Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Informática

# Identificação de Diretrizes para a Construção de Meta-modelos na Infra-estrutura de MDA

Andreza de Sousa Vieira

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande – Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Engenharia de Software

Dr. Franklin de Souza Ramalho
(Orientador)

Campina Grande, Paraíba, Brasil

**FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA CENTRAL DA UFCG**

V658
2010 Vieira, Andreza de Sousa.

Identificação de diretrizes para a construção de meta-modelos na infra-estrutura de MDA / Andreza de Sousa Vieira. ─ Campina Grande, 2010.
169f. : il.

Dissertação (Mestrado em Ciência da Computação) – Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática.
Referências.
Orientador: Prof. Dr. Franklin de Souza Ramalho.

1. Engenharia de Software. 2. Meta-modelos. 3. MDA - *Model-Driven Architecture*. 4. Mineração de Dados. 5. Informática – Padrões e Diretrizes. I. Título.

CDU – 004.41(043)

"IDENTIFICAÇÃO DE DIRETRIZES PARA A CONSTRUÇÃO DE META-MODELOS NA INFRA-ESTRUTURA DE MDA"

Andreza de Sousa Vieira

ANDREZA DE SOUSA VIEIRA

DISSERTAÇÃO APROVADA EM 16.04.2010

FRANKLIN DE SOUZA RAMALHO, Dr.
Orientador(a)

JORGE CESAR ABRANTES DE FIGUEIREDO, D.Sc
Examinador(a)

RITA SUZANA PITANGUEIRA MACIEL, Dra.
Examinador(a)

CAMPINA GRANDE - PB

# Resumo

MDA (*Model-Driven Architecture*) é uma abordagem com o objetivo principal de deslocar o esforço e tempo gastos durante as tarefas de testes e implementação do ciclo de vida de desenvolvimento de um *software* para tarefas de modelagem, meta-modelagem e transformações. Toda uma infra-estrutura e um conjunto de formalismos têm sido propostos dentro de MDA, a exemplo de UML, OCL, MOF e QVT. Por outro lado, diretrizes são guias que auxiliam e orientam pessoas na realização de determinadas atividades. No âmbito da computação, por exemplo, existem diretrizes para boas práticas de programação e para especificação de requisitos de um sistema. Todavia, dentro da infra-estrutura de MDA, a identificação de diretrizes que auxiliam na elaboração de seus artefatos (modelos, meta-modelos e transformações) ainda é incipiente e pouco se tem produzido. Nesse sentido, propomos um conjunto de diretrizes com foco, especificamente, na construção de meta-modelos. Estas diretrizes foram identificadas através da realização de duas abordagens: (i) automática, a partir da aplicação do processo KDD (*Knowledge Discovery in Databases*). Neste caso, uma ferramenta de suporte foi desenvolvida para auxiliar na realização de algumas etapas do processo KDD; e (ii) manual, a partir de uma análise detalhada de um conjunto de meta-modelos. Como resultado de ambas as abordagens, apresentamos um catálogo com 13 diretrizes documentadas de acordo com um *template* baseado no proposto pela GoF para padrões de projeto. Uma ferramenta de suporte foi desenvolvida para aplicar as diretrizes em meta-modelos já existentes de forma automática. A utilização destas diretrizes proporciona uma maior facilidade de compreensão, construção, manutenção, evolução e reuso dos meta-modelos. A avaliação deste trabalho foi realizada por meio da aplicação das diretrizes em seis meta-modelos, no intuito de analisarmos a aplicabilidade de cada uma delas.

# Abstract

MDA (*Model-Driven Architecture*) is an approach whose main objective is to shift the effort and time spent during the tests and implementation tasks of the software development lifecycle to modeling, metamodeling and transformations tasks. An entire infrastructure and a set of formalisms have been proposed in the MDA infrastructure, such as UML, OCL, MOF and QVT. By other hand, guidelines are directions to help people with little or none experience in carrying out certain activities. In the computing context, for instance, there are guidelines for best-practices programming and for requirements specification of a system. However, in the MDA infrastructure, the identification of guidelines that assist the elaboration of their artifacts (models, metamodels and transformations) is still incipient and little has been produced. We propose in this work a set of guidelines focused, specifically, on the construction of metamodels. These guidelines were identified by means of two approaches: (i) automatic, by applying the KDD (*Knowledge Discovery in Databases*) process. In this case, a tool was developed to support some KDD steps; and (ii) manual, by analyzing a set of metamodels. As a result of both approaches, we present a catalog with 13 guidelines documented according to a template based on that proposed by the GoF for design patterns. A tool support was developed to automatically apply the guidelines to existent metamodels. The adoption of these guidelines provides for the metamodels better understanding, construction, maintenance, development and reuse. The evaluation of this work was performed by applying the guidelines to six well-known metamodels in order to analyze the applicability of each guideline.

# Agradecimentos

Agradeço primeiramente a Deus pelo dom da vida e pela bênção de passar por mais uma etapa na minha vida: a conclusão do meu mestrado.

Aos meus pais, ao meu irmão, ao meu noivo e aos meus amigos, os quais participaram efetivamente de toda essa trajetória, dando conselhos, incentivos e levantando a minha cabeça nos momentos difíceis. Obrigada pela paciência e compreensão dos momentos em que estive ausente em suas vidas.

Ao meu orientador, o professor Franklin Ramalho. Sou muito grata pela paciência, compreensão e auxílio, pela dedicação empenhada, pelas motivações, pelos rápidos e construtivos *feedbacks* e por toda a força. Agradeço também a professora Patrícia Machado pela participação inicial no projeto.

Aos professores e demais funcionários que fazem o Curso de Pós-Graduação em Ciência da Computação da UFCG, que prestaram auxílio direta ou indiretamente à minha pesquisa. Em especial, ao professor Marcus Sampaio por ter me ajudado bastante quando precisei, sempre muito solícito e prestativo.

Aos meus colegas e amigos da UFCG que, direta ou indiretamente, me ajudaram durante a trajetória do curso dando força e conselhos tanto profissionais quanto pessoais: Andréa, Arthur, Bel, Giuseppe, Jemerson, Katiusco, Lilian, Makelli, Paulo, Roberto, Rute e Sebastião. Em especial a Anderson, Ane Caroline, Antônio Júnior, Camila, Diego, Everton, Fábio e Waldemar, pois sempre que precisei demonstraram disponibilidade e atenção em atender minhas dúvidas técnicas. Agradeço também a HP (*Hewlett-Packard*), que colaborou financeiramente para a realização deste projeto.

# Conteúdo

# Lista de Símbolos

ARFF - *Attribute-Relation File Format*
ATL - *ATLAS Transformation Language*
CIM - *Computational Independent Model*
CSV - *Comma-Separated Values*
CWM - *Common Warehouse Metamodel*
EMF - *Eclipse Modeling Framework*
GoF - *Gang of Four*
GPL - *General Public License*
GQM - *Goal, Question, Metric*
KDD - *Knowledge Discovery in Databases*
MDA - *Model-Driven Architecture*
MDD - *Model-Driven Development*
ML - *Model Language*
MOF - *Meta Object Facility*
OCL - *Object Constraint Language*
PIM - *Platform Independent Model*
PNML - *Petri Net Markup Language*
PSM - *Platform Specific Model*
QVT - *Query/View/Transformation*
SPEM - *Software and Systems Process Engineering Meta-Model*
U2TP - *UML 2 Testing Profile*
UML - *Unified Modelling Language*
WEKA - *Waikato Environment for Knowledge Analysis*
XMI - *XML Metadata Interchange*

# Lista de Figuras

# Lista de Tabelas

# Lista de Códigos Fonte

# Capítulo 1

# Introdução

Cada vez mais surgem novas abordagens e tecnologias no âmbito da engenharia de *software* no intuito de oferecer, dentre outros aspectos, uma maior automação e, conseqüentemente, maior produtividade no processo de desenvolvimento de *software*. Como exemplo de uma abordagem que atua nesse sentido temos o MDA (*Model-Driven Architecture*) [MDA, 2010]. Trata-se de uma arquitetura proposta pela OMG (*Object Managment Group*) [OMG, 2010] com o objetivo principal de deslocar o esforço e tempo gastos durante as tarefas de implementação e testes do ciclo de vida de desenvolvimento de um *software* para tarefas de modelagem, meta-modelagem e transformações entre modelos. Assim, podemos construir sistemas consistentes e independentes de plataforma.

Dentro da infra-estrutura de MDA, meta-modelagem é a tarefa de construir meta-modelos que, por sua vez, são gramáticas abstratas. Dentre outras atribuições, os meta-modelos possuem fundamental importância na definição de linguagens, dentre as quais podemos citar as linguagens de modelagem, de processo e de programação. Atualmente, estes artefatos podem ser definidos através de duas principais linguagens existentes para meta-modelagem, chamadas MOF (*Meta Object Facility*) [MOF, 2006] e Ecore [Ecore, 2009]. Um exemplo de meta-modelo bem definido e conhecido é a especificação da UML (*Unified Modelling Language*) [UML, 2007].

MDA reconhece a importância dos modelos no processo de desenvolvimento, tornando-os um ponto-chave. Nesse sentido, MDA sugere que os modelos de sistemas sejam especificados nas seguintes visões [Kleppe et al., 2003]: (i) CIM (*Computational Independent Model*) - especifica os modelos independentes de computação, representando o domínio e os

requisitos do funcionamento do sistema; (ii) PIM (*Platform Independent Model*) - especifica os requisitos e projeto de *software* independentes de qualquer plataforma de implementação; (iii) PSM (*Platform Specific Model*) - especifica os modelos com todos os detalhes da plataforma para a qual o *software* será implementado; e (iv) Código - produto final e executável.

Muitos projetos estão aderindo a esta nova abordagem que MDA propõe. Isto se deve, dentre outros fatores, ao grande número de vantagens que ela oferece. A primeira delas é a produtividade. Grande parte do código será gerada de forma automática a partir da transformação PSM ⇒ código, assim, acarretando na redução de tempo e custo do desenvolvimento do projeto. A portabilidade é mais uma vantagem que merece destaque, uma vez que o PIM é independente de plataforma, ele pode ser transformado automaticamente em vários PSMs, dessa maneira, permitindo que um sistema opere em diferentes plataformas.

Outro benefício de MDA é a facilidade na manutenção e documentação, tendo em vista que a manutenção não é mais realizada no código, mas sim em um nível mais alto de abstração, no CIM, mantendo a documentação atualizada constantemente. A interoperabilidade é outra de suas vantagens. Embora os PSMs especificados para plataformas diferentes não sejam interoperáveis diretamente, podemos utilizar o conceito de *bridges* para realizarmos a comunicação entre PSMs e também entre códigos de diferentes plataformas.

## 1.1 Motivação

Além das vantagens que a abordagem de MDA oferece, a adoção de padrões ou diretrizes que indiquem boas práticas para o desenvolvimento dos seus artefatos é mais um grande benefício. Segundo [Alexander et al., 1977], “um padrão é uma entidade que descreve um problema que ocorre repetidamente em um ambiente e então descreve a essência de uma solução para este problema, de tal forma que você possa usar essa solução várias vezes, sem nunca utilizá-la do mesmo modo”. Por outro lado, as diretrizes são guias que auxiliam e orientam pessoas na realização de determinadas atividades.

O uso de diretrizes ou padrões no desenvolvimento de *software* proporciona uma série de vantagens, dentre as quais podemos mencionar a aprendizagem com a experiência de outras pessoas, a facilidade para construção, manutenção, compreensão, evolução e reuso do *software*. Desta forma, haverá também um aumento da sua qualidade.

Dentro da infra-estrutura de MDA, em seus diferentes artefatos, a identificação de padrões ou diretrizes ainda é uma atividade incipiente e pouco se tem produzido. A maioria

dos trabalhos existentes neste ramo ainda se concentra em identificar maneiras de relacionar ou representar padrões de projeto [Gamma et al., 1995] no escopo desta infra-estrutura, como os apresentados em [France et al., 2004] e [Dong et al., 2003]. Por outro lado, alguns trabalhos abordam camadas ou padrões específicos da arquitetura de MDA, como por exemplo, padrões para especificação OCL (*Object Constraint Language*) [Ackermann, 2006] e padrões para transformação de modelos [Bézivin et al., 2005] [Iacob et al., 2008].

Há uma carência ainda maior, em especial, na identificação de diretrizes ou padrões para auxiliar na construção de meta-modelos. Nesse sentido, [Mili et al., 1998] identifica três padrões para meta-modelagem. Contudo, eles são específicos para o meta-modelo de *SmallTalk* [Goldberg et al., 1989]. Além do mais, não é utilizada a infra-estrutura de MDA, nem os seus formalismos, como MOF ou OCL [Warmer et al., 2003]. Neste mesmo escopo, [García et al., 2009] propõe uma diretriz para facilitar a definição de Linguagens de Modelagem (*ML – Model Languages*) através de meta-modelos. Entretanto, o desenvolvedor deve especificar as características estruturais e demais informações da linguagem que deseja modelar e, assim, a diretriz irá propor o meta-modelo para especificar esta linguagem. Este trabalho não trata de um conjunto de diretrizes que auxiliem o desenvolvedor na meta-modelagem ou de padrões aplicáveis a qualquer meta-modelo.

Em suma, podemos observar que poucos trabalhos na área de MDA têm contribuído no sentido de propor boas práticas de desenvolvimento. Esta carência se estende por todos os artefatos dessa abordagem, em especial, pelos meta-modelos. Os trabalhos já existentes ou (i) não abordam uma camada ou padrão específico de MDA; ou (ii) o fazem de forma bem restrita.

## 1.2 Objetivos

Este trabalho tem como objetivo, de maneira geral, identificar diretrizes no âmbito da infra-estrutura de MDA, em particular, nos seus meta-modelos. Desta forma, amenizando a lacuna existente nesta infra-estrutura, como mencionado anteriormente. Uma vez que as diretrizes tenham sido identificadas, este trabalho propõe a elaboração de um catálogo para a documentação de todas elas, de acordo com um *template* pré-estabelecido.

Com as diretrizes identificadas, pretende-se auxiliar pessoas no desenvolvimento de novos meta-modelos. Além disso, a aplicação de tais diretrizes em meta-modelos já existentes poderá oferecê-los grandes benefícios, como por exemplo, uma maior facilidade de construção, compreensão, manutenção, reuso e, conseqüentemente, evolução.

Adicionalmente, faz parte da finalidade deste trabalho a implementação de uma ferramenta de suporte para auxiliar os desenvolvedores na aplicação das diretrizes em meta-modelos já existentes de maneira automática. Desta forma, além de guiar as pessoas na construção de novos meta-modelos, as diretrizes são úteis também na melhoria dos já existentes.

## 1.3 Escopo do Trabalho

Tendo em vista que MDA dispõe de uma área muito ampla para a investigação de diretrizes, este trabalho restringe seu foco para um dos seus artefatos, os meta-modelos. A construção de meta-modelos não é uma tarefa trivial, principalmente, devido à carência de métodos que sirvam como guia para a sua realização. Com este trabalho, faremos uma contribuição inicial para a comunidade de MDA no que se refere à identificação de diretrizes em meta-modelos. Dessa forma, esperamos que elas sirvam como auxílio para que os desenvolvedores obtenham uma maior produtividade nas tarefas de meta-modelagem.

O desenvolvimento de meta-modelos exige um prévio conhecimento tanto de meta-modelagem quanto da linguagem que se deseja meta-modelar, além do mais, é necessária muita cautela e um pouco de experiência. Entretanto, nem todas as pessoas atendem a tais requisitos para a realização desta atividade, e ainda por cima, algumas delas confundem meta-modelagem com modelagem, por exemplo, em UML. Portanto, com a ausência de artifícios que auxiliem os desenvolvedores na meta-modelagem, eles constroem seus meta-modelos sem nenhuma referência a ser seguida e sem nenhuma base sólida que os indique as melhores decisões a serem tomadas.

Desta forma, podemos notar a grande necessidade de tais artifícios dentro da infra-estrutura de MDA. Este problema pode ser amenizado com a criação de padrões, tais quais os padrões de projeto, padrões de testes e padrões de processo ou com a definição de diretrizes que indiquem boas práticas de meta-modelagem.

## 1.4 Metodologia de Trabalho

Inicialmente, foi realizada uma pesquisa bibliográfica em busca de trabalhos que espelham o atual estado da arte no contexto de padrões, diretrizes ou qualquer outro mecanismo que auxilie as pessoas na construção dos artefatos de MDA.

Com a finalidade de identificarmos diretrizes para meta-modelos de maneira automática aplicamos a mineração de dados, que é uma tarefa para a descoberta de conhecimento a partir de uma grande quantidade de dados [Han e Kamber, 2001]. Trata-se de uma área multidisciplinar envolvendo não apenas bancos de dados e *data warehouse*, mas também estatística, aprendizagem de máquina, visualização de dados, recuperação de informação e computação de alto desempenho, dentre outros aspectos.

Portanto, para aplicarmos a mineração de dados neste trabalho tivemos que seguir cada uma das etapas do processo KDD (*Knowledge Discovery in Databases*) [Fayyad et al., 1996], no qual a mineração de dados está inserida como uma delas. As três primeiras etapas foram realizadas com o auxílio de uma ferramenta de suporte desenvolvida neste trabalho, exclusivamente, para este objetivo. Antes de iniciarmos a etapa de mineração de dados, foi realizado um estudo comparativo entre diversas ferramentas de mineração existentes no mercado a fim de selecionarmos uma para a execução dos algoritmos de análise de associação. Entretanto, não obtivemos os resultados esperados com a aplicação da mineração de dados, conseguimos apenas algumas regras de associação que serviram para a identificação de uma única diretriz, bem como outras regras que são úteis para validação de meta-modelos.

No intuito de propormos mais diretrizes, realizamos também a identificação destas por meio de um procedimento manual, a partir de uma minuciosa análise em um conjunto de meta-modelos construídos por organizações diferentes, dentre elas a OMG. Ao final deste procedimento, conseguimos identificar um total de 12 diretrizes independentes de domínio.

Como resultado de todo o processo de identificação, tanto de forma automática quanto manual, conseguimos encontrar um total de 13 diretrizes, documentadas de acordo com um *template* pré-estabelecido. Além disso, uma segunda ferramenta de suporte foi desenvolvida para aplicar, automaticamente, as diretrizes em meta-modelos já existentes.

Uma vez que as diretrizes foram identificadas e documentadas, realizamos um processo de avaliação com base no método GQM (*Goal, Question, Metric*) no intuito de analisarmos a aplicabilidade de cada uma delas. Para aplicarmos as diretrizes e realizarmos a análise, escolhemos um conjunto de meta-modelos que já são bem consolidados e reconhecidos, dentre os quais alguns são da OMG. Portanto, são meta-modelos consistentes e confiáveis, construídos e validados por especialistas da área.

## 1.5 Relevância e Contribuições Esperadas

Inicialmente, vale ressaltar a grande relevância da meta-modelagem dentro da infra-estrutura

de MDA. Existem dois principais motivos para que ela seja tão importante nesta infra-estrutura. Primeiro, precisamos de um mecanismo para definir linguagens (de modelagem, de processo, de programação, etc.), de forma que sejam definidas sem ambigüidade. Dessa forma, uma ferramenta de transformação poderá ler, escrever e entender os modelos. Em MDA, tais linguagens são definidas através de meta-modelos. Em segundo lugar, as regras de transformação que constituem a definição de transformação descrevem como um modelo numa linguagem de origem pode ser transformado em um modelo numa linguagem de destino. Essas regras usam os meta-modelos das linguagens de origem e destino para definir as transformações [Kleppe et al., 2003].

Visto que os meta-modelos possuem importantes papéis em MDA e que o desenvolvimento de tais artefatos não é uma tarefa fácil, uma das maiores relevâncias deste trabalho é o catálogo composto por 13 diretrizes que propomos para meta-modelagem. Vale ressaltar que tais diretrizes são independentes de domínio, ou seja, podem ser utilizadas em qualquer meta-modelo. Este catálogo pode ser considerado como uma contribuição inicial para a comunidade de MDA em relação à identificação de diretrizes em meta-modelos, de forma que ele possa ser estendido, futuramente, ao passo que surge a necessidade de novos artifícios que auxiliem na meta-modelagem.

Todas as diretrizes presentes no catálogo proposto são documentadas de acordo com um *template* baseado no adotado pela GoF (*Gang of Four*) para padrões de projeto. Nosso *template* descreve o objetivo, a motivação, a solução, as conseqüências, a aplicabilidade, as diretrizes relacionadas e exemplos práticos da aplicação de cada uma em meta-modelos variados. Mais uma relevância deste trabalho e contribuição para a comunidade de MDA é a ferramenta de suporte desenvolvida no intuito de aplicar, automaticamente, as diretrizes em meta-modelos já existentes, desta forma, aumentando a qualidade dos mesmos.

Outra contribuição deste trabalho é todo o legado que deixamos para futuros estudos e aplicações da mineração de dados. As etapas e os procedimentos que realizamos podem servir com base para outras pessoas aplicarem a mineração de dados para a descoberta automática de conhecimento em outras áreas.

Por fim, este trabalho pode ser considerado como um ponto de partida para uma investigação mais intensa de diretrizes ou até mesmo de padrões nos demais artefatos da infra-estrutura de MDA.

## 1.6 Estrutura do Documento

Esta dissertação encontra-se organizada em sete capítulos, incluindo esta introdução. A seguir, vejamos uma breve descrição de cada um dos capítulos restantes:

- Capítulo 2: Fundamentação Teórica. Apresenta os conceitos básicos para o entendimento deste trabalho. Serão abordados fundamentos: (i) da infra-estrutura de MDA e de alguns dos elementos que ela envolve, tais como meta-modelos, OCL e MOF; (ii) da mineração de dados, suas tarefas e ferramentas que auxiliam neste processo; e (iii) de padrões e diretrizes em um âmbito geral;

- Capítulo 3: Abordagens para Descoberta de Diretrizes. Apresenta, detalhadamente, todas as etapas realizadas para a identificação de diretrizes tanto de forma automática, por meio da mineração de dados, quanto de forma manual, por meio de uma análise em um conjunto de meta-modelos. Além disso, o capítulo apresenta a ferramenta de suporte desenvolvida neste trabalho para auxiliar na realização de algumas tarefas da abordagem automática;

- Capítulo 4: Conjunto de Diretrizes Identificadas para Meta-modelos. Apresenta o catálogo de diretrizes identificadas tanto de forma automática quanto manual, documentadas de acordo com um *template* pré-estabelecido. Além disso, são destacadas as diretrizes que também podem ser aplicadas em modelos, seguidas de um exemplo. Por fim, o capítulo descreve a ferramenta de suporte desenvolvida para a aplicação automática das diretrizes em meta-modelos;

- Capítulo 5: Avaliação Experimental. Apresenta todo o processo avaliativo deste trabalho seguindo a metodologia proposta pela abordagem GQM, a fim de analisar a aplicabilidade das diretrizes que identificamos;

- Capítulo 6: Trabalhos Relacionados. Apresenta os trabalhos considerados mais relevantes que possuem alguma relação, direta ou indireta, com a nossa pesquisa e que foram úteis para o desenvolvimento da mesma;

- Capítulo 7: Conclusões. Apresenta as conclusões do trabalho desenvolvido, bem como sugestões de alguns trabalhos que podem ser realizados futuramente.

# Capítulo 2

# Fundamentação Teórica

Este capítulo tem a finalidade de apresentar os conceitos necessários para um melhor entendimento do presente trabalho. Inicialmente, serão abordadas definições sobre MDA e alguns dos formalismos que esta tecnologia engloba. Em seguida, serão apresentados conceitos básicos de mineração de dados e suas atividades envolvidas. Este capítulo também fala um pouco acerca de padrões e diretrizes em um âmbito geral. Por fim, será apresentada uma visão geral de GQM, que foi o método utilizado para a avaliação deste trabalho.

## 2.1 MDA (*Model-Driven Architecture*)

De acordo com [Mellor et al., 2004], "o modelo de um sistema é um conjunto de elementos que o descreve". Normalmente, utiliza-se uma linguagem de modelagem, como a UML, para representar tais modelos. Uma linguagem de modelagem, por sua vez, define quais elementos podem ser utilizados para a criação de modelos.

Segundo [Kleppe et al., 2003], MDA é um *framework* para desenvolvimento de *software* proposto pela OMG. A chave para a sua compreensão e utilização é a grande importância dada aos modelos no processo de desenvolvimento de *software.* A abordagem de MDA representa uma visão de MDD (*Model-Driven Development*) [Stahl et al., 2006], que por sua vez, expressa a idéia do desenvolvimento orientado a modelos.

O principal objetivo de MDA é deslocar o esforço e tempo empregados durante o ciclo de vida de um *software* das tarefas de implementação e testes para tarefas de modelagem,

meta-modelagem e transformações de modelos [Ramalho, 2007]. Em outras palavras, a finalidade é separar as especificações de um sistema dos detalhes de sua implementação. No intuito de alcançar tal objetivo, MDA sugere que os modelos de sistemas sejam especificados nas seguintes visões:

(i) CIM – especifica os modelos independentes de computação, representando o domínio e os requisitos do funcionamento do sistema;

(ii) PIM – especifica os requisitos e projeto de *software* independentes de qualquer plataforma de implementação;

(iii) PSM – especifica os modelos com todos os detalhes da plataforma para a qual o *software* será implementado;

(iv) Código – produto final e executável.

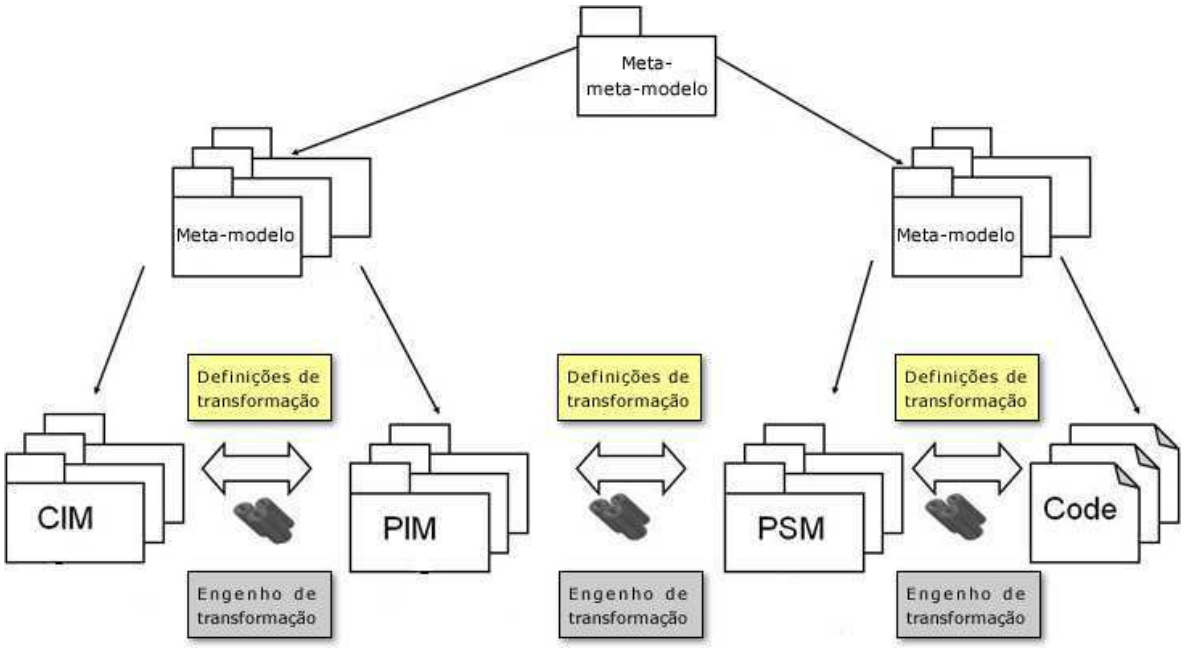



**Figura 1: Arquitetura MDA. Figura adaptada de [Ramalho, 2008].**

De acordo com a arquitetura MDA apresentada na Figura 1, o CIM serve de entrada para a geração do PIM, geralmente através de um processo manual. O PIM, por sua vez, serve de entrada para a geração do PSM. Finalmente, o PSM serve de entrada para geração do código de implementação. Vale destacar que MDA possibilita que as ferramentas transformem o PIM em um PSM ou diretamente em código. Além disso, também são

possíveis as transformações em outras direções, como por exemplo: PIM⇒CIM, PSM⇒PIM e Código⇒PSM.

A geração automática de um modelo destino a partir de um modelo origem ocorre por meio de transformações, de acordo com uma definição de transformação. Esta definição de transformação é um conjunto de regras que, juntas, descrevem como um modelo de origem (CIM, PIM, PSM ou código) pode ser transformado em um ou mais modelos de destino (CIM, PIM, PSM ou código). Estas regras são expressas através de uma linguagem de transformação que deve ser entendida por ferramentas denominadas engenhos de transformação, capazes de total e automaticamente executá-las. Como exemplos de linguagens de transformação podemos citar QVT (*Query/View/Transformation*) [QVT, 2008] e ATL (*ATLAS Transformation Language*) [ATL, 2008].

### 2.1.1 Meta-modelos

Meta-modelos são modelos que descrevem modelos. Estes formalismos são usados para definir a sintaxe abstrata de linguagens, tais como as linguagens de modelagem, de processo e de programação. Atualmente, os meta-modelos podem ser definidos através de duas principais linguagens existentes para meta-modelagem: MOF e Ecore. Um exemplo de meta-modelo bem definido e conhecido é o meta-modelo da UML, o qual provê os elementos dos modelos das aplicações. Em UML podemos usar classes, atributos, associações e outros tipos de elementos porque no seu meta-modelo existem esses elementos definidos.

Como um meta-modelo também é um modelo, ele mesmo deve ser descrito em uma linguagem bem definida, chamada meta-linguagem. Assim, o modelo que contém os elementos utilizados para a criação de um meta-modelo é denominado meta-meta-modelo.

A OMG definiu uma arquitetura de quatro camadas para representar os diferentes níveis de abstração existentes entre instâncias, modelos, meta-modelos e meta-meta-modelos. Como podemos observar na Figura 2, estas camadas são chamadas de M0, M1, M2 e M3. Vejamos a descrição de cada uma delas [Kleppe et al., 2003]:

(i) M0 (instâncias) – contém os dados da aplicação, sejam eles como objetos em um sistema orientado a objetos ou como registros em uma tabela de um banco de dados. Por exemplo, os objetos P1 e P2 como sendo instâncias da classe Pessoa.

Esta é a camada com o menor nível de abstração, representando o sistema em tempo de execução. Cada elemento desta camada é sempre uma instância de um elemento da camada M1;

(ii) M1 (modelos) – contém os modelos da aplicação. Por exemplo, um diagrama de classes UML que define a classe Pessoa. Cada elemento desta camada é sempre uma instância de um elemento da camada M2;

(iii) M2 (meta-modelos) – contém o meta-modelo de uma determinada linguagem. Por exemplo, o meta-modelo da UML. Cada elemento desta camada é sempre uma instância de um elemento da camada M3;

(iv) M3 (meta-meta-modelo) – contém o meta-meta-modelo que descreve os elementos que um meta-modelo pode exibir. Nesse sentido, a OMG definiu um modelo, chamado MOF, para especificar meta-modelos. Trata-se de um meta-meta-modelo responsável por definir elementos básicos para a criação de meta-modelos. Por exemplo, o meta-modelo da UML é uma instância do MOF. Os elementos desta camada são instâncias da própria camada, ou seja, a camada M3 se auto-descreve. Dessa forma, não há a necessidade de criar novas camadas de modelagem.

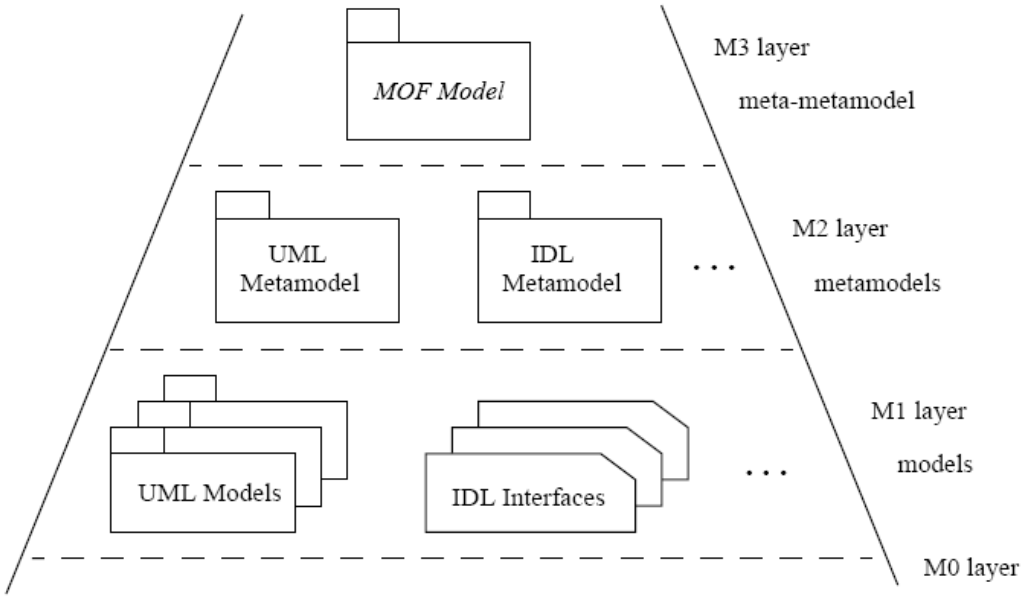



**Figura 2: Arquitetura em quatro camadas da OMG [MOF, 2002].**

### 2.1.2 MOF (*Meta Object Facility*)

MOF é um padrão definido pela OMG que pertence à camada M3 da arquitetura de MDA. Trata-se de uma linguagem para especificação de meta-modelos, sendo referenciada como um meta-meta-modelo. Este padrão "fornece um *framework* para o gerenciamento de meta-dados, bem como um conjunto de serviços de meta-dados que permitem o desenvolvimento e a interoperabilidade de sistemas dirigidos por modelos e meta-dados" [MOF, 2006].

Os meta-modelos utilizam MOF para definir a sintaxe abstrata do seu conjunto de construção de modelagem. É importante destacar que a semântica destes meta-modelos é especificada através da linguagem natural e de OCL. Como exemplos de meta-modelos definidos com MOF, podemos mencionar: SPEM (*Software and Systems Process Engineering Meta-Model*) [SPEM, 2008], CWM (*Common Warehouse Metamodel*) [CWM, 2003], U2TP (*UML 2 Testing Profile*) [U2TP, 2005] e UML.

