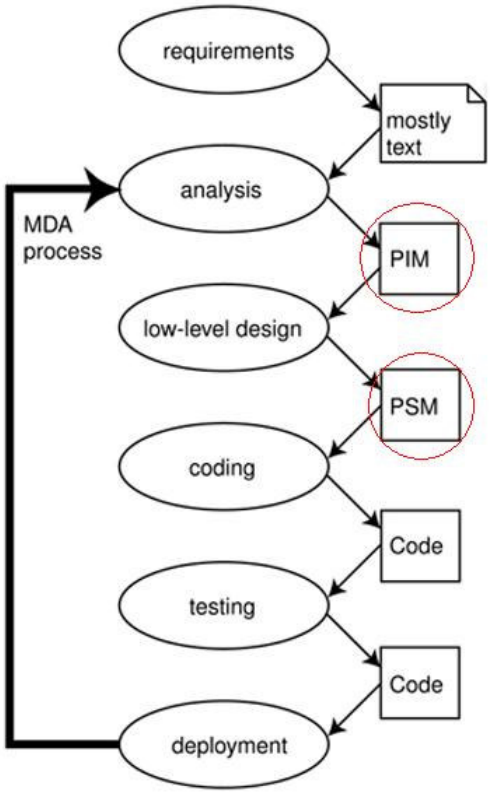

Figura 2.7. Ciclo de desenvolvimento MDA

Da análise da Figura 2.7, percebemos a presença de dois elementos desconhecidos no desenvolvimento de software tradicional: PIM e PSM.

O *Plataform Independent Model* (PIM) é um modelo com alto nível de abstração construído para ser independente de qualquer tecnologia de implementação. Durante a elaboração do PIM, a atenção é voltada para a construção de um modelo que melhor atende o negócio do sistema, permitindo abstração de detalhes técnicos.

É importante notar que o PIM é o artefato gerado pela etapa de análise, e é entrada para a etapa de projeto do sistema. Como resultado da etapa de projeto, temos o PSM.

O *Plataform Specific Model* (PSM) é um modelo que descreve o sistema em função de uma determinada tecnologia. Dessa forma, se estamos lidando com um PSM para um banco de dados relacional, vamos encontrar termos como tabela, coluna, chave estrangeira, dentre outros. Fica claro que para o entendimento do PSM é necessário o conhecimento prévio sobre a tecnologia que ele representa. Um PIM pode ser transformado em vários PSM's, já que normalmente se utiliza mais de uma tecnologia para implementar um sistema.

Finalmente, cada PSM é transformado em código-fonte, e como cada um deles está de acordo com uma determinada tecnologia, essa transformação é relativamente direta. Cada artefato gerado nesse ciclo representa um nível diferente de abstração, e permite que o programador possa focar sua atenção no PIM ou PSM e lidar mais facilmente com sistemas complexos.

2.4.5 Automação das transformações

Todas as transformações entre modelos propostas pela MDA são realizadas, teoricamente, por computadores. Com isso, o programador precisa apenas se preocupar em elaborar o PIM, que através de processos automáticos, dará origem a vários PSM's.

Porém, gerar código-fonte a partir de um modelo específico para uma plataforma (PSM) não é uma novidade. Temos muitas ferramentas na indústria que geram código-fonte, dado um modelo bem definido. A diferença é que a maioria dessas ferramentas produz código em formato de *template*, enquanto que na MDA a proposta é gerar o código-fonte completamente.

Então, a grande inovação proposta por MDA está na construção do PSM através do PIM. Como sabemos, MDA ainda é uma área em expansão, e conseqüentemente, não podemos encontrar ainda ferramentas que realizem totalmente esse objetivo.

A Figura 2.8 mostra claramente como as transformações são feitas em cada etapa da MDA, através da utilização de ferramentas de automação das transformações.

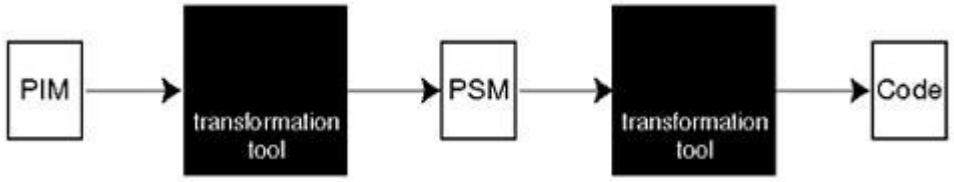

Figura 2.8. Automação das transformações

2.4.6 Vantagens de MDA

Agora que sabemos o que é a MDA, o que ela propõe e quais seus principais componentes, vamos avaliar como ela resolve, ou tenta resolver, os principais problemas de desenvolvimento de *software* mencionados na seção 2.4.2.

- **Produtividade**

  O ganho de produtividade é percebido pelo fato de o foco de atenção do programador deixar de ser detalhes técnicos de implementação e passar a ser o desenvolvimento do PIM. Porém, é importante ressaltar que o trabalho de implementar a transformação PIM-PSM deverá ser realizado pelo menos uma vez, no primeiro momento, podendo ser aproveitado em momentos futuros.

  Com a atenção voltada para a elaboração do PIM, o desenvolvedor se dedica a fazer com que o *software* melhor atenda ao négocio, e que seja feito mais rapidamente.

  Vale lembrar que, como o PIM será interpretado por um computador que produzirá outros PSM's, ele deve estar totalmente

consistente e deve refletir a realidade por completo, para que os PSMs gerados também estejam corretos.

- **Portabilidade**

A portabilidade deixa de ser um problema justamente porque o PIM é um modelo independente de plataforma. Como um PIM será mapeado em um PSM para cada plataforma, o primeiro passa a ser completamente portável.

Ainda assim, haverá a necessidade de implementação das transformações PIM-PSM para cada plataforma, mas como isso será algo muito recorrente, o próprio fornecedor de uma nova tecnologia será responsável por tal implementação.

- **Interoperabilidade**

Para prover interoperabilidade é necessário que "pontes" entre dois PSM's sejam criadas. Essas pontes possibilitam que as duas entidades possam trocar informações facilmente.

Teoricamente, se dois PSM's diferentes podem ser gerados do mesmo PIM, é possível saber como mapear um elemento do PIM em um elemento do primeiro PSM, e em um elemento do segundo PSM. Conseqüentemente, é possível criar a ponte entre os dois PSM's. MDA propõe que essa geração de pontes também seja automatizada por ferramentas.

A Figura 2.9 deixa claro como as pontes são construídas.

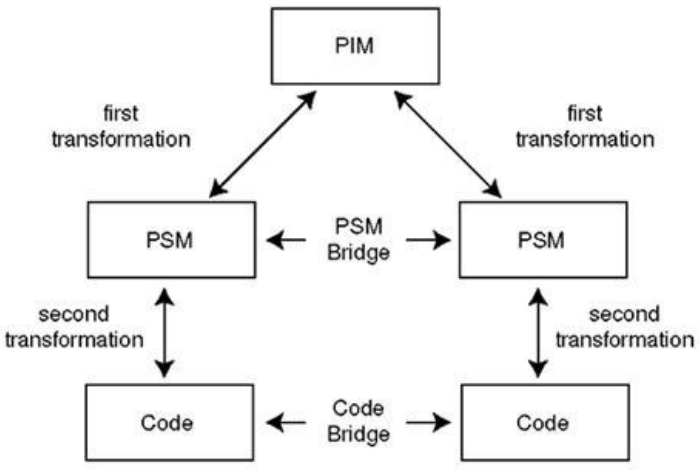

Figura 2.9. Pontes entre modelos

Vemos na Figura 2.9 que as pontes podem ser estabelecidas entre dois PSM's e entre dois códigos-fonte gerados por PSM's diferentes. Fica claro também que é possível estabelecer tais pontes dado que os dois PSM's vêm do mesmo PIM.

- **Manutenção e Documentação**

  O PIM representa um documento de alto nível de abstração para um *software*. O grande problema da documentação, que é manter compatibilidade entre código-fonte e documentos, será resolvido, já que, as alterações serão feitas no próprio PIM, e novos PSMs serão gerados.

  Ainda assim, não será possível abandonar por completo os documentos em papel. Algumas observações e decisões de projeto terão que ser mantidas em documentos para posterior consulta.

3. Linguagens de transformação de modelos

Neste capítulo serão apresentadas algumas linguagens de transformação de modelos, que se diferenciam principalmente em relação à forma de definição das regras de transformação e quanto ao suporte técnico provido ao desenvolvedor das transformações. Por suporte técnico, entendemos APIs e facilidades da linguagem de transformação que auxiliam na construção das regras.

As linguagens de transformação podem ser classificadas em: *Declarativas, Imperativas e Híbridas*. Inicialmente apresentamos duas linguagens imperativas de uso geral, *XSLT* e *JMI*, e depois apresentamos linguagens específicas para o desenvolvimento baseado em modelos, como a linguagem *Kermeta*, que foi a linguagem escolhida na implementação deste trabalho por prover um amplo ambiente de desenvolvimento de transformações e metamodelagem.

3.1 XSLT

XSLT (*XSL Transformations*) é uma linguagem de definição de transformações aplicáveis em um documento XML para produzir outro documento no mesmo formato. Foi criada para ser utilizada como parte de XSL, que é uma linguagem de definição de esquema para XML. Além de XSLT, XSL inclui um vocabulário de XML para especificar formatação [16].

Trata-se de uma linguagem declarativa que consiste em um conjunto de modelos de regras, onde cada regra é responsável por especificar o que deve ser adicionado no documento final, tendo como base um fragmento do documento de origem e um algoritmo [9]. Em uma visão mais prática, uma transformação expressa em XSLT descreve como aplicar essas regras para transformar uma árvore-destino em uma árvore-resultado.

O processo de transformação em XSLT é descrito pela Figura 3.1 abaixo.

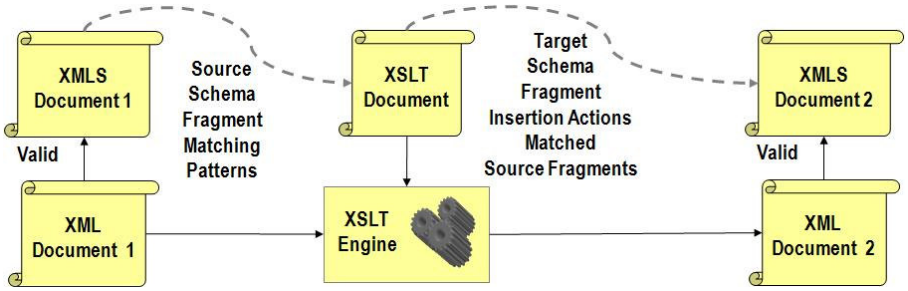

Figura 3.1. Processo de transformação em XSLT.

Vemos na Figura 3.1 que existe um motor (*engine*) que recebe um documento XML como entrada, realiza as transformações, e gera um outro documento XML como saída. O documento de entrada, deve estar válido de acordo com seu *schema*, para que o motor possa procurar fragmentos de texto que casem com os padrões definidos no documento XSLT. Após essa etapa, o motor criará como saída do processo o arquivo XML transformado, que estará, também, de acordo com o seu *schema*.

No início XSLT era muito utilizado, pois modelos são usualmente gravados em arquivo um formato XML padrão, chamado XMI. Infelizmente, por ser uma linguagem de caráter geral, as linguagens XSLT são bastante verborrágicas, o que dificulta o trabalho do desenvolvedor de transformações.

3.2 JMI

A especificação de JMI (*Java Metadata Interchange*) propõe a implementação de uma infra-estrutura independente de plataforma que gerencie a criação, armazenamento, acesso, busca e troca de metadados. Ela é baseada na especificação de MOF (*Meta Object Facility*) da OMG, que consiste em um padrão de gerenciamento de metadados reconhecido pela indústria. O padrão MOF consiste de um conjunto de artefatos básicos de modelagem descritos usando UML. Modelos de qualquer tipo de metadado (metamodelos) podem ser construídos a partir desses artefatos.

JMI define interfaces padronizadas para esses componentes de modelagem, permitindo assim, a busca e o acesso de metadados independentemente de plataforma. Permite também que metadados possam ser descobertos, consultados, acessados e manipulados tanto em tempo de *design* quanto em tempo de execução. Também provê a troca de metamodelos e metadados através de XML, utilizando a especificação de XMI (*XML Metadata Interchange*).

A API JMI se tornou um linguagem de grande apelo pois pode facilmente ser integrada com outros programas Java, e pode ser aprendida pela comunidade de programadores desta linguagem com a mesma facilidade. Infelizmente, ela possui complexidades inerentes às linguagens de programação, as quais o desenvolvimento baseado em modelos tenta resolver.

3.3 QVT

QVT (*Queries/Views/Transformations*) é um padrão de transformação de modelos definido pela OMG. Foi definido para acompanhar o crescente avanço nas pesquisas de MDA e para servir como padrão compatível com a sua suíte de recomendações, tais como UML, MOF, OCL, dentre outros.

A linguagem QVT também pode ser vista como uma DSL para transformações de modelos, que possibilita a construção e manutenção de transformações de forma simples, com um elevado poder de expressão e com estruturas avançadas [9].

Nesse contexto, entede-se *query (consulta)* como sendo um processo que recebe um modelo como entrada e seleciona elementos específicos desse modelo. As *views (visões)* são modelos derivados de outros modelos, e as *transformations (transformações)* recebem um modelo como entrada e o modificam ou criam um novo modelo a partir do primeiro [10].

É baseada em OCL e tem seu lado declarativo e seu lado imperativo. Em sua arquitetura, define três linguagens específicas de domínio (DSL) que são:

relations, core e operational mapping. A Figura 3.2 mostra detalhadamente essas linguagens e seus relacionamentos.

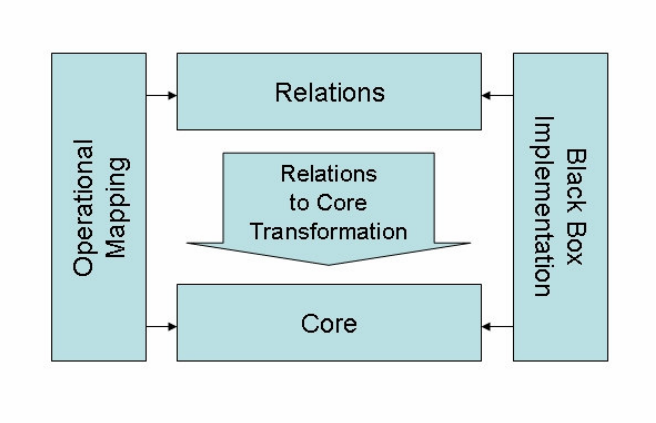

Figura 3.2. Linguagens definidas por QVT

As linguagens *Relations e Core* formam o lado declarativo de QVT, enquanto que a linguagem *Operational Mapping* representa o paradigma imperativo. A linguagem *Relations* serve para especificar os relacionamentos entre os elementos dos modelos, enquanto que a linguagem *Core* serve como ponto de referência para definir a semântica da linguagem de relacionamentos. Já a linguagem *Operational Mapping* tem a função de extender as linguagens declarativas com elementos tipicamente imperativos (laços, condições, etc).

Vale ressaltar que existe um mapeamento normativo entre as linguagens *Relations* e *Core*, já que elas representam níveis de abstração diferentes. Vemos também na Figura 3.2 um elemento chamado *BlackBox*. Esse elemento também é uma parte importante da especificação de QVT e traz facilidades na invocação de transformações escritas em outras linguagens, além de ser extremamente útil na integração com bibliotecas não-QVT.

Até onde o autor desta monografia sabe, nenhuma linguagem segue estritamente este padrão da OMG, porém várias linguagens, como algumas citadas adiante, aderem a este padrão de uma forma ou de outra acrescentando facilidades e melhorias a ele.

3.4 ATL

ATL (*Atlas Transformation Language*) é uma linguagem híbrida, já que possui construções imperativas e declarativas, feita para expressar transformações de modelos como requeridas pela abordagem de MDA. É descrita por uma sintaxe abstrata, um metamodelo MOF, uma sintaxe textual concreta e uma notação gráfica que permite que visões parciais de regras de transformação possam ser definidas.

Um programa de transformações em ATL é composto de regras que definem como elementos em um primeiro modelo são mapeados e navegados para criar e inicializar elementos em um segundo modelo. Ela está sendo utilizada por vários grupos de pesquisa em diferentes domínios e também no ensino.

Uma máquina virtual orientada à transformação de modelos foi definida e implementada para prover suporte de execução para ATL enquanto mantém um certo grau de flexibilidade. ATL torna-se executável simplesmente porque existe uma transformação do seu metamodelo para o *bytecode* da máquina virtual. Portanto, estender ATL é uma questão de especificar a semântica de execução de cada nova *feature* em termos de instruções simples.

Uma IDE foi desenvolvida para ATL por cima do Eclipse: *ATL Development Tools* (ADT). Ela usa EMF, *Eclipse Modeling Framework*, para lidar com modelos: serializá-los, desserializá-los, navegar e modificá-los. Um editor de código foi implementado com *syntax highlighting* e uma visão dos resultados do programa. Essa IDE também inclui uma extensão de ATL para o *framework* de depuração do Eclipse, permitindo assim que códigos que representam transformações também possam ser depurados [11].

Um pequeno exemplo de ATL é mostrado na Figura 3.3.

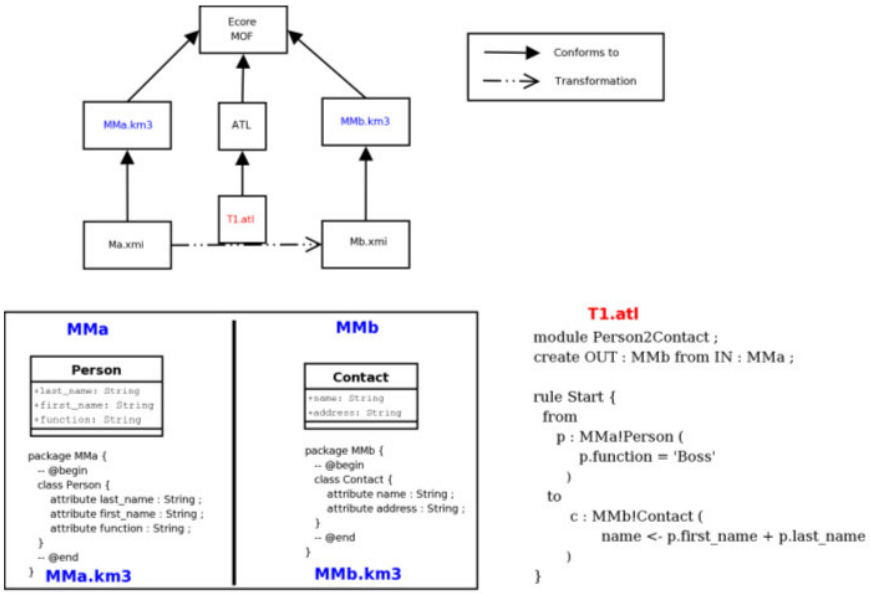

Figura 3.3. Exemplo de transformação escrita em ATL

Nesta Figura, são mostradas definições de duas entidades: Pessoa e Contato. A entidade Pessoa possui como atributos: primeiro nome, último nome e função. A entidade Contato possui como atributos: nome e endereço. No canto direito da Figura, é mostrado um pequeno programa de transformação (T1.atl). Nele, é definida uma regra da seguinte forma: Ao encontrar uma pessoa que tenha como função "Chefe", crie um Contato tendo como nome a concatenação dos seus primeiro e último nomes.

O diagrama exibido na parte de cima da Figura deixa claro que, os dois modelos criados estão de acordo com o metamodelo de **Ecore**, e são gerados arquivos **.xmi** que estão de acordo com esses modelos. Ainda se vê que existe uma transformação entre Ma.xmi e Mb.xmi.

Tendo sido construída na mesma época da definição do padrão QVT, a linguagem ATL possui várias similaridades em relação ao padrão, o que a torna simples e fácil de usar [19]. Infelizmente, tais facilidades só se tornam mais

evidentes em sua aplicação em transformações simples, onde exatamente um elemento do modelo de origem é mapeado em exatamente um elemento no modelo de destino. Isto se deve ao grande enfoque da linguagem em regras declarativas e pouca flexibilidade da linguagem em sua parte imperativa ou nos padrões de casamento destas regras.

3.5 Kermeta

Kermeta é uma linguagem de metamodelagem que permite a descrição da estrutura e do comportamento de modelos. Foi desenvolvida para ser compatível com a linguagem de metamodelagem da OMG, EMOF (parte da especificação de MOF 2.0) e com Ecore, do Eclipse.

Tem o objetivo de ser usada como a linguagem principal de uma plataforma orientada a modelos. Foi desenvolvida para ser uma base comum para implementação de linguagens de metadados, linguagens de ações, linguagens de restrições ou linguagens de transformações.

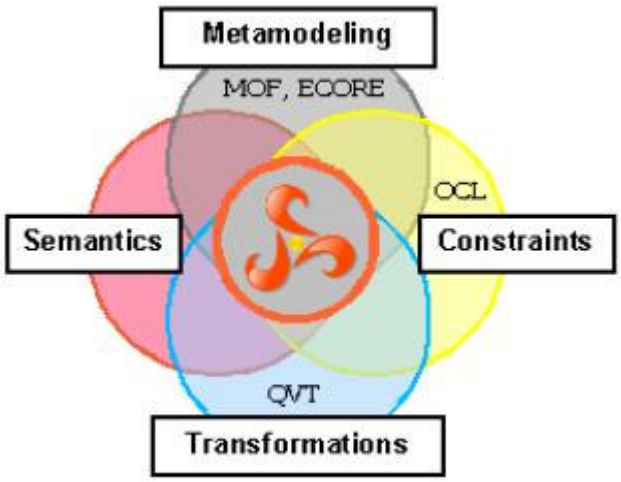

Figura 3.4. Posicionamento de Kermeta

Através da análise da Figura 3.4 vemos o posicionamento da linguagem Kermeta em relação a diversas variáveis. A Figura deixa claro que Kermeta pode ser utilizada para se trabalhar com MOF ou ECore como linguagens de metamodelagem, com OCL para definir restrições (*constraints*), com transformações de modelos e para definir a semântica de modelos.

Kermeta é uma linguagem orientada a modelo, imperativa, orientada a objetos e estaticamente tipada. Além disso, inclui também alguns conceitos típicos de orientação a modelos como associações, multiplicidades ou gerenciamento de conteúdo de objetos. Sua sintaxe foi inspirada na linguagem Eiffel e sua execução é feita por um interpretador.

Falaremos sobre as principais características da linguagem a partir desse ponto, mostrando exemplos e destacando diferenças entre Kermeta e outras linguagens, principalmente Java.

Classes, Abstração e Herança

Conceitos característicos de linguagens orientadas a objetos, como Java, estão presentes também em Kermeta, tais como: *classes*, *herança*, *exceções* e até *genericidade*. No exemplo da Figura 3.5 podemos observar esses conceitos sendo aplicados.

```
// persons who write documents
class Writer {
    attribute name : kermeta::standard::String
}

// generic concept for every document
abstract class Document {
    reference author : Writer
    attribute text : kermeta::standard::String
}

// a "Document" from the real world
class Book inherits Document {}

// a specialized "Book"
class ChildBook inherits Book {
    attribute minimalAge : kermeta::standard::Integer
}
```

Figura 3.5. Exemplo de classes e herança

Podemos perceber a definição de classes (*Writer, Document, Book e ChildBook*), sendo *Document* uma classe abstrata, *Book* herdando de *Document*, e *ChildBook* herdando de *Book*. Percebemos também que atributos de classes podem ser declarados usando a palavra *attribute* ou *reference*. *Attribute* deve ser utilizado quando existe uma composição entre duas entidades, ou quando existe a noção de contâiner. Já *reference*, deve ser utilizado quando se quer apenas estabelecer uma associação entre duas entidades.

Genericidade

Como foi dito anteriormente, *Kermeta* também permite a criação de classes e operações genéricas. Um pequeno exemplo é mostrado na Figura 3.6.

```
// This is a parametric class
class A<G> {
}

// This is the type variable binding : G is binded with Integer
var a : A<Integer>
a := A<Integer>.new
```

Figura 3.6. Exemplo de classe genérica

Vemos na Figura 3.6 que é possível criar definições de classes genéricas e instâncias delas para qualquer tipo. Neste caso, é criada uma classe genérica *A* e depois é declarada uma variável *a*, cujo tipo é *A<Integer>,* e depois a variável *a* é inicializada, recebendo uma instância da classe. Neste ponto podemos destacar uma diferença de *Kermeta* para Java, quando da criação de objetos. Ao contrário

de Java, *Kermeta* não possui a noção de construtores, sendo necessário utilizar a palavra reservada **new** para a criação do objeto. Portanto, para qualquer classe não-abstrata *X*, se queremos criar uma instância dessa classe, basta que façamos: X.**new**.

Operations x Methods

Os métodos Java são chamados de operações em Kermeta. Portanto, na definição de uma operação deve-se utilizar a palavra reservada **operation**. Deve-se utilizar a palavra **method** apenas quando for necessário implementar uma operação abstrata de uma classe abstrata. Neste caso, a declaração do método na classe que o implementa deve ser idêntica ao da classe original, trocando apenas a palavra operation pela palavra method.

Result

Para retornar valores em operações ou métodos em Kermeta, deve-se utilizar a variável **result**. Essa variável é "pré-definida", e nela deve ser guardado o valor que se deseja retornar através de uma simples atribuição.

Herança Múltipla

Em Kermeta é possível que uma classe herde de uma ou mais classes, conceito esse que não existe em Java. Porém, é sabido que a herança múltipla traz diversos problemas quando da sua utilização. Os principais conflitos relacionados a ela são:

- Construções com o mesmo nome podem ser herdadas de super-classes diferentes, causando um conflito de nomes.
- Diferentes implementações de uma mesma operação podem ser herdadas de super-classes diferentes.