O meta-modelo da UML, por sua vez, descreve exatamente como um modelo da UML é estruturado. Vejamos na Figura 3 um pequeno trecho deste meta-modelo.

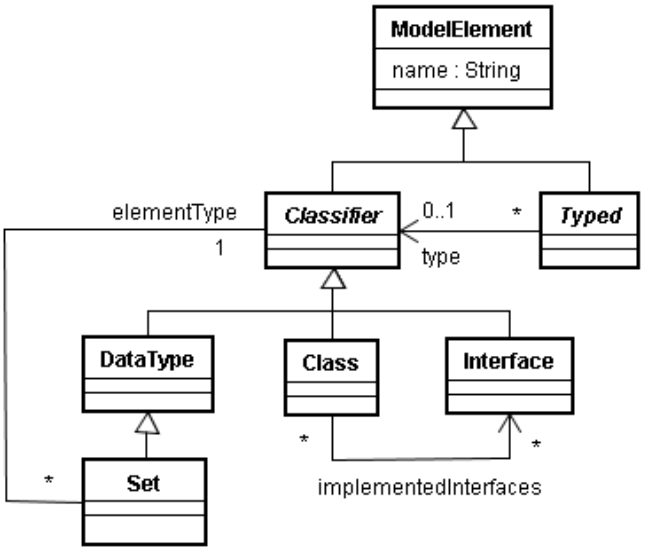



**Figura 3: Pequeno trecho do meta-modelo da UML [UML, 2007].**

A partir do trecho do meta-modelo da UML ilustrado na Figura 3, pode-se deduzir o seguinte: (i) todas as meta-classes presentes neste trecho são elementos do modelo (`ModelElement`) e possuem um nome (`name`); (ii) a meta-classe abstrata *`Classifier`* é uma generalização de uma classe (`Class`), interface (`Interface`) e tipo de dado (`DataType`); e (iii) uma classe (`Class`) pode implementar zero ou mais interfaces.

MOF desempenha um papel fundamental na infra-estrutura de MDA, pois, ao permitir que os mapeamentos das transformações sejam definidos em termos de construções MOF, é possível definir transformações entre modelos de diferentes meta-modelos.

### 2.1.3 OCL (*Object Constraint Language*)

Existem casos em que um diagrama UML não possui todas as informações relevantes para uma especificação. Lacunas desta natureza podem ser preenchidas adicionando anotações textuais aos modelos gerados. É neste contexto que entra a importância de se utilizar um padrão definido pela OMG chamado OCL.

"OCL é uma linguagem de restrição utilizada para descrever restrições em modelos de forma clara e precisa" [Warmer et al., 2003]. Além disso, ela especifica restrições em modelos de maneira formal. Esta linguagem é utilizada no processo de desenvolvimento de *software* no intuito de aumentar a expressividade de certos artefatos gerados durante a fase de análise e projeto do sistema.

Dessa forma, o modelo passará a ter maior precisão e formalidade. Os modelos gerados devem ser bem definidos no processo de desenvolvimento da abordagem de MDA, pois eles devem ser entendidos pelo computador e devem permitir que o processo de transformação entre modelos seja feito de maneira automatizada. Para atingir estas e outras finalidades, OCL torna-se um ingrediente chave dentro do processo de desenvolvimento MDA. Em contrapartida, o papel de OCL dentro da abordagem de MDA não está limitado apenas em aumentar a precisão dos modelos gerados. Esta linguagem de restrição também tem um importante papel na definição de meta-modelos, transformações entre modelos e novas linguagens através da utilização de perfis UML. Além disso, ela pode ser utilizada para outros propósitos, como para especificar: (i) valores iniciais de atributos; (ii) pré e pós-condições em operações; (iii) o corpo de operações de consulta; (iv) regras de derivação para atributos ou associações; e (v) invariantes em classes e tipos em um modelo de classes [Kleppe et al., 2003].

Como podemos perceber, a qualidade dos modelos pode ser melhorada por meio da combinação das linguagens UML e OCL, a primeira atuando na modelagem do sistema e a segunda na definição das regras e expressões aplicáveis ao modelo. Juntas, UML e OCL formam o coração de MDA, pois servem de base para muitos outros padrões desta infra-estrutura, como MOF e QVT.

## 2.2 Mineração de Dados

"A mineração de dados é a atividade de descoberta de padrões interessantes a partir de uma grande quantidade de dados armazenados em diferentes repositórios de dados" [Han et al., 2001]. Ela é seguida de várias técnicas e metodologias que atuam em conjunto para nos permitir analisar um grande número de dados e decidir qual informação é mais relevante, nos favorecendo na tomada de decisões com maior rapidez e precisão.

De acordo com [Han et al., 2001], o sistema da mineração de dados pode gerar muitos padrões[1], no entanto, um padrão é denominado interessante se ele for: facilmente entendido por humanos, válido em novos dados ou dados de teste com algum grau de certeza, potencialmente útil e novo. Um padrão também pode ser considerado interessante se ele validar a hipótese que o usuário pretende confirmar. Enfim, um padrão interessante representa conhecimento.

### 2.2.1 Processo KDD

Muitas pessoas consideram o termo "mineração de dados" como sinônimo de Descoberta de Conhecimento em Banco de Dados (*Knowledge Discovery in Databases* - KDD). Na verdade, KDD é o processo completo que engloba várias etapas para a descoberta de conhecimento, enquanto que mineração de dados é apenas uma das etapas deste processo. Segundo [Fayyad et al., 1996], "descoberta de conhecimento em bancos de dados é o processo não trivial de identificar padrões em dados que sejam válidos, novos (previamente desconhecidos), potencialmente úteis e compreensíveis, visando melhorar o entendimento de um problema ou um procedimento de tomada de decisão".

Vale ressaltar que, apesar de ser apenas uma das etapas do processo KDD, o termo "mineração de dados" é uma expressão costumeiramente usada pelas pessoas para referenciar o processo KDD como um todo. Nesse sentido, é importante destacar que, apesar de termos aplicado todas as etapas deste processo em nosso trabalho, utilizamos este mesmo termo para nos referirmos ao processo KDD como um todo ao longo deste documento.

[1] O termo "padrão" utilizado na Mineração de Dados não é o mesmo utilizado na Engenharia de *Software* para denominar padrões (*patterns*), tais como, os padrões de projeto, padrões de teste, dentre outros. Na Mineração de Dados, um padrão representa o conhecimento descoberto nos dados. Já em nosso trabalho, quando falamos em padrões ou diretrizes, estamos nos referindo ao termo utilizado na Engenharia de *Software*.

O processo KDD é interativo, iterativo e exploratório, envolvendo várias etapas. A Figura 4 ilustra cada uma delas, vejamos uma breve descrição:

(i) Seleção dos Dados (*Selection*) – um conjunto de dados relevantes ao usuário deve ser selecionado, o qual funcionará como alvo para a realização das descobertas;

(ii) Pré-processamento (*Preprocessing*) – esta fase consiste na eliminação de ruídos, dados estranhos ou inconsistentes, como também, na preparação dos dados, atuando na limpeza, integração e formatação;

(iii) Transformação (*Transformation*) – uma transformação ocorre nos dados pré-processados para que eles sejam armazenados de maneira adequada, isto é, no formato apropriado para aplicação de algoritmos de mineração;

(iv) Mineração de Dados (*Data Mining*) – nesta fase do processo alguns algoritmos são executados com o intuito de capturar padrões de interesse existentes nos dados;

(v) Interpretação e Avaliação (*Interpretation/Evaluation*) – com a finalidade de verificar as novas descobertas, os padrões identificados na etapa de Mineração de Dados passarão por uma interpretação e avaliação de acordo com algum critério do usuário;

(vi) Implantação do Conhecimento (*Knowledge*) – por fim, o conhecimento adquirido no decorrer das etapas anteriores deverá ser documentado e encaminhado aos interessados.

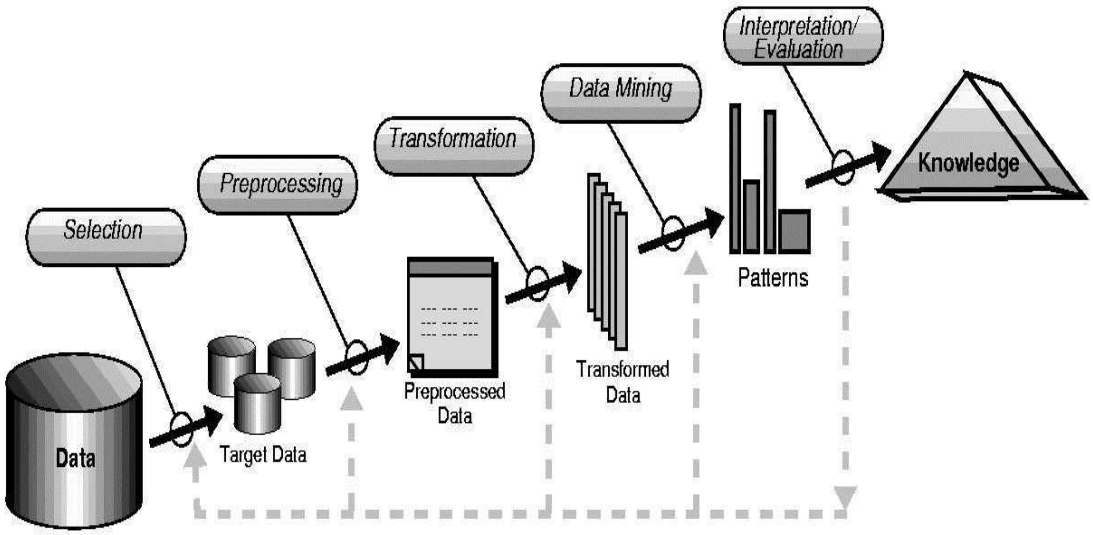



**Figura 4: Etapas do processo KDD [Fayyad et al., 1996].**

### 2.2.2 Tarefas da Mineração de Dados

Uma tarefa da mineração de dados consiste na especificação do que estamos querendo buscar nos dados, que tipo de regularidades ou categoria de padrões temos interesse em encontrar, ou que tipo de padrões poderiam nos surpreender. Dentre as tarefas existentes na mineração de dados podemos citar a Classificação, Análise de Associação, Agrupamento e Regressão.

Em nosso trabalho, utilizamos a tarefa de análise de associação juntamente com o algoritmo *Apriori* [Agrawal et al., 1994] no intuito de identificarmos diretrizes para meta-modelos. Por este motivo, descreveremos em detalhes apenas esta tarefa e o *Apriori* no tópico a seguir.

#### Análise de Associação e o Algoritmo *Apriori*

A tarefa de análise de associação tem a finalidade de encontrar tendências que podem ser usadas para entender e explorar padrões no comportamento dos dados. A utilização desta tarefa em uma base de dados pode gerar um grande número de regras de associação. Uma regra de associação é uma regra na forma $X \rightarrow Y$, onde $X$ e $Y$ são conjuntos de itens [Han et al., 2001]. Isto significa dizer que se $X$ ocorrer em um registro (instância) na base de dados, então $Y$ também tende a ocorrer. O conjunto de itens $X$ é o antecedente da regra, enquanto o conjunto de itens $Y$ é o conseqüente. O antecedente e o conseqüente são conjuntos disjuntos, ou seja, eles não possuem itens em comum.

Algumas das regras geradas a partir da tarefa de análise de associação podem não ser interessantes, pois elas podem ocorrer com uma baixa freqüência nos dados ou ter pouca confiança. Desta forma, para nos auxiliar na análise das regras geradas podemos considerar duas importantes métricas desta tarefa: o *suporte* e a *confiança*. Elas determinam a quantidade e qualidade de tais regras, respectivamente.

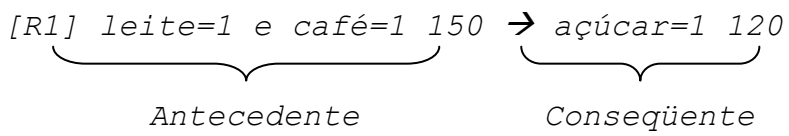
Considerando que os conjuntos de itens *X* e *Y* estão sendo avaliados, o *suporte* indica a freqüência com que o antecedente (*X*) e o conseqüente (*Y*) da regra ocorrem juntos em relação à totalidade de registros da base de dados [Agrawal et al., 1993]. O suporte de uma regra pode ser calculado da seguinte forma:

$$\text{suporte(R)} = \frac{\text{núm. de registros } (X \text{ e } Y)}{\text{núm. total de registros}}$$

A *confiança* de uma regra indica o percentual de registros que satisfaz o antecedente (*X*) e o conseqüente (*Y*) em relação ao número de registros que satisfaz o antecedente, medindo a força da regra ou a sua precisão [Agrawal et al., 1993]. Ela pode ser calculada da seguinte forma:

$$\text{confiança(R)} = \frac{\text{núm. de registros } (X \text{ e } Y)}{\text{núm. de registros } (X)}$$

Vejamos um simples exemplo na regra R1, considerando uma base de dados com um total de 200 registros, onde cada campo pode assumir valor 0 ou 1, indicando se ele está ausente ou presente no registro, respectivamente. O antecedente da regra R1 diz que `leite` e `café` foram comprados juntos em 150 registros da base de dados. O conseqüente da regra R1 diz que quem comprou `leite` e `café` também comprou `açúcar` em 120 registros da base de dados. Portanto, para sabermos a freqüência com que estes três itens ocorrem juntos na base de dados temos que calcular o suporte, dividindo-se o número de registros que contém os três itens pelo número total de registros da base: 120/200 = 0.60. Por outro lado, para sabermos o grau de certeza com que estes três itens ocorrem juntos temos que calcular a confiança, dividindo-se o número de registros que contém tais itens pelo número de registros com que os itens do antecedente ocorrem juntos: 120/150 = 0.80. Portanto, esta regra pode ser interpretada da seguinte forma: com 60% de suporte e 80% de certeza, podemos afirmar que, em determinado supermercado, quem compra leite e café geralmente compra açúcar.




A ferramenta Weka (apresentada na subseção 2.2.3) nos permite configurar vários parâmetros a respeito dos algoritmos de análise de associação e, dentre eles, estão o limite mínimo e máximo de suporte e confiança. Desta forma, é possível definirmos os valores de tais métricas para obtermos conjuntos diversificados de regras.

Existem vários algoritmos apropriados para a tarefa de análise de associação, como por exemplo, o *Apriori*. Proposto por [Agrawal et al., 1994], este algoritmo é o mais utilizado para minerar regras de associação nas bases de dados e foi desenvolvido, especificamente, para trabalhar com este tipo de aplicação. O algoritmo *Apriori* é basicamente composto por duas etapas: (i) encontrar todos os conjuntos de itens freqüentes, ou seja, com valor de suporte superior ou igual ao suporte mínimo especificado pelo usuário. Em termos de custo, esta é a etapa mais pesada; e (ii) usar os conjuntos de itens freqüentes para induzir as regras de associação, com valores de suporte e confiança superior ou igual ao suporte e confiança mínimos especificados pelo usuário.

### 2.2.3 Weka: Uma Ferramenta para Mineração de Dados

Atualmente, existem diversas ferramentas que auxiliam na mineração de dados, como por exemplo a Weka, que foi desenvolvida na Universidade de Waikato, Nova Zelândia. A Weka é uma coleção de algoritmos de aprendizagem de máquina para tarefas de mineração de dados [Weka, 2009]. Dentre os diversos algoritmos que a Weka implementa podemos citar alguns de análise de associação: *Apriori*, *FilteredAssociator* [FilteredAssociator, 2010], *PredictiveApriori* [PredictiveApriori, 2010] e *Tertius* [Tertius, 2010].

É importante destacar que a Weka aceita vários formatos de arquivos para a entrada dos dados (base de dados), como por exemplo, ARFF (*Attribute-Relation File Format*), CSV (*Comma-Separated Values*), C4.5 e binário. Por outro lado, ela não trabalha com bases de dados relacionais. O formato mais comumente utilizado nesta ferramenta como base de dados é o ARFF. Trata-se de um arquivo texto composto por três partes: (i) *relation*, referente ao nome da base de dados; (ii) *attribute*, referente aos atributos (campos) e seus respectivos tipos; e (iii) *data*, referente aos registros (instâncias), isto é, ao conjunto de dados. O arquivo

ARFF é, na verdade, uma tabela única contendo todos os dados de forma desnormalizada. Vejamos um exemplo deste arquivo na Figura 5, a seguir.

```
@relation heart-disease-simplified

@attribute age numeric
@attribute sex { female, male}
@attribute chest_pain_type { typ_angina, asympt, non_anginal, atyp_angina}
@attribute cholesterol numeric
@attribute exercise_induced_angina { no, yes}
@attribute class { present, not_present}

@data
63,male,typ_angina,233,no,not_present
67,male,asympt,286,yes,present
67,male,asympt,229,yes,present
38,female,non_anginal,?,no,not_present
...
```

**Figura 5: Exemplo de arquivo ARFF [ARFF, 2008].**

A Weka é um *software* livre distribuído sob a licença universal *GNU General Public License* [GNU, 2008]. Dentre as principais vantagens que ele oferece está a liberdade do código, de forma que seus usuários possam implementar novos algoritmos ou modificar os existentes para uma melhor adaptação às suas necessidades. Além dos recursos interessantes que a ferramenta oferece, há ainda uma vasta documentação *on-line* que apresenta, de forma detalhada, todas as suas funcionalidades e explica como utilizar cada uma delas. Toda a documentação do projeto Weka é gerada automaticamente a partir do código-fonte, sendo assim, mesmo a ferramenta estando em crescimento contínuo, sua documentação estará sempre atualizada.

## 2.3 Padrões e Diretrizes

Um padrão (*pattern*) no âmbito da computação visa de uma forma geral apresentar regras, processos, serviços e correlatos que contribuam para que haja a reutilização de soluções analisadas e aprovadas em problemas usuais no desenvolvimento de aplicações, evitando duplicação de trabalho. Em princípio, os padrões buscam isolar as partes que são estáveis daquelas que são alteradas com freqüência. O resultado que se espera com isso é ter aplicações que são mais fáceis de manter, compreender e reutilizar.

Os padrões nascem a partir da necessidade de se identificar e registrar soluções para problemas que podem ocorrer diversas vezes em escopos diferentes, de forma que as soluções desenvolvidas e conhecidas por especialistas sejam bem definidas, testadas e documentadas. Dessa forma, tornando-se padrões por serem reutilizadas várias vezes em projetos diferentes e por terem eficácia comprovada. Como exemplos de padrões temos os Padrões de Projeto, criados por Erich Gamma, John Vlissides, Ralph Jonhson e Richard Helm, os quais ficaram conhecidos como "*Gang of Four*". Estes padrões descrevem soluções para problemas recorrentes no desenvolvimento de sistemas orientados a objetos, de maneira que uma mesma solução possa ser usada para resolver problemas ocorridos em âmbitos diferentes.

Além do aspecto reutilizável dos padrões, temos ainda outras motivações para a sua utilização:

(i) Aprendizagem com a experiência de outras pessoas – uma vez que problemas comuns em engenharia de *software* são identificados e catalogados, outras pessoas podem utilizar estas soluções testadas e bem documentadas;

(ii) Vocabulário comum – a definição de um vocabulário comum para a discussão de problemas e soluções de projeto torna o sistema menos complexo, uma vez que será adotado um nível mais alto de abstração;

(iii) Facilidade na documentação e compreensão – a compreensão de sistemas existentes fica mais fácil quando os padrões adotados são conhecidos, facilitando na manutenção da arquitetura do *software*;

(iv) Refatoramento – com o uso de padrões desde o início do projeto, a necessidade de refatoramento pode diminuir.

Por outro lado, o conceito de diretrizes é mais limitado que o de padrões. Diretrizes são guias que servem para auxiliar e orientar as pessoas na realização de determinadas atividades. Assim como um padrão, uma diretriz também é definida no intuito de se reutilizar uma mesma solução que é adotada para resolver problemas recorrentes. Contudo, diretrizes não são consideradas padrões. Um padrão tem uma definição mais rigorosa e, muitas vezes, é baseado em um conjunto de diretrizes. Segundo a definição encontrada em [Barroso, 1994], uma diretriz é "conjunto de instruções ou indicações para se tratar e levar a termo um plano, uma ação, um negócio".

As diretrizes podem atuar em diversos contextos. Por exemplo, existem diretrizes que indicam como se deve redigir uma boa redação, como falar bem em público, como se fazer uma boa apresentação de projeto, dentre outras. Na área de computação, podemos citar as diretrizes para boas práticas de programação [Balena et al., 2005], para se fazer a modelagem de um domínio [Šilingas, 2006] e para se fazer uma pesquisa empírica em engenharia de *software* [Kitchenham et al., 2002]. Além disso, processos de desenvolvimento de *software* freqüentemente definem várias diretrizes para ajudar os desenvolvedores durante as atividades ao longo das fases de todo o processo, tais como especificação, implementação e testes. Por exemplo, [Carr, 2005] propõe várias diretrizes direcionadas ao processo de desenvolvimento ágil. Por outro lado, [Rational, 1998] apresenta diretrizes voltadas para o desenvolvimento de *software* utilizando RUP (*Rational Unified Process*) [RUP, 2010].

No contexto de meta-modelagem, as diretrizes direcionam os desenvolvedores na tarefa de meta-modelagem, assim, melhorando a qualidade dos meta-modelos construídos e facilitando outras atividades de MDA relacionadas, tais como a verificação de semântica estática, a construção de modelos e a especificação de transformações.

## 2.4 A Abordagem GQM (*Goal, Question, Metric*)

Atualmente, existem várias metodologias que propõem avaliar processos experimentais. Uma abordagem que atua nesse sentido é o GQM. De acordo com [Basili et al., 2001], GQM é definido como "um método orientado a objetivos para o desenvolvimento e a manutenção de um programa significativo de métricas que se baseia em três níveis: objetivos, questões e métricas". Trata-se de uma abordagem que apóia a definição e implementação *top-down* de metas operacionais e mensuráveis para a melhoria de *software*, como também apóia a

interpretação *bottom-up* dos dados coletados [Briand et al., 1997]. GQM tem sido bastante adotado para medir e melhorar a qualidade em organizações de desenvolvimento de *software*.

A abordagem GQM define um modelo de avaliação dividido em três níveis de realização [Basili et al., 2001]:

- Nível Conceitual (*Goal*): objetiva definir o escopo da avaliação, isto é, o objeto que será medido, como por exemplo: processos, produtos ou recursos;
- Nível Operacional (*Question*): objetiva elaborar questões a fim de caracterizar o objeto de medição para identificar os itens e critérios de qualidade esperados;
- Nível Quantitativo (*Metric*): objetiva definir dados relacionados às questões elaboradas no nível anterior, para que se possa respondê-las de forma quantitativa.

Alguns autores acreditam que a utilização de GQM deve ser vista em termos de fases, cada uma associada a um conjunto de atividades. Estas fases devem ser integradas com o planejamento e gerenciamento do projeto, além de possuírem relações de dependência umas com as outras. Nesse sentido, [Solingen et al., 1999] define quatro fases para o método GQM:

- Definição: visa definir e documentar os objetivos, as questões, as métricas e as hipóteses do processo de avaliação;
- Planejamento: visa elaborar o plano de projeto. Dentre outras atividades, ela envolve a seleção do projeto que será avaliado, a seleção e treinamento dos membros da equipe, bem como a definição, caracterização e planejamento do projeto;
- Coleta de Dados: visa coletar os dados a partir da execução do experimento com base nas métricas definidas, resultando em informações prontas para a interpretação;
- Interpretação: visa analisar os dados coletados anteriormente, considerando os objetivos, as questões e as métricas definidas.

A utilização de GQM para a avaliação de um experimento oferece diversos benefícios, como por exemplo, o melhor entendimento dos objetivos, melhor comunicação e interação entre os membros da equipe de projeto, melhor definição e execução do processo, ganhos financeiros, dentre outros [Solingen et al., 1999]. Todas estas vantagens levam a uma considerável melhoria no grau de qualidade do processo como um todo.

# Capítulo 3

# Abordagens para Descoberta de Diretrizes

Este capítulo apresenta as duas abordagens utilizadas para descobrirmos diretrizes em meta-modelos. A primeira foi automática, a partir da aplicação de técnicas de mineração de dados e com o auxílio de uma ferramenta de suporte que desenvolvemos, enquanto que a segunda foi manual, a partir de uma análise detalhada em um conjunto de meta-modelos.

## 3.1 Automática: Aplicando Técnicas de Mineração de Dados

A utilização da mineração de dados como mecanismo para a descoberta automática de conhecimento requer a realização de algumas etapas, que vão desde a preparação da base de dados (com as informações necessárias para a descoberta) até a implantação do conhecimento obtido. Nesse sentido, o processo seguido neste trabalho para a identificação automática das diretrizes foi resumido em quatro etapas. Vejamos uma breve descrição de cada uma delas, comparando-as com as etapas do processo KDD:

(i) Preparação da Base de Dados – responsável por definir a estrutura e os dados que constituíram a base, a qual foi construída no formato adequado para a execução de algoritmos. Ela envolve as etapas de Seleção, Pré-processamento e Transformação do processo KDD, suas atividades serão melhor abordadas na subseção 3.1.1;

(ii) Mineração de Dados – responsável por executar os algoritmos de análise de associação (uma das tarefas da mineração de dados) e apresentar as regras encontradas. Ela se refere à etapa de Mineração de Dados do processo KDD, suas atividades serão melhor abordadas na subseção 3.1.2;

(iii) Interpretação/Avaliação – responsável por interpretar as regras de associação encontradas depois da mineração dos dados no intuito descobrir conhecimento suficiente para a concepção de diretrizes para meta-modelos. Ela se refere à etapa de Interpretação/Avaliação do processo KDD, suas atividades serão melhor abordadas na subseção 3.1.3;

(iv) Implantação do Conhecimento – responsável por implantar, em forma de diretrizes para meta-modelos, todo o conhecimento obtido. Ela se refere à etapa de Implantação do Conhecimento do processo KDD, suas atividades serão melhor abordadas na subseção 3.1.3.

### 3.1.1 Preparação da Base de Dados

Para um melhor entendimento de como ocorreu a construção da nossa base de dados, esta subseção explica as atividades realizadas em três partes: (i) seleção dos campos e dos dados; (ii) definição de uma base de dados relacional; e (iii) construção da base de dados: arquivo ARFF.

#### 3.1.1.1 Seleção dos Campos e dos Dados

Todo o procedimento de seleção dos campos usados para estruturar a base de dados ocorreu manualmente, a partir de uma análise da especificação de MOF 2.0. Durante esta análise, selecionamos os campos mais relevantes, isto é, os elementos de MOF mais comumente utilizados para a especificação de meta-modelos, como por exemplo, meta-classes, propriedades, operações e associações entre meta-classes. Além disso, com base em algumas suspeitas que tivemos, escolhemos também os campos que poderiam ser mais favoráveis para a descoberta de diretrizes interessantes. Alguns exemplos destes campos são apresentados na Figura 6, na qual podemos ver duas meta-classes do meta-modelo de MOF com alguns atributos que foram utilizadas como campos na nossa base de dados.

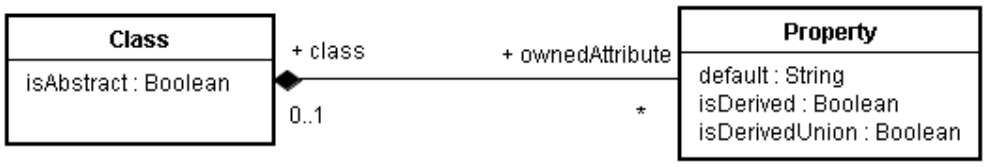



**Figura 6: Exemplo de alguns campos da base de dados retirados de MOF.**

A especificação de MOF define um meta-modelo que deve ser seguido para a definição de meta-modelos, ou seja, alguns meta-modelos são construídos em conformidade com o MOF – estes são os chamados meta-modelos MOF. Por este motivo, a especificação de MOF foi escolhida para a extração dos campos da base de dados e, além disso, por ela ser o padrão oficial definido pela OMG para a definição de meta-modelos.

Por outro lado, o conjunto de dados utilizado para povoar a base foi extraído da especificação da UML 2.1.2, que é mais um padrão definido pela OMG. O meta-modelo da UML descrito nesta especificação está disponível também em um arquivo no formato XMI (*XML Metadata Interchange*), o qual foi utilizado como fonte para a extração das informações necessárias para povoar a base. Todo o procedimento para a seleção dos dados foi realizado automaticamente, por meio de uma ferramenta de suporte que desenvolvemos.

Uma vez que os campos tenham sido definidos, a ferramenta de suporte: (i) obtém os dados do arquivo XMI (etapa de Seleção do processo KDD); (ii) elimina qualquer inconsistência presente neles, bem como preenche os campos que possuírem valores vazios com seus respectivos valores *default* definidos na especificação de MOF (etapa de Pré-processamento do processo KDD); e (iii) gera a base de dados povoada no formato adequado para a execução dos algoritmos de mineração de dados (etapa de Transformação do processo KDD).

É importante deixarmos bem claro os papéis das especificações de MOF e de UML neste trabalho: a primeira foi utilizada para extrairmos os campos da base de dados e a última para extrairmos as instâncias (registros) que serviram para povoar esta base, as quais constituem relacionamentos, meta-classes, propriedades, operações, enfim, todos os elementos que envolvem o meta-modelo da UML [Vieira et al.].

#### 3.1.1.2 Definição de uma Base de Dados Relacional

Antes de definirmos e construirmos a estrutura de dados que, de fato, seria utilizada para a execução dos algoritmos, definimos uma base de dados relacional. A definição desta base foi

realizada maualmente, ela não foi construída em nenhum SGBD (Sistema de Gerenciamento de Banco de Dados). Esta base de dados relacional teve a finalidade de nos ajudar a entender melhor os conceitos e a relação existente entre os elementos da especificação de MOF: as meta-classes, os relacionamentos, as propriedades, dentre outros. Além disso, ela nos ajudou a pensarmos em uma maneira de estruturarmos, adequadamente, as tabelas e os campos no formato de dados aceito pela ferramenta de mineração de dados que adotamos.

Para construirmos a base de dados relacional foi necessário criamos uma tabela para cada meta-classe da especificação de MOF. Além disso, as propriedades (atributos) de cada meta-classe foram transformadas em campos da tabela. No entanto, é importante notarmos que, desta forma, há um problema na construção da base de dados relacional: ao criarmos uma tabela para cada meta-classe perdemos algumas informações sobre as propriedades que, por ventura, esta meta-classe tenha herdado de outra. Por esta razão, observamos todos os casos de herança existentes nas meta-classes e, manualmente, copiamos todas as propriedades das superclasses para suas respectivas subclasses, sejam elas concretas ou abstratas. Este procedimento foi realizado até o último nível de herança, acumulando todas as propriedades herdadas nas subclasses.

Nesse sentido, vejamos dois exemplos de tabelas relacionais na Figura 7, criadas a partir das meta-classes da especificação de MOF ilustradas na Figura 6 da subseção anterior. O campo `name` da tabela `Class` foi obtido a partir da meta-classe *`Type`* que, por sua vez, estende a meta-classe *`NamedElement`*, a qual, de fato, possui a propriedade `name`. Da mesma forma, o campo `package` da tabela `Class` foi obtido a partir da meta-classe *`Type`*, pois a meta-classe `Class` estende *`Type`*. Por fim, o campo `isAbstract` desta tabela foi obtido da própria meta-classe `Class`. Já na tabela `Property`, os campos `type` e `name` foram obtidos a partir da meta-classe *`TypedElement`*, pois `Property` estende esta meta-classe que, por sua vez, possui a propriedade `type` e estende *`NamedElement`*, a qual, de fato, possui a propriedade `name`. Os demais campos da tabela `Property` foram obtidos da própria meta-classe `Property`.

| Tabela: Class | |
|---|---|
| isAbstract | bool |
| name | varchar |
| package | varchar |

| Tabela: Property | |
|---|---|
| type | varchar |
| name | varchar |
| default | varchar |
| isDerived | bool |
| isDerivedUnion | bool |

**Figura 7: Exemplo de algumas tabelas relacionais.**

### 3.1.1.3 Construção da Base de Dados: Arquivo ARFF

Embora a base de dados relacional, descrita anteriormente, tenha nos oferecido um melhor entendimento dos elementos da especificação de MOF, ela não pôde ser utilizada pela Weka, pois esta ferramenta não suporta esse tipo de base.

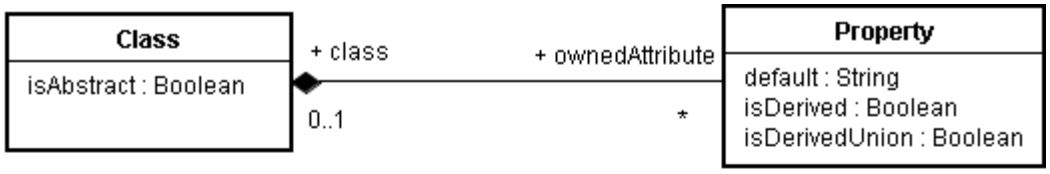
Para a escolha da ferramenta da qual reutilizamos os algoritmos para a mineração de dados, foi realizado um estudo comparativo entre diversas ferramentas existentes no mercado: Weka, Tanagra [Tanagra, 2005], Orange [Orange, 2010] e KDB2000 [KDB2000, 2009]. Este estudo considerou vários critérios e pode ser visto na íntegra em [Leal e Ramalho, 2009]. Como resultado, selecionamos a Weka por diversos motivos, dentre eles, por ser uma ferramenta livre e amplamente adotada para fins acadêmicos. Portanto, alguns algoritmos implementados na Weka foram reutilizados na ferramenta de suporte que desenvolvemos neste trabalho para nos auxiliar na etapa de mineração de dados.

Tendo em vista que a Weka não suporta bases de dados relacionais, decidimos adotar o formato de arquivo de dados mais popular aceito por esta ferramenta para representar a nossa base: o ARFF. A construção do ARFF foi realizada por meio de duas atividades: (i) a aglutinação, de forma manual, das tabelas relacionais em apenas uma única tabela; e (ii) a geração automática do ARFF com o auxílio da ferramenta de suporte que desenvolvemos.

A aglutinação foi realizada para montarmos a estrutura do arquivo ARFF. Cada registro nesta tabela única contém informações sobre uma meta-classe específica, suas propriedades, seus relacionamentos, dentre outras. Vale ressaltar que o arquivo ARFF é composto por dados não normalizados, isto é, nenhuma das formas normais é aplicada para este tipo de base de dados. O processo de aglutinação é ilustrado, a seguir, com um simples exemplo. A Figura 8-A apresenta um trecho do meta-modelo de MOF com uma associação de composição entre as meta-classes `Class` e `Property`, ilustrando apenas algumas das propriedades. Para cada uma destas meta-classes foi criada uma tabela relacional, como mostra a Figura 8-B e a Figura 8-C. Por fim, as duas tabelas relacionais foram aglutinadas em

apenas uma, chamada *BigTable*. O resultado pode ser conferido na Figura 8-D, uma única tabela construída com todos os campos pertencentes às tabelas `Class` e `Property`.