Para esses conflitos existem duas soluções conhecidas na literatura. A primeira delas é ter um mecanismo de escolha implícito, que escolhe o método que será herdado de acordo com uma determinada política. A segunda consiste em incluir na linguagem construções que dêem o poder explícito de resolução de conflitos ao programador.

Na versão atual de Kermeta, existe um mecanismo mínimo de seleção que permite ao usuário escolher explicitamente o método herdado que deve ser sobreescrito, se mais de uma implementação para este método for herdada. Esse mecanismo, no entanto, não permite resolver alguns conflitos de nomes, rejeitando programas que tenham tais conflitos.

Tratamento de Exceções

O tratamento de exceções em Kermeta é muito similar ao de Java. O que muda são apenas as palavras reservadas utilizadas. Para lançar uma exceção utiliza-se **raise** ao contrário de **throw**, e para capturá-las utiliza-se um bloco chamado **rescue** ao invés de **catch**.

Em Kermeta, métodos que levantam exceções não precisam indicar esse fato explicitamente em suas assinaturas, como em Java. A Figura 3.7 mostra um pequeno bloco de código, onde é lançada uma exceção e outra é capturada.

```
do
    var excep : Exception

    excep := Exception.new
    stdio.writeln("Throwing an exception ! ")
    raise excep
end
var v1 : Integer init 2
var v2 : Integer init 3

do
    var v3 : Integer
    v3 := v1 + v2
rescue (myError : Exception)
    // something with myError
    // ...
end
```

Figura 3.7. Exemplo de tratamento de exceções.

Leitura e Escrita de Modelos

Kermeta possui construções que permitem que modelos EMF possam ser lidos, navegados, modificados e salvos em outro arquivo de recurso. Essa funcionalidade será melhor exemplificada no capítulo 5, onde será mostrado como foi feito o carregamento e a escrita de modelos no contexto desse trabalho.

Coleções

Kermeta define alguns tipos de coleções para lidar com conjuntos de valores. Os tipos de coleções disponíveis são resultado da combinação de duas restrições: **unique** (única) e **ordered** (ordenada). Uma coleção é dita *unique* quando não possui duplicatas, e é dita *ordered* quando a posição de um objeto pode ser modificada.

Na tabela 3.1 são mostrados os tipos de coleções disponíveis, uma breve descrição e seus comportamentos em relação as restrições.

Nome	**Restrições**	
	Unique	**Ordered**
set	Sim	Não
oset	Sim	Sim
seq	Não	Sim
bag	Não	Não

Tabela 3.1. Coleções disponíveis em Kermeta

Apesar de não possuir o conceito de *arrays*, *Kermeta* possui a noção de multiplicidade, que pode ser utilizada para definir coleções com um limitante inferior e um limitante superior. Sintaticamente, esses limitantes são definidos entre colchetes e separados por dois pontos, e podem ser um inteiro ou um asterisco, no caso do limitante superior, para indicar que não existe.

Comparação de Objetos

Assim como em Java, se o programador quiser comparar referências de objetos em Kermeta, ele deve utilizar o '==', e caso queira comparar o conteúdo de objetos, deve utilizar o método *equals*. Toda classe pode sobreescrever o método *equals* com a sua comparação própria, e para tipos primitivos basta utilizar o '=='.

Expressões Lambda e Funções

Kermeta possui uma implementação simples de expressões Lambda, que é útil, principalmente, para implementar funções no estilo OCL. Na Figura 3.8 ilustramos como deve ser a declaração e a chamada de uma função com expressão Lambda.

```
// With one parameter
operation FUNCTION_NAME ( LAMBDA_NAME : <TYPE->RETURN_TYPE> )
// With many parameters
operation FUNCTION_NAME ( LAMBDA_NAME : <[TYPE_1, TYPE_2, ...]->RETURN_TYPE> )

anInstance.FUNCTION_NAME ( TYPE | SOME_CODE )
anInstance.FUNCTION_NAME ( TYPE_1, TYPE_2 | SOME_CODE )
```

Figura 3.8. Exemplo de função e expressão lambda.

Funções e expressões lambda foram muito úteis na implementação deste trabalho, principalmente as funções já definidas na linguagem Kermeta que operam em coleções. Um exemplo do uso da função *each* é mostrado na Figura 3.9 abaixo.

```
aCollection.each { e |
    stdio.writeln("I am complex code!"
    stdio.writeln("Element : " + e.toString)
    var i : Integer init 5
    i := i + 132458
}
```

Figura 3.9. Exemplo do uso da função each

Da análise da Figura 3.9, vemos que é possível aplicar um pedaço de código para cada elemento de uma determinada coleção. Funções como *each, forAll, select, reject, collect, detect e exists* foram muito úteis para navegar nos modelos, durante a implementação das transformações desse trabalho.

Algumas linguagens de orientação a objetos, como C#, já incorporaram a noção de expressões lambda, e aumentaram muito o poder do programador, e a possibilidade de fazer código simples e conciso.

4. Metamodelo para UML-RT

Antes de apresentarmos as transformações de modelo implementadas neste trabalho, precisamos definir o metamodelo dos modelos utilizados nas transformações. Como dissemos anteriormente, as transformações estabelecem regras que mapeiam elementos de um modelo de entrada em elementos de um modelo de saída. Um pré-requisito para a definição destas regras é a definição semântica de quais são os elementos que compõem o modelo, e quais são suas relações.

O metamodelo representa, de maneira simplificada, o metamodelo da linguagem UML-RT, focando nos elementos comuns a outras linguagens de descrição arquitetural. Os elementos contidos no metamodelo representam as três principais visões arquiteturais de UML-RT: diagrama de classes, diagrama de estrutura e diagrama de estados. Para facilitar a compreensão, o metamodelo será dividido em três partes, que correspondem exatamente a cada uma das três principais visões de UML-RT. A consistência entre as partes do metamodelo é feitas por elementos que servem de "ponte", destacados a seguir em cada uma das partes.

4.1 Visão de Diagrama de Estados

O diagrama de estados serve para definir o comportamento interno de uma cápsula. Ele deve ser utilizado quando se quer representar o comportamento interno das cápsulas "folhas" bem como das que possuem sub-cápsulas, para especificar restrição de ordem nos sinais de um protocolo, ou ainda para especificar restrição na ordem de invocação dos métodos de uma classe [13].

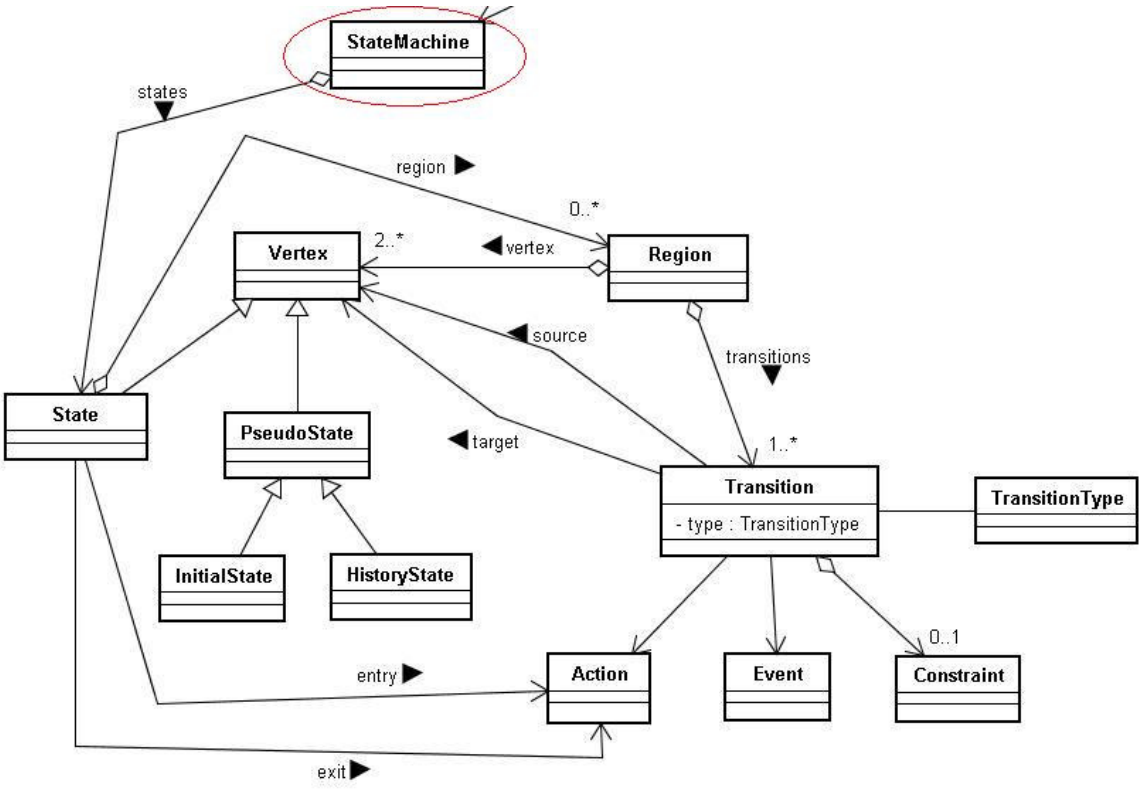

Figura 4.1. Metamodelo - Diagrama de Estados

A Figura 4.1 mostra como foi modelada a noção de diagramas de estados no metamodelo aqui proposto. A leitura começa da entidade *StateMachine* (inscrita em um círculo vermelho), que será a ponte de ligação com o diagrama de classes explicado adiante. Uma máquina de estados é composta de estados, fato esse ilustrado através do relacionamento de agregação *states,* entre as entidades *StateMachine* e *State*.

Por definição, um estado é uma condição de um objeto no qual este realiza alguma atividade ou espera por determinado evento [13]. Ele é composto por nome, ação de entrada e ação de saída. Não está sendo mostrado nessa Figura, mas todas as entidades que possuem nome herdam de uma entidade chamada *NamedEntity*. Já as ações de entrada e de saída estão representadas através de dois relacionamentos de *State* com *Action*. Ainda sobre *State* vemos que ele herda da entidade *Vertex*, que foi definida já que temos estados e pseudo-estados. Os pseudo-estados são representados pela entidade *PseudoState* que também é um *Vertex*. As entidades *InitialState e HistoryState* representam os pseudo-estados estado inicial e estado história.

Um estado pode estar contido em zero ou mais regiões (*Region*). Uma *Region* possui dois ou mais vértices (*Vertex*), e uma ou mais transições (*Transition*). Uma transição, por definição, é um relacionamento direcionado entre dois estados (origem e destino) [13]. Transições são compostas por: dois estados (origem e destino), representados pelos relacionamentos entre *Transition* e *Vertex*; evento de disparo, representado pelo relacionamento com a entidade *Event;* zero ou mais condições de guarda representadas pelo relacionamento com a entidade *Constraint*; uma ação, representada pela entidade *Action*. Uma transição também tem um tipo, que pode ser interna ou externa, e esse tipo está sendo representado pela entidade *TransitionType*.

4.2 Visão de Diagrama de Classes

O diagrama de classes serve para definir quais são as cápsulas, protocolos e classes do sistema e como cada uma dessas entidades se relaciona com as demais.

A Figura 4.2 mostra como foi modelada a noção de diagrama de classes no metamodelo proposto. A entidade inscrita em um círculo azul, *StateMachine*, representa a ponte dessa Figura com o diagrama de estados mostrado na Figura 4.1, enquanto que a entidade inscrita em um círculo vermelho, *CompositeCapsule*, representa a ponte dessa Figura com o diagrama de estrutura, que será explicado posteriormente.

A leitura desta Figura começa pela entidade *Model.* Essa entidade representa um modelo qualquer que será manipulado pelas transformações. Um modelo possui pacotes, sendo esse fato mostrado através do relacionamento entre *Model* e a entidade *Package*.

Neste diagrama, vemos as entidades *NamedEntity e NamedDefinitionEntity*, que representam, respectivamente, uma entidade qualquer que possua um nome como atributo, e uma entidade de definição que possui nome. Por entidade de definição, nesse caso, estamos falando de Protocolos, Classes e Cápsulas. Pra evitar poluição do diagrama, retiramos a associação de algumas entidades com *NamedEntity*, mas, as entidades que a generalizam são: *Port, Method, Package, Attribute* e *Signal*.

Uma das *NamedDefinitionEntity* é a entidade *Protocol* que representa um protocolo de comunicação entre cápsulas, e possui sinais de entrada e sinais de saída. Esses sinais são representados através de dois relacionamentos entre *Protocol e Signal* (*incomings e outcomings*). Outra entidade de definição é *Class*, que representa uma Classe que possui uma lista de atributos e de métodos. O conjunto de atributos é representado através do relacionamento entre *Class* e *Attribute,* enquanto que o conjunto de métodos é representado através do relacionamento entre *Class* e *Method*.

A última entidade de definição é a cápsula, representada pela entidade *CapsuleDefinition*. Essa por sua vez, possui portas, que são instâncias de protocolos. As portas são representadas pela entidade *Port*, que podem ser *EndPort e RelayPort*. Neste contexto, só mostramos *EndPort*, dado que *RelayPort* será explicada na próxima Figura. *EndPort* é uma porta que direciona os sinais que passam por ela diretamente para o *statechart* da cápsula.

Uma cápsula pode ser uma cápsula com comportamento (*BehaviouralCapsule)*, ou uma cápsula composta de outras cápsulas (*CompositeCapsule*). Uma cápsula com comportamento só possui *endports* e possui também uma máquina de estados, formando assim a ponte entre os dois diagramas mostrados até agora.

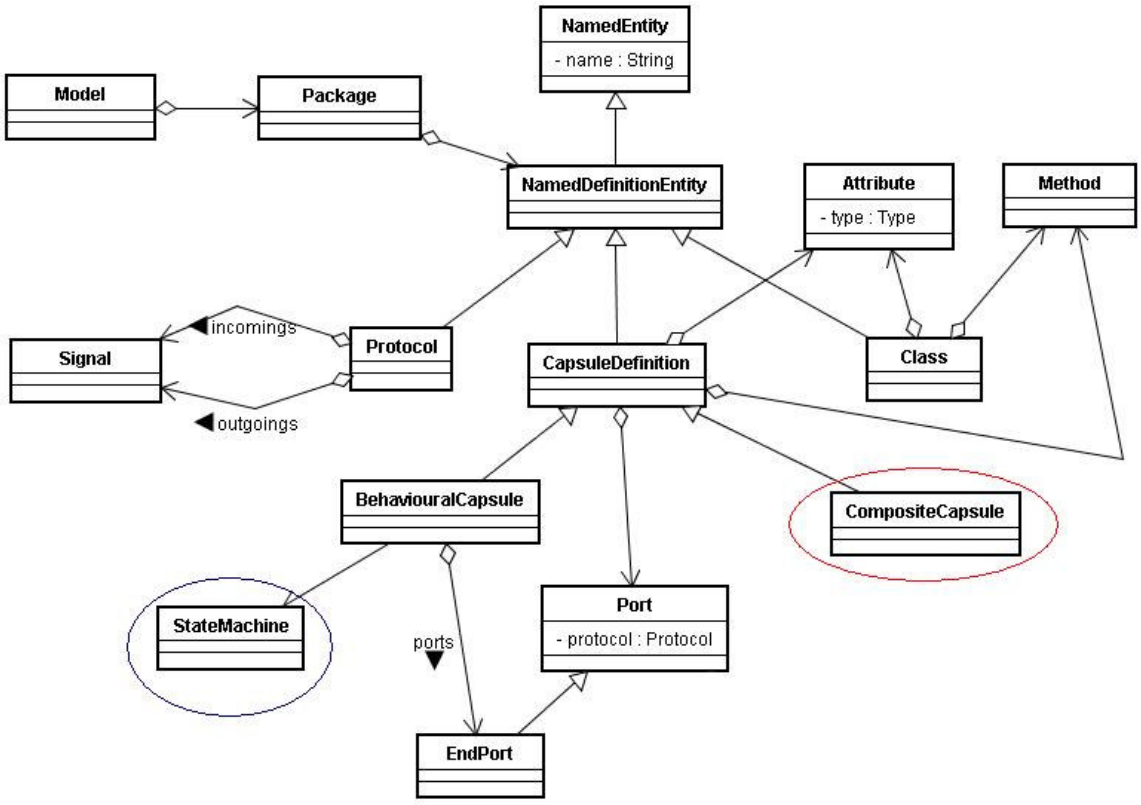

Figura 4.2. Metamodelo - Diagrama de Classes

4.3 Visão de Diagrama de Estrutura

O diagrama de estrutura serve para definir as conexões entre as instâncias das cápsulas, e deve ser utilizado para indicar quais portas de cada instância estão conectadas entre si. Ele pode também ser considerado um diagrama de colaboração. O diagrama de estrutura é composto de: instâncias de cápsulas, portas e conexões. A Figura 4.3 mostra como foi modelada a noção de diagrama de estrutura no metamodelo proposto.

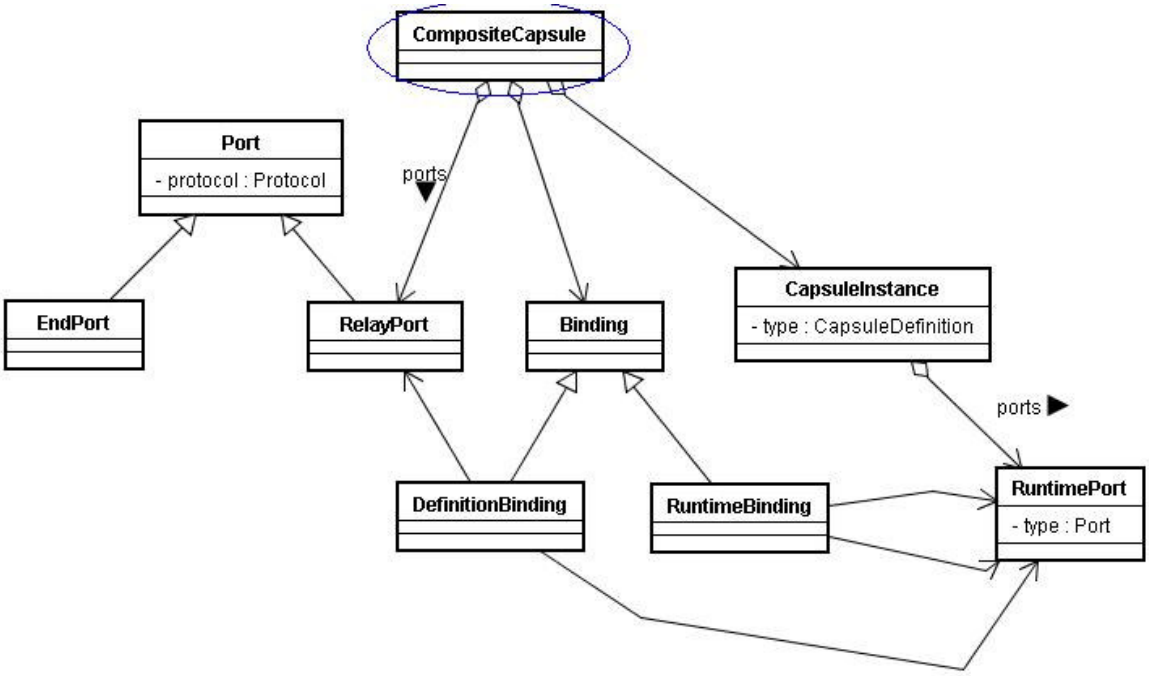

Figura 4.3. Metamodelo - Diagrama de Estrutura

A leitura deste diagrama começa pela entidade *CompositeCapsule*, que representa uma cápsula composta de outras cápsulas. Uma *CompositeCapsule* possui *RelayPort(s)*, que são portas que permitem a comunicação de cápsulas externas diretamente com as sub-cápsulas[13]. Uma cápsula composta também possui instâncias de cápsulas (*CapsuleInstance*), que são a valoração de uma cápsula, e conexões (*Bindings*), que representam o canal de comunicação por ondem passam as mensagens [13].

Essas conexões podem ser *DefinitionBinding* ou *RuntimeBinding*. Uma conexão é do tipo *DefinitionBinding* quando liga uma *RelayPort* a uma *RuntimePort*, e é dita *RuntimeBinding* quando liga duas *RuntimePort(s)*. Uma *RuntimePort* nada mais é do que a instância da definição de uma porta.

5. Transformações abordadas

Neste capítulo, mostraremos quais são as transformações abordadas por este trabalho e como foram implementadas utilizando a linguagem *Kermeta*. As definições detalhadas de todas as transformações, contendo descrição, condições de aplicação nos dois sentidos e os modelos de origem e destino, são encontradas no Apêndice deste trabalho. Escolhemos um sub-conjunto das leis de transformações de modelos UML-RT definidas em [14], que são listadas abaixo.

Declarar Cápsula
Introduzir Associação Cápsula-Cápsula
Introduzir Associação Cápsula-Classe
Introduzir Associação Cápsula-Protocolo
Introduzir Método
Introduzir Sinal de Saída

Tabela 5.1. Leis de transformação implementadas neste trabalho

Para cada lei de transformação existe uma ilustração que estabelece como se encontra o modelo antes, e como ficará o modelo final, após sua aplicação. As leis de transformações foram definidas sempre em dois sentidos, sendo sua aplicação feita da esquerda para direita ou da direita para a esquerda. Portanto, na prática, cada lei de transformação pode ser entendida como duas transformações, tendo em vista que as regras para aplicação são diferentes dependendo do sentido de sua utilização.

A condição requerida somente para aplicações da lei da esquerda para direita é antecedida pelo símbolo (→), que define claramente o seu sentido, e a condição necessária somente para aplicações da lei da direita para esquerda vem precedida pelo símbolo (←).

Para maior entendimento do que é uma transformação, analisemos a Figura 5.1 abaixo.

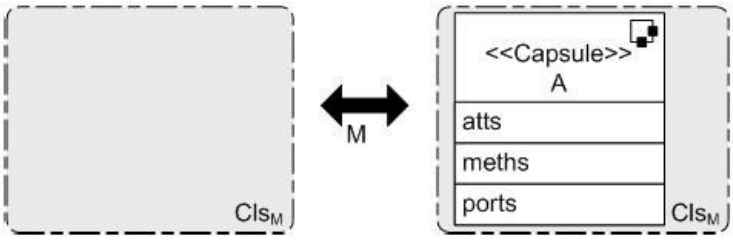

Figura 5.1. Declarar Cápsula

Vemos na Figura 5.1 que a transformação pode ser facilmente entendida apenas através da análise dos modelos, antes e depois da transformação ser aplicada. Neste caso, a transformação em questão é **Declarar Cápsula**, e vemos do lado esquerdo da Figura um modelo inicial sem nenhum elemento; já do lado direito, vemos um modelo contendo uma cápsula com o nome **A** que possui um conjunto de atributos, métodos e portas, representados, respectivamente, por **atts, meths** e **ports**.

As condições de aplicação dessa transformação são:

(→) **CIsm** não possui a declaração de nenhum elemento, no mesmo pacote, chamado **A**.

(←) Nenhuma cápsula em **M** tem uma relação com a cápsula **A** em qualquer diagrama.

De uma forma geral, leis relacionadas à declaração de elementos, usualmente, possuem como condição para a introdução de um elemento (aplicação da esquerda para direita da lei) que este não possua o mesmo nome de outro elemento já utilizado no modelo, e para a remoção de um elemento (aplicação da direita para esquerda da lei) que o elemento não seja utilizado pelo restante do modelo [14]. A título de esclarecimento, **CIsm** representa o diagrama de classes do modelo e **Strm** representa o diagrama de estrutura de uma cápsula. Essas visões são usadas para melhor entendimento do funcionamento de cada transformação.

Apesar de sua simplicidade, esta lei é bastante importante, já que uma cápsula só pode fazer parte da arquitetura do sistema modelado, caso ela tenha sido declarada previamente.

A implementação das transformações neste trabalho possui uma limitação quando da verificação de algumas regras que necessitariam de uma linguagem de ações. Tais ações representam códigos de uma linguagem de programação que são inseridos no modelo para complementar a descrição do comportamento de algumas partes do sistema. Apesar de existirem alguns diagramas comportamentais que poderiam substituir tais trechos de código, o uso de linguagens de ações ainda é comum devido à sua expressividade. A dificuldade na verificação em uma linguagem de ações vem da dificuldade de criar-se um metamodelo para tal linguagem. A lei de transformação **Introduzir Método**, mostrada na Figura 5.2, é um exemplo desse caso.

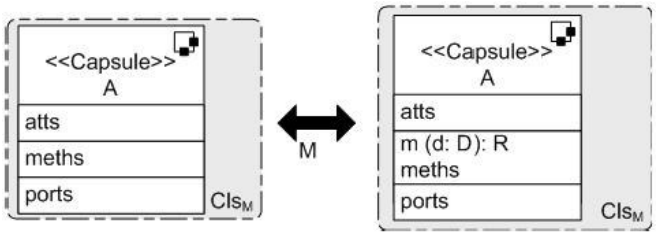

Figura 5.2. Introduzir Método

Condições:

(→) Não existe um método chamado **m** em **A**.

(←) O método **m** não é utilizado por outro método em **meths**, nem tampouco no diagrama de estados de **A**, ou em um predicado de **A**.

A condição da direita para a esquerda (←) deixa claro que precisaríamos de uma forma de verificar se um método de uma cápsula utiliza o método **m** em questão. No entanto, não está no escopo deste trabalho realizar tal verificação, já que para isso, como foi dito anteriormente, precisaríamos de uma linguagem de ação para definirmos o comportamento dos métodos em **meths**. Uma possível implementação deveria varrer a estrutura das ações de todos os elementos no modelo em busca de referencias à **m**.

5.1 Implementação

Nesta seção entraremos em maiores detalhes técnicos, falando das tecnologias utilizadas para a implementação, do mapeamento do metamodelo para linguagem *Kermeta*, do padrão de projeto utilizado e da leitura e escrita dos modelos.