A




B

| Tabela: Class | |
|---|---|
| isAbstract | bool |
| name | varchar |
| package | varchar |

C

| Tabela: Property | |
|---|---|
| type | varchar |
| name | varchar |
| default | varchar |
| isDerived | bool |
| isDerivedUnion | bool |

D

| Tabela: BigTable | |
|---|---|
| class.isAbstract | bool |
| class.name | varchar |
| class.package | varchar |
| class.property.type | varchar |
| class.property.name | varchar |
| class.property.default | varchar |
| class.property.isDerived | bool |
| class.property.isDerivedUnion | bool |

**Figura 8: A. Trecho do meta-modelo de MOF com duas meta-classes; B. Tabela relacional para a meta-classe Class; C. Tabela relacional para a meta-classe Property; D. A tabela única.**

Vale salientar que, para este simples exemplo, as tabelas relacionais ilustraram somente alguns dos seus campos e, dentre eles, estão alguns que foram herdados de superclasses, como explicado na subseção anterior.

Como conseqüência da aglutinação das tabelas relacionais em apenas uma e considerando alguns campos adicionais que criamos, obtivemos uma única tabela contendo um total de 96 campos. Eles foram povoados com os elementos do meta-modelo da UML (instanciando os elementos de MOF descritos na tabela única), o que gerou uma base de dados com 57.037 registros. Todos os campos são do tipo nominal, que é um dos tipos de dados suportados pela ferramenta Weka. Na Tabela 1, vejamos a descrição de alguns deles. A tabela completa com todos os campos pode ser conferida no Apêndice A deste documento.

**Tabela 1: Alguns dos campos presentes na base de dados.**

| Campo | Descrição |
|---|---|
| *class.isAbstract* | Especifica se uma meta-classe é abstrata. |
| *class.isNavigable* | Especifica se uma meta-classe é navegável a partir de outra em um relacionamento. |
| *class.lower* | Especifica o limite mínimo de multiplicidade de uma meta-classe. |
| *class.upper* | Especifica o limite máximo de multiplicidade de uma meta-classe. |
| *class.composition* | Especifica se uma meta-classe: (i) representa "o todo" (*Owner*) de uma associação de composição; (ii) representa "a parte" (*Owned*) desta associação; ou (iii) não possui associação de composição (*None*). |
| *class.isReadOnly* | Especifica se uma meta-classe pode ser escrita depois de sua inicialização. |
| *class.property.isDerived* | Especifica se o valor do atributo de uma meta-classe é derivado de outra informação. |
| *class.operation.parameters.direction* | Especifica a direção dos parâmetros de uma operação, que pode ser: (i) *in* – de entrada; (ii) *inout* – de entrada/saída; (iii) *out* – de saída; ou (iv) *return* – de retorno. |

Depois que toda a estrutura do arquivo ARFF foi definida, utilizamos a ferramenta de suporte que desenvolvemos para gerar, automaticamente, a base de dados completa: o arquivo ARFF contendo todos os campos e dados. Em outras palavras, a ferramenta preenche a nossa tabela única com as instâncias (dados).

### 3.1.2 Mineração dos Dados

Em seguida, serão apresentadas as regras de associação consideradas mais interessantes obtidas com a mineração dos dados, as quais serão interpretadas na subseção seguinte. Para a etapa de mineração de dados do processo KDD, decidimos adotar a tarefa de análise de associação, uma vez que a nossa finalidade era encontrar regras que apresentassem correlações e associações interessantes entre os elementos do meta-modelo da UML. Dentre os algoritmos de análise de associação, selecionamos o *Apriori* devido à sua simplicidade e versatilidade em grandes bases de dados e, além do mais, por ele possuir estruturas que oferecem uma boa flexibilidade para a geração de regras.

**Análise de Associação**

Na tarefa de análise de associação, o algoritmo *Apriori* permite que o usuário configure seus parâmetros antes de sua execução, tais como a confiança mínima, o suporte mínimo, o número de campos da base de dados e o número de regras a serem geradas. Estes parâmetros devem ser definidos de acordo com o tipo de regra que o usuário deseja obter.

Inicialmente, o nosso objetivo era obter regras mais fortes (alta confiança) e mais recorrentes (alto suporte), envolvendo 36 campos específicos da base de dados. Portanto, configuramos estes parâmetros com valores bem altos para o mínimo de suporte e confiança (90% para ambos), e definimos um total de 36 campos da nossa base de dados, como mostra a Tabela 2. Com esta nossa configuração inicial obtivemos um total de 1.000 regras de associação, cada uma delas envolvendo alguns dos 36 campos informados. Contudo, muitas destas regras eram redundantes e não apresentavam nenhum conhececimento inovador. Portanto, no intuito de obtermos regras mais diversificadas (com valores variados para o suporte e a confiança) e envolvendo diferentes combinações dos campos, reduzimos gradativamente os valores de cada um dos parâmetros. Desta forma, conseguimos obter uma quantidade bem maior de regras de associação. A Tabela 2 apresenta algumas das configurações que realizamos para a execução do *Apriori*. Como podemos observar, quanto menor os valores que informamos para os parâmetros, maior o número de regras geradas, uma vez que aumenta a variedade de regras.

**Tabela 2: Algumas configurações do Apriori realizadas na mineração de dados.**

| Confiança Mín. | Suporte Mín. | Num. de Campos | Num. de Regras |
|---|---|---|---|
| 90% | 90% | 36 | 1.000 |
| 85% | 80% | 15 | 10.000 |
| 80% | 50% | 11 | 50.000 |
| 70% | 15% | 8 | 80.000 |

É importante destacar que, embora o Apêndice A deste documento apresente um total de 96 campos para a nossa base de dados, o algoritmo *Apriori* não foi executado considerando

todos eles de uma só vez[2]. A cada execução selecionamos combinações diferentes destes campos, a fim de obtermos regras mais diversificadas.

No intuito de encontrarmos regras mais interessantes e diferentes daquelas geradas pelo *Apriori*, outros algoritmos também foram executados, como por exemplo, o *FilteredAssociator*, o *PredictiveApriori* e o *Tertius*. Contudo, observamos que as regras geradas foram praticamente as mesmas que obtivemos com o *Apriori*.

Antes de partirmos para a análise das regras de associação, é importante esclarecermos os conceitos referentes ao `owned` e `owner` de uma associação de composição. No exemplo que segue, vejamos o que cada um deles representa. A Figura 9 mostra uma associação de composição entre as meta-classes `Class` e `Property` pertencentes ao meta-modelo de MOF. A primeira meta-classe representa o "todo" desta associação, isto é, o `owner`. Já a segunda representa a "parte", isto é, o `owned`.

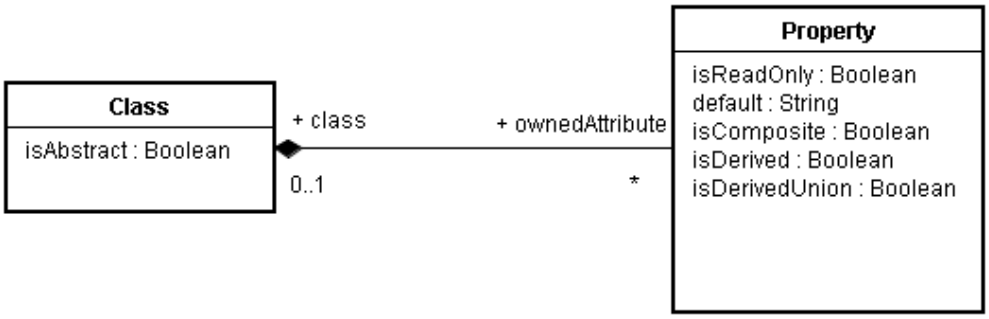



**Figura 9: Exemplo de *Owner* e *Owned* em uma associação de composição.**

Dentre as regras de associação obtidas com a execução do *Apriori* serão descritas as regras R1, R2, R3, R4, R5, R6 e R7, nas quais todos os seus antecedentes se referem ao mesmo contexto: associação de composição entre meta-classes (*class.composition*). As quatro primeiras regras mostram informações sobre meta-classes que representam a parte (`owned`) de uma associação de composição. Já as três últimas, ilustram informações sobre meta-classes que representam o todo (`owner`). A seguir, vejamos a descrição de cada uma destas regras. Vale observar que cada uma delas apresenta um valor numérico no seu (i) antecedente, que é o número de registros da base de dados que contem o(s) item(ns) do antecedente; e (ii)

[2] Alguns campos foram criados a partir da derivação de outros, e até mesmo a partir da união de dois ou mais campos, como por exemplo, o campo `class.isCompositionOwnerLower1` foi criado a partir da união dos campos `class.isOwner` e `class.lower`. Foi por este motivo que não executamos o algoritmo *Apriori* com a totalidade dos campos.

conseqüente, que é o número de registros da base de dados que contem o(s) item(ns) tanto do antecedente quanto do conseqüente.

A regra R1 afirma, com 19% de suporte e 100% de confiança, que quando uma meta-classe representar a parte da associação de composição ela deve ser sempre navegável. Vale ressaltar que, embora o valor de suporte desta regra tenha sido bem menor em relação ao valor de confiança, ele não é considerado baixo, pois o seu valor é relativo à natureza do meta-modelo, uma vez que nem todo meta-modelo possui associações de composição. Em outras palavras, quanto maior a quantidade de associações de composição em um meta-modelo, maior será o suporte das regras que envolvem este relacionamento. Além disso, é importante destacar que o valor de 19% de suporte foi obtido a partir de uma base com grande quantidade de dados. Por outro lado, a regra R1 possui uma confiança absoluta, pois em todos os registros da base em que o antecedente da regra ocorria o conseqüente ocorria também. Desta forma, podemos dizer que esta regra é válida para todas as meta-classes que representam a parte de uma associação de composição. Com a regra R1 podemos concluir que, neste tipo de associação, a meta-classe que representa o todo sempre deve ter acesso direto às suas partes.

*[R1] class.composition = owned  11046 →*
*class.isNavigable = true  11046*

A regra R2 afirma, com 17% de suporte e 87% de confiança, que quando uma meta-classe representar a parte da associação de composição ela não deve ser abstrata. Embora esta regra não atinja 100% do meta-modelo da UML, ela é útil para formarmos diretrizes que envolvem informações sobre associações de composição. Com esta regra podemos concluir que, para a maior parte das associações de composição, a meta-classe que representa a parte deve ser concreta.

*[R2] class.composition = owned  11046 →*
*class.isAbstract = false  9570*

Como a regra R2 não atinge os 100% de confiança, existem casos em que ela não é verdadeira. Um exemplo disso pode ser conferido no trecho do meta-modelo da UML ilustrado na Figura 10, em que um `Package` é composto por zero ou mais meta-classes

*Type*. Como podemos ver, a meta-classe que representa a parte desta associação é abstrata e não concreta, como diz a regra.

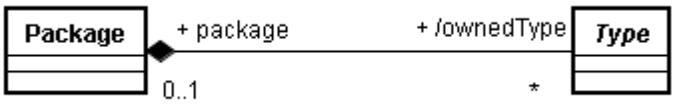



**Figura 10: Trecho do meta-modelo da UML em que R2 não se aplica.**

A regra R3 afirma, com 16% de suporte e 84% de confiança, que quando uma meta-classe representar a parte da associação de composição o limite mínimo de sua multiplicidade deve ser `0`. Portanto, podemos concluir que neste tipo de associação quase todas as meta-classes que representam a parte devem ter o limite mínimo de multiplicidade especificado como `0`.

*[R3] class.composition = owned  11046 →*
*class.lower = 0  9320*

Como a regra R3 não atinge os 100% de confiança, existem casos em que ela não é verdadeira. Um exemplo disso pode ser conferido no trecho do meta-modelo da UML ilustrado na Figura 11, em que um `Constraint` é composto por um *`ValueSpecification`*. Como podemos ver, a meta-classe que representa a parte desta associação possui o limite mínimo de multiplicidade igual a `1` e não `0`, como diz a regra.

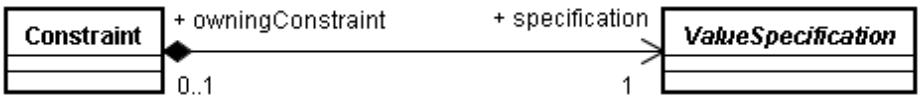



**Figura 11: Trecho do meta-modelo da UML em que R3 não se aplica.**

As regras R4 e R5 afirmam, com 19% e 11% de suporte, respectivamente, e ambas com 99% de confiança, que quando a meta-classe representar a parte ou o todo da associação de composição elas não devem ser somente leitura. Estas duas regras não são absolutamente verdadeiras devido ao fator de confiança. Com as regras R4 e R5 podemos concluir que em uma associação de composição tanto as meta-classes que representam a parte quanto as que representam o todo quase sempre podem ser escritas depois de inicializadas.

*[R4] class.composition = owned  11046 →*
*class.isReadOnly = false  10982*

*[R5] class.composition = owner  6160 →*
*class.isReadOnly = false  6078*

Como as regras R4 e R5 não atingem os 100% de confiança, existem casos em que elas não são verdadeiras. Um exemplo disso pode ser conferido no trecho do meta-modelo da UML ilustrado na Figura 12, em que um *Namespace* é composto por zero ou mais meta-classes *NamedElement*. Como podemos ver, tanto a meta-classe que representa o todo quanto a que representa a parte desta associação é somente leitura, não condizendo com as regras.

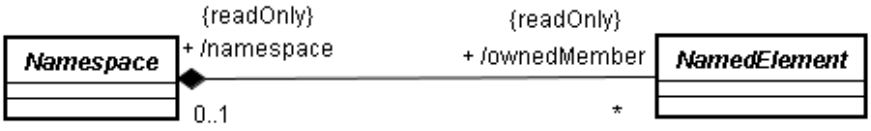



**Figura 12: Trecho do meta-modelo da UML em que R4 e R5 não se aplicam.**

A regra R6 afirma, com 11% de suporte e 100% de confiança, que quando a meta-classe representar o todo da associação de composição o limite máximo de sua multiplicidade deve ser sempre 1. Com esta regra podemos concluir que toda meta-classe que representa o todo de uma associação de composição deve ter o limite máximo especificado como 1.

*[R6] class.composition = owner  6160 →*
*class.upper = 1  6160*

A regra R7 afirma, com 11% de suporte e 99% de confiança, que quando a meta-classe representar o todo da associação de composição os seus atributos não devem ser derivados. Com isto podemos concluir que, em associações de composição, quase todas as meta-classes que representam o todo não devem possuir atributos derivados de qualquer outra informação.

*[R7] class.composition = owner  6160 →*
*class.property.isDerived = false  6078*

Como a regra R7 não atinge os 100% de confiança, existem casos em que ela não é verdadeira. Um exemplo disso pode ser conferido no trecho do meta-modelo da UML ilustrado na Figura 13, em que um `Operation` é composto por zero ou mais meta-classes `Parameter`. Como podemos ver, a meta-classe que representa o todo desta associação possui atributos derivados, não condizendo com a regra.

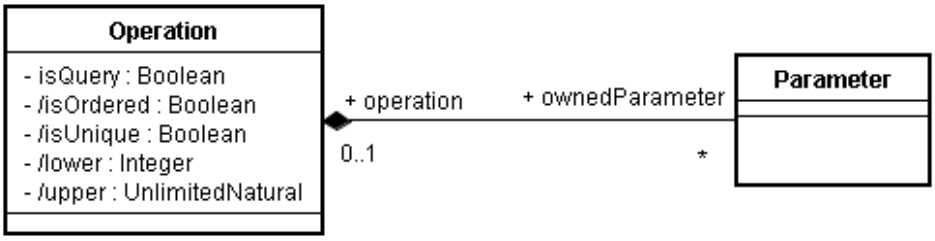



**Figura 13: Trecho do meta-modelo da UML em que R7 não se aplica.**

### 3.1.3 Interpretação/Avaliação e Implantação do Conhecimento

Nesta subseção, as regras ilustradas anteriormente serão interpretadas no intuito de compormos diretrizes úteis para o escopo de meta-modelos. Além disso, esta subseção também mostra algumas regras de associação que, no lugar de formar diretrizes, servem para validarmos meta-modelos já existentes, pois elas refletem informações precisas acerca dos mesmos.

#### 3.1.3.1 Composição de Diretrizes

Durante a etapa de Interpretação/Avaliação do processo KDD, as regras consideradas mais interessantes foram analisadas a fim de descobrirmos características e comportamentos em comum. Tivemos que analisar cada uma, individualmente, comparando com a especificação do meta-modelo da UML. Dessa forma, verificamos se de fato a regra era válida.

Ao analisarmos os atributos e os valores das regras R1 até a R7 percebemos que há algumas similaridades entre elas, como por exemplo, todas apresentam características de associação de composição em seus antecedentes. Já os seus conseqüentes são formados por atributos distintos, os quais deduzem aspectos importantes para uma associação de composição. Considerando também a forte confiança que estas regras apresentam, a Figura 14 ilustra um trecho de meta-modelo construído a partir da junção de todos os conceitos que elas

representam: a meta-classe `Owner` representa o todo de uma associação de composição e a meta-classe `Owned` representa a parte.

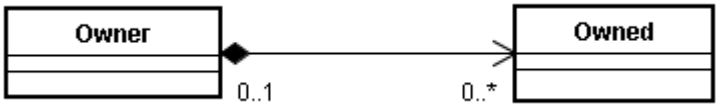



**Figura 14: Estrutura construída a partir de algumas regras de associação.**

Este trecho especifica que, em associações de composição: (R1) a parte deve ser navegável a partir do todo; (R2) a parte deve ser uma meta-classe concreta; (R3) a parte deve ter o limite mínimo de multiplicidade igual a `0`; (R4 e R5) tanto a parte quanto o todo devem possuir atributos que podem ser escritos depois de inicializados; (R6) o todo deve ter o limite máximo de multiplicidade igual a `1`; e (R7) o todo não deve possuir atributos derivados.

Como podemos observar, a estrutura formada a partir destas regras de associação já nos oferece a idéia de uma diretriz relacionada a associações de composição. Portanto, através destas sete regras e de uma análise manual realizada em um conjunto de meta-modelos diversos, definimos a diretriz chamada D11 – *Defining Association Member Ends Features* (apresentada no capítulo 4). Esta diretriz é bastante útil para auxiliar os desenvolvedores de meta-modelos na definição de associações de composição, uma vez que apresenta direções de como devem ser feitas. Desta forma, os meta-modelos se tornam mais consistentes, fáceis de compreender, de construir e, conseqüentemente, de fazer manutenções.

Depois de realizadas todas as etapas do processo KDD, o último passo é a implantação do conhecimento adquirido. No nosso trabalho, em particular, realizamos a implantação do conhecimento por meio de um documento descrevendo a diretriz identificada automaticamente por meio da mineração de dados. Portanto, neste documento consta apenas a diretriz D11 que, por sua vez, é bastante forte e pautada em várias regras de associação.

#### 3.1.3.2 Validação de Meta-modelos

Algumas das regras de associação geradas com a mineração de dados descreviam informações que não apresentavam nenhum conhecimento inovador que pudesse ser útil para a composição de diretrizes. Entretanto, tais regras são úteis como um mecanismo para a validação de meta-modelos e para auxiliar os desenvolvedores na construção de novos, uma vez que elas são totalmente confiáveis (100% de confiança). Algumas destas regras são

apresentadas no Apêndice B deste documento. Por exemplo, a regra R8, com 71% de suporte e 100% de confiança, especifica que operações definidas por meio de uma restrição OCL `body` são operações de consulta.

*[R8] class.operation.rule.constraintKind = body 40378 →*
*class.operation.isQuery = true 40378*

Outro exemplo é ilustrado na regra R9. Com 61% de suporte e 100% de confiança, ela especifica que operações com parâmetros de retorno são operações de consulta.

*[R9] class.operation.parameters.direction = return 34510 →*
*class.operation.isQuery = true 34510*

### 3.1.4 Suporte Ferramental

Tendo em vista que utilizamos todo o processo KDD para a descoberta automática das diretrizes, uma ferramenta de suporte foi desenvolvida para nos auxiliar na realização de algumas de suas etapas, tais como a Seleção, o Pré-processamento e a Transformação. Além disso, ela foi utilizada para a geração automática da nossa base de dados, o arquivo ARFF.

Como podemos observar na arquitetura ilustrada na Figura 15, a ferramenta foi dividida em três módulos principais:

- *Extractor*: Módulo responsável tanto pela leitura do arquivo XMI que descreve o meta-modelo da UML quanto pela extração das informações necessárias neste arquivo para a mineração dos dados;
- *Generator*: Módulo responsável pela geração da base de dados, o arquivo ARFF, a partir das informações obtidas no módulo *Extractor*;
- *AlgorithmsManipulator*: Módulo responsável pela aplicação de algoritmos de mineração de dados na nossa base. A implementação destes algoritmos foi reutilizada da ferramenta Weka. O resultado obtido é traduzido em forma de diretrizes (*guidelines*) para meta-modelos.

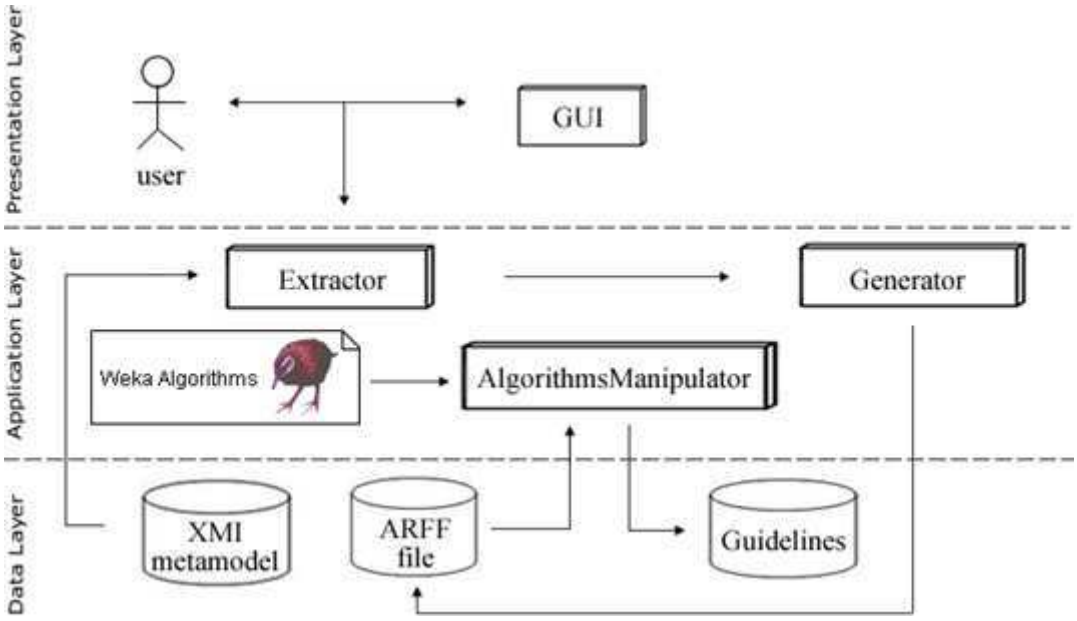



**Figura 15: Arquitetura da ferramenta de suporte para a mineração de dados.**

A ferramenta foi desenvolvida com base em uma arquitetura de três camadas: Apresentação (*Presentation Layer*), Aplicação (*Application Layer*) e Dados (*Data Layer*). Ela foi implementada em Java e possui uma licença GPL. Portanto, a ferramenta encontra-se disponível em [Vieira et al., 2009] para toda a comunidade, de forma que qualquer outra pessoa possa ter acesso e fazer modificações na sua implementação.

A Figura 16, ilustrada a seguir, mostra uma das telas da ferramenta de suporte desenvolvida, em que o usuário pode escolher o algoritmo de análise de associação que deseja executar e configurar os seus parâmetros no botão *Choose*. Esta tela mostra a execução do algoritmo *Apriori* com um total de 10 regras geradas, as quais são apresentadas na guia *Associator output*.

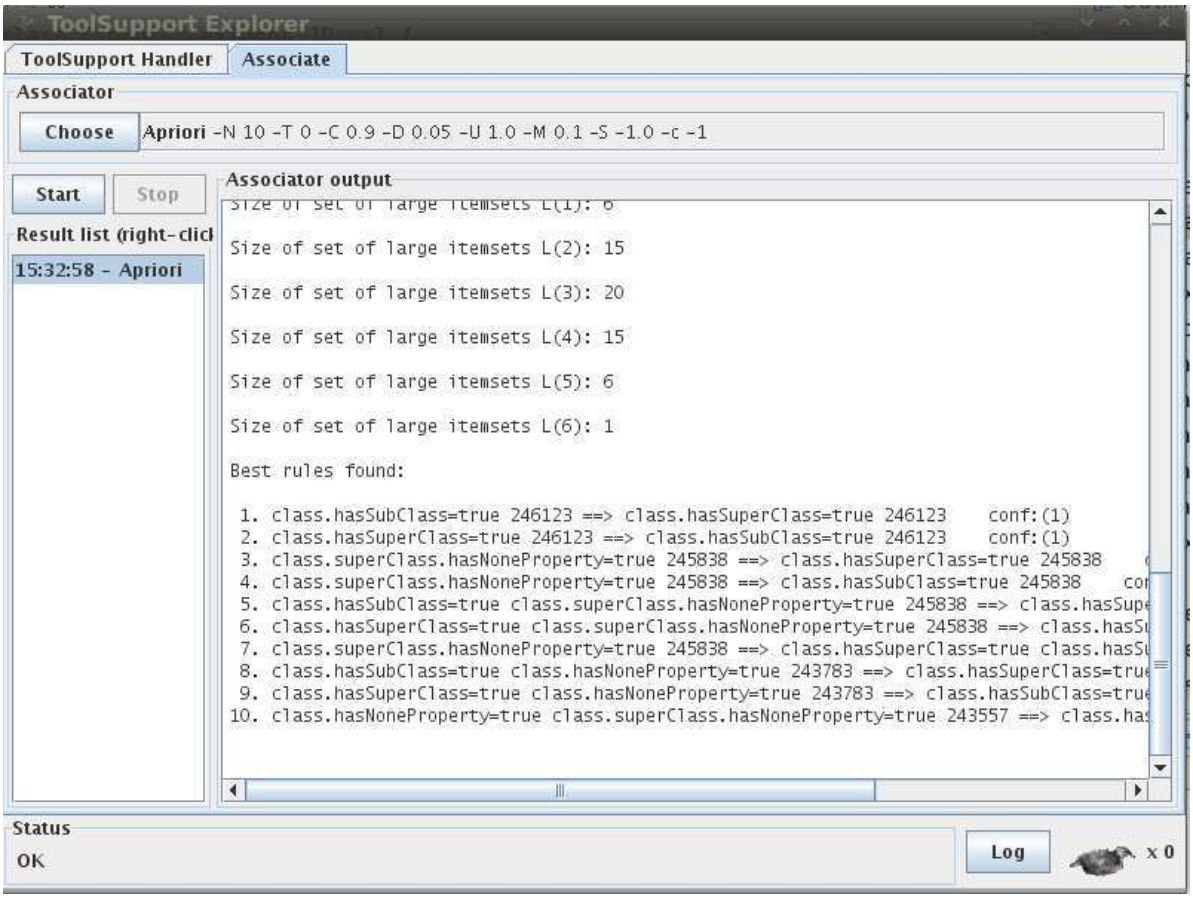



**Figura 16: Uma das telas da ferramenta de suporte desenvolvida.**

### 3.1.5 Limitações da Abordagem

A utilização da mineração de dados para a descoberta automática de conhecimento, seja em qual for a área, exige a dedicação de muito tempo desde a etapa de preparação dos dados até a análise dos resultados. Apesar de todo esse tempo empregado, a mineração de dados pode resultar na descoberta de nenhum conhecimento inovador ou que possa ter alguma utilidade na área envolvida. A realidade é que não há garantias de que, mesmo se todas as etapas forem realizadas corretamente, a aplicação da mineração de dados trará bons resultados.

Neste trabalho, cerca de 60% do tempo foi dedicado às etapas da mineração de dados. Como podemos notar, a utilização desta abordagem foi a atividade de maior custo e tempo empregado ao longo do nosso trabalho. A etapa mais custosa foi a de preparação da base de dados, na qual tivemos que implementar uma ferramenta de suporte para construir a nossa base. Apesar de todo o custo e tempo gasto, o resultado final que obtivemos através desta abordagem não foi o que esperávamos, mas ainda conseguimos identificar uma diretriz e utilizar algumas das regras óbvias identificadas como forma de validação de meta-modelos.

Possivelmente, um dos motivos pelos quais a aplicação da mineração de dados não teve o sucesso esperado em nosso trabalho é o fato de termos selecionado apenas o meta-modelo da UML para compor a nossa base de dados, apesar deste já ser um meta-modelo bastante extenso e complexo. Contudo, para cada novo meta-modelo que desejássemos incluir na base de dados, teríamos que passar pela etapa de preparação da base de dados e, conseqüentemente, adaptar a ferramenta de suporte[3]. Com a base de dados ainda maior e com meta-modelos diversificados, talvez fosse possível identificarmos mais diretrizes. Entretanto, devido a restrições de tempo, não foi possível realizarmos tais extensões ao longo de um trabalho de mestrado.

Apesar de todas as limitações, a experiência com a utilização da mineração de dados foi válida. Além de termos descoberto uma diretriz, todo o estudo que realizamos pode servir como base para futuras tentativas de descoberta de conhecimento em outras áreas, como em transformações de modelos, por exemplo.

## 3.2 Manual: Analisando um Conjunto de Meta-modelos

Tendo em vista que os resultados obtidos com a descoberta automática não foram os esperados, pois apenas a diretriz D11 (que se refere às associações de composição entre meta-classes) foi descoberta, realizamos também um procedimento manual para a descoberta de diretrizes.

Inicialmente, diversos meta-modelos foram minuciosamente investigados a fim de descobrirmos características recorrentes que eles apresentavam em comum. Estes meta-modelos foram construídos por organizações diferentes, dentre elas está a OMG, em virtude de se tratar de um consórcio em padronização bastante reconhecido e consolidado, oferecendo meta-modelos consistentes e confiáveis. Durante o processo de descoberta, observamos todos os elementos dos meta-modelos, tais como meta-classes, atributos, pacotes, associações e até mesmo regras especificadas com OCL. Por exemplo, analisamos a estrutura de pacotes (considerando seu conteúdo e semântica) dos meta-modelos com o intuito de descobrirmos

[3] Nem todos os meta-modelos são especificados seguindo um mesmo padrão, portanto, a forma pela qual nossa ferramenta extrai as informações do arquivo XMI difere. Por exemplo, o formato do arquivo XMI referente a um meta-modelo especificado em MOF é diferente do formato do arquivo XMI referente a um meta-modelo especificado em *Ecore*.

como diferentes meta-modelos a especificam. Desta forma, dentre outras características, descobrimos que eles geralmente especificam um pacote contendo apenas os tipos primitivos.

Vale ressaltar que, durante o processo de identificação das diretrizes, descartamos aquelas que eram dependentes de domínio, isto é, que apresentavam características comuns apenas a um meta-modelo específico de acordo com o domínio ou linguagem meta-modelada. Para cada diretriz identificada em um meta-modelo, observamos se ela também ocorria nos outros meta-modelos analisados, de forma a averiguar se ela realmente era independente do domínio sendo meta-modelado. Além disso, estudamos cada diretriz individualmente a fim de compreendermos a razão pela qual ela era importante (sua motivação), ou seja, procuramos identificar o problema que ela ajuda a resolver. Por exemplo, percebemos que vários meta-modelos da OMG definem um pacote chamado `Abstractions`. Portanto, tentamos descobrir por que é importante especificarmos este pacote no meta-modelo, bem como as suas possíveis conseqüências.

Como resultado de todas as investigações desta descoberta manual, conseguimos identificar um total de 12 diretrizes bastante interessantes para a aplicação em meta-modelos. Cada uma delas é descrita na Seção 4. Tendo em vista a variedade dos meta-modelos analisados, as diretrizes que identificamos não apresentam tendência à nenhum meta-modelo específico, isto é, elas são independentes de domínio.

### 3.2.1 Limitações da Abordagem

Apesar de termos obtido melhores resultados por meio da identificação manual de diretrizes, esta abordagem possui algumas limitações: (i) está suscetível a falhas, uma vez que depende inteiramente da análise e intervenção humana para a descoberta de diretrizes; (ii) muitas diretrizes podem passar despercebidas pelo pesquisador e este, por sua vez, não detectar aquelas que poderiam ser bastante úteis na construção de meta-modelos; (iii) exige um elevado tempo empregado durante o processo de investigação; e (iv) se torna impraticável quando desejamos identificar diretrizes considerando uma grande quantidade de meta-modelos.

# Capítulo 4

# Conjunto de Diretrizes Identificadas para Meta-modelos

Este capítulo tem por finalidade apresentar o conjunto de diretrizes para meta-modelos que foram identificadas tanto de forma manual quanto automática. Estas diretrizes serão descritas de maneira padronizada, conforme um *template* pré-estabelecido. Por fim, o capítulo discute acerca de uma ferramenta de suporte desenvolvida para automatizar a aplicação de algumas diretrizes em meta-modelos já existentes.

## 4.1 Descrição das Diretrizes

As diretrizes propostas neste trabalhado têm a finalidade de auxiliar desenvolvedores de meta-modelos na construção destes artefatos, facilitando a sua compreensão, manutenção, evolução e reuso, além de reduzir possíveis redundâncias. Nesse contexto, a presente subseção destina-se à descrição detalhada de cada uma delas, apresentando suas características de acordo com um *template* que definimos com base no adotado pela GoF para padrões de projeto. Este *template* descreve o objetivo, a motivação, a solução, as conseqüências, a aplicabilidade, as diretrizes relacionadas e exemplos práticos da aplicação de cada uma em meta-modelos variados.

No decorrer desta subseção, cada uma das diretrizes será referenciada por uma sigla, a qual é formada pela letra D e seu número seqüencial, como por exemplo, a primeira diretriz

será chamada de D1. O catálogo consiste em um total de 13 diretrizes, podendo ser aplicadas em diferentes elementos de um meta-modelo, sejam eles relacionamentos, atributos, *enumerations*, dentre outros. A seguir, vejamos a descrição de cada uma delas.

### 4.1.1 D1 - *Abstracting Common Attributes*

**Objetivo:** Evitar a redundância de atributos em comum (com as mesmas características e a mesma semântica) especificados em diferentes meta-classes de um meta-modelo.

**Motivação:** Muitas vezes quando os meta-modelos vão tomando proporções maiores, acontece de um mesmo atributo ser definido em várias meta-classes diferentes e o desenvolvedor não perceber, mesmo que ele já seja experiente. Contudo, isto pode ocorrer até mesmo em meta-modelos pequenos, por falta de atenção. Dessa forma, o meta-modelo fica saturado com a repetição desnecessária de atributos que possuem as mesmas características (nome, tipo, visibilidade, etc.) e a mesma semântica em meta-classes diferentes. Além do problema da redundância, se quisermos alterar um atributo temos que fazer a modificação individualmente em todas as meta-classes que o especificam, tendo em vista que tal atributo é definido da mesma forma em todas elas.

**Solução:** Devemos criar uma meta-classe abstrata para cada atributo em comum que foi especificado em meta-classes diferentes. Dependendo da semântica do meta-modelo, podemos criar uma meta-classe abstrata para generalizar mais de um atributo em comum. As meta-classes que especificam um ou mais atributos em comum passam a estender a meta-classe abstrata criada, assim, herdando o(s) atributo(s) e não precisando mais defini-lo(s). Vale ressaltar que todos os atributos definidos na meta-classe abstrata devem ter a visibilidade *public* para que as subclasses possam herdá-los. Nesse sentido, é importante destacarmos que todos os atributos em comum nas meta-classes devem ter a visibilidade *public* para que não haja problemas semânticos ao definirmos tais atributos como *public* na meta-classe abstrata criada.

De acordo com a diretriz D5, detalhada na subseção 4.1.5, toda meta-classe abstrata reutilizável deve ser definida em um pacote chamado `Abstractions`. Nesse sentido, toda meta-classe abstrata criada pela diretriz D1 deve ser definida no pacote `Abstractions` do meta-modelo.

**Conseqüência:** Cada atributo em comum especificado em diferentes meta-classes do meta-modelo irá se concentrar apenas na meta-classe abstrata criada para generalizá-lo. Portanto, qualquer modificação neste atributo deve ser feita apenas nesta meta-classe. Dessa forma, haverá a eliminação da redundância de atributos no meta-modelo, bem como maior facilidade de manutenção, compreensão e evolução. Há uma facilidade também no seu reuso, pois se uma nova meta-classe possuir um atributo já especificado em uma meta-classe abstrata, basta estendê-la para que o atributo seja herdado.

**Aplicabilidade:** Pode ser aplicada em qualquer meta-modelo que possua mais de uma meta-classe com atributos em comum, desde que tais atributos tenham a visibilidade *public*. A quantidade de aplicações da diretriz D1 nos meta-modelos é contabilizada pelo número de meta-classes abstratas que foram criadas após a sua aplicação.

**Diretrizes Relacionadas:** D2, D3, D5.

**Exemplo:** Vejamos no exemplo que segue como a diretriz D1 pode ser aplicada no meta-modelo Maven [Maven, 2005]. A Figura 17 mostra que as meta-classes `Xmlns`, `Goal` e `AntTaskDef` compartilham de uma mesma característica: todas elas são nomeáveis, isto é, definem um atributo `name`.

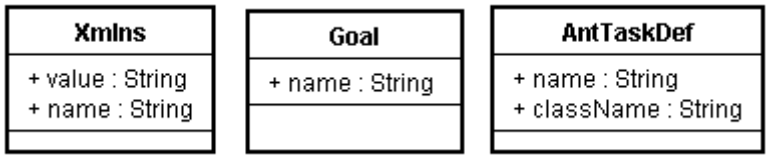



**Figura 17: Meta-classes do meta-modelo de Maven com um atributo em comum.**

Sendo assim, a diretriz especifica que devemos criar uma meta-classe abstrata e nela especificarmos o atributo `name` com visibilidade *public*. As meta-classes que definem este atributo em comum devem estender *`NamedElement`*, a meta-classe abstrata criada, e assim, ter acesso ao mesmo. A diretriz aplicada pode ser vista na Figura 18. Neste exemplo, a quantidade de aplicações da diretriz D1 foi igual a 1, pois foi necessário criar apenas uma meta-classe abstrata.

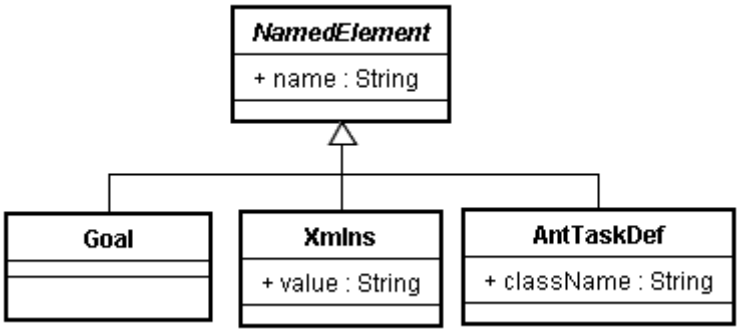



**Figura 18: Meta-modelo de Maven depois da aplicação da diretriz D1.**

Além deste exemplo, a diretriz D1 pode ser aplicada também em outros meta-modelos, como por exemplo: (i) J2SE5 [J2SE5, 2007], em que suas meta-classes `VariableDeclarationStatement`, `VariableDeclarationExpression`, `SingleVariableDeclaration` e *`BodyDeclaration`* possuem um mesmo atributo em comum, chamado `modifiers`; (ii) HTML [HTML, 2005], em que as meta-classes `Embed`, `Img` e `Applet` possuem três atributos em comum, chamados `height`, `width` e `align`; e (iii) U2TP, em que 13 meta-classes possuem o atributo `name` em comum.

### 4.1.2 D2 - *Abstracting Common Associations*

**Objetivo:** Evitar a redundância de associações em comum (com as mesmas características e a mesma semântica) entre meta-classes de um meta-modelo.

**Motivação:** Observando vários meta-modelos, com ênfase em suas associações, podemos notar que existem casos em que meta-classes distintas se associam com uma determinada meta-classe, e os lados destas associações especificam a mesma multiplicidade, a mesma semântica e o mesmo nome da associação ou dos papéis – quando algum papel é definido. Dessa forma, o meta-modelo apresenta uma grande quantidade de associações que pode sobrecarregá-lo e confundir o leitor. Em meta-modelos pequenos e mais simples talvez este problema não seja tão perceptível. Por outro lado, se considerarmos exemplos maiores e mais complexos, principalmente envolvendo muitas associações, iremos sentir certa dificuldade de compreensão. Conseqüentemente, isto será refletido negativamente durante a manutenção do meta-modelo.

**Solução:** Considere que duas meta-classes distintas (A e B) possuem uma associação em comum com uma mesma meta-classe (C). A solução proposta para diminuir esta redundância é criar uma meta-classe abstrata para generalizar estas meta-classes distintas (A e B). Portanto, as associações em comum serão especificadas somente entre a nova meta-classe abstrata criada e a meta-classe (C). As associações devem ser retiradas das meta-classes distintas (A e B) e estas, por sua vez, devem estender a meta-classe abstrata criada, assim, herdando a associação. Logo, sempre que uma nova meta-classe precisar de uma associação com a meta-classe (C), basta estender a meta-classe abstrata. Vale ressaltar que, para os casos em que a associação possui um papel especificado, convencionamos os nomes das meta-classes abstratas criadas como sendo: *Owner* + *Nome_do_papel*. Esta é apenas uma convenção para apresentarmos os exemplos neste documento, o desenvolvedor pode escolher um nome mais apropriado de acordo com a semântica do meta-modelo.

Vale ressaltar que toda meta-classe abstrata criada pela diretriz D2 deve ser definida no pacote `Abstractions` do meta-modelo.

**Conseqüência:** As associações em comum serão definidas apenas na meta-classe abstrata criada, logo, qualquer modificação deve ser feita apenas na(s) associacão(ões) com esta meta-classe. Dessa forma, além de evitarmos a redundância de associações, diminuímos a dificuldade de entendimento, de manutenção e de evolução do meta-modelo. Adicionalmente, há uma facilidade no seu reuso, pois, uma vez que uma associação é especificada em uma meta-classe abstrata, ela poderá ser reutilizada por novas meta-classes que precisam defini-la.

**Aplicabilidade:** Deve ser aplicada de acordo com as seguintes condições: (i) duas ou mais meta-classes distintas devem possuir uma associação com uma "mesma" meta-classe; e (ii) os lados das associações devem especificar o mesmo limite mínimo e máximo de multiplicidade, a mesma semântica, e o mesmo nome de papéis, quando existirem. A quantidade de aplicações desta diretriz nos meta-modelos é o número de meta-classes abstratas que foram criadas como resultado de sua aplicação.

**Diretrizes Relacionadas:** D1, D5.

**Exemplo:** O exemplo que segue mostra como a diretriz deve ser aplicada no meta-modelo de Maven. A Figura 19 apresenta um pequeno trecho deste meta-modelo em que as meta-classes `Path` e `ClassPath` possuem uma associação de composição com a meta-classe `FileSet`.

Podemos observar que os lados destas duas associações especificam a mesma multiplicidade e nome dos papéis referentes à associação.

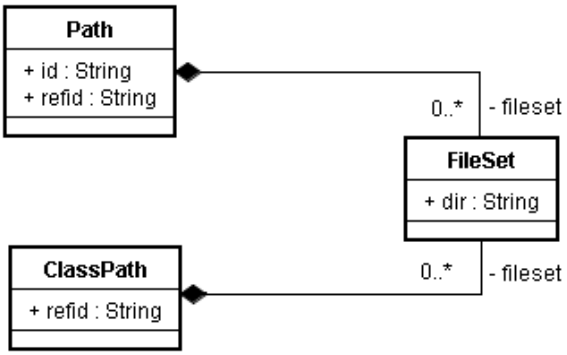



**Figura 19: Trecho do meta-modelo de Maven com associações em comum.**

Neste caso, para aplicarmos a diretriz devemos, primeiramente, criar uma meta-classe abstrata para generalizar as meta-classes que especificam a associação. Portanto, a associação de composição entre `ClassPath-FileSet` e `Path-FileSet` foi definida na meta-classe abstrata *`OwnerFileset`*, como pode ser vista na Figura 20. Por fim, as meta-classes `ClassPath` e `Path` estendem a meta-classe abstrata. Neste exemplo, houve apenas uma aplicação da diretriz D2, pois foi necessário criar apenas uma meta-classe abstrata.

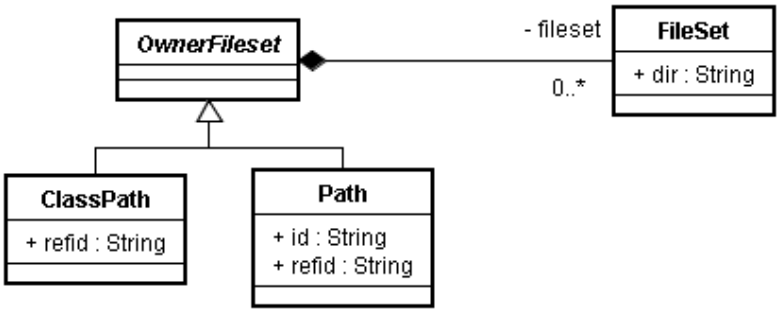



**Figura 20: Trecho do meta-modelo de Maven depois da aplicação da diretriz D2.**

Além deste exemplo, a diretriz D2 pode ser aplicada também em outros meta-modelos, como por exemplo: (i) Java [Java, 2006], em que as meta-classes `VariableDeclarationExpression`, `VariableDeclarationStatement` e `FieldDeclaration` possuem uma associação de composição chamada `fragments` com a meta-classe `VariableDeclarationFragment`; (ii) *Model to Text Transformation Language* [MtoTTL, 2008], em que as meta-classes `ForBlock` e `TemplateInvocation` possuem três associações, chamadas `before`, `after` e `each`, com a meta-classe

`OclExpression`; e (iii) BPEL [BPEL, 2006], em que as meta-classes `CatchAll`, `ElseIf`, `While`, `Then`, `Otherwise`, `Case`, `ForEach`, `Catch`, dentre outras, possuem uma associação de composição chamada `activity` com a meta-classe `Activity`.

### 4.1.3 D3 - *Generalizing Common Attributes*

**Objetivo:** Evitar a redundância de atributos comuns (com as mesmas características e a mesma semântica) especificados em meta-classes que estendem uma mesma superclasse de um meta-modelo.

**Motivação:** Muitas pessoas constroem relacionamentos de herança em seus meta-modelos sem observar os atributos que existem em comum nas subclasses. Dessa forma, acabam definindo os mesmos atributos nas subclasses e, às vezes, na superclasse também. Como conseqüência, o meta-modelo torna-se redundante, apresentando elementos desnecessários. A compreensão também pode ser afetada uma vez que, com uma análise superficial do meta-modelo, podemos fazer interpretações diferentes para atributos que, de fato, possuem o mesmo significado. Um meta-modelo redundante e de difícil compreensão, conseqüentemente, apresentará dificuldades durante a sua utilização e manutenção.

**Solução:** A diretriz D3 especifica que, em relacionamentos de herança, deve-se verificar quais são os atributos que existem em comum em todas as subclasses de uma mesma superclasse e, então, defini-los apenas na superclasse, com a visibilidade *public*. Dessa forma, as subclasses terão acesso aos atributos da superclasse.

**Conseqüência:** Em um relacionamento de herança, todos os atributos existentes em comum nas subclasses irão se concentrar apenas na superclasse, assim, evitando que o meta-modelo apresente redundância e dificuldade no entendimento. Além disso, há uma melhoria na manutenção, uma vez que qualquer alteração será feita apenas na superclasse. Adicionalmente, há uma facilidade de reuso, pois, uma vez que os atributos são generalizados, eles poderão ser estendidos por novas meta-classes que precisam defini-los. Eles poderão ser acessados normalmente pelas subclasses, uma vez que estes são definidos com visibilidade *public* na superclasse.

Vale salientar que, como conseqüência da aplicação desta diretriz, podem existir meta-classes sem nenhum atributo especificado, caso seu(s) atributo(s) tenha(m) sido

generalizado(s) para a superclasse. Portanto, nestes casos, é importante que o desenvolvedor analise o meta-modelo a fim de verificar se tais meta-classes possuem algum outro relacionamento (além do relacionamento de herança) ou se elas possuem semânticas diferentes. Dessa forma, o desenvolvedor poderá saber se tais meta-classes ainda têm razão de existir.

**Aplicabilidade:** Pode ser aplicada em qualquer meta-modelo que possua relacionamento de herança, de forma que as subclasses tenham pelo menos um atributo em comum. A quantidade de aplicações da diretriz D3 é definida pelo número de atributos que foram transferidos das subclasses para a superclasse.

**Diretrizes Relacionadas:** D1.

**Exemplo:** O exemplo que segue mostra um trecho do meta-modelo de Express [Express, 2007] sem a utilização da diretriz e, em seguida, ilustra como ela deve ser aplicada. A Figura 21 apresenta um relacionamento de herança, em que as meta-classes `DomainRule`, `UniqueRule` e *`Attribute`* estendem *`TypeElement`*, todas elas pertencentes ao pacote `Core` do meta-modelo. Como podemos ver, as três subclasses possuem um atributo em comum, chamado `position`.

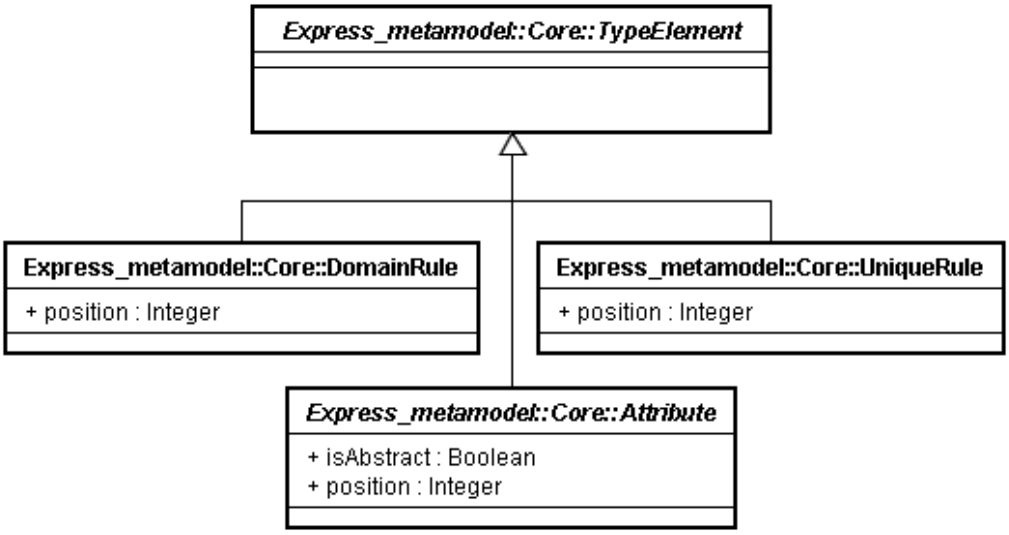



**Figura 21: Trecho do meta-modelo de Express sem a utilização da diretriz D3.**

Portanto, para aplicarmos a diretriz D3 devemos definir o atributo `position` apenas na superclasse *`TypeElement`*, com a visibilidade *public*, e então retirá-lo das subclasses. A

diretriz aplicada pode ser observada na Figura 22. Neste exemplo, houve apenas uma aplicação da diretriz.

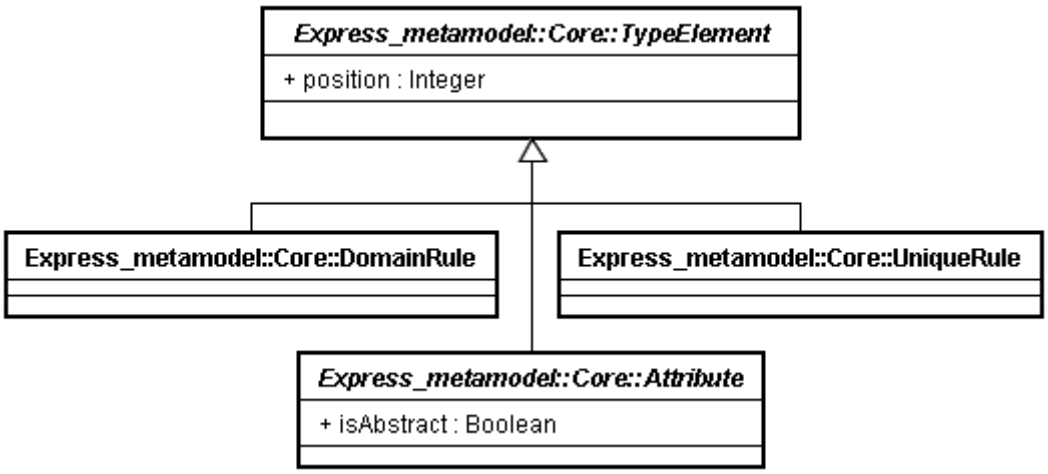



**Figura 22: Trecho do meta-modelo de Express com aplicação da diretriz D3.**

### 4.1.4 D4 - *Grouping Related Metaclasses*

**Objetivo:** Considere uma associação entre duas meta-classes A e B, onde B possui quatro subclasses e apenas duas delas, de fato, precisam herdar a associação com a meta-classe A. O objetivo desta diretriz é evitar o uso desnecessário de restrições OCL para especificar quais são as subclasses de B que devem se associar com A.

**Motivação:** Para melhor ilustrar a motivação desta diretriz, considere a situação apresentada anteriormente. No intuito de especificarmos quais subclasses de B, de fato, possuem a associação com a meta-classe A, definimos algumas restrições OCL. Entretanto, o uso de tais restrições não é a melhor solução para especificarmos este trecho do meta-modelo, pois o entendimento do mesmo pode ser comprometido, uma vez que nem todas as pessoas têm conhecimentos em OCL. Geralmente, os desenvolvedores adotam esse tipo de solução por inexperiência ou pela ausência de algum auxílio que possa direcioná-los na construção do meta-modelo.

**Solução:** Esta diretriz recomenda a criação de uma superclasse S para generalizar apenas as subclasses da meta-classe B que, de fato, precisam da associação com a meta-classe A. Logo, a associação passa a ser entre a meta-classe A e a nova superclasse criada, S.

**Conseqüência:** O meta-modelo será melhor compreendido. Além disso, haverá uma diminuição da redundância, uma vez que evitará o uso de restrições OCL desnecessárias. Dessa forma, haverá também uma maior facilidade de manutenção e evolução do meta-modelo. Muitas vezes, o uso de tais restrições pode dificultar o entendimento do meta-modelo e confundir o leitor, pois nem todas as pessoas são familiares com OCL.

**Aplicabilidade:** Pode ser aplicada nos meta-modelos que utilizam restrições OCL para a finalidade apresentada anteriormente na motivação. A quantidade de aplicações da diretriz D4 é definida pelo número de superclasses que foram criadas como resultado de sua aplicação.

**Diretrizes Relacionadas:** Nenhuma.

**Exemplo:** Este exemplo mostra como a diretriz deve ser aplicada no meta-modelo de QVT. Vale salientar que o trecho do meta-modelo ilustrado na Figura 23 foi modificado apenas para mostrar como seria um meta-modelo sem o uso da diretriz D4, pois, na realidade, ele já está em conformidade com a mesma. Na Figura 23, podemos observar uma associação de composição entre as meta-classes `ImperativeOperation` e *`Element`* que, por sua vez, possui quatro subclasses. Entretanto, apenas as subclasses `MappingBody` e `ConstructorBody` devem se relacionar com `ImperativeOperation`.

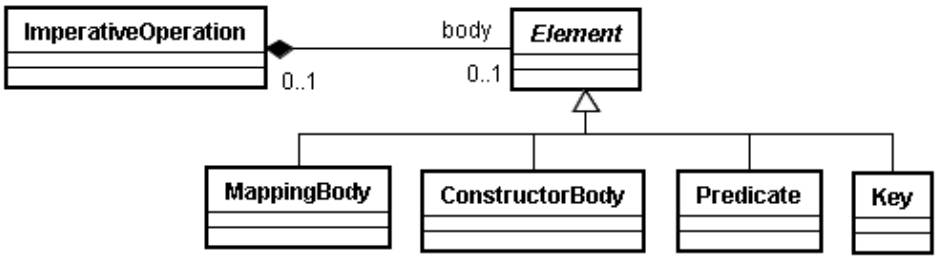



**Figura 23: Trecho do meta-modelo de QVT.**

Nesse sentido, definimos a seguinte restrição OCL para especificarmos que apenas as subclasses `MappingBody` e `ConstructorBody` devem se relacionar com `ImperativeOperation`:

*context ImperativeOperation*

*inv: self.body.oclIsTypeOf(MappingBody) or self.body.oclIsTypeOf(ConstructorBody)*

Agora, vejamos como se deve aplicar a diretriz D4 neste meta-modelo. Basta criarmos a superclasse *OperationBody* para generalizar as subclasses que, de fato, devem se relacionar com ImperativeOperation. Podemos observar o resultado na Figura 24. Neste exemplo, houve apenas uma aplicação da diretriz, pois somente uma superclasse foi criada, a *OperationBody*.

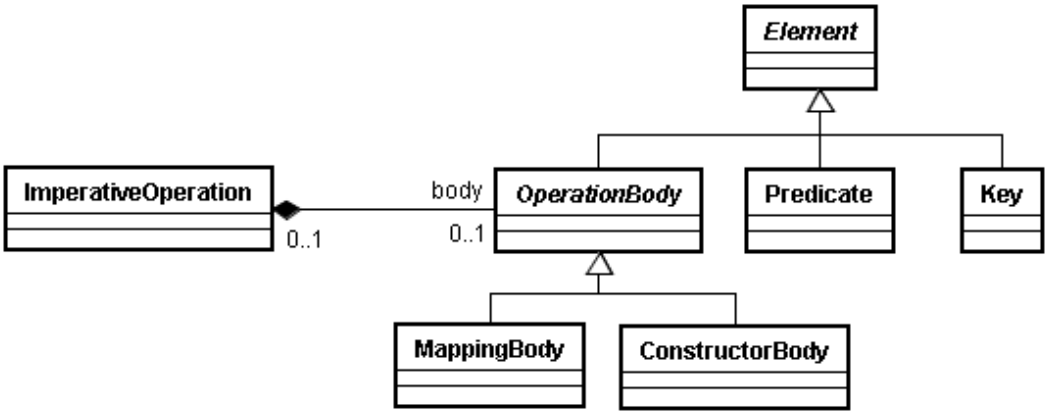



**Figura 24: Trecho do meta-modelo de QVT com a aplicação da diretriz D4.**

## Diretrizes para Organização em Pacotes

Neste tópico, serão apresentadas três diretrizes – D5, D6 e D7 – que visam melhorar os meta-modelos a partir da organização em pacotes, bem como facilitar o reuso destes pacotes em outros meta-modelos.

**Motivação:** Alguns projetos de meta-modelagem não utilizam um recurso bastante útil para a estruturação e organização das meta-classes: os pacotes. Neste contexto, um pacote é um mecanismo de propósito geral para organizar elementos semanticamente relacionados em grupos. Um meta-modelo não organizado em pacotes é mais difícil de ser compreendido, principalmente, se ele for muito grande. Além disso, o reuso de partes do meta-modelo também se torna uma tarefa difícil. Outro problema que ocorre durante a manutenção de meta-modelos é como localizar alguns elementos, como por exemplo, meta-classes abstratas já existentes que podemos reutilizar ou os tipos de dados primitivos. As diretrizes a seguir propõem resolver estes problemas.

**Aplicabilidade:** Podem ser aplicadas em qualquer meta-modelo. Caso ele já possua uma organização em pacotes bem definida, esta pode ser adaptada para ficar em conformidade

com a diretriz. A quantidade de aplicações de cada uma das diretrizes descritas neste tópico pode ser no máximo igual a 1, pois cada uma delas sugere a criação de apenas um pacote.

### 4.1.5 D5 - *Adding Abstractions Package*

**Objetivo:** Facilitar a localização das meta-classes que podem ser reutilizadas em todo o meta-modelo, como aquelas criadas a partir da aplicação das diretrizes D1 e D2.

**Solução:** Esta diretriz especifica que na raiz do meta-modelo deve ser criado um pacote, chamado `Abstractions`, para conter apenas as meta-classes abstratas (com seus elementos relacionados, como atributos e associações) que podem ser reutilizadas em todo o meta-modelo. Logo, o meta-modelo como um todo será melhor compreendido e, conseqüentemente, a sua manutenção torna-se mais fácil, uma vez que as meta-classes reutilizáveis são facilmente localizadas. Esta diretriz também oferece uma melhor modularização do projeto em pacotes, tendo em vista que o pacote `Abstractions` pode conter outros pacotes para organizar as meta-classes de acordo com alguma relação semântica. Além disso, outros pacotes do próprio meta-modelo somente podem ter acesso as meta-classes abstratas se eles realizarem um *merge* ou *import* do pacote `Abstractions`.

**Conseqüência:** Todas as meta-classes que possuem características reutilizáveis irão se concentrar em um único pacote, o `Abstractions`, facilitando a localização destas para reuso. Visto que tais meta-classes estarão melhor organizadas neste pacote, haverá também uma maior facilidade na manutenção, compreensão e evolução do meta-modelo.

**Diretrizes Relacionadas:** D1, D2, D6, D7.

**Exemplos:** Este exemplo ilustra a aplicação da diretriz no meta-modelo de Maven. Depois de aplicada a diretriz D1 em uma parte deste meta-modelo, uma nova meta-classe foi criada: a *`NamedElement`*, como ilustrado anteriormente na Figura 18. Esta meta-classe pode ser reutilizada por qualquer outra que seja nomeável. Da mesma forma, depois de aplicada a diretriz D2 em uma parte do meta-modelo de Maven, a meta-classe *`OwnerFileset`* foi criada, como ilustrado anteriormente na Figura 20.

Portanto, como mostra a Figura 25, a diretriz D5 sugere criarmos o pacote `Abstractions` e nele incluirmos as meta-classes: *`NamedElement`*, a qual possui o

atributo `name` que foi generalizado, e *`OwnerFileset`*, a qual possui a associação de composição com `Maven::FileSet` que também foi generalizada. Além disso, este pacote deve conter todas as outras meta-classes que forem reutilizáveis do meta-modelo de Maven.

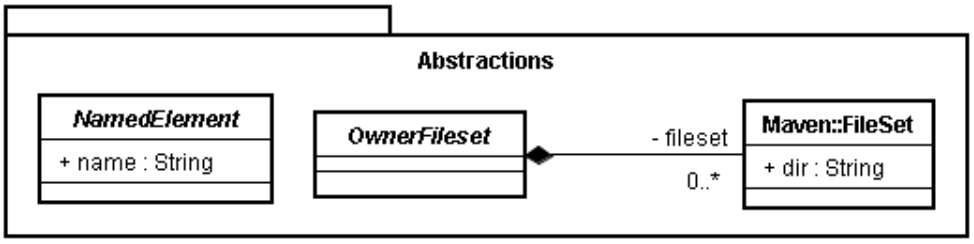



**Figura 25: Pacote Abstractions para o meta-modelo de Maven.**

Além deste exemplo, a diretriz D5 pode ser aplicada também em outros meta-modelos, dos quais nenhum possui um pacote `Abstractions` ou algo semelhante que contenha apenas as meta-classes reutilizáveis, a fim de facilitar o reuso. São eles: (i) XHTML [XHTML, 2005], constituído por apenas dois pacotes, chamados `PrimitiveTypes` e `XHTML`; (ii) J2SE5, também constituído por apenas dois pacotes, chamados `PrimitiveTypes` e `J2SE5`; e (iii) IRL [IRL, 2006].

### 4.1.6 D6 - *Adding PrimitiveTypes Package*

**Objetivo:** Facilitar a localização de todos os tipos de dados primitivos existentes no meta-modelo.

**Solução:** Esta diretriz sugere que na raiz do meta-modelo seja criado um pacote, chamado `PrimitiveTypes`, contendo todos os tipos primitivos existentes no meta-modelo, sejam eles *String*, *Integer*, *Boolean*, *Enumeration*, dentre outros. Todos estes tipos de dados devem ser definidos em meta-classes com o estereótipo `<<primitive>>` para indicar que se trata de um tipo primitivo. Deste modo, torna-se mais fácil identificar os tipos de dados que o meta-modelo dispõe. Além disso, há uma facilidade no reuso, tendo em vista que outros meta-modelos podem necessitar dos mesmos tipos de dados primitivos.

**Conseqüência:** Uma vez que todos os tipos de dados primitivos existentes no meta-modelo serão facilmente encontrados em um único pacote, o `PrimitiveTypes`, outros meta-modelos podem importá-lo e estendê-lo com outros tipos de dados caso haja necessidade.

**Diretrizes Relacionadas:** D5, D7.

**Exemplo:** Vejamos como utilizar a diretriz D6 no meta-modelo de ATL [ATLtoProblem, 2005]. Na Figura 26, podemos observar um pequeno trecho do meta-modelo de ATL contendo alguns tipos de dados primitivos: *Boolean*, *String*, *Numeric* (*Real* ou *Integer*).

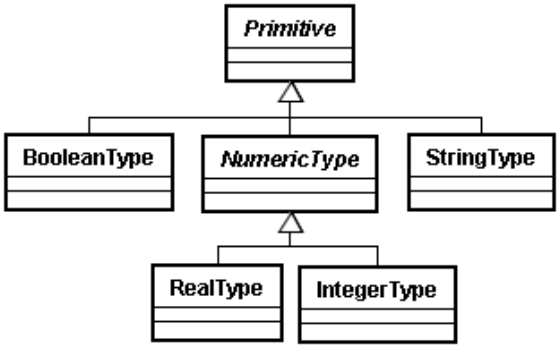



**Figura 26: Trecho do meta-modelo de ATL contendo alguns tipos de dados.**

Para utilizarmos a diretriz D6 neste meta-modelo, basta criarmos um pacote `PrimitiveTypes` e nele definirmos os tipos de dados primitivos que o meta-modelo necessita. Observe que cada meta-classe deve possuir um estereótipo `<<primitive>>`, o qual determina que ela deve definir, especificamente, elementos do tipo primitivo. O resultado pode ser conferido na Figura 27.

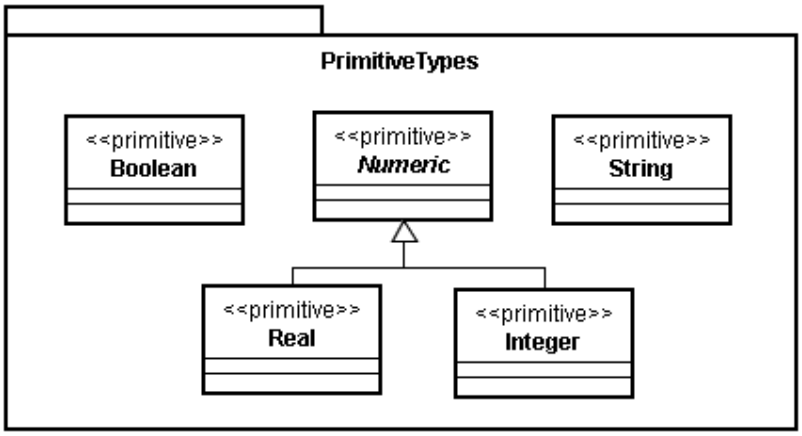



**Figura 27: Pacote PrimitiveTypes para o meta-modelo de ATL.**

Além deste exemplo, a diretriz D6 pode ser aplicada também em outros meta-modelos, os quais não possuem nenhum pacote `PrimitiveTypes` ou semelhante para definir os seus

tipos de dados primitivos. São eles: (i) CHORD [Silva, 2009]; (ii) Odyssey-FEX [Oliveira, 2005]; e (iii) KDM [KDM, 2007].

### 4.1.7 D7 - *Adding Core Package*

**Objetivo:** Facilitar a organização e o reuso das meta-classes intrínsecas do domínio e que servem como base para todo o meta-modelo.

**Solução:** No intuito de se obter uma melhor organização do meta-modelo, esta diretriz propõe a criação de um pacote principal, o qual devemos chamar de `Core` ou de qualquer outro nome que identifique o meta-modelo. Este pacote, diferentemente dos demais, deve conter todos os elementos intrínsecos do domínio do meta-modelo. A intenção é que outros meta-modelos reutilizem a totalidade ou parte do pacote `Core`, permitindo que eles se beneficiem com as semânticas e sintaxes abstratas que já foram definidas.