Eclipse Modeling Framework (EMF)

Para a construção do metamodelo, foi necessário representá-lo em uma linguagem de metamodelagem. Para tal, foi utilizado o EMF (*Eclipse Modeling Framework)*, um framework de modelagem e geração de código do Eclipse para a construção de ferramentas e aplicações baseadas em modelos de dados estruturados, como XML [15].

O EMF consiste de três partes principais:

- **Core**: Inclui um metamodelo (Ecore) para a descrição de modelos, suporte em tempo de execução, incluindo notificações de mudanças, suporte a persistência com serialização XMI, e uma API para manipular objetos EMF genericamente.
- **Edit**: Inclui classes genéricas e reusáveis para a construção de editores para modelos definidos em EMF.
- **CodeGen**: Realiza a geração de código necessária para a construção dos editores gráficos para modelos EMF.

Para este trabalho, a única parte do EMF utilizada foi o **Core**.

Através do EMF, o Eclipse permite que possamos criar o metamodelo aqui proposto em formato de arquivo **.ecore**, que por sua vez pode ser mapeado em arquivo no formato padrão XMI. A ferramenta que permite a elaboração do **.ecore** permite que representemos o grafo do metamodelo de maneira estruturada, como ilustrado na Figura 5.3.

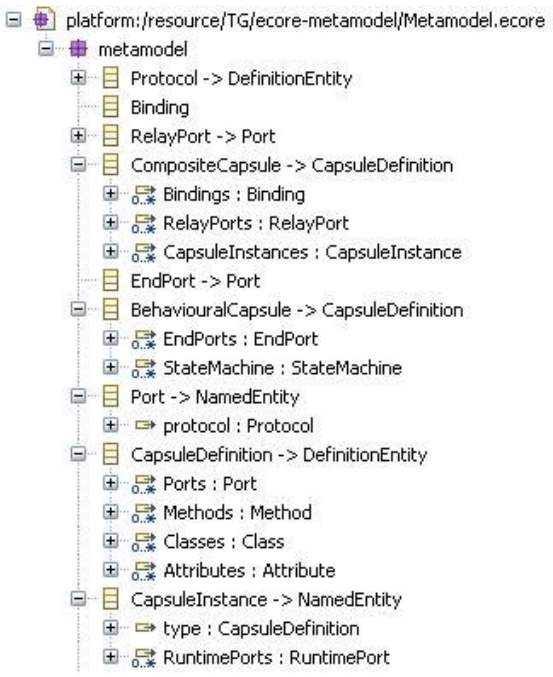

Figura 5.3. Metamodelo em formato .ecore

Vemos na Figura 5.3 que todo o modelo (nesse caso, um metamodelo) pode ser representado fielmente, incluindo seus relacionamentos, heranças e multiplicidades, no arquivo **.ecore**.

Metamodelo Kermeta

Após essa etapa, as entidades do metamodelo definido no **.ecore** podem então ser mapeadas na linguagem *Kermeta* para serem utilizadas diretamente nas transformações. Por conveniência, o ambiente de desenvolvimento *Kermeta* do *Eclipse* permite que metamodelos *Kermeta* possam ser gerados automaticamente a partir dos arquivos **.ecore**. Nesse caso, cada entidade definida no modelo será mapeada em uma classe *Kermeta* com mesmo nome, atributos e referências, como mostramos na Figura 5.4.

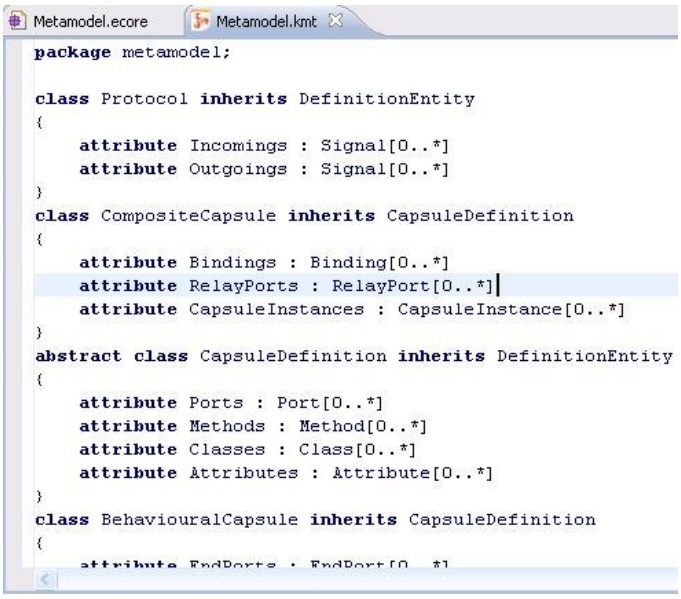

Figura 5.4. Metamodelo em Kermeta

Vemos, na Figura 5.4, que as entidades definidas no **.ecore** foram mapeadas identicamente em *Kermeta*. Como exemplo, temos a entidade abstrata *CapsuleDefinition* com seu conjunto de atributos, portas, métodos e classes, declarados através de atributos da classe abstrata. Percebe-se aqui também a presença da sintaxe de *Kermeta* para definição de cardinalidade. A grande vantagem do uso de *Kermeta* está na homogeneidade da linguagem de metamodelagem e das transformações, facilitando assim o desenvolvimento de linguagens de transformação de modelos.

Implementação das transformações

Tendo sido definido o metamodelo em *Kermeta*, partimos para a implementação propriamente dita das transformações. A implementação foi feita utilizando-se o padrão de projeto *Command*. Na programação orientada a objetos, o padrão *Command* é utilizado para encapsular ações ou comandos em objetos, utilizando uma interface padronizada. Dessa forma, diferentes tipos de requisições podem ser executados utilizando-se uma mesma interface, criando um alto nível de abstração.

Em nosso trabalho definimos uma classe abstrata chamada *Command* que representa uma abstração sobre o que todas as transformações devem conter e quais métodos devem implementar. A Figura 5.5 ilustra, através de um modelo UML, como foi estruturada a implementação com o padrão *Command*.

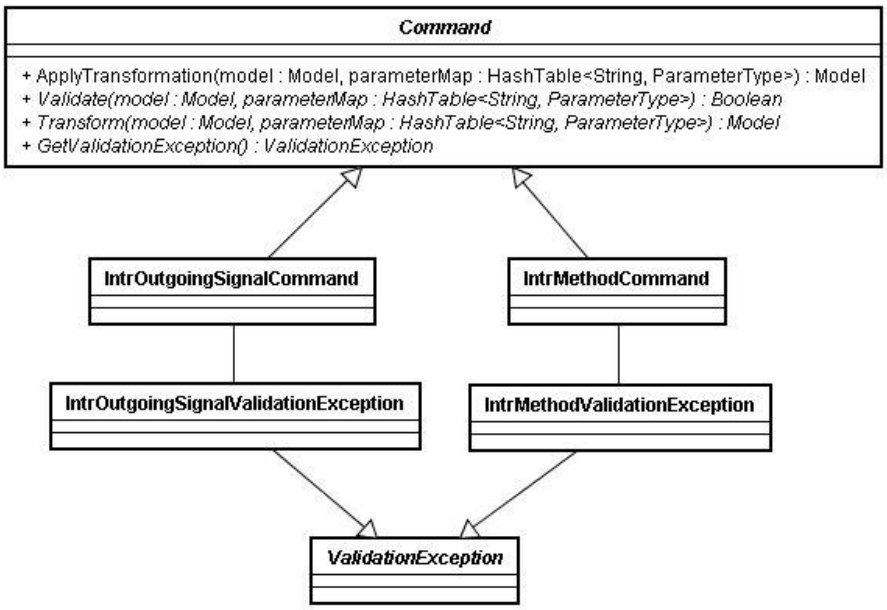

Figura 5.5. Transformações utilizando Command

Como foi dito anteriormente, vemos na Figura 5.5, que a classe *Command* é uma classe abstrata e contém os seguintes métodos: *ApplyTransformation, Validate, Transform, GetValidationException*, onde o primeiro método utiliza os outros três métodos, que são abstratos, para realizar a transformação. Dessa forma, alcançamos um nível de abstração tal que, torna-se transparente para quem for usar a API, como cada transformação implementa seus métodos.

Ainda analisando a Figura, percebemos que para cada transformação implementada são criadas duas classes que herdam da classe *Command* e implementam seus métodos abstratos com sua lógica. Por uma questão de simplicidade, e para não poluir a Figura, só é exibida uma classe para cada transformação, mas, na prática, são criadas classes para os dois sentidos da aplicação da transformação.

Também percebemos a presença da classe abstrata *ValidationException*, que representa uma exceção de validação na regra da transformação. Para cada transformação implementada também deve ser criada uma classe que representa uma exceção levantada quando da sua aplicação. Essa classe herda de *ValidationException*, tem seu nome definido como sendo o nome da transformação concatenado com a palavra *ValidationException*, e implementa o método *GetExceptionMessage* com sua mensagem específica.

Ainda sobre os métodos presentes em todos os *Command(s)*, o método *Validate* é responsável por fazer, de fato, a validação da regra. Já o método *Transform* é responsável por efetuar as mudanças no modelo, necessárias para a correta aplicação da transformação. O método *GetValidationException* é responsável por retornar uma instância de uma classe que herde de *ValidationException*. Finalmente, o método *ApplyTransformation* se utiliza dos três métodos explicados anteriormente, para validar a transformação, realizá-la se ela for de fato válida, ou levantar uma exceção caso contrário.

A Figura 5.6 abaixo mostra um pequeno exemplo de implementação. Nesse caso, trata-se da transformação **Introduzir Associação Cápsula-Cápsula**. Destacamos na Figura a herança utilizada, e a assinatura dos métodos *Validate e Transform*.

```
class IntrCapsuleCapsuleAssociationCommandLTR inherits Command
{
    method Validate(model : Model, parameterMap : Hashtable<String, ParameterType>) : Boolean is do
        var capsuleA : CapsuleDefinition
        var capsuleB : CapsuleDefinition
        var associationName : String
        var isValid : Boolean init false
        var composite : CompositeCapsule

        capsuleA ?= parameterMap.getValue("capsuleA")
        capsuleB ?= parameterMap.getValue("capsuleB")
        associationName := parameterMap.getValue("associationName").stringPrimitive

        if CompositeCapsule.isInstance(capsuleA) then
            composite ?= capsuleA
            isValid :=
                not (composite.CapsuleInstances.exists { cRef | cRef.name.equals(associationName) })
        end

        result := isValid
    end

    method Transform(model : Model, parameterMap : Hashtable<String, ParameterType>) : Model is do
        var composite : CompositeCapsule
        var componentRef : CapsuleInstance
        var capsuleA : CapsuleDefinition
        var capsuleB : CapsuleDefinition

        capsuleA ?= parameterMap.getValue("capsuleA")
        capsuleB ?= parameterMap.getValue("capsuleB")

        composite ?= capsuleA
        componentRef := CapsuleInstance.new
        componentRef.type := capsuleB
```

Figura 5.6. Exemplo de implementação.

Criação dos Modelos

A maioria das ferramentas gráficas de edição de modelos UML, mapeia o modelo definido pelo usuário em um arquivo XMI com o extensão **.xmi**. Um grande problema nestas ferramentas é que, além de mapear informações do modelo, elas estendem a estrutura XML destes arquivos para conter detalhes relativos à parte gráfica da aplicação, como posição dos elementos na tela, dentre outros. A adição de detalhes específicos da ferramenta em tais arquivos faz com que não exista uma padronização rígida em seu formato, fazendo com que cada fabricante possua um formato proprietário.

A criação de uma ferramenta gráfica de edição de modelos, e o posterior mapeamento dos elementos deste modelo em um arquivo **.xmi** padrão foge ao escopo deste trabalho, não deixando de ser uma boa oportunidade de trabalho futuro. Sendo assim, para testar e validar o comportamento de cada transformação, criamos scripts responsáveis pela criação de arquivos **.xmi** que representem modelos de entrada para cada uma delas.

Para criar arquivos **.xmi** através da API de *Kermeta*, basta que o usuário instancie um modelo, modifique-o e depois salve-o no arquivo. A Figura 5.7 mostra um pequeno trecho de código que exemplifica essa situação.

```
class DeclareCapsuleModel
{
    operation main() : Void is do
        var repository : EMFRepository init EMFRepository.new
        var resource : EMFResource
        resource ?= repository.createResource("../../input-models/Model-IN.xmi",
            "../../ecore-metamodel/Metamodel.ecore")

        var model : Model init Model.new
        //Criando o primeiro pacote: 'Pacote 1'
        var ~package1 : Package init Package.new
        ~package1.name := "Pacote 1"
        model.Packages.add(~package1)

        //Criando o segundo pacote: 'Pacote 2'
        var ~package2 : Package init Package.new
        ~package2.name := "Pacote 2"
        model.Packages.add(~package2)

        resource.instances.add(model)
        resource.save()
```

Figura 5.7. Exemplo de criação de modelo em Kermeta.