Além deste pacote principal, um meta-modelo pode ser organizado em vários outros, de forma a melhor estruturar suas definições. Por exemplo, o meta-modelo de SPEM possui um pacote principal chamado `Core` e vários outros, como por exemplo, o `ProcessStructure`. O pacote `Core` contém as meta-classes que servem como base para todos os outros pacotes deste meta-modelo. O `ProcessStructure`, por sua vez, contém os elementos estruturais básicos para definir processos de desenvolvimento. Portanto, um pacote `Core` deve ser mais geral e reutilizável em outros pacotes do mesmo ou de diferentes meta-modelos. Já os demais pacotes devem conter definições de conceitos mais específicos, como no exemplo citado, `ProcessStructure` define processos de desenvolvimento.

**Conseqüência:** Outros meta-modelos poderão reutilizar com maior facilidade partes ou a totalidade do meta-modelo desenvolvido de acordo com a diretriz D7, tendo em vista a sua boa organização em pacotes.

**Diretrizes Relacionadas:** D5, D6.

**Exemplo:** Para ilustrarmos a aplicação desta diretriz, vejamos como exemplo o Meta-modelo para Especificação de Arquiteturas de *Software* em Camadas [Silva e Paula, 2001]. Como podemos observar na Figura 28, este meta-modelo não foi estruturado em nenhum pacote, o que dificulta seu reuso, manutenção e evolução.

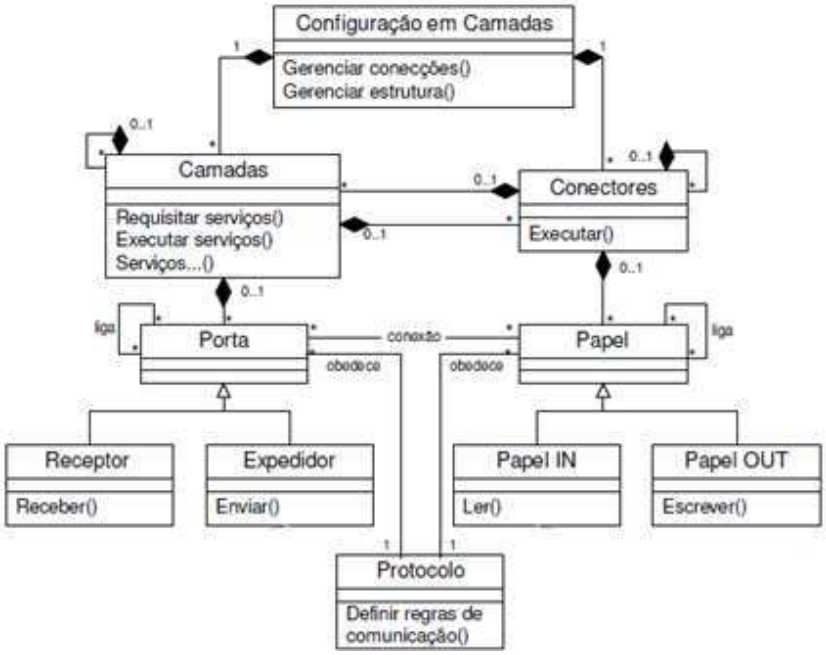



**Figura 28: Meta-modelo para Especificação de Arquiteturas de *Software* em Camadas.**

Para deixarmos este meta-modelo em conformidade com a diretriz D7, criamos um pacote global chamado `Core`, ilustrado na Figura 29, contendo as suas meta-classes. Contudo, vale ressaltar que, caso seja necessário, este pacote ainda pode ser estruturado em subpacotes, de forma a melhor organizar o meta-modelo.

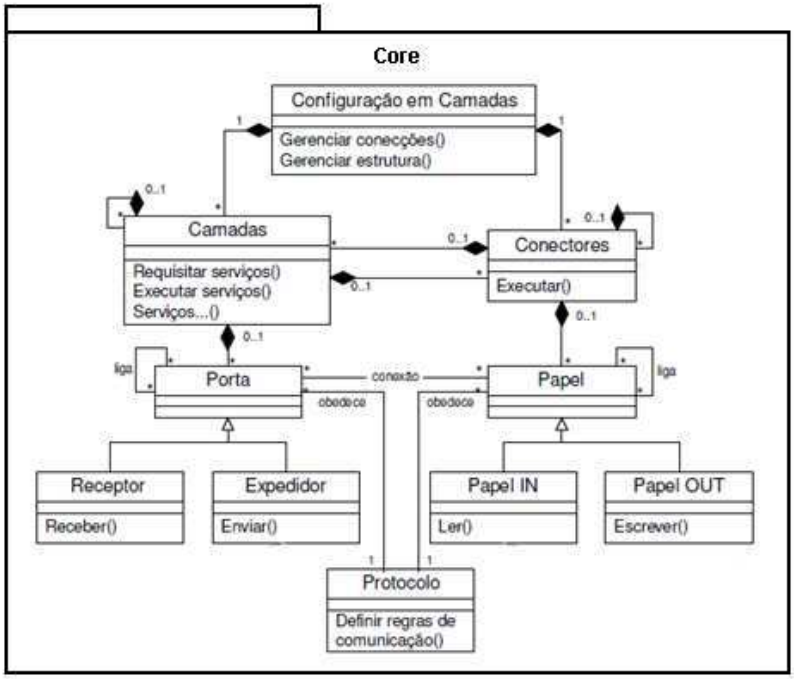



**Figura 29: Pacote Core para o Meta-modelo para Especificação de Arquiteturas de *Software* em Camadas.**

Além deste exemplo, a diretriz D7 pode ser aplicada também em outros meta-modelos, como por exemplo, no Meta-modelo de um Método de Desenvolvimento de Sistema [Jackowski, 2003], o qual não possui nenhum pacote para organizar suas meta-classes.

### 4.1.8 D8 - *Adding Utility Operations*

**Objetivo:** Evitar a repetição de expressões que servem para uma mesma finalidade e ocorrem mais de uma vez em restrições OCL de meta-classes.

**Motivação:** Quando estamos desenvolvendo meta-modelos muitas vezes utilizamos OCL para especificarmos restrições em meta-classes. Ao longo da meta-modelagem, pode ocorrer de precisarmos de uma mesma informação em restrições diferentes, isto é, uma mesma expressão OCL (ou até mesmo expressões definidas de maneiras diferentes) pode ser usada em diferentes partes do meta-modelo para uma mesma finalidade. Por exemplo, várias meta-classes do meta-modelo da UML precisam obter o limite máximo de uma multiplicidade (*upperBound*) para ser usado em suas restrições. Logo, se esta diretriz não for aplicada, a expressão OCL que obtém esta informação estará repetida em todas as meta-classes que a utilizam.

Dessa forma, o meta-modelo possuirá expressões OCL repetitivas e, muitas vezes, expressões diferentes que servem para o mesmo fim. Isto dificulta o seu entendimento e manutenção, pois se a expressão tiver que ser alterada, a modificação deverá ser feita em todas as restrições do meta-modelo que a definem. Além disso, o meta-modelo estará propício a erros devido às repetições e às diferentes formas de se definir as expressões.

**Solução:** A solução para evitarmos essa repetição de expressões nas restrições OCL do meta-modelo é criarmos uma operação adicional para cada expressão utilizada com freqüência. Caso tais operações possuam expressões apenas de consulta, sem causar nenhum efeito colateral, elas devem ser especificadas dentro de um *definition* (def). Por outro lado, se as operações possuírem expressões que causam efeitos colaterais, elas devem ser especificadas dentro de um *postcondition* (post). Portanto, esta diretriz sugere que ao invés de repetir a expressão em cada restrição OCL que a utiliza, tais restrições apenas devem invocar a operação adicional para obter a informação desejada.

**Conseqüência:** Haverá uma maior facilidade na manutenção e compreensão, uma vez que as expressões OCL úteis em várias partes do meta-modelo serão definidas apenas em um único local, nas operações. Por exemplo, se houver alguma mudança na expressão, ou seja, na forma de se obter o limite máximo de uma multiplicidade (*upperBound*), apenas a operação que obtém esta informação deverá ser alterada, e não todas as partes do meta-modelo que a utilizam. Outra vantagem é a diminuição da redundância e de erros que, possivelmente, poderiam ocorrer se o conjunto de restrições fosse definido nas diversas partes do meta-modelo, pois cada definição é independente e pode ser feita de formas diferentes. Além disso, as operações adicionais poderão ser reusadas em todo o meta-modelo, isto é, haverá um aumento do reuso.

**Aplicabilidade:** Pode ser aplicada em meta-modelos cujas restrições de algumas meta-classes possuam expressões OCL em comum, definidas de forma similar e para a mesma finalidade, isto é, quando há a repetição de expressões nas restrições do meta-modelo. A quantidade de aplicações da diretriz D8 é definida pelo número de operações adicionais criadas no meta-modelo com o intuito de evitar a repetição de expressões.

**Diretrizes Relacionadas:** Nenhuma.

**Exemplo:** Este exemplo mostra como a diretriz D8 deve ser aplicada no meta-modelo da UML. Vale salientar que algumas restrições OCL deste meta-modelo foram modificadas apenas para mostrar como seria um meta-modelo sem a utilização de operações utilitárias, pois, na realidade, ele já está em conformidade com a diretriz D8. No meta-modelo da UML, a restrição para obter o limite máximo de uma multiplicidade (*upperBound*) é útil em algumas de suas meta-classes, como *`MultiplicityElement`*, `Property` e `ExtensionEnd`. A seguir, vejamos como três restrições deveriam ser especificadas sem a utilização de uma operação adicional que obtenha o valor para o *upperBound* nestas três meta-classes, respectivamente:

[1] Especifica que uma multiplicidade deve definir, pelo menos, uma cardinalidade válida maior que zero. O atributo `upper` informa o limite máximo de multiplicidade, ele está definido em *`MultiplicityElement`*.

```
context MultiplicityElement
inv: let upperBound: UnlimitedNatural = if upper->notEmpty() then upper else 1 endif
in upperBound->notEmpty() implies upperBound > 0
```

[2] Especifica que a multiplicidade do todo em uma associação de composição não deve ter um limite máximo maior que `1`. Como `Property` estende a meta-classe *`StructuralFeature`*, a qual estende *`MultiplicityElement`*, o atributo `upper` pode ser usado por meio da herança.

```
context Property
inv: let upperBound: UnlimitedNatural = if upper->notEmpty() then upper else 1 endif
in isComposite implies (upperBound->isEmpty() or upperBound <= 1)
```

[3] Especifica que a multiplicidade de um `ExtensionEnd` deve ser `0..1` ou `1`. Como `ExtensionEnd` estende `Property`, o atributo `upper` pode ser usado por meio da herança.

```
context ExtensionEnd
inv: let upperBound: UnlimitedNatural = if upper->notEmpty() then upper else 1 endif
in (self->lowerBound() = 0 or self->lowerBound() = 1) and upperBound = 1
```

Como podemos observar nas expressões em destaque das restrições 1, 2 e 3, o valor do `upperBound` é igualmente especificado e utilizado em todas elas. Portanto, como esta é uma informação útil em várias partes do meta-modelo, podemos criar uma operação adicional para obtê-la. Ela deve ser criada em *`MultiplicityElement`*, pois a informação obtida pertence à esta meta-classe. Vejamos como o meta-modelo da UML definiu tal operação, a qual retorna o limite máximo de uma multiplicidade:

```
context MultiplicityElement
def: upperBound() : UnlimitedNatural  =  if upper->notEmpty() then upper else 1 endif
```

Dessa forma, basta as restrições que necessitam desta informação invocarem a operação *upperBound()*, simplificando bastante o meta-modelo. Vejamos o resultado a seguir:

```
[1] context MultiplicityElement
    inv: upperBound()->notEmpty() implies upperBound() > 0

[2] context Property
    inv: isComposite implies (upperBound()->isEmpty() or upperBound() <= 1)

[3] context ExtensionEnd
    inv:  (self->lowerBound() = 0 or self->lowerBound() = 1) and upperBound() = 1
```

Neste exemplo, a diretriz D8 foi aplicada uma única vez, pois apenas a operação *upperBound()* foi criada. Contudo, esta simples aplicação já nos leva a notar que a repetição de algumas expressões OCL foi reduzida, o que refletiu diretamente em três restrições do meta-modelo.

A diretriz D8 pode ser aplicada em vários meta-modelos: (i) KobrA2 [Atkinson et al., 2009], em que meta-classes, tais como `StructuralElement`, `ConstraintElement`, `BehavioralElement` e `StructuralBehavioralElement`, possuem restrições OCL utilizando expressões em comum; e (ii) CHRD [Silva, 2009], em que a meta-classe `CHR::CHRD::Program::Rule` possui quatro invariantes com expressões semelhantes, as quais podem ser definidas em uma operação adicional.

## Diretrizes para Definição de Valores *Default*

Neste tópico, serão apresentadas duas diretrizes que propõem uma melhoria nos meta-modelos a partir da definição de valores *default*: uma no escopo de atributos do tipo *boolean* e outra no escopo de *enumerations*.

**Motivação:** Quando estamos construindo a meta-classe de um meta-modelo, às vezes não paramos para observar os valores que seus atributos booleanos e seus *enumerations* podem assumir, de forma a tentarmos encontrar valores *default* caso nenhum valor seja informado para qualquer um destes elementos na instância da meta-classe. É importante definirmos valores *default*, principalmente, para atributos que nunca devem possuir valor nulo e sempre devem ter um valor associado caso nenhum seja informado, como é o caso dos atributos booleanos e dos *enumerations*.

**Conseqüência:** A utilização de valores *default* evitará que atributos tenham valores nulos, caso não seja definido nenhum valor no momento de sua instância. Além disso, vamos supor que a maioria das instâncias de uma meta-classe possui o mesmo valor para um determinado atributo. Logo, se definirmos este valor como *default* para o atributo, não será necessário defini-lo para cada instância da meta-classe.

### 4.1.9 D9 - *Defining Boolean Attributes Default Value*

**Objetivo:** Evitar a especificação de atributos booleanos em meta-classes de um meta-modelo sem valores *default* definidos.

**Solução:** Esta diretriz sugere a definição de valores *default* para os atributos booleanos presentes no meta-modelo, isto é, valores iniciais pré-definidos. Logo, caso o valor de um determinado atributo não seja informado na instância da sua meta-classe, ele possuirá o valor *default* estabelecido. Por outro lado, mesmo que um atributo tenha um valor *default* definido, este valor poderá ser sobrescrito sempre que for preciso. Logo, somente se fará necessário definir os valores, explicitamente, quando o valor do atributo é diferente do valor *default*.

Diante de várias análises em atributos booleanos de alguns meta-modelos, percebemos que não há um comportamento padrão para a definição de valores *default*. Em sua grande maioria, os atributos são especificados como `false`. Contudo, não podemos identificar isto como um padrão, de fato, porque existem casos em que eles são especificados como `true`, como é o caso de alguns atributos dos seguintes meta-modelos: UML 2.1.2 *Superstructure*, UML 2.1.2 *Infrastructure*, QVT, SPEM e CWM. Portanto, esta diretriz não tem a intenção de informar qual o valor que atributos do tipo *boolean* devem assumir como *default*, mas sim de mostrar a importância da definição de tais valores.

**Aplicabilidade:** Pode ser aplicada em qualquer atributo booleano do meta-modelo. A quantidade de aplicações desta diretriz está relacionada ao número de atributos do tipo *boolean* que passaram a ter valores *default* definidos.

**Diretrizes Relacionadas:** D10.

**Exemplo:** O exemplo que segue ilustra a aplicação da diretriz D9 no meta-modelo de SCADE [SCADE, 2006], o qual possui várias meta-classes com atributos sem nenhum valor *default*

definido. A Figura 30 ilustra duas meta-classes: `StorageUnit` e `LocalVariable`. Elas possuem alguns atributos booleanos, contudo, nenhum deles tem valor *default*.

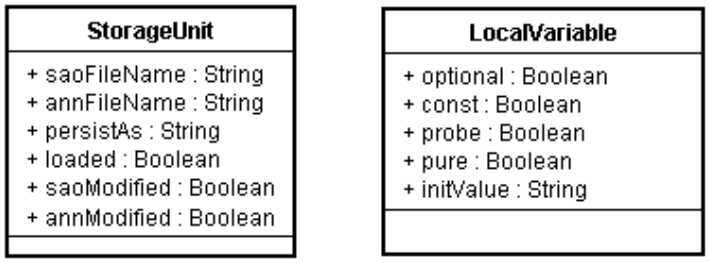



**Figura 30: Atributos do meta-modelo de SCADE sem valores *default*.**

Ao aplicarmos a diretriz D9 nestas meta-classes, devemos escolher um valor que deve ser o *default* para cada atributo. Vejamos na Figura 31 uma sugestão de como alguns atributos devem ser definidos depois da aplicação da diretriz. Neste exemplo, a diretriz D9 foi aplicada 12 vezes, considerando as duas meta-classes.

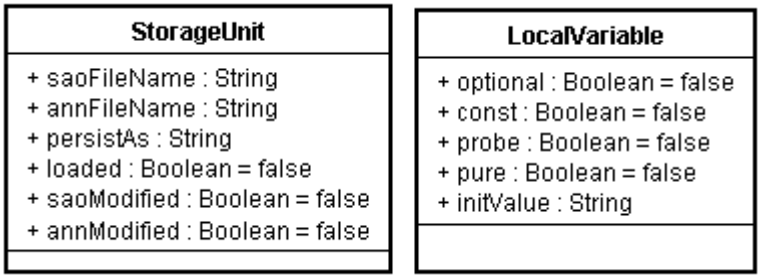



**Figura 31: Atributos do meta-modelo de SCADE com a aplicação da diretriz D9.**

Além deste exemplo, a diretriz D9 pode ser aplicada em outros meta-modelos que não possuem valor *default* definido para os atributos booleanos, como por exemplo: (i) Grafcet [Grafcet, 2005], com os atributos: `isInitial` e `isActive` pertencentes à meta-classe `Step`; (ii) QVT, com os atributos `one`, `isInverse` e `isDeferred`, pertencentes à meta-classe `ResolveExp` do pacote `QVTOperational`, dentre diversas outras meta-classes; e (iii) CWM, com os atributos `isUnique`, `isSorted` e `isPartitioning` pertencentes à meta-classe `Index`.

### 4.1.10 D10 - *Defining Enum Default Value*

**Objetivo:** Evitar a especificação de *enumerations* em um meta-modelo sem valores *default* definidos.

**Solução:** Esta diretriz sugere a definição de valores *default* para os *enumerations* existentes no meta-modelo, isto é, valores iniciais pré-definidos. Logo, caso o valor de um determinado *enumeration* não seja informado no momento de sua instância, ele possuirá o valor *default* estabelecido. Por outro lado, mesmo que um *enumeration* tenha um valor *default* definido, este valor poderá ser sobrescrito sempre que for preciso. Dessa forma, somente se fará necessário definir os valores, explicitamente, quando o valor do *enumeration* é diferente do valor *default*.

Diante de várias análises nos *enumerations* presentes em alguns meta-modelos, percebemos que não há um comportamento padrão para a definição de valores *default*. Em sua grande maioria, o *default* de um *enumeration* é especificado como sendo o primeiro valor declarado na lista de valores que ele pode assumir. Contudo, não podemos identificar isto como um padrão, de fato, porque existem casos em que o *default* é especificado como sendo qualquer outro valor da lista, e não o primeiro, como é o caso de alguns *enumerations* pertencentes ao meta-modelo da UML 2.1.2 *Superstructure*. Portanto, esta diretriz não tem a intenção de informar qual o valor que os *enumerations* devem assumir como *default*, mas sim de mostrar a importância da definição de tais valores.

**Aplicabilidade:** Pode ser aplicada em qualquer *enumeration* do meta-modelo. A quantidade de aplicações desta diretriz está relacionada ao número de *enumerations* que passaram a ter valores *default* definidos.

**Diretrizes Relacionadas:** D9.

**Exemplo:** A seguir, o exemplo mostra a aplicação da diretriz D10 no meta-modelo de QVT, o qual possui três *enumerations* sem nenhum valor *default* definido. São eles: `ImportKind`, `DirectionKind` e `EnforcementMode`. Podemos conferi-los na Figura 32.

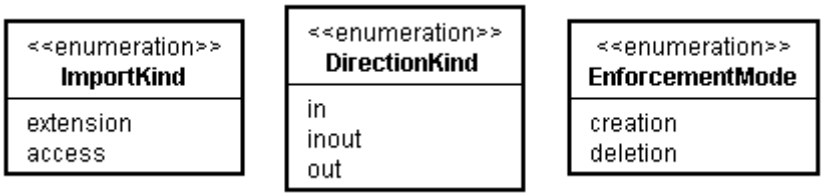



**Figura 32: *Enumerations* do meta-modelo de QVT sem valores *default*.**

Para aplicarmos a diretriz D10 nestes *enumerations* devemos escolher um dos possíveis valores que cada um pode assumir para ser definido como valor *default*. De preferência, sugere-se a escolha do valor que seja mais utilizado no meta-modelo. Dessa forma, este será o valor do *enumeration* caso ele não seja definido durante sua instância. Neste exemplo, a diretriz teve um total de três aplicações.

Além deste exemplo, a diretriz D10 pode ser aplicada também em outros meta-modelos que não possuem valor *default* definido para os seus *enumerations*, como por exemplo: (i) SPEM, com os *enumerations* chamados `WorkSequenceKind` e `OptionalityKind`; (ii) U2TP, com os *enumerations* chamados `OpKind` e `InteractionOperator`; e (iii) OCL, com o *enumeration* chamado `CollectionKind`.

### 4.1.11 D11 - *Defining Association Member Ends Features*

**Objetivo:** Considerando uma associação de composição entre meta-classes, esta diretriz sugere quais devem ser as características dos lados referentes ao todo e à parte da associação, de forma a evitar erros semânticos no meta-modelo.

**Motivação:** Para uma motivação desta diretriz, vejamos a seguinte situação hipotética: o meta-modelo da UML define uma meta-classe chamada `Class` para descrever como uma classe deve ser especificada em modelos e uma outra meta-classe chamada `Association` para descrever como uma associação deve ser especificada. Ambas as meta-classes são compostas por "`*`" meta-classes `Property`, a qual define como especificar uma propriedade em modelos. A parte desta associação (a meta-classe `Property`) é concreta e navegável a partir das duas meta-classes que representam o todo. Tendo em vista que cada meta-classe define como sua instância deve ser especificada, se definirmos a multiplicidade igual a `1` para as meta-classes `Class` e `Association`, como ilustra a Figura 33, estaremos dizendo que: uma propriedade obrigatoriamente deve pertencer a uma associação e a uma classe ao mesmo

tempo. Contudo, isto não é verdade, pois uma propriedade ou pertence a uma associação ou a uma classe, nunca aos dois simultaneamente.


**Figura 33: Trecho do meta-modelo da UML [com a multiplicidade alterada].**

Sem a noção destes conceitos de multiplicidade, muitas pessoas a definem de forma equivocada para o todo de uma associação de composição. Dessa forma, os meta-modelos poderão apresentar erros semânticos, comprometendo o seu entendimento e manutenção. Entretanto, vale salientar que a proposta desta diretriz é direcionada para os casos em que a parte da associação pode pertencer a mais de um todo, desde que não seja de forma simultânea. A utilização de tais associações exige muita atenção, principalmente, porque não há regras ou algum manual que indique como defini-las em meta-modelos. Além disso, algumas pessoas confundem modelagem com meta-modelagem durante a especificação de associações de composição.

**Solução:** Esta diretriz sugere que devemos, primeiramente, definir a multiplicidade como sendo `0..1` nas meta-classes que representam o todo da associação de composição, uma vez que, de acordo com a semântica do meta-modelo, seja permitido à meta-classe parte pertencer a mais de um todo, de forma não simultânea. Em outras palavras, temos que verificar se, semanticamente, faz sentido a meta-classe parte pertencer a mais de um todo. A fim de facilitar a evolução do meta-modelo, este procedimento também pode ser feito mesmo quando a parte estiver se relacionando com apenas um todo. Dessa maneira, de acordo com o exemplo citado na motivação, uma propriedade não estará obrigada a pertencer a uma classe e a uma associação ao mesmo tempo, mas sim apenas a uma destas meta-classes. Já o limite máximo da multiplicidade deve ser `1` para o todo porque em uma associação de composição, de fato, uma instância da parte só pode pertencer no máximo a uma instância do todo de cada vez.

Além da multiplicidade nas meta-classes que representam o todo de uma associação, devemos observar mais algumas características dos dois lados (*memberEnds*) da associação.

A diretriz D11 sugere que: (i) a parte seja uma meta-classe concreta, navegável a partir do todo e tenha o limite mínimo de multiplicidade igual a `0`; (ii) tanto a parte quanto o todo possuam atributos que podem ser escritos depois de inicializados; e (iii) o todo não possua atributos derivados.

**Conseqüência:** Os meta-modelos construídos de acordo com esta diretriz possuirão associações de composição mais fáceis de fazer manutenção. Além disso, haverá uma maior facilidade de futuras evoluções do meta-modelo, bem como de entendimento do domínio que se está meta-modelando.

**Aplicabilidade:** Pode ser aplicada nas associações de composição em que, de acordo com a semântica do meta-modelo, a meta-classe parte da associação pode pertencer a mais de um todo, contudo, não simultaneamente. A quantidade de aplicações desta diretriz é determinada pelo número de associações de composição em que qualquer um de seus lados tenha características alteradas como resultado da aplicação.

**Diretrizes Relacionadas:** Nenhuma.

**Exemplo:** O seguinte exemplo ilustra como esta diretriz deve ser aplicada no meta-modelo de PNML (*Petri Net Markup Language*) [PNML, 2005]. Como podemos ver na Figura 34, algumas características destas duas associações de composição já estão em conformidade com a diretriz D11: a parte é uma meta-classe concreta, navegável a partir das duas meta-classes que representam o todo e possui o limite mínimo de multiplicidade igual a `0`. Por outro lado, podemos notar que as meta-classes *`NetContent`* e `NetElement` são compostas por uma mesma meta-classe, chamada `Name`. Como a multiplicidade destas meta-classes está definida como `1`, o meta-modelo descreve que um `Name` deve pertencer tanto a um *`NetContent`* quanto a um `NetElement`, simultaneamente. De fato, isto não é verdade, pois a instância de um `Name` não pode pertencer a um *`NetContent`* e a um `NetElement` ao mesmo tempo, apenas a um deles.

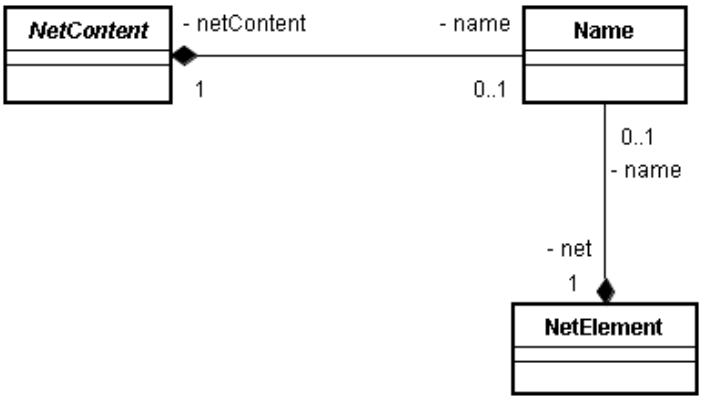



**Figura 34: Trecho do meta-modelo de PNML com duas associações de composição.**

A Figura 35 mostra como aplicarmos a diretriz D11: A multiplicidade das meta-classes `NetElement` e *`NetContent`* foi definida para `0..1`. Dessa forma, não haverá mais erros semânticos, uma vez que a instância de um `Name` somente pertencerá a um *`NetContent`* ou a um `NetElement` por vez. Neste exemplo, a diretriz D11 foi aplicada duas vezes, pois duas associações de composição tiveram suas multiplicidades redefinidas após a aplicação.

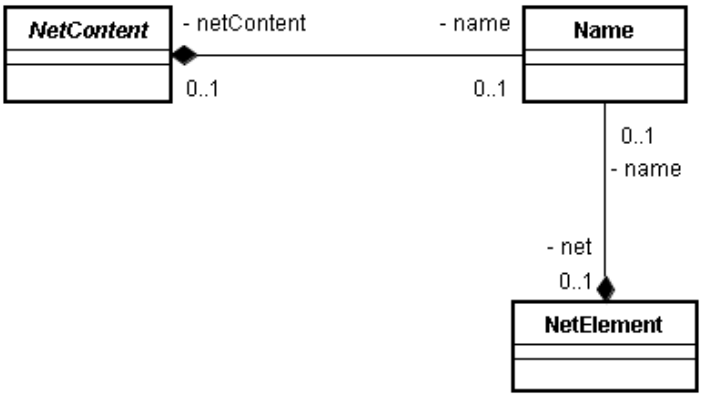



**Figura 35: Meta-modelo de PNML com a aplicação da diretriz D11.**

Além deste exemplo, a diretriz D11 pode ser aplicada também em outros meta-modelos, como por exemplo: (i) U2TP, em que as meta-classes `TestContext` e `TestCase` são compostas pela meta-classe `Behavior`; (ii) DTD [DTD, 2005], em que a meta-classe *`AttributeDescription`* é composta pela meta-classe *`AttributeType`*; e (iii) AgilPro [Bauer et al., 2007], em que as meta-classes `Data` e `Application` são compostas pela meta-classe `Parameter`.

## Diretrizes para Definição de Nomes

Neste tópico, serão apresentadas duas diretrizes que visam uma melhor definição de nomes, tanto de atributos booleanos quanto de *enumerations*, de forma que isto se torne uma convenção para facilitar o entendimento e a manutenção de meta-modelos.

**Objetivo:** O objetivo das duas diretrizes que serão discutidas neste tópico é apresentar as melhores práticas e convenções que, geralmente, vêm sendo empregadas na definição de nomes de atributos e *enumerations*.

**Motivação:** Durante a meta-modelagem alguns desenvolvedores podem ter várias dúvidas relacionadas à melhor forma de se definir um nome para um atributo booleano ou para um *enumeration* do meta-modelo. Na dúvida, alguns acabam definindo nomes de difícil entendimento ou sem relação alguma com o contexto, assim, dificultando a compreensão e a manutenção do meta-modelo.

**Conseqüência:** Um meta-modelo construído de acordo com estas diretrizes apresentará uma grande melhoria da compreensão e manutenção.

**Diretrizes Relacionadas:** Nenhuma.

### 4.1.12 D12 - *Redefining Boolean Attribute Names*

**Solução:** Devemos prestar bastante atenção na hora de definirmos os nomes de atributos do meta-modelo, pois é importante que eles sejam significativos, únicos (dentro de seu contexto) e legíveis. Em relação aos atributos booleanos, em especial, uma boa prática é definir seus nomes sempre começando pelos prefixos *is* ou *has*, logo, eles serão facilmente identificados e entendidos no meta-modelo. Por exemplo, é bem mais fácil entendermos o significado de um atributo booleano chamado `isRoot` do que chamado `root`. O prefixo *is* no nome nos faz entender que o atributo pode assumir apenas dois valores: *true* ou *false*. Desta forma, facilmente podemos identificar que se trata de um booleano. Já o atributo chamado `root` é muito vago, além de não transparecer os possíveis valores que ele pode assumir, é mais difícil conseguirmos distinguir se ele é do tipo *boolean*, *String*, *Integer*, dentre outros.

**Aplicabilidade:** Pode ser aplicada para redefinir nomes de qualquer atributo booleano do meta-modelo. A quantidade de aplicações desta diretriz é determinada pelo número de atributos do tipo *boolean* que tiverem seus nomes redefinidos.

**Exemplo:** Vejamos uma meta-classe do meta-modelo de SCADE para ilustrarmos a aplicação da diretriz D12. A Figura 36 mostra que a meta-classe `LocalVariable` possui cinco atributos, dentre os quais, quatro são do tipo *boolean*: `optional`, `const`, `probe` e `pure`.

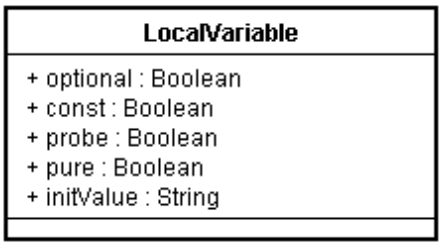



**Figura 36: Meta-classe do meta-modelo de SCADE com atributos booleanos.**

Contudo, a diretriz sugere que os nomes de atributos booleanos se iniciem com os prefixos *is* ou *has*. Então, ao aplicarmos a diretriz D12 neste meta-modelo, teremos os atributos com os seguintes nomes redefinidos: `isOptional`, `isConst`, `isProbe` e `isPure`. Neste exemplo, a diretriz teve um total de quatro aplicações, pois quatro de seus atributos booleanos tiveram seus nomes redefinidos.

Além deste exemplo, a diretriz D12 pode ser aplicada também em outros meta-modelos cujas meta-classes não possuem atributos booleanos com o prefixo *is* ou *has*, como por exemplo: (i) J2SE5, com os atributos `expressionInitialized`, `static`, `constructor` e `proxy`, pertencentes às meta-classes `SwitchCase`, `ImportDeclaration`, `MethodDeclaration` e *`NamedElement`*, respectivamente; (ii) em vários atributos do meta-modelo de CHORD, como por exemplo, `closedClasses`, `closedInstances`, `closedClassFeatures` e `closedInstanceFeatures`, pertencentes à meta-classe `WorldAssumption`; e (iii) em vários atributos do meta-modelo de XHTML, como por exemplo, `checked`, `disabled` e `readonly` pertencentes à meta-classe `Input`, e nos atributos `multiple` e `disabled` pertencentes à meta-classe `Select`.

### 4.1.13 D13 - *Redefining Enum Names*

**Solução:** Esta diretriz propõe que todo nome de *enumeration* possua o sufixo *Kind*. Os nomes de *enumerations* e de qualquer meta-classe sempre devem possuir a primeira letra maiúscula, até mesmo quando o nome é formado por mais de uma palavra. Estas convenções de nomes podem auxiliar bastante o entendimento e a manutenção do meta-modelo, uma vez que os *enumerations* serão facilmente identificados.

**Aplicabilidade:** Pode ser aplicada em qualquer *enumeration* do meta-modelo. A quantidade de aplicações desta diretriz é determinada pelo número de *enumerations* que tiverem seus nomes redefinidos.