Vemos na Figura 5.7 três pedaços de código destacados. O primeiro pedaço em vermelho se refere a criação do recurso (nesse caso o arquivo **.xmi**) através do método **createResource** da classe *EMFRepository*. Este método recebe duas strings como parâmetros, onde a primeira representa o caminho e o nome do arquivo a ser gerado, e a segunda representa o caminho para encontrar o arquivo .ecore que representa o metamodelo.

O bloco de código contido no marcador azul representa a criação do modelo e de seus elementos. Finalmente, o segundo pedaço de código em vermelho adiciona o modelo na coleção de instâncias do recurso e ordena que ele seja salvo. Após a execução desse código, será gerado o arquivo "Model-IN.xmi" no caminho especificado. A Figura 5.8 ilustra um arquivo **.xmi** gerado para um modelo que contém duas cápsulas: *Produtor* e *Consumidor*.

```
<?xml version="1.0" encoding="ASCII"?>
<metamodel:Model xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:metamodel="platform:/resource/TG/ecore-metamodel/Metamodel.ecore">
  <Packages name="Pacote 1">
    <Entities xsi:type="metamodel:BehaviouralCapsule" name="Produtor"/>
  </Packages>
  <Packages name="Pacote 2">
    <Entities xsi:type="metamodel:BehaviouralCapsule" name="Consumidor"/>
  </Packages>
</metamodel:Model>
```

Figura 5.8. Arquivo .xmi representando um modelo

Aplicação das transformações

Após ter sido montado o modelo de entrada em um arquivo **.xmi**, a transformação pode ser aplicada recebendo-o como entrada. O trecho de código

mostrado na Figura 5.9 mostra como *Kermeta* permite que carreguemos o arquivo .xmi, e como a transformação é invocada.

```
resource ?= repository.createResource("../../input-models/Model-IN.xmi",
    "../../ecore-metamodel/Metamodel.ecore")
resource.load

model ?= resource.instances.one
var ~package : Package
    init model.Packages.detect { pack | pack.name.equals("Pacote 1") }
var capsule : BehaviouralCapsule init BehaviouralCapsule.new
capsule.name := "Nova Capsula"
var declareCapsuleLTR : DeclareCapsuleCommandLTR
    init DeclareCapsuleCommandLTR.new
var parameterMap : Hashtable<String, ParameterType>
    init Hashtable<String, ParameterType>.new

parameterMap.put("capsule", capsule)
parameterMap.put("package", ~package)

do
    model := declareCapsuleLTR.ApplyTransformation(model, parameterMap)
    resource.saveWithNewURI("../../output-models/Model-OUT.xmi")
rescue (validationException : DeclareCapsuleValidationException)
    stdio.writeln(validationException.GetExceptionMessage())
end
```

Figura 5.9. Carregando o arquivo e aplicando a transformação

Vemos na primeira parte destaca em vermelho que, assim como é possível salvar uma instância de recurso, também é possível carregar o modelo através do recurso (arquivo **.xmi**). A segunda parte destacada mostra como podemos invocar a transformação através do método *ApplyTransformation*, passando como parâmetros o modelo, e um mapeamento com o nome do parâmetro e seu valor. Em seguida, invocamos o método *saveWithNewURI* do recurso passando qual o caminho onde deve ser gerado o arquivo de saída.

Finalmente, podemos ver o resultado da aplicação da transformação quando analisamos o arquivo gerado. Esse arquivo é mostrado na Figura 5.10 abaixo. Perceba que existe um novo elemento em relação ao arquivo mostrado na Figura 5.8, que é justamente a nova cápsula declarada.

```
<?xml version="1.0" encoding="ASCII"?>
<metamodel:Model xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:metamodel="platform:/resource/TG/ecore-metamodel/Metamodel.ecore">
  <Packages name="Pacote 1">
    <Entities xsi:type="metamodel:BehaviouralCapsule" name="Produtor"/>
    <Entities xsi:type="metamodel:BehaviouralCapsule" name="Nova Capsula"/>
  </Packages>
  <Packages name="Pacote 2">
    <Entities xsi:type="metamodel:BehaviouralCapsule" name="Consumidor"/>
  </Packages>
</metamodel:Model>
```

Figura 5.10. Arquivo .xmi após a aplicação da transformação

Neste capítulo mostramos a etapa de implementação das transformações detalhando seus vários passos. Primeiramente, mostramos a definição de uma transformação e suas condições de aplicação nos dois sentidos. Em seguida, falamos da principal limitação na verificação dessas condições utilizando o exemplo da transformação **Inserir Método**.

Após isso, falamos do framework EMF do Eclipse, utilizado para criar uma representação mais concreta do metamodelo através de um arquivo **.ecore**. Mostramos também que é possível gerar automaticamente, através do Eclipse, o código *Kermeta* que representa o metamodelo definido no **.ecore**. Após isso, mostramos como foi utilizado o padrão *Command* na implementação das transformações e como esse padrão deu, para elas, maior poder de abstração.

Em seguida, mostramos como foram criados os modelos, representados pelos arquivos **.xmi**, e ilustramos como uma transformação é aplicada, exibindo seus modelos de entrada e de saída.

6. Exemplos

Neste capítulo, mostraremos como as transformações implementadas neste trabalho podem ser, de fato, aplicadas em situações do cotidiano. Para isso, utilizaremos um exemplo já conhecido na literatura, o exemplo do *Produtor* e *Consumidor*. Neste exemplo, temos essas duas entidades, onde a primeira produz determinado artefato, e a segunda o consome, e como se tratam de cápsulas, é preciso haver um protocolo de comunicação entre elas.

Através deste exemplo, poderemos validar o comportamento da maioria das transformações propostas. A construção desse exemplo se dará paulatinamente, através da aplicação sucessiva das transformações. Para cada passo, serão mostrados o modelo UML e o arquivo **.xmi**, antes e depois da aplicação da transformação, para que possamos comparar o seu efeito no modelo através dessas duas visões.

Apesar de utilizar a abordagem de aplicação sucessiva de transformações para partir de um modelo inicial vazio e obter um modelo final com as entidades desejadas, é provado em [14] que todas essas transformações mantém o comportamento das entidades envolvidas. O autor resolveu utilizar essa abordagem apenas por uma questão de didática, para ilustrar melhor o que foi feito, tendo em vista que o conjunto de transformações escolhido foca, principalmente, em mudanças na declaração de elementos de projeto de UML-RT.

É importante lembrar que a execução das transformações se dá através da leitura de um arquivo **.xmi,** que representa o modelo de entrada, e posterior geração de outro arquivo no mesmo formato, que representa o modelo resultante. Portanto, os modelos aqui mostrados, não foram lidos e nem gerados pelas transformações, são apenas uma forma gráfica de mostrar o comportamento de cada uma delas.

Partindo de um modelo vazio, isto é, sem nenhum elemento, utilizamos a transformação **Declarar Cápsula** para declarar a cápsula **Produtor**. Utilizaremos o padrão, daqui em diante, de mostrar ao lado esquerdo da Figura o modelo inicial, e ao lado direito, o modelo resultante. Para os arquivos **.xmi**, em cima estará o arquivo inicial, representado pelo número 1, e em baixo estará o arquivo de saída, representado pelo número 2.

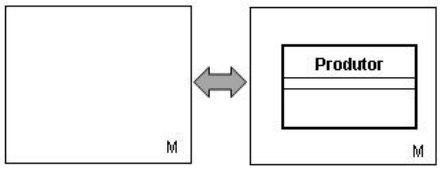

Figura 6.1. Visão do modelo - Declarar Cápsula Produtor

```
<?xml version="1.0" encoding="ASCII"?>                                    1
<metamodel:Model xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
    xmlns:metamodel="platform:/resource/TG/ecore-metamodel/Metamodel.ecore">
  <Packages name="Pacote 1"/>
</metamodel:Model>
```

```
<?xml version="1.0" encoding="ASCII"?>                                    2
<metamodel:Model xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:metamodel="platform:/resource/TG/ecore-metamodel/Metamodel.ecore">
  <Packages name="Pacote 1">
    <Entities xsi:type="metamodel:BehaviouralCapsule" name="Produtor"/>
  </Packages>
</metamodel:Model>
```

Figura 6.2. Visão do .xmi - Declarar Cápsula Produtor

Vemos na Figura 6.2 que a diferença entre os arquivos de entrada e de saída, a inserção da cápsula **Produtor**, está destacada em um retângulo vermelho.

Após ter declarado a cápsula **Produtor**, partimos para a declaração da cápsula **Consumidor**. Essa etapa utiliza a mesma transformação aplicada no passo anterior.

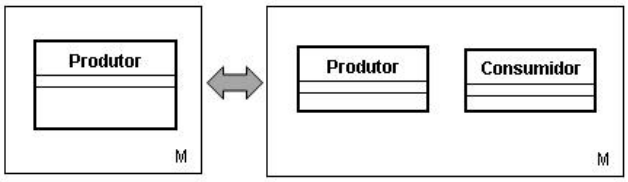

Figura 6.3. Visão do modelo - Declarar Cápsula Consumidor

```
<?xml version="1.0" encoding="ASCII"?>
<metamodel:Model xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"    1
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:metamodel="platform:/resource/TG/ecore-metamodel/Metamodel.ecore">
  <Packages name="Pacote 1">
    <Entities xsi:type="metamodel:BehaviouralCapsule" name="Produtor"/>
  </Packages>
</metamodel:Model>
```

```
<?xml version="1.0" encoding="ASCII"?>                                    2
<metamodel:Model xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:metamodel="platform:/resource/TG/ecore-metamodel/Metamodel.ecore">
  <Packages name="Pacote 1">
    <Entities xsi:type="metamodel:BehaviouralCapsule" name="Produtor"/>
    <Entities xsi:type="metamodel:BehaviouralCapsule" name="Consumidor"/>
  </Packages>
</metamodel:Model>
```

Figura 6.4. Visão do .xmi - Declarar Cápsula Consumidor

Para que duas cápsulas se comuniquem é preciso que exista um protocolo entre elas. A transformação que declara um protocolo não foi implementada neste trabalho, mas seria semelhante a declaração de cápsulas. Portanto, para prosseguir com este exemplo, um protocolo foi introduzido manualmente no arquivo **.xmi**. O protocolo recebeu o nome de **Comunicacao**, e a próxima transformação, **Introduzir Sinal de Saída,** trata da inserção de um sinal de saída neste protocolo.

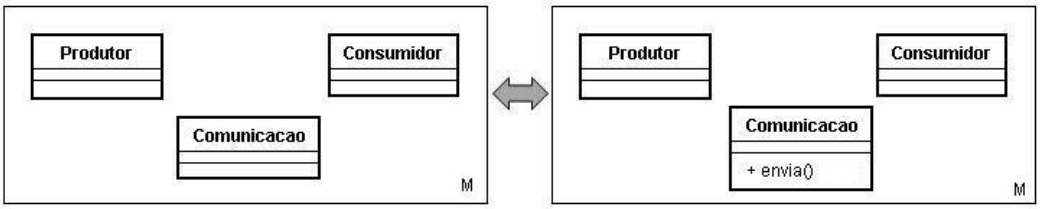

Figura 6.5. Visão do modelo - Introduzir Sinal de Saída

1

```
<?xml version="1.0" encoding="ASCII"?>
<metamodel:Model xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:metamodel="platform:/resource/TG/ecore-metamodel/Metamodel.ecore">
  <Packages name="Pacote 1">
    <Entities xsi:type="metamodel:BehaviouralCapsule" name="Produtor"/>
    <Entities xsi:type="metamodel:BehaviouralCapsule" name="Consumidor"/>
    <Entities xsi:type="metamodel:Protocol" name="Comunicacao"/>
  </Packages>
</metamodel:Model>
```

2

```
<?xml version="1.0" encoding="ASCII"?>
<metamodel:Model xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:metamodel="platform:/resource/TG/ecore-metamodel/Metamodel.ecore">
  <Packages name="Pacote 1">
    <Entities xsi:type="metamodel:BehaviouralCapsule" name="Produtor"/>
    <Entities xsi:type="metamodel:BehaviouralCapsule" name="Consumidor"/>
    <Entities xsi:type="metamodel:Protocol" name="Comunicacao">
      <Outgoings name="envia"/>
    </Entities>
  </Packages>
</metamodel:Model>
```

Figura 6.6. Visão do .xmi - Introduzir Sinal de Saída

Vemos através das Figuras 6.5 e 6.6 a introdução de um sinal de saída no protocolo **Comunicacao**. Esse sinal de saída é necessário para que a cápsula **Produtor** possa se comunicar com a cápsula **Consumidor**, enviando assim o resultado de sua produção. No arquivo **.xmi** mostrado acima, vemos que o nome do sinal de saída é **envia**.

Agora que já temos um protocolo para intermediar a comunicação entre as duas cápsulas, é necessário que criemos uma associação entre cada cápsula com esse protocolo. O resultado da associação, em cada cápsula, é a criação de uma porta do tipo **Comunicacao** que fica armazenada na sua lista de portas. Para evitar repetição, a transformação **Introduzir Associação Cápsula-Protocolo**, foi executada duas vezes, criando uma associação com cada cápsula, e será mostrado o resultado de sua segunda aplicação.

Vemos na Figura 6.7 abaixo que, além de possui um tipo de protocolo, uma porta também possui um nome.