**Exemplos:** O exemplo que segue ilustra a aplicação da diretriz D13 no meta-modelo de XHTML, o qual possui um total de 11 *enumerations*. Contudo, nenhum deles está em conformidade com a diretriz. A Figura 37 ilustra três *enumerations* deste meta-modelo, chamados `Scope`, `Direction` e `Shape`:

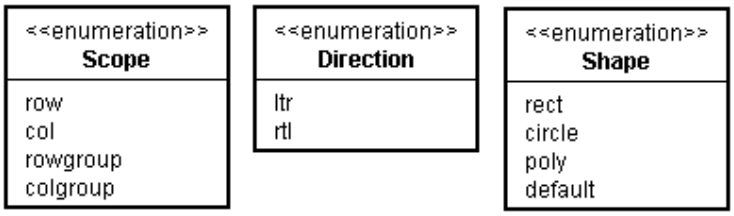



**Figura 37: Três enumerations do meta-modelo de XHTML.**

Para aplicarmos a diretriz D13 nestes três *enumerations*, devemos adicionar o sufixo *Kind* em cada um dos seus nomes. Portanto, após a aplicação desta diretriz, os nomes passam a ser: `ScopeKind`, `DirectionKind` e `ShapeKind`. No exemplo ilustrado, a diretriz teve apenas três aplicações. Contudo, se considerarmos todo o meta-modelo de XHTML, teremos um total de onze aplicações, pois nenhum de seus *enumerations* possui o sufixo *Kind*.

Além deste exemplo, a diretriz D13 pode ser aplicada também em outros meta-modelos cujas meta-classes não possuem o sufixo *Kind* nos nomes dos *enumerations*, como por exemplo: (i) XSchema [XSchema, 2005], com o *enumeration* chamado `AttributeUseType`; (ii) CHRD, com o *enumeration* chamado `Connective`; e (iii) COBOL [Cobol, 2005], com o *enumeration* chamado `COBOLUsageValue`.

**Sumário das Diretrizes**

| Sigla | Nome |
|---|---|
| **D1** | *Abstracting Common Attributes* |
| **D2** | *Abstracting Common Associations* |
| **D3** | *Generalizing Common Attributes* |
| **D4** | *Grouping Related Metaclasses* |
| **D5** | *Adding Abstractions Package* |
| **D6** | *Adding PrimitiveTypes Package* |
| **D7** | *Adding Core Package* |
| **D8** | *Adding Utility Operations* |
| **D9** | *Defining Boolean Attributes Default Value* |
| **D10** | *Defining Enum Default Value* |
| **D11** | *Defining Association Member Ends Features* |
| **D12** | *Redefining Boolean Attribute Names* |
| **D13** | *Redefining Enum Names* |

## 4.2 Diretrizes Aplicáveis em Modelos

Apesar de pertencerem à camadas diferentes da arquitetura de quatro camadas proposta pela OMG, a construção de modelos e meta-modelos possui algumas similaridades. Nesse sentido, muitas das diretrizes para meta-modelos apresentadas anteriormente podem ser aplicadas também em modelos. A seguir, vejamos exemplos de aplicação destas diretrizes em modelos.

**D3 – *Generalizing Common Attributes*:** No exemplo ilustrado na Figura 38-A podemos ver um relacionamento de herança, no qual tanto a classe `Student` quanto `Teacher` estende *`Person`*. Como podemos notar, as duas subclasses da classe *`Person`* possuem um atributo em comum, chamado `telephone`. Ao aplicar a diretriz D3, este atributo foi especificado apenas na superclasse, como mostra a Figura 38-B.

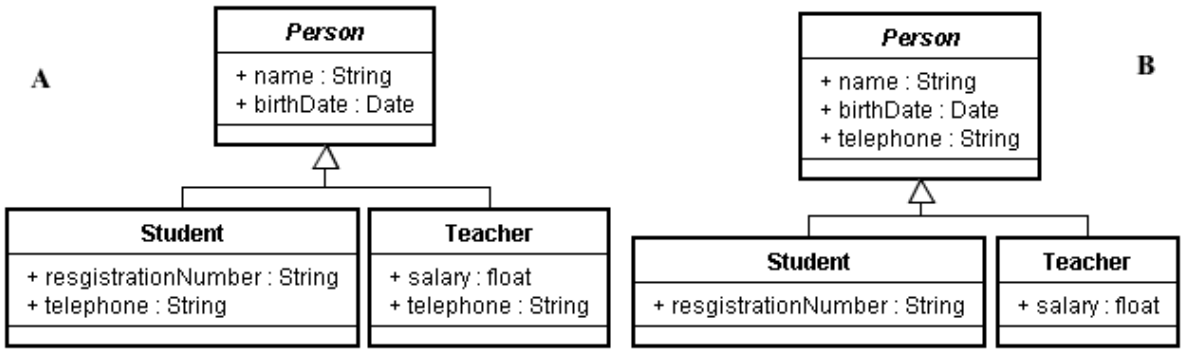



**Figura 38: A. Modelo antes da aplicação da D3; B. Modelo depois da aplicação da D3.**

**D4 – *Grouping Related Metaclasses*:** No exemplo ilustrado na Figura 39 podemos ver um relacionamento de associação entre as classes `Parcel` e *`Payment`*. Contudo, apenas os pagamentos com `Visa` ou `Master` podem ser parcelados.

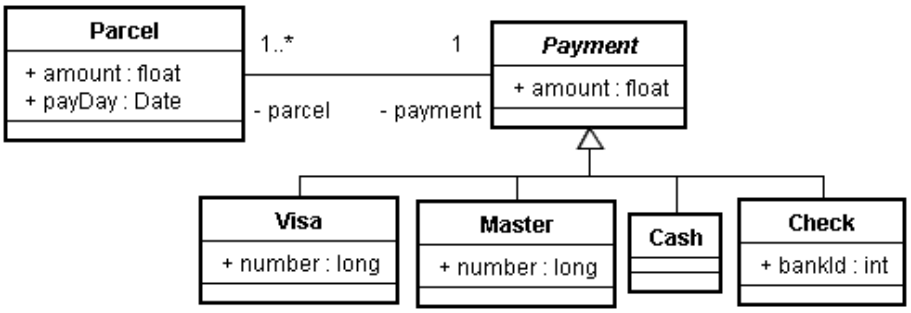



**Figura 39: Modelo antes da aplicação da D4.**

Nesse sentido, a seguinte restrição OCL foi definida para especificarmos que apenas as subclasses `Visa` e `Master` devem se relacionar com `Parcel`.

```
context Parcel
inv: self.payment.oclIsTypeOf(Visa) or self.payment.oclIsTypeOf(Master)
```

Ao aplicar a diretriz D4, a classe abstrata *`CreditCard`* foi criada para generalizar apenas as formas de pagamento que podem ser parceladas. Portanto, não houve mais a necessidade da restrição OCL apresentada anteriormente. O resultado da aplicação da D4 pode ser conferido na Figura 40.

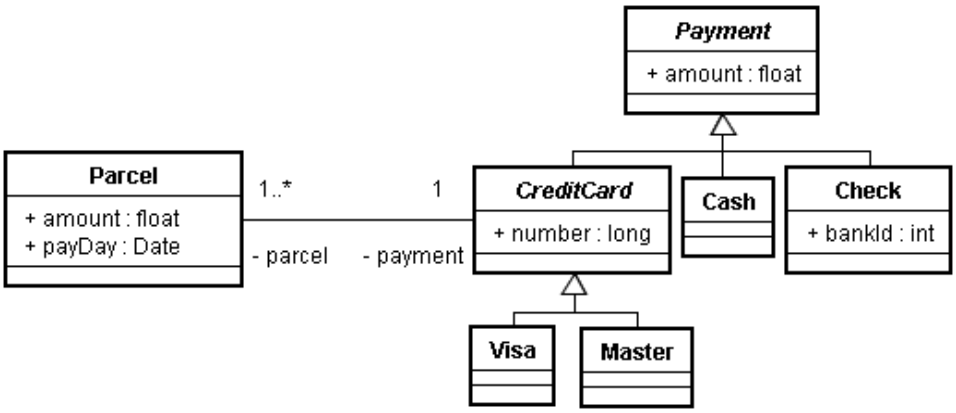



**Figura 40: Modelo depois da aplicação da D4.**

**D7 – *Adding Core Package*:** No exemplo ilustrado na Figura 41-A podemos ver um simples modelo que foi especificado sem a utilização de nenhum pacote. Ao aplicar a diretriz D7, um pacote global chamado `Core` foi criado, como mostra a Figura 41-B, contendo as classes intrínsecas do modelo.

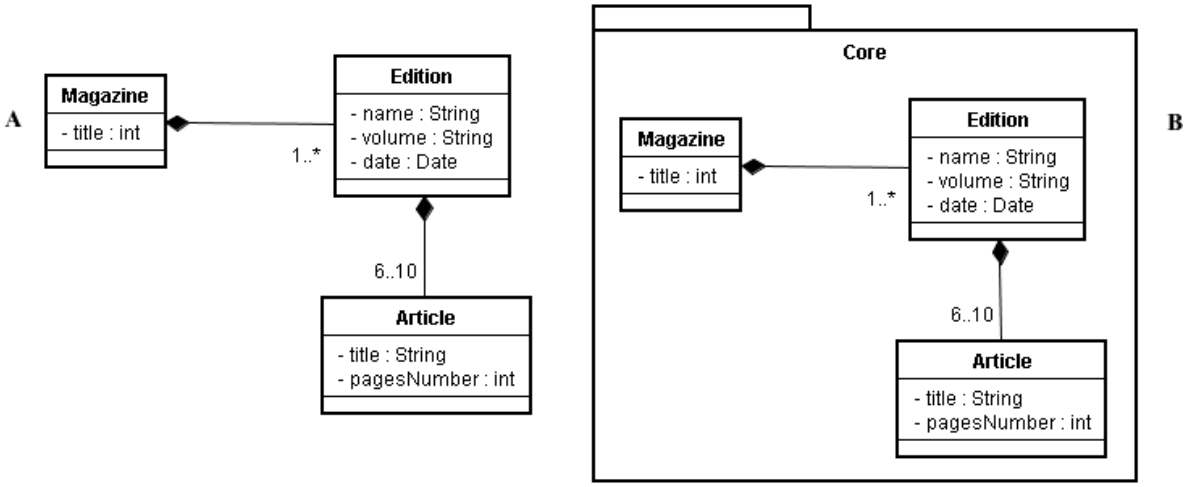



**Figura 41: A. Modelo antes da aplicação da D7; B. Modelo depois da aplicação da D7.**

**D8 – *Adding Utility Operations*:** No modelo ilustrado na Figura 42 podemos ver que uma pessoa pode ter vários dependentes. Contudo, as subclasses `Client` e `Employee` especificam algumas restrições OCL para seus dependentes, as quais são apresentadas a seguir.

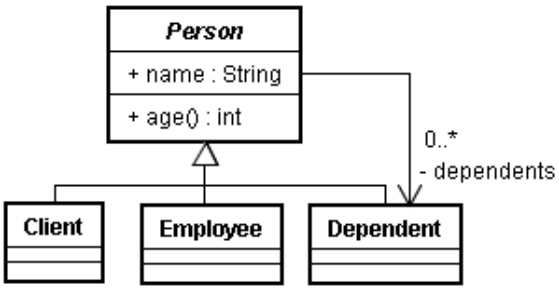



**Figura 42: Simples modelo para ilustrar a aplicação da D8.**

[1] Especifica que um cliente pode ter no máximo cinco dependentes.

```
context Client
inv: let hasDependent: Boolean = if self.dependents->notEmpty() then true else false endif
in hasDependent implies self.dependents->size() <= 5
```

[2] Especifica que um empregado apenas pode ter dependentes de até 18 anos.

```
context Employee
inv: let hasDependent: Boolean = if self.dependents->notEmpty() then true else false endif
in hasDependent implies self.dependents->forAll(d | d.age() <= 18)
```

Como podemos observar, nas restrições [1] e [2] existe uma expressão em comum. Portanto, ao aplicarmos a diretriz D8 especificamos uma operação adicional chamada `hasDependent()` no contexto da classe *`Person`*, uma vez que `Client` e `Employee` a estendem. Vejamos a nova operação a seguir.

```
context Person
def: hasDependent() : Boolean  =  if self.dependents->notEmpty() then true else false endif
```

As restrições [1] e [2] passam agora a invocar a operação criada, desta forma, diminuindo a redundância das expressões. Vejamos o resultado a seguir.

```
[1] context Client
    inv: self.hasDependent() implies self.dependents->size() <= 5
```

```
[2] context Employee
    inv: self.hasDependent() implies self.dependents->forAll(d | d.age() <= 18)
```

**D9 – *Defining Boolean Attributes Default Value*:** No pequeno trecho de modelo ilustrado na Figura 43-A podemos ver que uma `Library` pode conter vários `Books`. A classe `Book`, por sua vez, possui um atributo booleano chamado `available` (sem nenhum valor *default* definido) para indicar se o livro está ou não disponível. Ao aplicar a diretriz D9, definimos `true` como sendo o valor *default* para o atributo booleano `available`, como ilustra a Figura 43-B. Desta forma, um livro já é cadastrado na biblioteca como disponível.

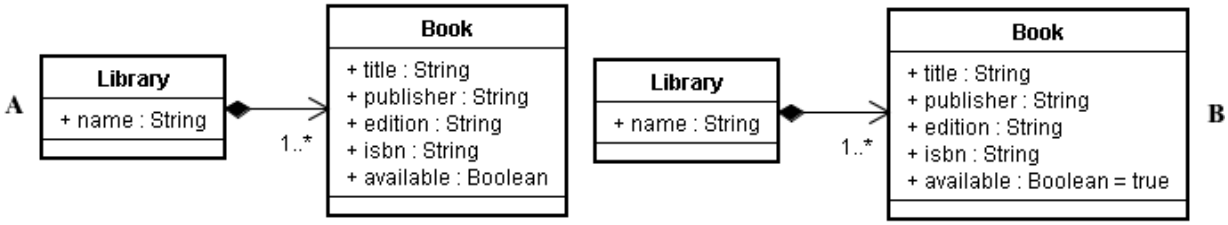



**Figura 43: A. Modelo antes da aplicação da D9; B. Modelo depois da aplicação da D9.**

**D10 – *Defining Enum Default Value*:** O exemplo que segue ilustra, na Figura 44, uma classe `Thread` que possui um atributo `priority` que, por sua vez, é do tipo de um *enumeration* especificado no modelo. Para aplicar a diretriz D10 neste simples exemplo, podemos definir o valor *default* do *enumeration* `Priority` como sendo `normal`. Desta forma, uma *thread* é inicialmente criada com uma prioridade `normal`, caso nenhuma prioridade seja definida para ela.

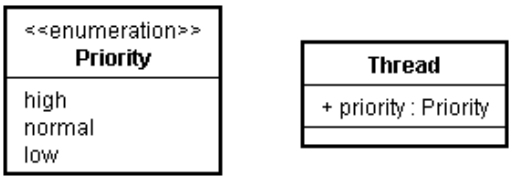



**Figura 44: Exemplo de modelo com um *enumeration*.**

**D12 – *Redefining Boolean Attribute Names*:** Um exemplo de atributo booleano sem a aplicação desta diretriz pode ser visto na classe `Book` apresentada anteriormente na Figura 43, o qual se chama `available`. Aplicando a diretriz D12 neste atributo, o seu nome passa a ser is`Available`.

**D13 – *Redefining Enum Names*:** Um exemplo de *enumeration* sem a aplicação desta diretriz pode ser visto na Figura 44 (apresentada anteriormente), o qual se chama `Priority`. Aplicando a diretriz D13 neste *enumeration*, o seu nome passa a ser `PriorityKind`.

## 4.3 Suporte Ferramental

Uma ferramenta de suporte foi desenvolvida com a finalidade de auxiliar os desenvolvedores na aplicação das diretrizes em seus meta-modelos já existentes de forma automática. Esta ferramenta foi implementada em ATL. Além disso, ela possui uma licença GPL. Desta forma, ela está disponível em [Vieira e Ramalho, 2009] para toda a comunidade, de forma que qualquer outra pessoa possa ter acesso e estendê-la com a implementação de novas diretrizes.

A arquitetura da ferramenta foi definida como ilustra a Figura 45. Para cada diretriz foi implementada uma regra ATL correspondente, como por exemplo, a regra chamada `AbstractingCommonAttributes.atl`, que foi implementada para a diretriz D1. Então, para a execução de qualquer uma das regras ATL, é necessário que o usuário informe um meta-modelo de entrada no formato *.ecore*, isto é, o meta-modelo em que deseja aplicar as diretrizes. Logo, a ferramenta irá executar a regra ATL que o usuário deseja no meta-modelo informado. Como resultado da execução, será gerado um arquivo de saída no formato *.ecore* correspondente ao meta-modelo informado pelo usuário com a diretriz já aplicada.

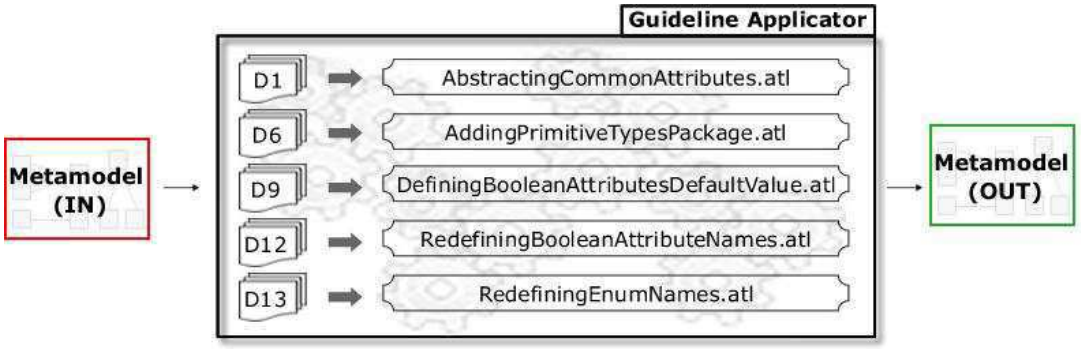



**Figura 45: Arquitetura da ferramenta de suporte para aplicação das diretrizes.**

Para ilustramos a implementação destas regras ATL, vejamos alguns trechos de código da regra chamada `RedefiningEnumNames.atl`, como mostra o Código Fonte 1. Neste exemplo, a *matched rule*[4] é a regra `RedefiningEnumNames` (linhas 1-5). Ela obtém cada

[4] São regras que podem possuir código tanto declarativo quanto imperativo. Elas permitem ao desenvolvedor especificar como os elementos do meta-modelo de entrada devem gerar os elementos no meta-modelo de saída.

pacote do meta-modelo de entrada, o copia no meta-modelo de saída (como podemos ver no bloco *to* da linha 3), e invoca a *called rule*[5] `copyAllEnums`. Esta, por sua vez, verifica se os *enumerations* do pacote de entrada estão em conformidade com a diretriz D13, caso não estejam, invoca a *called rule* `applyD13` (como podemos ver na linha 11) para copiar os *enumerations* deste pacote com os nomes já redefinidos no pacote previamente copiado no meta-modelo de saída (como podemos ver no bloco *to* da linha 16).

**Código Fonte 1: Regra RedefiningEnumNames.atl**

```
1  rule RedefiningEnumNames {
2       from pkgIn: EMF!EPackage
3       to pkgOut: EMF!EPackage ( name <- pkgIn.name )
4       do { self.copyAllEnums(pkgOut, pkgIn); ... }
5  }
6  rule copyAllEnums(pkgOut: EMF!EPackage, pkgIn: EMF!EPackage) {
7       do { ...
8            for (enum in pkgIn.eClassifiers->select(
9                 e|e.oclIsTypeOf(EMF!EEnum))) {
10                if (not enum.name.endsWith('Kind')
11                     self.applyD13(enum, pkgOut);
12           }
13      }
14 }
15 rule applyD13(enumIn: EMF!EEnum, pkgOut: EMF!EPackage) {
16      to enumOut: EMF!EEnum ( name <- enumIn.name + 'Kind' )
17      do {
18           for (literal in enumIn.eLiterals) {
19                self.copyEnumLiteral(literal, enumOut);
20           }
21           pkgOut.eClassifiers <- pkgOut.eClassifiers->including(enumOut); ...
22      }
23 }
```

[5] São regras que possuem apenas código imperativo, a partir do qual o desenvolvedor pode gerar os elementos no meta-modelo de saída.

Como se pode observar, as regras ATL não copiam todo o meta-modelo de entrada no meta-modelo de saída, mas apenas os elementos envolvidos na aplicação da diretriz.

Para uma versão inicial desta ferramenta de suporte escolhemos cinco das diretrizes identificadas, tomando como critério a facilidade de implementação. Portanto, implementamos as mais simples e disponibilizamos a ferramenta para uma futura extensão. Vejamos cada uma destas diretrizes e uma breve descrição de como suas respectivas regras ATL funcionam:

- *D1 - Abstracting Common Attributes*: procura por todos os atributos que existem em comum nas meta-classes, considerando todos os pacotes e subpacotes. Então, caso existam, cria uma meta-classe abstrata para generalizar cada um deles, de forma que as meta-classes que antes definiam tais atributos passam a estendê-la. As novas meta-classes criadas são adicionadas no pacote `Abstractions` do meta-modelo, caso ele ainda não exista, será criado;

- *D6 - Adding PrimitiveTypes Package*: procura por todos os tipos de dados primitivos existentes no meta-modelo, considerando todos os pacotes e subpacotes. Então, cria um pacote chamado `PrimitiveTypes`, se ainda não existir, e adiciona nele esses tipos primitivos encontrados;

- *D9 - Defining Boolean Attributes Default Value*: procura por todos os atributos booleanos do meta-modelo que ainda não tenham valores *default* definidos e apresenta-os para o usuário;

- *D12 - Redefining Boolean Attribute Names*: procura por todos os atributos booleanos do meta-modelo cujos nomes não estejam adequados para o que eles representam e os redefine de acordo com algumas convenções;

- *D13 - Redefining Enum Names*: procura por todos os *enumerations* do meta-modelo cujos nomes não estejam de acordo com a convenção e os redefine.

# Capítulo 5

# Avaliação Experimental

Este capítulo descreve como foi realizada a avaliação da aplicabilidade das diretrizes propostas neste trabalho. O processo de avaliação foi baseado nos conceitos de uma abordagem bastante adotada na engenharia de *software* para a medição de produtos e processos de *software*, chamada GQM (*Goal, Question, Metric*). Na seqüência, o capítulo descreve cada fase do processo de avaliação com base neste método.

## 5.1 Fase de Definição do Processo de Avaliação - GQM

De acordo com a abordagem GQM, a fase de definição deve contemplar o plano GQM, o qual tem por finalidade estabelecer o(s) objetivo(s) da avaliação, a(s) questão(ões) e a(s) métrica(s). A Figura 46 mostra o plano GQM da nossa avaliação experimental destacando cada um destes três elementos.

Como podemos observar nesta figura, no contexto do nosso trabalho foi definido um objetivo macro, a partir do qual duas questões foram extraídas. A primeira determina duas métricas para avaliar as diretrizes de forma quantitativa. Já a segunda trata-se de uma questão subjetiva, a qual apresenta uma discussão de caráter qualitativo para demonstrar a nossa análise particular em relação à melhoria da qualidade dos meta-modelos após a aplicação das diretrizes. Vale ressaltar que para esta análise qualitativa não foi avaliada nenhuma métrica, trata-se apenas de um sentimento particular nosso. As métricas definidas para a Questão 1 serão abordadas na subseção 5.1.3, enquanto a discussão qualitativa para melhor respondermos a Questão 2 será apresentada na subseção 5.4.2.

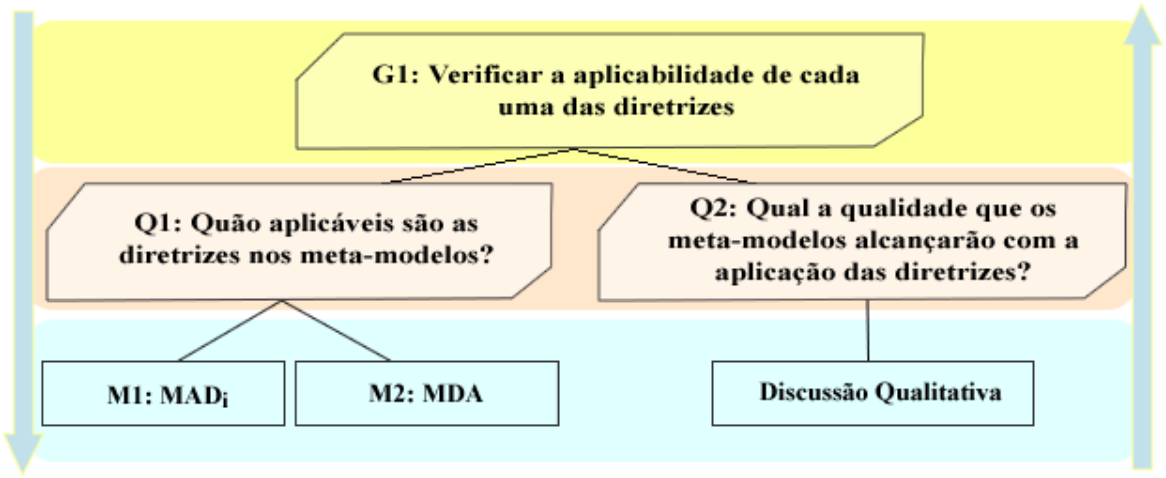



**Figura 46: Plano GQM.**

De acordo com a Figura 46, podemos interpretar a relação entre objetivo, questões e métricas da seguinte maneira: o objetivo identifica a ação que estamos querendo realizar; as questões definem o caminho para alcançar este objetivo e caracterizam o objeto de estudo no contexto de qualidade; e, por fim, as métricas e a discussão qualitativa são apresentadas através de dados objetivos e subjetivos necessários para responder às questões elaboradas e quantificar o objetivo. Vejamos, ao longo desta subseção, a descrição detalhada de cada um destes elementos.

### 5.1.1 Objetivo

O processo de avaliação descrito neste capítulo foi realizado com o único e macro objetivo de verificar a aplicabilidade de cada uma das diretrizes propostas para meta-modelos. Este objetivo foi definido a partir da necessidade de se analisar cada uma das diretrizes, individualmente, a fim de descobrir quão aplicáveis e úteis elas podem ser. Nesta avaliação, as diretrizes foram aplicadas em meta-modelos diversificados construídos por diferentes autores. Dessa forma, foi possível verificarmos se elas são independentes de domínio, podendo ser aplicadas em meta-modelos que definem a sintaxe abstrata de diversas linguagens, como por exemplo, de processo de desenvolvimento e de programação.

Com a finalidade de melhorar o entendimento do objetivo da avaliação e do contexto geral em que ele se encontra, o método GQM propõe a definição de alguns elementos para a identificação de metas. Tais elementos são apresentados na Tabela 3.

**Tabela 3: Definição do nosso objetivo de acordo com o modelo GQM.**

| | |
|---|---|
| **Objeto de Estudo** | Diretrizes |
| **Propósito** | Análise |
| **Foco de Qualidade** | Aplicabilidade |
| **Ponto de Vista** | Equipe do projeto |
| **Contexto** | Laboratório de pesquisa no qual a equipe do projeto aplicou as diretrizes em meta-modelos consolidados |

De acordo com a Tabela 3: o (i) "objeto de estudo" identifica o que será analisado, ou seja, as diretrizes identificadas neste trabalho; (ii) o "propósito" identifica o motivo pelo qual o objeto será analisado, ou seja, para análise; (iii) o "foco de qualidade" identifica o atributo do objeto que será analisado, ou seja, a aplicabilidade; (iv) o "ponto de vista" identifica quem utilizará as métricas coletadas para analisar o objeto, ou seja, a equipe do projeto; e o (v) "contexto" identifica o ambiente onde o processo de avaliação será executado, ou seja, o laboratório de pesquisa no qual a equipe do projeto aplicou as diretrizes em meta-modelos consolidados. Logo, a tabela pode ser interpretada da seguinte forma: o propósito deste estudo é analisar as diretrizes com foco na aplicabilidade a partir do ponto de vista da equipe do projeto no contexto do laboratório de pesquisa pelo qual o presente trabalho foi desenvolvido.

### 5.1.2 Questões

Após a definição do objetivo macro da avaliação, algumas questões que caracterizam o objeto de estudo foram elaboradas a fim de que suas respostas possam melhor esclarecer o objetivo da avaliação. A seguir, vejamos cada uma delas:

- **Questão 1: Quão aplicáveis são as diretrizes nos meta-modelos?**

  Cada diretriz possui o seu nível de aplicabilidade. Algumas podem ser aplicadas com maior freqüência do que outras em determinado meta-modelo e pode ocorrer de algumas nem serem aplicadas, caso o meta-modelo já esteja em conformidade com elas. Com esta questão, queremos saber a aplicabilidade de cada uma das diretrizes nos meta-modelos analisados nesta avaliação experimental. As métricas consideradas para responder esta questão foram: (i) $MAD_i$ (Média de Aplicações de uma Diretriz), onde o índice *i* representa a diretriz; e (ii) MDA (Média de Diretrizes Aplicadas).

- **Questão 2: Qual a qualidade que os meta-modelos alcançarão com a aplicação das diretrizes?**

  Os meta-modelos irão obter grandes benefícios com a aplicação das diretrizes, mesmo que haja um pequeno número de aplicações, pois o meta-modelo pode já estar parcialmente em concordância com as diretrizes, faltando aprimorar apenas algumas particularidades. Dessa forma, haverá uma considerável elevação da qualidade dos meta-modelos, que é justamente o que esta questão pretende focalizar. Para responder esta questão subjetiva, foi considerada a discussão de caráter qualitativo apresentada na subseção 5.4.2.

### 5.1.3 Métricas

Uma vez que as questões foram elaboradas, o próximo passo foi definir as métricas. Estas, por sua vez, foram definidas apenas para a Questão 1, pois apenas esta oferece uma análise de caráter quantitativo. Vejamos cada uma delas seguida de uma breve descrição:

- **$MAD_i$: Média de Aplicações de uma Diretriz**

  Em um meta-modelo, uma mesma diretriz pode ser aplicada várias vezes. Portanto, esta métrica é referente à média do número de vezes que determinada diretriz foi aplicada nos meta-modelos avaliados. Ela é calculada para cada diretriz, a qual representamos com o índice *i*. O número de aplicações de uma diretriz em determinado meta-modelo é o número de vezes que ela foi aplicada nele, e é calculado de maneira específica para cada diretriz. Por exemplo, na diretriz D1 (*Abstracting Common Attributes*) o número de aplicações é a quantidade de meta-classes abstratas criadas ao final da sua aplicação. Já na diretriz D8 (*Adding Utility Operations*), é a quantidade de operações adicionais criadas. A *média de aplicações da diretriz "i"* é calculada como segue: obtemos o somatório do número de aplicações desta diretriz em cada meta-modelo analisado e depois dividimos o resultado pelo número de meta-modelos analisados. Vejamos a métrica a seguir:

$$MAD_i = \sum_{j=1}^{m} NA_{i,j} \; / \; m$$

Onde, *NA* é o Número de Aplicações da diretriz *i*; *i* é a diretriz que se deseja calcular a média de aplicações; *j* é cada um dos meta-modelos analisados em que a diretriz foi aplicada; e *m* é o número de meta-modelos analisados.

- **MDA: Média de Diretrizes Aplicadas**

  Em um meta-modelo pode haver a aplicação de mais de uma diretriz. Portanto, esta métrica informa a média do número de diretrizes que foram aplicadas em todos os meta-modelos avaliados. Logo, o valor máximo que esta métrica pode assumir é 13, caso todas as diretrizes sejam aplicadas em todos os meta-modelos, pois temos um total de 13 diretrizes. A *média de diretrizes aplicadas* é calculada como segue: obtemos o somatório do número de diretrizes aplicadas em cada meta-modelo analisado e depois dividimos o resultado pelo número de meta-modelos analisados. Vejamos a métrica a seguir:

$$\text{MDA} = \sum_{j=1}^{m} \text{NDA}_j \; / \; m$$

  Onde, *NDA* é o Número de Diretrizes Aplicadas em cada meta-modelo; *j* é cada um dos meta-modelos analisados; e *m* é o número de meta-modelos analisados.

## 5.2 Fase de Planejamento do Processo de Avaliação - GQM

Inicialmente, foi necessário obtermos a infra-estrutura utilizada na avaliação experimental: as diretrizes e os meta-modelos avaliados. As diretrizes já foram identificadas e descritas no capítulo anterior, portanto, restou apenas realizarmos uma pesquisa em busca de alguns meta-modelos candidatos para a avaliação. Ao final da pesquisa, foi selecionado um conjunto de seis meta-modelos: (i) três especificados pela OMG – são os meta-modelos de OCL [OCL, 2006], QVT e SPEM; e (ii) três especificados por organizações diferentes – são os meta-modelos de KobrA2, Ant [Ant, 2005] e Java. Estes seis meta-modelos foram escolhidos tendo em vista que eles definem a sintaxe abstrata de linguagens variadas, como por exemplo, linguagens de: processo de desenvolvimento de *software* (SPEM e KobrA2), programação (Java), transformações de modelos (QVT), especificação de restrições (OCL) e construção de *software* (Ant). Portanto, esta grande variedade reforça ainda mais a avaliação da aplicabilidade das diretrizes em qualquer meta-modelo.

Optamos por meta-modelos da OMG no intuito de fazermos um breve contraste com os meta-modelos especificados por outras organizações, uma vez que a OMG é um dos maiores consórcios do mundo em padronização, bem reconhecido e consolidado. Além disso, a OMG possui diversos meta-modelos não triviais construídos e validados por especialistas da área, sendo assim, mais consistentes e confiáveis. Adicionalmente, algumas das diretrizes que identificamos foram inspiradas por meta-modelos definidos pela própria OMG. Por conseguinte, já é esperado que os meta-modelos desta organização tenham uma quantidade menor de aplicações das diretrizes do que os demais.

### 5.2.1 Seleção do Contexto e dos Participantes

O contexto experimental é composto pelas condições em que o experimento é executado. Ele pode ser caracterizado conforme quatro dimensões:

- Processo: Neste experimento, o processo não foi um estudo desenvolvido na indústria, mas sim na academia. Contudo, ele pode ser utilizado nestes dois ambientes;

- Participantes: A equipe de projeto designada para participar deste experimento foi constituída por um único indivíduo. Durante todo o processo de avaliação não houve a necessidade da realização de entrevistas, da aplicação de questionários ou de qualquer outra técnica que vise extrair determinadas informações de pessoas externas ao projeto. Vale salientar que apesar de um mesmo indivíduo ter definido as diretrizes, aplicado-as nos meta-modelos e avaliado os resultados, o processo de avaliação deste trabalho não foi tendencioso, tendo em vista que: (i) as diretrizes foram identificadas com base em uma grande variedade de meta-modelos; (ii) uma vez identificadas, as diretrizes foram discutidas com um especialista da área a fim de constatar sua utilidade; (iii) as diretrizes foram aplicadas em meta-modelos diversificados, isto é, que definem diferentes linguagens; e (iv) os resultados foram analisados de forma individual para cada meta-modelo avaliado;

- A realidade (o problema pode ser real ou modelado): Em nosso caso o experimento foi real, pois os meta-modelos não foram simulados ou modelados. Ao contrário, foram considerados meta-modelos reais e já existentes, construídos por diferentes organizações;

- Generalidade (pode ser específico ou geral): o contexto possuiu caráter específico, pois foi focado na aplicação e avaliação das diretrizes exclusivamente em meta-modelos.

### 5.2.2 Variáveis: Independentes e Dependentes

Um dos elementos principais para a avaliação de um experimento são as variáveis. Existem dois tipos de variáveis: as independentes e as dependentes. As primeiras se referem à entrada do processo de avaliação e representam a causa que afeta o resultado deste processo. Já as segundas se referem à saída da avaliação, elas representam o efeito causado pelas variáveis independentes [Travassos et al., 2002].

No contexto do nosso trabalho, as variáveis independentes foram formadas pelo conjunto de meta-modelos utilizado para avaliação e pelas diretrizes que foram aplicadas nestes meta-modelos. Como resultado da aplicação das diretrizes nos meta-modelos, obtivemos a nossa variável dependente, isto é, os meta-modelos modificados.

## 5.3 Fase de Coleta de Dados do Processo de Avaliação - GQM

A presente subseção está estruturada da seguinte forma: cada meta-modelo participante da avaliação será apresentado, juntamente com um quantitativo dos seus elementos e, em seguida, será ilustrada a aplicação de todas as diretrizes que são aplicáveis nele. Nesta subseção, será apresentada a avaliação de três meta-modelos (KobrA2, Java e QVT), a avaliação dos outros três (Ant, OCL e SPEM) pode ser vista no Apêndice C deste documento. O resultado desta avaliação será interpretado e discutido na fase de Interpretação (subseção 5.4).

### 5.3.1 Meta-modelo de KobrA2

Segundo [Atkinson et al., 2002], KobrA2 é uma metodologia de desenvolvimento de sistemas que combina engenharia de linha de produto e desenvolvimento de *software* baseado em componentes, utilizando UML para especificar componentes. Na seqüência, a Tabela 4 apresenta alguns elementos que quantificamos neste meta-modelo, os quais foram alvo para a aplicação das diretrizes. A contagem destes elementos está agrupada de acordo com os três

pacotes que o meta-modelo de KobrA2 possui: `SUM`, `Views` e `Transformation`. Ao final da tabela, é informado o somatório de cada elemento em todos os pacotes.

**Tabela 4: Quantitativo de elementos do meta-modelo de KobrA2.**

| **Pacote / Elemento** | *Meta-classes* | *Atributos* | *Enumerations* | *Associações* |
|---|---|---|---|---|
| *SUM* | 104 | 7 | 0 | 40 |
| *Views* | 67 | 10 | 3 | 1 |
| *Transformation* | 87 | 0 | 0 | 43 |
| *Total* | 258 | 17 | 3 | 84 |

A seguir, a Tabela 5 relaciona cada diretriz com o número de vezes que ela pode ser aplicada no meta-modelo de KobrA2. Ao final desta tabela, é informado o número total de aplicações de todas as diretrizes neste meta-modelo, bem como o número de diretrizes que foram aplicadas.

**Tabela 5: Quantitativo de aplicações das diretrizes no meta-modelo de KobrA2.**

| **Diretriz** | **Núm. Aplicações** |
|---|---|
| D1 | 2 |
| D2 | 2 |
| D5 | 1 |
| D8 | 3 |
| D10 | 3 |
| D9 | 12 |
| D11 | 14 |
| D12 | 7 |
| *Número de Aplicações Total* | 44 |
| *Número de Diretrizes Aplicadas*: 8 | |

Como podemos observar, a diretriz D6 não foi aplicada, pois, apesar de não possuir um pacote `PrimitiveTypes`, o meta-modelo de KobrA2 já possui um pacote que contém todos os tipos primitivos do projeto, chamado `Type`. Da mesma forma ocorre com a diretriz D7, pois, apesar de não possuir um pacote chamado `Core`, o meta-modelo de KobrA2 já possui um pacote principal chamado `KobrA2`, o qual contém todos os principais subpacotes do projeto. As demais diretrizes – D3, D4 e D13 – não foram aplicadas devido ao fato de o meta-modelo de KobrA2 já se apresentar em conformidade com todas elas.

**D1 - *Abstracting Common Attributes* (KobrA2)**

Esta diretriz possui duas aplicações no meta-modelo de KobrA2, envolvendo três meta-classes do pacote `KobrA2::SUM::Constraint::Common`. Como podemos observar na Figura 47, as três meta-classes, chamadas *`Constraint`*, `ExpressionInOcl` e *`OclExpression`*, têm características em comum: os atributos `boolean` e `query`, os quais possuem a mesma finalidade em todas elas.

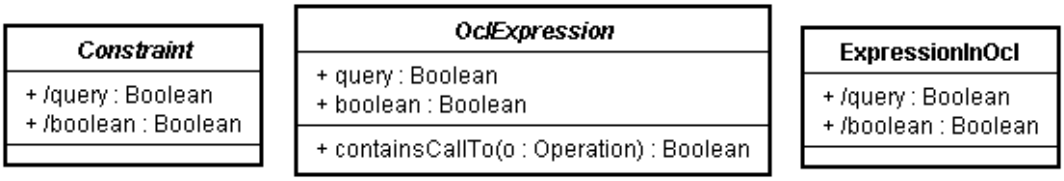



**Figura 47: Meta-classes do meta-modelo de KobrA2 com atributos em comum.**

Para aplicarmos a diretriz D1 criamos duas meta-classes abstratas: *`BooleanableElement`* e *`QueryableElement`*. A primeira possui o atributo chamado `boolean`, enquanto a segunda possui o atributo `query`, ambos com a visibilidade *public*. A diretriz sugere que estas meta-classes abstratas sejam definidas em um pacote chamado `Abstractions` do meta-modelo. Por fim, as meta-classes *`Constraint`*, `ExpressionInOcl` e *`OclExpression`* estendem tanto de *`BooleanableElement`* quanto de *`QueryableElement`* para ter acesso aos atributos. Vejamos o resultado da aplicação na Figura 48.

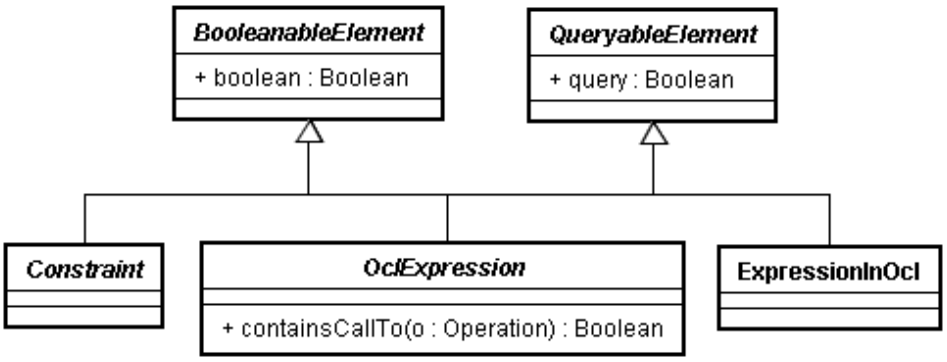



**Figura 48: Meta-modelo de KobrA2 com a aplicação da diretriz D1.**

**D2 - *Abstracting Common Associations* (KobrA2)**

A diretriz D2 tem duas aplicações no meta-modelo de KobrA2. A Figura 49 representa um trecho deste meta-modelo em que três meta-classes diferentes – `Derived`, `Init` e

`PropDef` – possuem uma associação com a meta-classe `Property`, todas com a mesma multiplicidade e nenhum nome de papel (*role*).

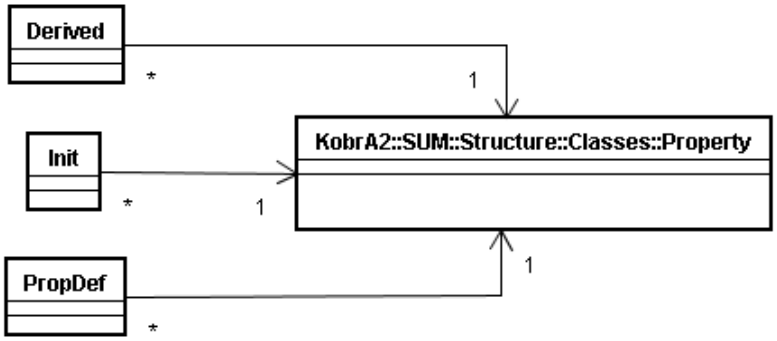



**Figura 49: Trecho do meta-modelo de KobrA2 com três associações em comum.**

A aplicação da diretriz envolveu três associações e pode ser vista na Figura 50. Como podemos observar, foi necessário criarmos uma meta-classe abstrata (no pacote `Abstractions`), a qual chamamos de *`ConstrainedElement`*, para generalizar as meta-classes que especificam a associação com a meta-classe `Property`. Logo, as meta-classes `Derived`, `Init` e `PropDef` estendem a meta-classe criada, assim, herdando a associação.

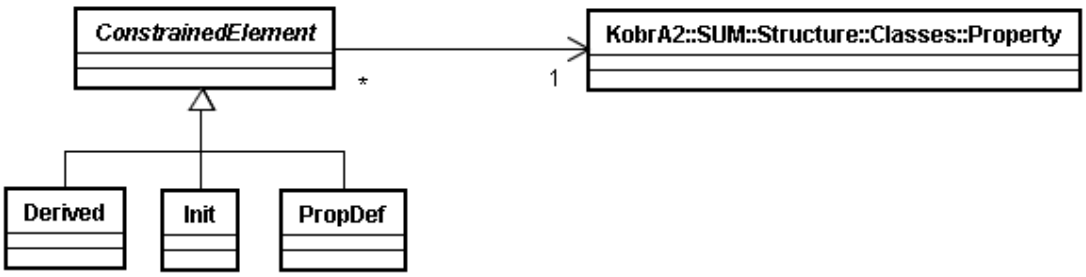



**Figura 50: Meta-modelo de KobrA2 com a aplicação da diretriz D2.**

Outra aplicação da diretriz D2 no meta-modelo de KobrA2 pode ser conferida na subseção D.2.1 do Apêndice D.

### D5 - *Adding Abstractions Package* (KobrA2)

O meta-modelo de KobrA2 não possui nenhum pacote que contenha todas as meta-classes reutilizáveis do projeto. Portanto, ao aplicarmos esta diretriz definimos um pacote chamado `Abstractions`, como ilustrado na Figura 51.

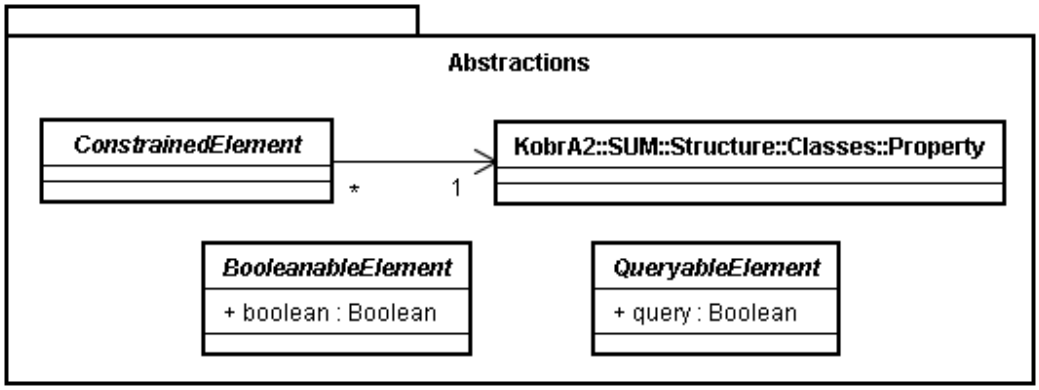



**Figura 51: Pacote Abstractions para o meta-modelo de KobrA2.**

O pacote `Abstractions` contém as meta-classes que podem ser reutilizadas em todo o meta-modelo, como por exemplo, *`BooleanableElement`*, *`QueryableElement`* e *`ConstrainedElement`*, as quais foram oriundas da aplicação das diretrizes D1 e D2. A Figura 51 mostra apenas algumas das meta-classes que podem pertencer ao pacote `Abstractions`, o qual crescerá ao passo que novas meta-classes reutilizáveis são adicionadas ao meta-modelo.

**D8 - *Adding Utility Operations* (KobrA2)**

A diretriz D8 possui três aplicações no meta-modelo de KobrA2. A aplicação que será ilustrada envolve 22 restrições com expressões repetitivas. Como exemplo, vejamos apenas três destas restrições em que as expressões destacadas em negrito se repetem:

[1] context StructuralElement
inv: let kindAllowed = Set{TypeElement, InstanceElement}
in (**kindAllowed->exists(e: Element | self.oclIsKindOf(e))**)

[2] context InstanceElement
inv: let kindAllowed = Set{ComponentObject, Slot, Link}
in (**kindAllowed->exists(e: Element | self.oclIsKindOf(e))**)

[3] context StructuralConstraintElement
inv: let kindAllowed = Set{Inv, Derived, Init, PropDef}
in (**kindAllowed->exists(e: Element | self.oclIsKindOf(e))**)

Esta expressão se repete 22 vezes em todo o meta-modelo de KobrA2. Portanto, ao aplicarmos a diretriz D8 criamos uma operação adicional, chamada `existsElement(set: Set(OclAny))`, para realizar a mesma função que a expressão. A seguir, vejamos como foi definida a operação:

```
context Element
def: existsElement(set: Set(OclAny)) : Boolean =
set->exists(e: Element | self.oclIsKindOf(e))
```

Como todas as meta-classes que utilizam a expressão em destaque são subclasses de `Element`, definimos a operação dentro do seu contexto. Uma vez que a operação adicional foi definida, modificamos as restrições para que elas a invoque, como ilustrado a seguir:

```
[1] context StructuralElement
inv: let kindAllowed = Set{TypeElement, InstanceElement}
in self.existsElement(kindAllowed)
```

Outra aplicação da diretriz D8 no meta-modelo de KobrA2 pode ser conferida na subseção D.3.1 do Apêndice D.

**D9 - *Defining Boolean Attributes Default Value* (KobrA2)**

Após uma análise do meta-modelo de KobrA2 foi detectado que não há nenhum valor *default* definido para os 12 atributos booleanos que existem, portanto, a diretriz D9 tem um total de 12 aplicações neste meta-modelo. A Figura 52 mostra quais são alguns destes atributos booleanos para os quais devemos definir um valor *default*.

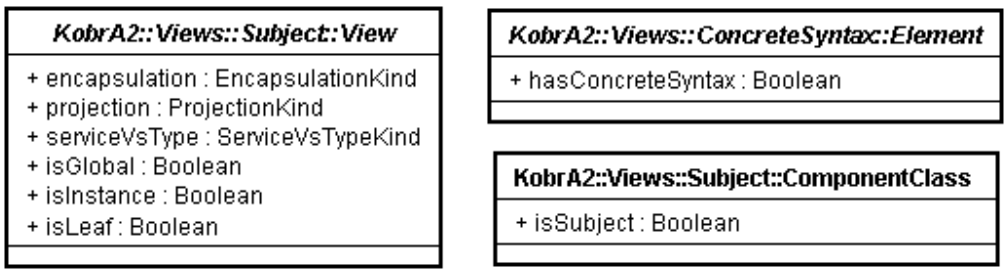



**Figura 52: Atributos do meta-modelo de KobrA2 sem valores *default*.**

Outras aplicações da diretriz D9 no meta-modelo de KobrA2 podem ser conferidas na subseção D.4.1 do Apêndice D.

**D10 - *Defining Enum Default Value* (KobrA2)**

Nenhum dos três *enumerations* existentes no meta-modelo de KobrA2 especifica valor *default*. Logo, a diretriz D10 possui três aplicações neste meta-modelo. A Figura 53 ilustra quais são estes *enumerations* para os quais devemos definir valores *default*.

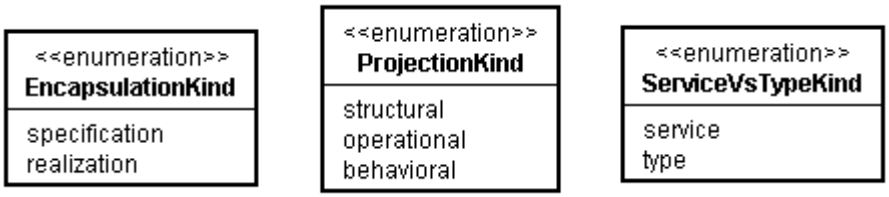



**Figura 53: *Enumerations* do meta-modelo de KobrA2 sem valores *default*.**

**D11 - *Defining Association Member Ends Features* (KobrA2)**

Podemos aplicar esta diretriz em 14 associações de composição do meta-modelo de KobrA2. Contudo, vale observar que antes de aplicarmos a diretriz D11 nestas associações, devemos analisar a semântica em cada caso para, de fato, sabermos se a diretriz é ou não aplicável. Temos que verificar se a associação realmente exige que a parte não pertença a mais de um todo ao mesmo tempo. Como exemplo, vejamos a sua aplicação na associação de composição entre as meta-classes `ProtocolStateMachine` e `Region`, ilustrada na Figura 54. Ambas as meta-classes pertencentes ao pacote `KobrA2::SUM::Behavior::ProtocolStateMachines`.

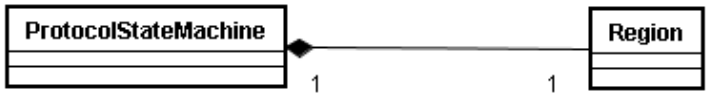



**Figura 54: Trecho do meta-modelo de KobrA2 com uma associação de composição.**

Como podemos observar, algumas características desta associação de composição já estão em conformidade com a diretriz D11: a parte (`Region`) é uma meta-classe concreta e navegável a partir da meta-classe que representa o todo (`ProtocolStateMachine`). Por outro lado, ela possui o limite mínimo de multiplicidade igual a `1`. Além disso, a multiplicidade do todo desta associação é `1`. Embora um `Region` pertença apenas a uma meta-classe (a `ProtocolStateMachine`) em todo o meta-modelo, a aplicação desta diretriz é válida para garantirmos uma melhor evolução do mesmo. Portanto, para aplicá-la apenas definimos a multiplicidade do todo desta associação para `0..1`. Apesar da meta-

classe `Region` possuir o limite mínimo de multiplicidade igual a `1`, ele não foi alterado devido à semântica do meta-modelo de KobrA2, pois um `ProtocolStateMachine` deve ter pelo menos um `Region`. Vejamos o resultado da aplicação na Figura 55.

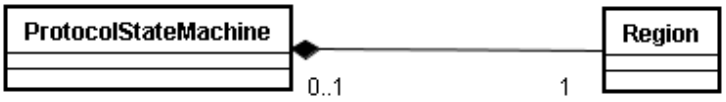



**Figura 55: Meta-modelo de KobrA2 com a aplicação da diretriz D11.**

Outras aplicações da diretriz D11 no meta-modelo de KobrA2 podem ser conferidas na subseção D.5.1 do Apêndice D.

**D12 - *Redefining Boolean Attribute Names* (KobrA2)**

Esta diretriz pode ser aplicada em sete atributos booleanos do meta-modelo de KobrA2. A seguir, a Figura 56 apresenta alguns deles: os atributos chamados `query` e `boolean` presentes nas meta-classes *`Constraint`* e `ExpressionInOcl` do pacote `KobrA2::SUM::Constraint::Common`, bem como o atributo chamado `atPre` pertencente à meta-classe *`FeatureCallExp`* do pacote `KobrA2::SUM::Constraint::OclExpressions`. Após a aplicação da diretriz D12, os nomes dos atributos passam a ser `isBoolean`, `isQuery` e `isPre`.

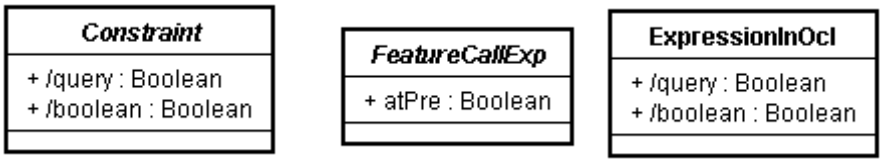



**Figura 56: Meta-classes do meta-modelo de KobrA2 com atributos booleanos.**

Outras aplicações da diretriz D12 no meta-modelo de KobrA2 podem ser conferidas na subseção D.6.1 do Apêndice D.

## 5.3.2 Meta-modelo de Java

Nesta subseção, é avaliada a aplicabilidade das diretrizes no meta-modelo de Java, uma linguagem de programação orientada a objetos. Este meta-modelo foi retirado de um repositório de meta-modelos disponibilizado por um grupo de pesquisas chamado AtlanMod

[AtlanMod, 1990]. A seguir, a Tabela 6 apresenta alguns elementos que quantificamos no meta-modelo de Java, os quais foram alvo para a aplicação das diretrizes.

**Tabela 6: Quantitativo de elementos do meta-modelo de Java.**

| Elemento | Quantidade |
|---|---|
| Meta-classes | 95 |
| Atributos | 46 |
| *Enumerations* | 4 |
| Associações | 150 |

A seguir, a Tabela 7 relaciona cada diretriz com o número de vezes que ela pode ser aplicada no meta-modelo de Java. Ao final desta tabela, é informado o número total de aplicações de todas as diretrizes neste meta-modelo, bem como o número de diretrizes que foram aplicadas.

**Tabela 7: Quantitativo de aplicações das diretrizes no meta-modelo de Java.**

| Diretriz | Núm. Aplicações |
|---|---|
| D1 | 1 |
| D2 | 18 |
| D5 | 1 |
| D9 | 29 |
| D11 | 103 |
| D12 | 29 |
| *Número de Aplicações Total* | 181 |
| *Número de Diretrizes Aplicadas*: 6 | |

Como podemos observar, a diretriz D6 não foi aplicada, pois o meta-modelo de Java já possui um pacote chamado `PrimitiveTypes`, o qual contém todos os tipos primitivos do projeto. Da mesma forma, a diretriz D7 não foi aplicada, uma vez que Java já possui um pacote principal chamado `JavaAbstractSyntax`. As demais diretrizes – D3, D4, D8, D10 e D13 – não foram aplicadas devido ao fato de o meta-modelo de Java já se apresentar em conformidade com todas elas.

### D1 - *Abstracting Common Attributes* (Java)

A diretriz D1 tem apenas uma aplicação no meta-modelo de Java. Como ilustra a Figura 57, as meta-classes chamadas `MethodRefParameter`, `MethodDeclaration` e `SingleVariableDeclaration` possuem uma característica em comum: o atributo booleano `varargs`.

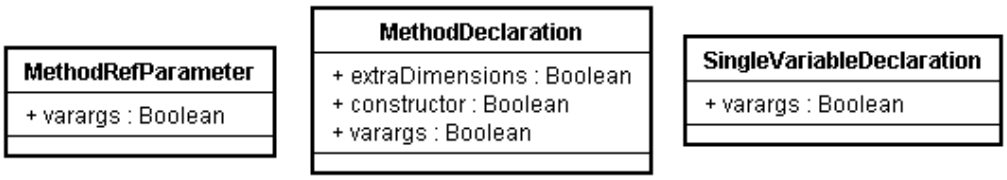



**Figura 57: Meta-classes do meta-modelo de Java com um atributo em comum.**

Para aplicarmos a diretriz D1, criamos uma meta-classe abstrata chamada `VarargsElement` com o atributo `varargs`. A diretriz sugere que esta meta-classe abstrata seja definida em um pacote chamado `Abstractions` do meta-modelo. Por fim, as três meta-classes que possuem o atributo `varargs` em comum estendem de `VarargsElement` para ter acesso ao atributo, como podemos ver na Figura 58.

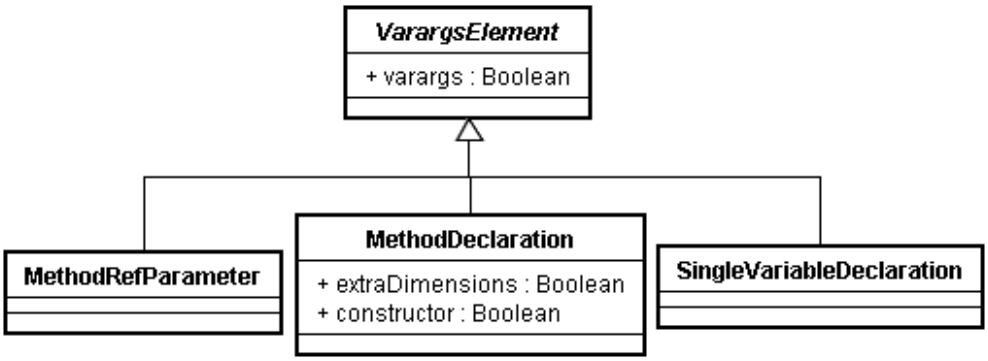



**Figura 58: Meta-modelo de Java com a aplicação da diretriz D1.**

### D2 - *Abstracting Common Associations* (Java)

Esta diretriz tem 18 aplicações no meta-modelo de Java. Como exemplo, a Figura 59 apresenta um trecho do meta-modelo em que as meta-classes `ThisExpression`, `MethodRef`, `SuperFieldAccess`, `MemberRef`, `SuperMethodInvocation` e `QualifiedName` possuem uma associação de composição com a meta-classe `Name`, todas com a mesma multiplicidade e o mesmo papel.

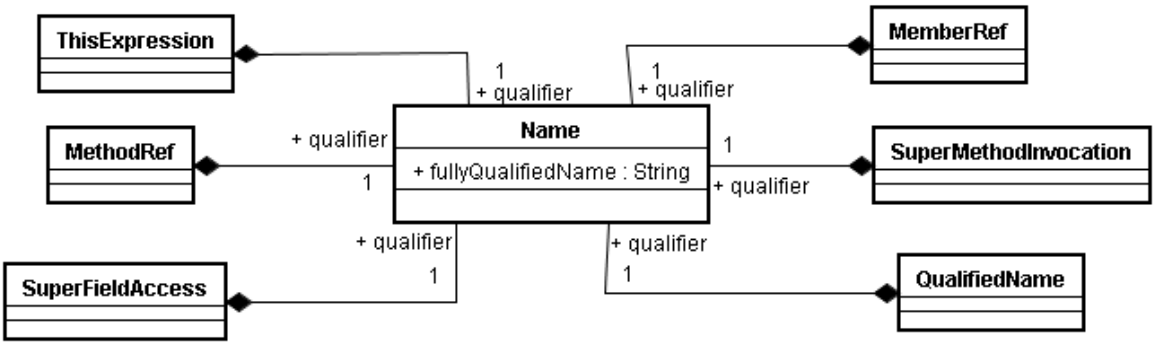



**Figura 59: Trecho do meta-modelo de Java com associações em comum.**

Podemos verificar a aplicação da diretriz na Figura 60. Foi necessário criarmos uma meta-classe abstrata (no pacote `Abstractions`), chamada *`OwnerQualifier`*, para generalizar as meta-classes que especificam a associação de composição com a meta-classe `Name`. O nome desta meta-classe é apenas uma convenção que adotamos, o desenvolvedor pode defini-lo de acordo com a semântica do meta-modelo. Logo, as seis meta-classes estendem a meta-classe criada, assim, herdando a associação.

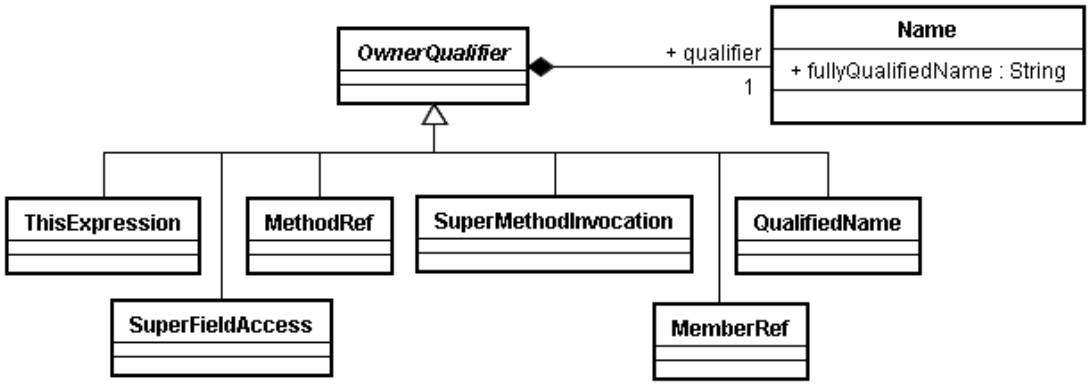



**Figura 60: Meta-modelo de Java com a aplicação da diretriz D2.**

Outra aplicação da diretriz D2 no meta-modelo de Java pode ser conferida na subseção D.2.3 do Apêndice D.

### D5 - *Adding Abstractions Package* (Java)

O meta-modelo de Java não possui nenhum pacote que contenha todas as meta-classes reutilizáveis do projeto. Portanto, ao aplicarmos esta diretriz temos um pacote chamado `Abstractions`, como ilustrado na Figura 61.

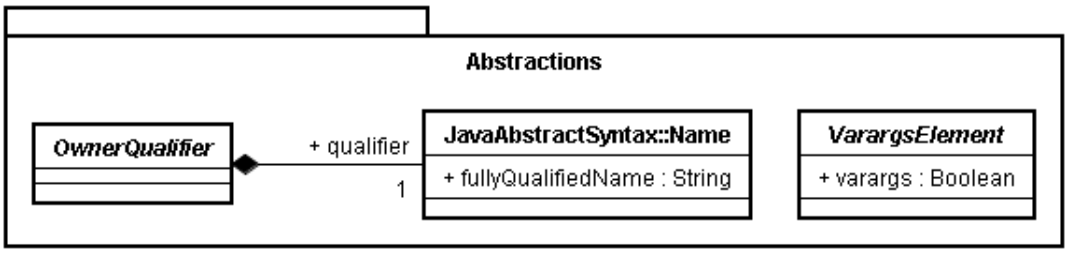



**Figura 61: Pacote Abstractions para o meta-modelo de Java.**

`Abstractions` contém as meta-classes que podem ser reutilizadas em todo o meta-modelo de Java, como por exemplo, *`VarargsElement`* e *`OwnerQualifier`*, as quais foram oriundas da aplicação das diretrizes D1 e D2, respectivamente.

**D9 - *Defining Boolean Attributes Default Value* (Java)**

Nenhum dos atributos booleanos do meta-modelo de Java possui valor *default*. Portanto, vejamos na Figura 62 alguns exemplos de atributos que devemos definir valores *default* ao aplicarmos esta diretriz.

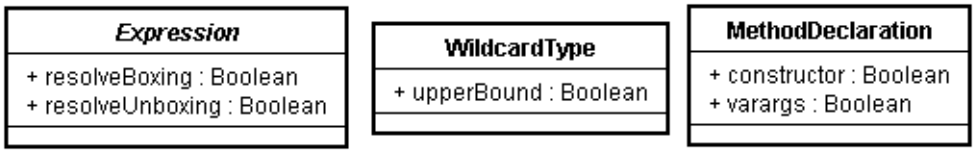



**Figura 62: Atributos do meta-modelo de Java sem valores *default*.**

Outras aplicações da diretriz D9 no meta-modelo de Java podem ser conferidas na subseção D.4.2 do Apêndice D.

**D11 - *Defining Association Member Ends Features* (Java)**

Esta diretriz pode ser aplicada em 103 associações de composição do meta-modelo de Java. Contudo, vale observar que antes de aplicarmos esta diretriz nestas associações, devemos analisar a semântica em cada caso para, de fato, sabermos se a diretriz é ou não aplicável. Temos que verificar se a associação realmente exige que a parte não pertença a mais de um todo ao mesmo tempo. Como exemplo, vejamos a sua aplicação nas associações de composição entre as meta-classes `SynchronizedStatement-Block` e `TryStatement-Block`, ilustradas na Figura 63.

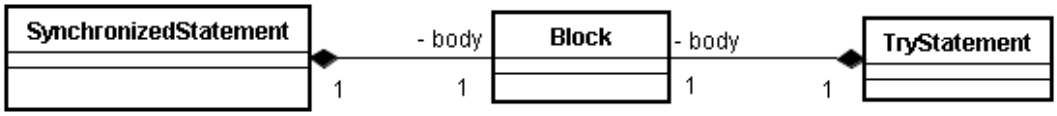



**Figura 63: Trecho do meta-modelo de Java com duas associações de composição.**

Como podemos observar, o limite mínimo de multiplicidade da meta-classe `Block` nas duas associações não é `0`, como sugere a diretriz. Além disso, as multiplicidades das meta-classes que representam o todo destas associações estão definidas como `1`. Isto significa dizer que um mesmo `Block` pertence tanto a um `SynchronizedStatement` quanto a um `TryStatement` ao mesmo tempo. Para deixarmos as associações em conformidade com a diretriz D11: (i) definimos a multiplicidade de `SynchronizedStatement` e `TryStatement` como `0..1`; e (ii) definimos o limite mínimo de multiplicidade igual a `0` para as duas associações de `Block`. Embora não seja muito comum, vale ressaltar que é sintaticamente correto termos um *statement* `Syncrhonized` ou `Try` com o corpo vazio. Podemos conferir o resultado na Figura 64.

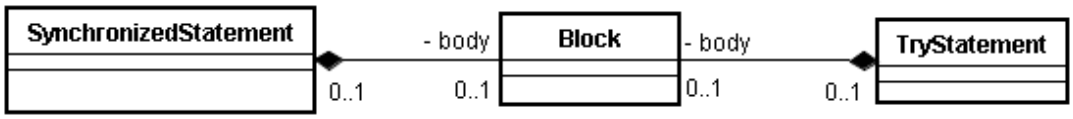



**Figura 64: Meta-modelo de Java com a aplicação da diretriz D11.**

Outras aplicações da diretriz D11 no meta-modelo de Java podem ser conferidas na subseção D.5.3 do Apêndice D.