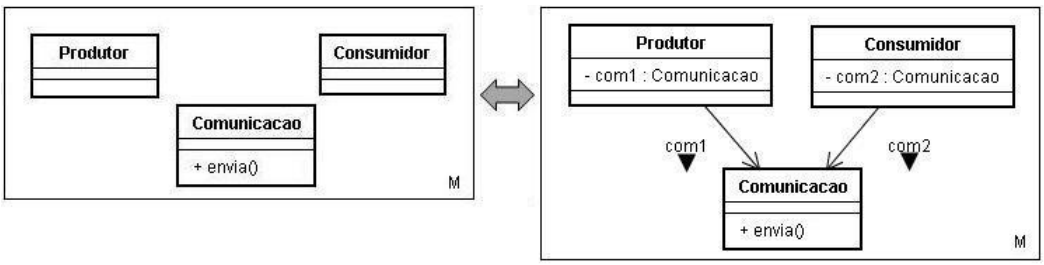

Figura 6.7. Visão do modelo - Introduzir Associação Cápsula-Protocolo

1

```
<?xml version="1.0" encoding="ASCII"?>
<metamodel:Model xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:metamodel="platform:/resource/TG/ecore-metamodel/Metamodel.ecore">
  <Packages name="Pacote 1">
    <Entities xsi:type="metamodel:BehaviouralCapsule" name="Produtor"/>
    <Entities xsi:type="metamodel:BehaviouralCapsule" name="Consumidor"/>
    <Entities xsi:type="metamodel:Protocol" name="Comunicacao">
      <Outgoings xsi:type="metamodel:Signal" name="envia"/>
    </Entities>
  </Packages>
</metamodel:Model>
```

2

```
<?xml version="1.0" encoding="ASCII"?>
<metamodel:Model xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:metamodel="platform:/resource/TG/ecore-metamodel/Metamodel.ecore">
  <Packages name="Pacote 1">
    <Entities xsi:type="metamodel:BehaviouralCapsule" name="Produtor">
      <Ports xsi:type="metamodel:Port" name="com2"/>
    </Entities>
    <Entities xsi:type="metamodel:BehaviouralCapsule" name="Consumidor">
      <Ports xsi:type="metamodel:Port" name="com1"/>
    </Entities>
    <Entities xsi:type="metamodel:Protocol" name="Comunicacao">
      <Outgoings xsi:type="metamodel:Signal" name="envia"/>
    </Entities>
  </Packages>
</metamodel:Model>
```

Figura 6.8. Visão do .xmi - Introduzir Associação Cápsula-Protocolo

A Figura 6.8 nos mostra, no arquivo gerado como saída, a presença das duas portas criadas como resultado da aplicação dessa transformação. Agora que já temos um sinal de saída no protocolo **Comunicacao**, precisamos adicionar métodos nas cápsulas **Produtor** e **Consumidor**. Utilizaremos a transformação **Introduzir Método** para criar os métodos **produz()** e **consome()**, o primeiro na cápsula **Produtor** e o segundo na cápsula **Consumidor**.

Para evitar repetição será mostrado o resultado da segunda aplicação dessa transformação.

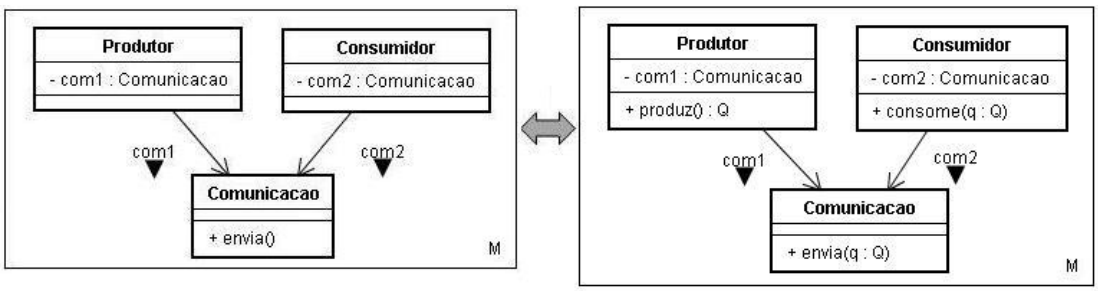

Figura 6.9. Visão do modelo - Introduzir Método

Vemos na Figura 6.9 que foi introduzido na cápsula **Produtor** o método **produz()**, que retorna uma entidade qualquer, aqui representada por **Q**. O sinal de saída do protocolo **Comunicacao**, e o método **consome()** da cápsula **Consumidor**, passam a receber como parâmetro essa entidade. Abaixo, na Figura 6.10, o arquivo **.xmi** com o resultado final das transformações.

```
<?xml version="1.0" encoding="ASCII"?>
<metamodel:Model xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"   1
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:metamodel="platform:/resource/TG/ecore-metamodel/Metamodel.ecore">
  <Packages name="Pacote 1">
    <Entities xsi:type="metamodel:BehaviouralCapsule" name="Produtor">
      <Ports xsi:type="metamodel:Port" name="com2"/>
    </Entities>
    <Entities xsi:type="metamodel:BehaviouralCapsule" name="Consumidor">
      <Ports xsi:type="metamodel:Port" name="com1"/>
    </Entities>
    <Entities xsi:type="metamodel:Protocol" name="Comunicacao">
      <Outgoings xsi:type="metamodel:Signal" name="envia"/>
    </Entities>
  </Packages>
</metamodel:Model>
```

```
<?xml version="1.0" encoding="ASCII"?>
<metamodel:Model xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"   2
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:metamodel="platform:/resource/TG/ecore-metamodel/Metamodel.ecore">
  <Packages name="Pacote 1">
    <Entities xsi:type="metamodel:BehaviouralCapsule" name="Produtor">
      <Ports xsi:type="metamodel:Port" name="com2"/>
      <Methods xsi:type="metamodel:Method" name="produz"/>
    </Entities>
    <Entities xsi:type="metamodel:BehaviouralCapsule" name="Consumidor">
      <Ports xsi:type="metamodel:Port" name="com1"/>
      <Methods xsi:type="metamodel:Method" name="consome"/>
    </Entities>
    <Entities xsi:type="metamodel:Protocol" name="Comunicacao">
      <Outgoings xsi:type="metamodel:Signal" name="envia"/>
    </Entities>
  </Packages>
</metamodel:Model>
```

Figura 6.10. Visão do .xmi - Introduzir Método

A Figura 6.10 mostra o arquivo .xmi final, resultante da aplicação das transformações aqui explicadas. Nela, estão destacados os métodos inseridos, **produz** e **consome**.

Neste capítulo foi possível mostrar a aplicação de quatro das seis transformações implementadas neste trabalho. Foi utilizado o exemplo do **Produtor** e **Consumidor**

para ilustrar como cada uma das transformações pode ser utilizada para manipular modelos de entrada, e gerar modelos diferentes na saída. As transformações aqui demonstradas foram: **Declarar Cápsula, Introduzir Associação Cápsula-Protocolo, Introduzir Sinal de Saída e Introduzir Método.**

Para validar o comportamento de cada uma delas, foram mostrados os arquivos **.xmi** antes e depois de suas aplicações, destacando as suas diferenças. Apesar de simples, todas as transformações se mostram de extrema utilidade, e proporcionam maior dinamismo na manipulação de cápsulas.

7. Conclusão

Neste trabalho, implementamos um conjunto de transformações de modelos para a linguagem de descrição arquitetural UML-RT. O conjunto de transformações escolhido foca, principalmente, em mudanças na declaração de elementos de projeto de UML-RT, como classes, cápsulas e protocolos.

Como já destacado anteriormente, o padrão de desenvolvimento proposto pela MDA (*Model Driven Architecture*) prega que o foco dos desenvolvedores de aplicações esteja voltado para a elaboração de modelos de alto nível, como o PIM (*Plataform Independent Model*) e o PSM (*Plataform Specific Model*), possibilitando a eles um melhor entendimento do negócio do sistema, e evitando preocupações com detalhes técnicos.

Porém, para que isso seja possível, é necessário que a criação de vários PSM's a partir de um PIM, e transformações aplicadas em um PSM, sejam de fato automatizadas. Vimos também que ainda existe muita carência na área, no que diz respeito a ferramentas que automatizem esses processos. Portanto, percebe-se que a automatização das transformações entre modelos é crucial para a utilização das práticas propostas pela MDA.

Um outro aspecto importante é que as transformações de modelos sejam consistentes, no sentido de preservarem semântica. Vários trabalhos relacionados a MDA sistematizam o processo, mas não possuem uma base sólida que possibilite provar a consistências das regras de transformação. Neste sentido, um diferencial do trabalho utilizado como base [14] para nosso projeto é que as transformações propostas são formalizadas e provadas a partir de uma semântica atribuída a UML-RT.

Para implementar as transformações abordadas neste trabalho, a linguagem *Kermeta* foi escolhida por prover um amplo ambiente de desenvolvimento de transformações e metamodelagem. Através das suas construções e sintaxe, foi possível definir classes que representam as entidades definidas no metamodelo, e os relacionamentos entre elas.

Na implementação, utilizamos o padrão de projeto *Command* que serve para encapsular ações ou comandos em objetos, utilizando uma interface padronizada. Dessa forma, diferentes tipos de requisições podem ser executados utilizando-se uma mesma interface, criando um alto nível de abstração. Assim, foi definida uma entidade abstrata que contém todas as operações necessárias para a realização da transformação, e todas as entidades que implementam transformações devem herdar seu comportamento e estrutura.

Essa estrutura montada para a implementação permite que novas transformações possam ser acopladas facilmente ao projeto da API, a medida que forem implementadas, fazendo assim com que a API seja totalmente extensível.

O processo de transformar um modelo em outro, no caso deste trabalho, envolve a leitura de um arquivo com formato XML, que descreve o modelo de entrada, um processamento feito em cima deste modelo, e a posterior escrita em outro arquivo, que representa o modelo de saída. A principal dificuldade durante a realização desse trabalho está relacionada ao ambiente de desenvolvimento *Kermeta e Eclipse*. O ambiente ainda não está muito maduro e por muitas vezes representava um empecilho

para o progresso do trabalho. O autor sente que existe uma real necessidade de melhora na ferramenta de desenvolvimento e execução de *Kermeta*.

Adicionalmente, uma outra importante contribuição deste trabalho é na proposição de um metamodelo para UML-RT. Esta linguagem pertence a uma ferramenta proprietária, Rose Real Time [22], e não dispõe de uma definição sobre os elementos da linguagem; a semântica da linguagem é disposta implicitamente na ferramenta e em poucos documentos disponibilizados pela Rational [23]. A semântica formal desta linguagem já foi previamente apresentada em [20], porém nenhum metamodelo foi encontrados pelo autor até então.

O metamodelo aqui proposto foi feito de forma simplificada e com o intuito de ser o mais abrangente possível, e os seus elementos representam as três principais visões arquiteturais da linguagem: diagrama de classes, diagrama de estrutura e diagrama de estados.

Os principais desafios encontrados no processo de refatoramento de modelos estão descritos em [24]. Esses desafios são classificados, dentre outros, em: *Preservação de Comportamento, Modelagem de Domínios Específicos e Suporte Ferramental.* Quanto à preservação de comportamento, todas as transformações aqui implementadas possuem provas formais [14] de que, em suas aplicações, o comportamento das entidades envolvidas é mantido, conforme mencionado anteriormente. Já no que diz respeito à modelagem de domínios específicos, este trabalho se encaixa perfeitamente, já que consiste em refatoramentos para uma linguagem específica, UML-RT. Finalmente, a maior contribuição deste trabalho está no quesito suporte ferramental, já que se trata da implementação de refatoramentos e da construção de uma estrutura para o posterior acoplamento de novas transformações.

Em [25] e [26], temos definições de metamodelos para linguagens de descrição arquitetural (ADL's). O metamodelo proposto em [25] possui algumas similaridades com o metamodelo proposto neste trabalho, principalmente no que diz respeito a sua divisão clara em três partes, nesse caso: *atividade*, *componentes*, *máquina de estado*. Apesar de possuir entidades que não se aplicam ao contexto deste trabalho, ele foi de fundamental importância na etapa de elaboração do metamodelo aqui apresentado. Já o metamodelo da linguagem ACME [26] é mais simplificado e possui alguns poucos elementos relacionados ao nosso metamodelo. Além disso, os relacionamentos entre as entidades apresentados em [26] não estão muito claros, dando margem a diferentes interpretações.

Sobre refatoramentos arquiteturais, [27] apresenta uma adaptação da Teoria da Decisão para minimizar riscos e esforços quando das decisões necessárias para este tipo de refatoramento. O artigo também apresenta métodos para avaliação de alternativas de refatoramento, até mesmo quando as conseqüências são de difícil interpretação e os valores são incertos.

Sobre refatoramentos de modelos, [28] apresenta uma abordagem bastante interessante no que diz respeito ao entendimento de modelos como grafos, e metamodelos como grafos de tipos. É interessante notar que a tecnologia utilizada neste trabalho, EMF, possibilita que tratemos nosso metamodelo também como um grafo. Ainda em [28], o autor mostra que transformações em modelos nada mais são

do que transformações executadas em grafos, destacando exemplos e regras de aplicações.

Uma proposta de *framework* para refatoramentos utilizando transformações de modelos e anotações semânticas é apresentada em [29]. Nesse artigo, os autores definem um metamodelo contendo alguns elementos presentes no nosso metamodelo, tais como protocolos e portas, instanciam modelos utilizando *annotations* como *soft-goals, métricas e restrições* e, finalmente, aplicam ações selecionadas em um conjunto de refatoramentos possíveis.

Após a conclusão deste trabalho, é possível identificar várias possibilidades de continuidade em oportunidades futuras. A primeira delas seria a implementação de todas as transformações definidas em [14], tendo em vista que este trabalho trata apenas um subconjunto delas. Outra possibilidade seria a implementação de um editor gráfico de modelos baseados no metamodelo aqui definido. Através desse editor gráfico, seria mais fácil construir modelos e trabalhar com eles, sendo necessário também que o modelo definido graficamente fosse corretamente mapeado para um arquivo .xmi.

Ainda existe a possibilidade da implementação da etapa de casamento de padrões, que seria responsável por procurar, em um modelo, características e elementos suscetíveis a aplicação de uma dessas transformações. Há também a possibilidade de estender a implementação atual das verificações das regras, para abordar também regras que consideram ações, e de utilizar a linguagem OCL. Finalmente, o uso prático da ferramenta pode ser potencializado se a mesma for implementada como *plug-in* de ferramentas de modelagem como o Rose-RT.

8. Referências

[1] ROTHENBERT, Jeff; WILLIAM, L.E; LOPARO, K.A; NELSON, N.R. *The Nature of Modeling*. Artificial Intelligence, Simulation and Modeling. New York, John Wiley and Sons, Inc., 1989, pp 75-92

[2] Dicionário da língua portuguesa Michaellis, 1998 Cia. Melhoramentos de São Paulo.

[3] BÉZIVIN, Jean. *Introduction to Model Engineering – A gentle introduction to a new way of considering the construction and maintenance of information systems*. ATLAS Group, INRIA and LINA, University of Nantes.

[4] WARMER, Jos; KLEPPE, Anneke; BAST, Wim. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison Wesley, 2003.

[5] STACHOWIAK, H. *Allgemeine Modelltheorie*. Springer-Verlag, Wien etc. 1973

[6] LUDEWIG, Jochen. *Models in software engineering – an introduction*. Institute of Software Techonology at Stuttgart University. 2003

[7] *Community site for meta-modeling and semantic modeling –* www.metamodel.com, último acesso em 24/01/2008.

[8] KENT, Stuart. *Model Driven Engineering*, 2002.

[9] TELES, Fabrício; RAMOS, Rodrigo. *Model Transformations*.

[10] GULLION, Tom. *MDA and QVT*. Borland, 2006.

[11] *ATL: The ATLAS Transformation Language*, disponível em www.eclipse.org/m2m/atl/doc/ATL_PresentationSheet.pdf, último acesso em 24/01/2008.

[12] *OMG Model Driven Architecture* - http://www.omg.org/mda/, último acesso em 24/01/2008.

[13] Aula de Projeto de Cápsulas da disciplina Análise e Projeto de Sistemas, Centro de Informática, Universidade Federal de Pernambuco. Disponível em www.cin.ufpe.br/~if718

[14] RAMOS, R. *Desenvolvimento Rigoroso com UML-RT*. Dissertação de mestrado, Universidade Federal de Pernambuco, Centro de Informática, 2005.

[15] *Eclipse Modeling Framework* – http://www.eclipse.org/modeling/emf/?project=emf, último acesso em 24/01/2008.

[16] *XSL Transformations (XSLT)* – http://www.w3.org/TR/1999/REC-xslt-19991116.html#section-Introduction, último acesso em 24/01/2008.

[17] Linguagem de metamodelagem *Kermeta* - www.kermeta.org, último acesso em 24/01/2008.

[18] SELIC, B; RUMBAUGH, J. *Using UML for Modeling Complex RealTime Systems.* Rational Software Corporation, 1998. Disponível em http://www. rational.com.

[19] JOUAULT, Fréderic; KURTEV, Ivan. *On the Architectural Alignment of ATL and QVT*. ATLAS Group, INRIA and LINA, University of Nantes.

[20] RAMOS, Rodrigo; SAMPAIO, Auguso; MOTA, Alexandre. *A Semantics for UML-RT Active Classes via Mapping into Circus*. Proc. 7th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'05), Vol. 3535 of LNCS, Athens, Greece,Springer (2005) 99-114.

[21] FLEUREY, Franck; DREY, Zoé; DIDIER, Vojtisek; FAUCHER, Cyril; MAHÉ, Vincent. *Kermeta Language – Reference Manual*.

[22] Site da ferramenta Rational Rose Real-Time, IBM - http://www.ibm.com/developerworks/rational/library/622.html, acessado em 24/01/2008.

[23] Site da Rational, IBM - http://www-306.ibm.com/software/br/rational, acessado em 24/01/2008.

[24] MENS, Tom; TAENTZER, Gabriele; MULLER, Dirk. *Challenges in Model Refactoring*. University of Mons-Hainaut, Bélgica; Philipps-Universität Marburg, Alemanha.

[25] Metamodelo da linguagem V3Studio. Disponível em http://gforge.inria.fr/plugins/scmcvs/cvsweb.php/patternmatching_projects/fr.irisa.triskell.kermeta.patternmatching.samples/V3Studio/metamodels/?cvsroot=kermeta, acessado em 24/01/2008.

[26] Metamodelo da linguagem ACME. Disponível em www.eclipse.org/m2m/atl/atlTransformations/METAH2ACME/METAH2ACME.pdf , acessado em 24/01/2008.

[27] WOHLFARTH, Sven; RIEBISCH, Matthias. *Evaluating Alternatives for Architecture-Oriented Refactoring.* Technical University Ilmenau, Alemanha.

[28] MENS, Tom. *On the use of graph transformations for model refactoring.* Software Engineering Lab, University of Mons-Hainault.

[29] IVKOVIC, Igor; KONTOGIANNIS, Kostas. *A Framework for Software Architecture Refactoring using Model Transformations and Semantic Annotations.* University of Waterloo, Canadá.

Apêndice

Neste apêndice serão mostradas as definições de todas as leis de transformação implementadas neste trabalho. Para cada uma delas, será mostrado o modelo antes e depois da aplicação da transformação. Vale ressaltar que as leis podem ser aplicadas nos dois sentidos e possuem condições específicas para cada um. Todas essas leis se encontram em [14].

1. Declarar Cápsula

Esta lei estabelece quando é possível introduzir uma nova cápsula ao modelo. Observe que apesar de o contexto no lado esquerdo da lei estar vazio, o M subscrito fixa o contexto para a aplicação da lei.

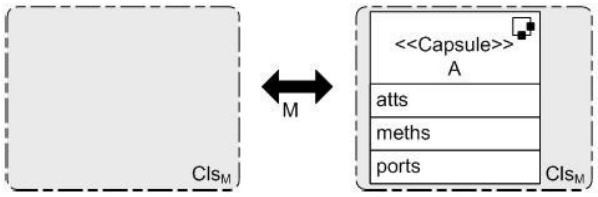

Figura A.1. Declarar Cápsula.

Condições:
(→) CIsm não possui a declaração de nenhum elemento, no mesmo pacote, chamado **A**.
(←) Nenhuma cápsula em **M** tem uma relação com a cápsula **A** em qualquer diagrama.

Usamos o símbolo (→) antes de uma condição para indicar que ela é requerida somente para aplicações da lei da esquerda para direita. Similarmente, utilizamos (←) para indicar que a condição é necessária somente para aplicações da lei da direita para a esquerda. CIsm representa a visão de diagrama de classes enquanto que Strm representa a visão de diagrama de estruturas.

2. Introduzir Associação Cápsula-Cápsula

Esta lei estabelece quando podemos adicionar ou remover uma associação entre cápsulas. Assumimos que, quando uma cápsula **A** é criada, é criado também seu diagrama de estrutura; este diagrama contém as instâncias de todas as cápsulas com as quais **A** possui uma associação.

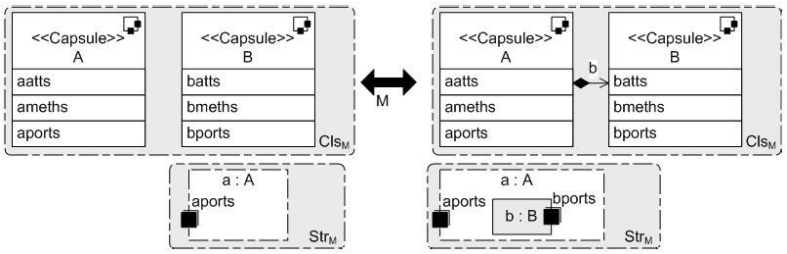

Figura A.2. Introduzir Associação Cápsula-Cápsula.

Condições:
(→) Não existe outra instância de cápsula chamada **b** no diagrama de estrutura de **A**.
(←) A instância **b** da cápsula **B** não está conectada a nenhuma outra na estrutura de **A**, inclusive à instância **a** que a contém.

3. Introduzir Associação Cápsula-Classe

Esta lei introduz um atributo em uma cápsula, como conseqüência da criação de uma associação da cápsula com a classe. Assumimos que a introdução de uma associação **b** entre uma cápsula **A** e uma classe **B** introduz implicitamente um atributo **b** em **A**, por esta razão a existência de uma condição que indica que não pode haver outro atributo chamado **b** em **A**.

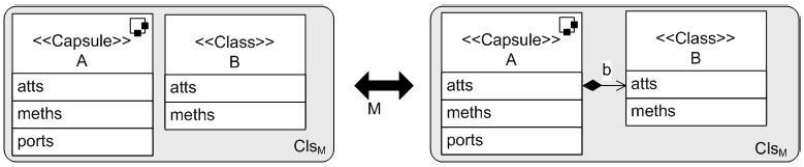

Figura A.3. Introduzir Associação Cápsula-Classe.

Condições:
(→) Não há um atributo chamado **b** em **A**.
(←) O atributo **b** não é utilizado por nenhum método, diagrama de estados ou predicado de **A**.

4. Introduzir Associação Cápsula-Protocolo

Esta lei estabelece quando podemos adicionar ou remover uma associação entre um protocolo e uma cápsula. Como conseqüência da introdução desta associação, é criada uma instância deste protocolo (porta) na cápsula.

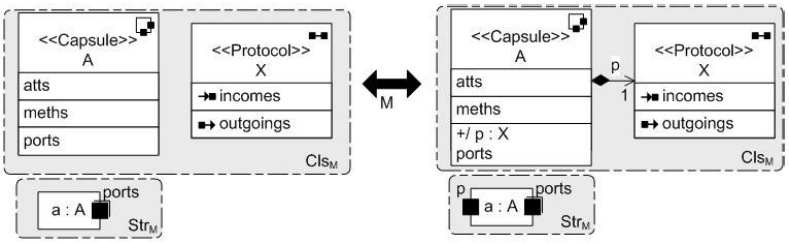

Figura A.4. Introduzir Associação Cápsula-Protocolo.

Condições:
(→) Não existe outra porta chamada **p** na cápsula **A**.
(←) A porta **p** não é utilizada no diagrama de estados de **A**; não existe conexão ligada a **p** em Strm.

5. Introduzir Método

Esta lei estabelece quando é permitido adicionar ou remover métodos em uma cápsula.

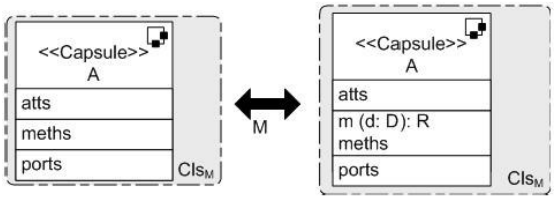

Figura A.5. Introduzir Método.

Condições:
(→) Não existe um método chamado **m** em **A**.
(←) O método **m** não é utilizado por outro método em **meths**, nem tampouco no diagrama de estados de **A**, ou em um predicado de **A**. Assumimos aqui que **meths** é o conjunto de métodos da cápsula **A**.

6. Introduzir Sinal de Saída

Esta lei estabelece quando é permitido adicionar ou remover sinais de saída em um protocolo.

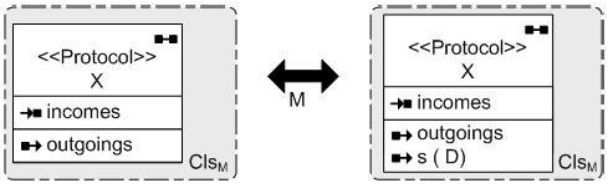

Figura A.6. Introduzir Sinal de Saída.

Condições:
(→) Não existe um sinal chamado **s** no protocolo **X**.
(←) Nenhum diagrama de estados em Stam usa o sinal **s**.

Orientador

_____

Prof. Ph.D. Augusto Cesar Alves Sampaio

Aluno

_____

Eliaquim Lima Sá Neto