### D12 - *Redefining Boolean Attribute Names* (Java)

Podemos aplicar a diretriz D12 em todos os atributos booleanos presentes no meta-modelo de Java. Vejamos na Figura 65 alguns destes atributos pertencentes às meta-classes `Modifier` e `ImportDeclaration`. Após a aplicação desta diretriz, os nomes dos atributos passam a ser (i) `isAbstract`, `isFinal`, `isNative`, `isNone`, `isPrivate`, `isProtected`, `isPublic`, `isStatic`, `isStrictfp`, `isSynchronized`, `isTransient` e `isVolatile` para a meta-classe `Modifier`; e (ii) `isOnDemand` e `isStatic` para a meta-classe `ImportDeclaration`.

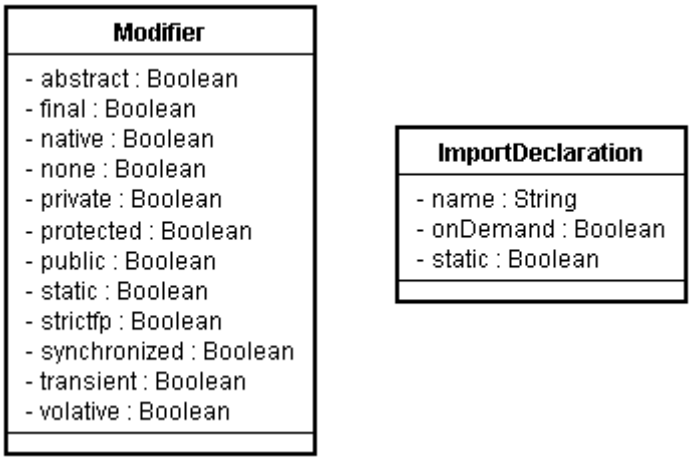



**Figura 65: Meta-classes do meta-modelo de Java com atributos booleanos.**

Outras aplicações da diretriz D12 no meta-modelo de Java podem ser conferidas na subseção D.6.2 do Apêndice D.

### 5.3.3 Meta-modelo de QVT

Nesta subseção, é avaliada a aplicabilidade das diretrizes no meta-modelo de QVT. Definida pela OMG, QVT é uma poderosa linguagem para transformação de modelos, podendo ser declarativa, imperativa ou híbrida. Na seqüência, a Tabela 8 apresenta alguns elementos que quantificamos neste meta-modelo, os quais foram alvo para a aplicação das diretrizes. A contagem destes elementos está agrupada de acordo com os seis pacotes que o meta-modelo de QVT possui: `QVTBase`, `QVTTemplate`, `QVTRelational`, `QVTOperational`, `ImperativeOCL` e `QVTCore`.

**Tabela 8: Quantitativo de elementos do meta-modelo de QVT.**

| **Pacote / Elemento** | *Meta-classes* | *Atributos* | *Enumerations* | *Associações* |
|---|---|---|---|---|
| *QVTBase* | 8 | 2 | 0 | 13 |
| *QVTTemplate* | 4 | 0 | 0 | 9 |
| *QVTRelational* | 8 | 1 | 0 | 17 |
| *QVTOperational* | 22 | 11 | 2 | 40 |
| *ImperativeOCL* | 34 | 4 | 1 | 44 |
| *QVTCore* | 11 | 2 | 1 | 12 |
| *Total* | 87 | 20 | 4 | 135 |

A seguir, a Tabela 9 relaciona cada diretriz com o número de vezes que ela pode ser aplicada no meta-modelo de QVT. Ao final desta tabela, é informado o número total de

aplicações de todas as diretrizes neste meta-modelo, bem como o número de diretrizes que foram aplicadas.

**Tabela 9: Quantitativo de aplicações das diretrizes no meta-modelo QVT.**

| Diretriz | Núm. Aplicações |
|---|---|
| D2 | 2 |
| D5 | 1 |
| D9 | 12 |
| D10 | 3 |
| D11 | 17 |
| D12 | 2 |
| D13 | 1 |
| *Número de Aplicações Total* | 38 |
| *Número de Diretrizes Aplicadas*: 7 | |

Como podemos observar, não foi necessário aplicar a diretriz D6, pois o meta-modelo de QVT reusa partes de EMOF/CMOF [MOF, 2006] que, por sua vez, importa o pacote `PrimitiveTypes` do meta-modelo da UML contendo todos os tipos primitivos. Da mesma forma ocorre com a diretriz D7, pois o meta-modelo de QVT já possui um pacote principal chamado `QVTCore`. As demais diretrizes – D1, D3, D4 e D8 – não foram aplicadas devido ao fato de o meta-modelo de QVT já se apresentar em conformidade com todas elas.

**D2 -** ***Abstracting Common Associations*** **(QVT)**

Esta diretriz possui duas aplicações no meta-modelo de QVT. Como exemplo, a Figura 66 apresenta um trecho deste meta-modelo em que as meta-classes `OrderedTupleLiteralPart` e `AnonymousTupleLiteralPart` possuem uma associação de composição com a meta-classe *`OclExpression`*, a qual foi importada do pacote `EssentialOCL` pertencente ao meta-modelo de OCL. Como podemos notar, as duas associações possuem a mesma multiplicidade e os mesmos papéis.

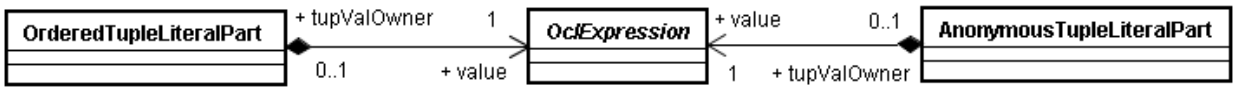



**Figura 66: Trecho do meta-modelo de QVT com duas associações em comum.**

Ao aplicarmos a diretriz D2, temos o trecho ilustrado na Figura 67. Uma meta-classe abstrata, chamada *`OwnerValue`*, foi criada (no pacote `Abstractions`) para generalizar as

meta-classes que especificam a associação de composição com a meta-classe *OclExpression*. O nome da meta-classe abstrata criada é apenas uma convenção que adotamos, o desenvolvedor pode defini-lo de acordo com a semântica do meta-modelo. Logo, as meta-classes `OrderedTupleLiteralPart` e `AnonymousTupleLiteralPart` estendem a meta-classe criada, assim, herdando a associação.

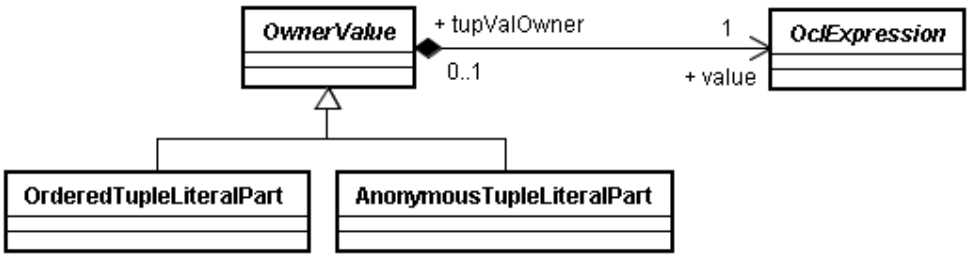



**Figura 67: Meta-modelo de QVT com a aplicação da diretriz D2.**

Outra aplicação da diretriz D2 no meta-modelo de QVT pode ser conferida na subseção D.2.4 do Apêndice D.

**D5 - *Adding Abstractions Package* (QVT)**

O meta-modelo de QVT não possui nenhum pacote que contenha todas as meta-classes reutilizáveis do projeto. Portanto, ao aplicarmos esta diretriz um pacote chamado `Abstractions` será criado, como ilustrado na Figura 68.

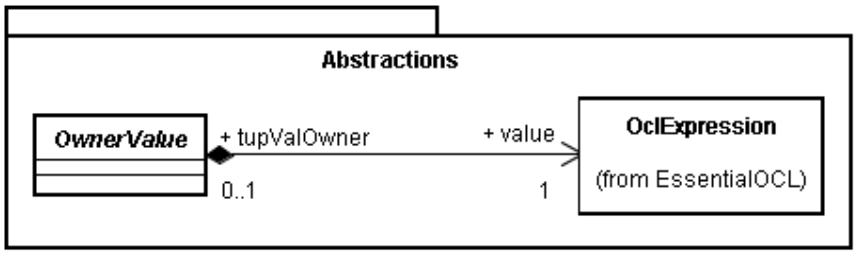



**Figura 68: Pacote Abstractions para o meta-modelo de QVT.**

Como podemos ver na Figura 68, inicialmente, o pacote `Abstractions` possui apenas a meta-classe *OwnerValue*. Contudo, este pacote vai crescendo ao passo que as diretrizes D1 e D2 são aplicadas, criando meta-classes reutilizáveis para todo o projeto.

**D9 - *Defining Boolean Attributes Default Value* (QVT)**

De todos os atributos booleanos existentes no meta-modelo de QVT, 12 não possuem valores *default* especificados. A seguir, vejamos na Figura 69 exemplos de alguns atributos que devem ter valores *default* definidos ao aplicar esta diretriz.

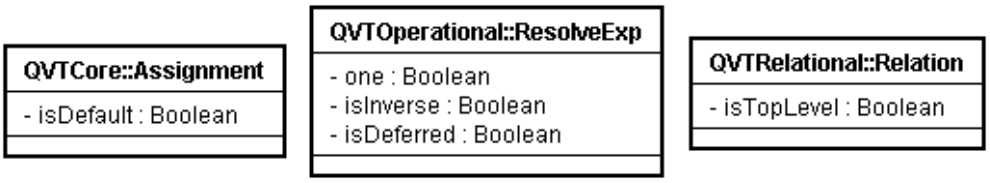



**Figura 69: Atributos do meta-modelo de QVT sem valores *default*.**

Outras aplicações da diretriz D9 no meta-modelo de QVT podem ser conferidas na subseção D.4.3 do Apêndice D.

**D10 - *Defining Enum Default Value* (QVT)**

O meta-modelo de QVT possui três *enumerations* que não especificam valor *default* para seus atributos. A Figura 70 ilustra quais são estes *enumerations* para os quais, ao aplicarmos a diretriz D10, devemos definir valores *default*.

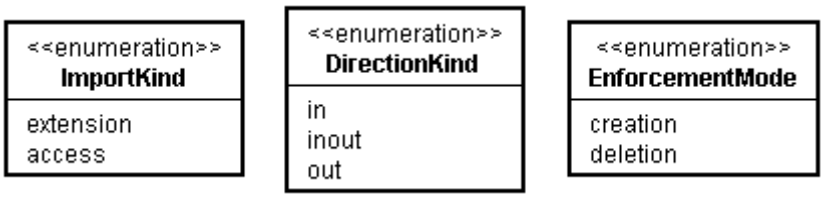



**Figura 70: *Enumerations* do meta-modelo de QVT sem valores *default*.**

**D11 - *Defining Association Member Ends Features* (QVT)**

Esta diretriz pode ser aplicada em 17 associações de composição do meta-modelo de QVT. Contudo, vale observar que antes de aplicarmos esta diretriz nestas associações, devemos analisar a semântica em cada caso para, de fato, sabermos se a diretriz é ou não aplicável. Temos que verificar se a associação realmente exige que a parte não pertença a mais de um todo ao mesmo tempo. Como exemplo, vejamos a sua aplicação na associação entre as meta-classes `Transformation` e *`Rule`*, ilustrada na Figura 71. Ambas as meta-classes pertencem ao pacote `QVTBase`.

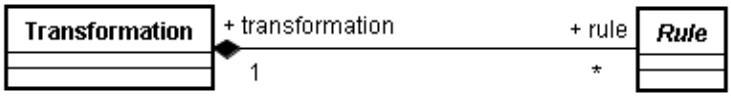



**Figura 71: Trecho do meta-modelo de QVT com uma associação de composição.**

Como podemos observar, a parte desta associação não é uma meta-classe concreta, como sugere a diretriz. Além do mais, a multiplicidade do todo desta associação é `1`. Embora um *`Rule`* pertença apenas a uma meta-classe (a `Transformation`) em todo o meta-modelo, a aplicação desta diretriz é válida para garantirmos uma melhor evolução do mesmo. Para aplicá-la, definimos a multiplicidade do todo como `0..1` e definimos a meta-classe parte da associação como sendo concreta. Podemos conferir o resultado na Figura 72.

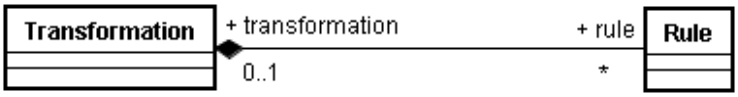



**Figura 72: Meta-modelo de QVT com a aplicação da diretriz D11.**

Outra aplicação da diretriz D11 no meta-modelo de QVT pode ser conferida na subseção D.5.4 do Apêndice D.

### D12 - *Redefining Boolean Attribute Names* (QVT)

De todos os atributos booleanos existentes no meta-modelo de QVT, dois não estão em conformidade com a diretriz D12. Como mostra a Figura 73, os atributos são: `withResult` e `one`, presentes nas meta-classes `ImperativeOCL::VariableInitExp` e `QVTOperational::ResolveExp`, respectivamente. Após a aplicação da diretriz D12, os nomes dos atributos passam a ser `hasResult` e `isOne`.

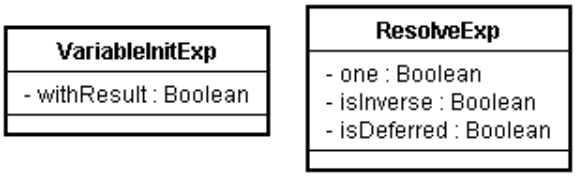



**Figura 73: Meta-classes do meta-modelo de QVT com atributos booleanos.**

### D13 - *Redefining Enum Names* (QVT)

A diretriz D13 precisa ser aplicada em apenas um *enumeration* do meta-modelo de QVT, o `EnforcementMode`, como ilustrado na Figura 74. Ao aplicarmos a diretriz D13, o nome do *enumeration* `EnforcementMode` passa a ser `EnforcementModeKind`.

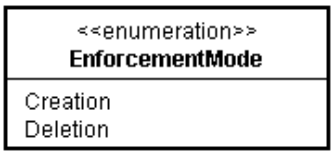



**Figura 74: *Enumeration* do meta-modelo de QVT.**

## 5.4 Fase de Interpretação do Processo de Avaliação - GQM

Os dados coletados anteriormente foram absorvidos e conclusões acerca dos mesmos foram definidas e analisadas. Cada uma das questões foi analisada considerando todas as diretrizes, conforme apresentado a seguir.

Vale salientar que a nossa avaliação experimental foi bem abrangente, envolvendo um conjunto de meta-modelos que já são bem consolidados e reconhecidos. Além do mais, é importante destacar que para a nossa avaliação foram consideradas as versões finais destes meta-modelos, ou seja, os meta-modelos avaliados já estão no último nível de otimização. E, mesmo assim, muitas diretrizes puderam ser aplicadas. Na realidade, o ideal seria que a avaliação fosse pautada em versões preliminares dos meta-modelos, pois em tais versões a aplicabilidade das diretrizes provavelmente teria um impacto maior. A nossa avaliação não considerou tais versões dos meta-modelos devido à grande dificuldade de encontrá-las, pois as pessoas, geralmente, não guardam as versões anteriores. Além do mais, As organizações das quais obtivemos os meta-modelos avaliados neste trabalho não disponibilizavam versões preliminares dos meta-modelos, apenas as finais.

### 5.4.1 Questão 1

Para responder a Questão 1 (Quão aplicáveis são as diretrizes nos meta-modelos?) foi utilizada uma análise quantitativa, envolvendo duas métricas: $MAD_i$ e MDA, as quais foram descritas na subseção 5.1.3. Cada uma delas será discutida a seguir.

#### Métrica 1: $MAD_i$ (Média de Aplicações de uma Diretriz)

Primeiramente, para responder a Questão 1 avaliamos a métrica $MAD_i$. A Tabela 10, ilustrada a seguir, apresenta o número de aplicações de cada diretriz em cada meta-modelo avaliado. A Tabela 10 também apresenta o total de aplicações para cada diretriz e, em seguida, a métrica

$MAD_i$ é calculada para cada uma delas. Dessa forma, podemos avaliar melhor a aplicabilidade de cada diretriz nos meta-modelos.

**Tabela 10: Análise da métrica $MAD_i$.**

| **Diretriz / Meta-modelo** | *KobrA2* | *Ant* | *Java* | *OCL* | *QVT* | *SPEM* | *Total* | $MAD_i$ |
|---|---|---|---|---|---|---|---|---|
| D1 | 2 | 5 | 1 | 0 | 0 | 0 | 8 | 1.3 |
| D2 | 2 | 4 | 18 | 0 | 2 | 2 | 28 | 4.6 |
| D3 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| D4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| D5 | 1 | 1 | 1 | 0 | 1 | 1 | 5 | 0.8 |
| D6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| D7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| D8 | 3 | 0 | 0 | 0 | 0 | 0 | 3 | 0.5 |
| D9 | 12 | 0 | 29 | 1 | 12 | 2 | 56 | 9.3 |
| D10 | 3 | 0 | 0 | 1 | 3 | 2 | 9 | 1.5 |
| D11 | 14 | 27 | 103 | 1 | 17 | 14 | 176 | 29.3 |
| D12 | 7 | 0 | 29 | 0 | 2 | 1 | 39 | 6.5 |
| D13 | 0 | 0 | 0 | 0 | 1 | 1 | 2 | 0.3 |

De acordo com a Tabela 10, das 13 diretrizes apresentadas apenas quatro (D3, D4, D6 e D7) não tiveram aplicação nenhuma nos meta-modelos, uma vez que eles já se encontravam em conformidade com tais diretrizes. O motivo para isto é o fato destas diretrizes serem mais direcionadas aos iniciantes em meta-modelagem, pois muitas vezes eles não conhecem as melhores práticas para especificar atributos (D3), relacionamentos de herança entre meta-classes (D4) e a estrutura de pacotes (D6 e D7), assim, acabam construindo meta-modelos reduntantes e mal estruturados. Como os meta-modelos que analisamos foram construídos por especialistas em meta-modelagem, estas quatro diretrizes já foram intuitivamente incorporadas neles.

Os meta-modelos avaliados já foram construídos seguindo algumas tendências, como: elevar para a superclasse todos os atributos em comum existentes em suas subclasses (D3), criar relacionamentos de herança adequados de forma a evitar o uso de restrições OCL desnecessárias (D4), e definir pacotes visando uma melhor organização do meta-modelo (D6 e D7). Além disso, tais meta-modelos já estão em suas versões finais e já passaram por vários refinamentos. Contudo, o conjunto de diretrizes que propomos não é direcionado apenas para a aplicação em meta-modelos já existentes, mas, principalmente, para auxiliar os desenvolvedores na construção de novos meta-modelos. Portanto, o fato de algumas diretrizes

não terem sido aplicadas não as torna, de maneira alguma, menos importantes ou sem utilidade.

Por outro lado, a Tabela 10 mostra que algumas diretrizes foram aplicadas muitas vezes. A diretriz D11 foi a que teve o maior número de aplicações (176) em comparação às demais, pois os meta-modelos apresentavam erros semânticos relacionados à multiplicidade das associações de composição ou poderiam ser melhorados de forma a facilitar sua evolução. Um dos motivos para tais erros é o fato de que muitas pessoas geralmente confundem modelagem com meta-modelagem. Em modelos, a associação de composição é um relacionamento forte em que o todo não deve existir sem a(s) sua(s) parte(s), então a sua multiplicidade é especificada como `1`. Já em meta-modelos, é possível especificarmos associações de composição em que o todo ainda tenha razão de existir mesmo se sua(s) parte(s) não existir(em) mais, então a sua multiplicidade é especificada como `0..1`.

Já as diretrizes D9 e D12, por exemplo, tiveram um total de 56 e 39 aplicações, respectivamente, pois os meta-modelos não definiam valores *default* para os seus atributos booleanos nem padronizavam os nomes de tais atributos. Há outras diretrizes que, apesar de uma quantidade não muito elevada de aplicações, foram aplicadas em quase todos os meta-modelos. Por exemplo, as diretrizes D2 e D10, pois a maioria dos meta-modelos tinha atributos com a mesma característica e semântica definidos em meta-classes diferentes, bem como possuíam *enumerations* sem nenhum valor *default* especificado.

Já a diretriz D5, como podemos ver na Tabela 10, foi aplicada somente uma vez em alguns meta-modelos pelo fato de ser uma diretriz voltada para a organização em pacotes, isto é, o pacote `Abstractions` só deve ser criado uma única vez no meta-modelo.

Do conjunto de meta-modelos avaliados, três foram especificados pela OMG (OCL, QVT e SPEM) e os demais (KobrA2, Ant e Java) foram definidos por organizações diferentes. Nesse sentido, a Tabela 11 mostra um comparativo da quantidade de aplicações das diretrizes em meta-modelos que são padrões da OMG e dos que não são. Como já esperávamos, todas as diretrizes, exceto D10 e D13, tiveram uma quantidade menor de aplicações nos meta-modelos definidos pela OMG do que pelas outras organizações. Um dos motivos para isto está relacionado ao número de elementos nestes meta-modelos. Por exemplo, os meta-modelos da OMG avaliados possuem um total de 10 *enumerations*, enquanto os meta-modelos de outras organizações possuem 7. Portanto, os meta-modelos com um maior número de elementos têm maior probabilidade de aplicação das diretrizes.

**Tabela 11: Quantidade de aplicações das diretrizes definidas pela OMG e por outras organizações.**

| **Diretriz / Organiz.** | *OMG* | *Outras Organizações* |
|---|---|---|
| D1 | 0 | 8 |
| D2 | 4 | 24 |
| D5 | 2 | 3 |
| D8 | 0 | 3 |
| D9 | 15 | 41 |
| D10 | 6 | 3 |
| D11 | 32 | 144 |
| D12 | 3 | 36 |
| D13 | 2 | 0 |

Avaliando de uma maneira geral, podemos dizer que a média de aplicações de cada diretriz, que varia de 0 a 29.3, foi satisfatória. Pudemos comprovar que, além de úteis, algumas diretrizes foram até aplicadas várias vezes em um mesmo meta-modelo, o que reforça ainda mais a sua aplicabilidade. Estes meta-modelos foram escolhidos propositalmente para a avaliação, de forma a mostrar o contraste do número de aplicações de cada diretriz em meta-modelos da OMG e de outras organizações.

**Métrica 2: MDA (Média de Diretrizes Aplicadas)**

Na seqüência, vejamos uma análise da Questão 1 com base na métrica MDA que, por sua vez, pode ser melhor analisada no gráfico ilustrado na Figura 75. O gráfico mostra, de forma sintetizada, cada meta-modelo seguido do número de diretrizes que foram aplicadas. Visto que o somatório destes números é igual a 35 e o número de meta-modelos analisados é igual a 6, o valor obtido da métrica MDA é 5,8. Então, podemos dizer que de um total de 13 diretrizes, tivemos uma média de 5,8 aplicadas nos meta-modelos analisados.

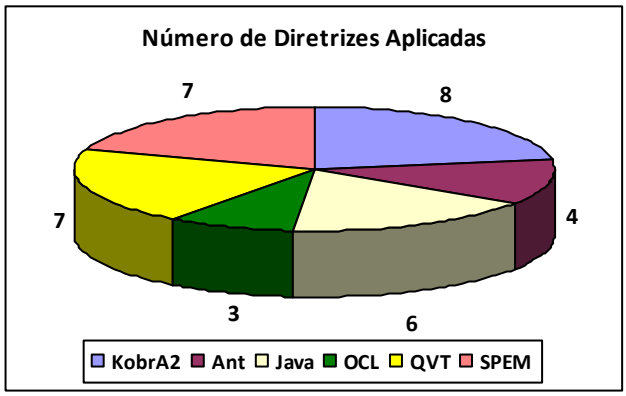



**Figura 75: Análise da métrica MDA.**

Analisando o gráfico, podemos notar que o maior número de diretrizes aplicadas encontra-se nos meta-modelos de KobrA2, Java, QVT e SPEM. A razão para isto pode ser justificada pelo fato destes serem os maiores meta-modelos participantes da avaliação. Para constatarmos este fato, a Tabela 12 apresenta o número total de elementos de cada um dos meta-modelos, considerando as meta-classes, atributos, *enumerations* e associações. Nesse sentido, com um total de 362, 295, 256 e 208 elementos, respectivamente, podemos ver que, de fato, estes quatro são os maiores meta-modelos da avaliação. Logo, podemos deduzir que quanto maior for o meta-modelo, maior será a probabilidade de aplicação das diretrizes.

Por outro lado, podemos ver na Tabela 12 que o meta-modelo de OCL, com apenas 95 elementos, é o menor dentre os avaliados e, de acordo com o gráfico da Figura 75, é o que teve o menor número de aplicações. Contudo, não se deve fazer deste fato uma regra, pois pode haver meta-modelos pequenos com grande aplicabilidade das diretrizes, dependendo de quão bem eles foram construídos. Da mesma forma, pode haver grandes meta-modelos sem a necessidade de aplicar nenhuma das diretrizes.

**Tabela 12: Número total de elementos para cada meta-modelo.**

| **Meta-modelo / Elemento** | *Meta-classes* | *Atributos* | *Enums* | *Associações* | *Total* |
|---|---|---|---|---|---|
| *OCL* | 55 | 6 | 1 | 33 | 95 |
| *Ant* | 48 | 93 | 0 | 27 | 168 |
| *SPEM* | 68 | 25 | 5 | 110 | 208 |
| *QVT* | 87 | 20 | 4 | 135 | 256 |
| *Java* | 95 | 46 | 4 | 150 | 295 |
| *KobrA2* | 258 | 17 | 3 | 84 | 362 |

Avaliando de uma maneira geral, podemos dizer que a média (5,8) de diretrizes aplicadas nos meta-modelos analisados foi satisfatória, pois, além de quase todas elas terem sido aplicadas (apenas quatro não foram), podemos observar que para cada um destes meta-modelos aplicamos mais de uma diretriz. Além disso, algumas delas foram aplicadas diversas vezes. Apesar desta média não representar nem metade do número de diretrizes propostas, vale considerar que elas foram aplicadas nas versões finais de meta-modelos já consolidados e construídos por especialistas. Até mesmo os meta-modelos da OMG tiveram um bom número de diretrizes aplicadas, apesar de ter sido em uma menor quantidade de aplicações. Possivelmente, o número de aplicações seria muito maior se tivéssemos considerado as versões preliminares dos meta-modelos.

### 5.4.2 Questão 2

Para responder a Questão 2 (Qual a qualidade que os meta-modelos alcançarão com a aplicação das diretrizes?) foi realizada uma discussão de caráter qualitativo, envolvendo seis aspectos:

- **Compreensão -** Discute a capacidade de compreensão do meta-modelo após a aplicação de cada diretriz;
- **Manutenção -** Discute cada diretriz quanto a manutenibilidade do meta-modelo depois de sua aplicação, ou seja, discute se o meta-modelo passa a ter uma maior facilidade de manutenção, como por exemplo, alteração de seus elementos;
- **Evolução -** Discute a capacidade e facilidade de evolução dos meta-modelos após a aplicação das diretrizes, como por exemplo, a extensão de um meta-modelo de forma a adicionar novas características;
- **Reuso -** Discute as diretrizes quanto ao aumento da capacidade de reuso dos meta-modelos depois de sua aplicação;
- **Redundância -** Existem meta-modelos que já foram construídos com alguma redundância e outros que a adquiriu ao longo de manutenções. Algumas das diretrizes têm como objetivo eliminar este problema. Portanto, neste aspecto discute-se se ocorreu diminuição da redundância existente nos meta-modelos após a aplicação das diretrizes;

- **Organização -** Discute quão organizados, no sentido de estruturação em pacotes, os meta-modelos se tornam após a aplicação das diretrizes.

No decorrer da fase de Coleta de Dados, a aplicação das diretrizes nos meta-modelos foi ilustrada em detalhes e, desta forma, foi possível observarmos os trechos dos meta-modelos antes e depois da aplicação. Portanto, podemos perceber que os meta-modelos apresentaram uma maior qualidade depois da aplicação das diretrizes. Nesse sentido, as diretrizes, de uma forma geral, oferecem uma melhor compreensão, manutenção, reuso e, conseqüentemente, maior capacidade de evolução do meta-modelo. Isto significa dizer que a aplicação das mesmas tende a atribuir cada vez mais qualidade ao meta-modelo. A seguir, vejamos uma breve discussão sobre os benefícios alcançados pelos meta-modelos avaliados após a aplicação das diretrizes:

A aplicação da **diretriz D1** eliminou a redundância de atributos com a mesma semântica que se repetiam em meta-classes diferentes dos meta-modelos. Outra vantagem está relacionada à evolução, reuso e manutenção, pois qualquer mudança nos atributos ou na própria semântica deverá ser feita apenas nas meta-classes abstratas. Por fim, houve também uma melhoria na compreensão do meta-modelo.

A aplicação da **diretriz D2** fez com que os meta-modelos avaliados se tornassem menos sobrecarregados, uma vez que eliminou a repetição das associações, isto é, a redundância. Os meta-modelos também obtiveram maior facilidade de reuso, futuras evoluções e de manutenção. Houve também uma melhoria na compreensão dos mesmos.

A aplicação da **diretriz D5** facilitou a localização das meta-classes reutilizáveis, assim, facilitando também o reuso de partes dos meta-modelos avaliados, uma vez que o pacote `Abstractions` foi criado. Outra facilidade está na manutenção, compreensão e evolução dos meta-modelos, uma vez que suas meta-classes reutilizáveis são melhor organizadas em um só pacote.

A aplicação da **diretriz D8** reduziu a repetição de expressões OCL, facilitando o entendimento das restrições e futuras manutenções. A definição de operações adicionais nos meta-modelos também é útil para que outras operações as reutilizem.

A aplicação da **diretriz D9** evitou que os atributos booleanos dos meta-modelos avaliados tivessem valores nulos, nos casos em que nenhum valor foi definido para eles. Isto é muito importante, porque atributos booleanos não devem ter valor nulo. Além disso, se, por

exemplo, a maioria das instâncias da meta-classe `ComponentClass` pertencente ao meta-modelo de KobrA2 (como mostrou a Figura 52) possuir o valor *false* para seu atributo `isSubject`, podemos defini-lo como sendo o seu valor *default*. Logo, não será mais necessário especificá-lo para cada instância da meta-classe que possua este mesmo valor, apenas para os que possuem valor diferente.

A aplicação da **diretriz D10** evitou que atributos do tipo de qualquer *enumeration* dos meta-modelos tivessem valores nulos, nos casos em que nenhum valor foi definido para eles. Além disso, se, por exemplo, todos ou a maioria dos atributos do tipo `CollectionKind`, *enumeration* pertencente ao meta-modelo de OCL (como mostrou a Figura 84), assumirem o valor `Set`, basta defini-lo como o *default* deste *enumeration* para não precisarmos defini-lo em cada instância da meta-classe.

A aplicação da **diretriz D11** facilitou bastante a evolução dos meta-modelos avaliados e, conseqüentemente, ofereceu uma maior facilidade de entendimento e manutenção. Como no exemplo do meta-modelo de Ant ilustrado na Figura 82, um `FileSet` não está obrigado a pertencer à `ClassPath` e `Path` simultaneamente. Logo, se futuramente ele pertencer a uma outra meta-classe, não haverá preocupação com as multiplicidades das associações já existentes, pois elas já estão consistentes e livres de erros semânticos.

A aplicação da **diretriz D12** ofereceu uma maior facilidade de entendimento e manutenção dos meta-modelos, pois os atributos booleanos de suas meta-classes se tornaram padronizados e mais facilmente identificados devido à convenção de nomes que adotamos em todo o meta-modelo.

A aplicação da **diretriz D13** ofereceu uma maior facilidade de entendimento e manutenção dos meta-modelos, pois os seus *enumerations* se tornaram padronizados e mais facilmente identificados devido à convenção de nomes que adotamos em todo o meta-modelo.

### 5.4.3 Considerações Finais

No intuito de verificarmos a aplicabilidade de cada uma das diretrizes no contexto de meta-modelos e avaliá-las de maneira quantitativa, consideramos duas métricas ($MAD_i$ e MDA). Na fase de planejamento do processo de avaliação, foram selecionados seis meta-modelos para ilustrarmos e analisarmos a aplicação das diretrizes. Por fim, foi realizada a interpretação dos resultados obtidos com o experimento.

Em geral, os resultados deste experimento foram bastante satisfatórios, principalmente, porque os meta-modelos avaliados já estão em suas versões finais, o que indica que as diretrizes podem servir de instrução até mesmo para os especialistas mais experientes em meta-modelagem. Adicionalmente, vale considerarmos também que cada aplicação envolve vários elementos, assim, efetivando-a ainda mais. Por exemplo, a diretriz D8 pôde ser aplicada três vezes no meta-modelo KobrA2, contudo, uma destas aplicações envolve 22 restrições OCL com expressões repetitivas. Logo, a operação criada com a aplicação desta diretriz será empregada em todas estas restrições, assim, aumentando a qualidade do meta-modelo.

# Capítulo 6

# Trabalhos Relacionados

Com a crescente disseminação da abordagem de MDA, está surgindo cada vez mais a necessidade de se utilizar algum mecanismo que auxilie as pessoas no desenvolvimento dos diferentes artefatos dessa infra-estrutura – modelos, meta-modelos e transformações. Com o propósito de preencher esta lacuna, alguns trabalhos vêm sugerindo diretrizes ou padrões que podem ser aplicados nestes artefatos.

Nas subseções a seguir serão apresentados alguns trabalhos considerados relevantes, os quais espelham o atual estado da arte no contexto de padrões, diretrizes ou qualquer outro mecanismo que sirva de auxílio para a construção de um dos artefatos de MDA: os meta-modelos. Como veremos, alguns destes trabalhos estão intensamente relacionados ao nosso, relatando diretrizes para meta-modelagem. Outros, apesar de não possuírem ligação direta, referem-se à aplicação de padrões em UML, o que não deixa de ser uma abordagem interessante para ser analisada e comparada, pois a própria UML foi uma das especificações que inspirou a identificação de algumas de nossas diretrizes.

Primeiramente, serão apresentados alguns trabalhos referentes à identificação de padrões no escopo de restrições OCL, transformações e meta-modelos. Em seguida, veremos alguns trabalhos que relatam a aplicação de padrões em UML, tais como os padrões de projeto e de processo. Por fim, serão citados outros trabalhos que não se referem nem à identificação nem à aplicação de padrões, mas que de alguma forma estão relacionados com a nossa pesquisa.

## 6.1 Padrões OCL

Foram encontrados poucos trabalhos que relatam a identificação de padrões para especificação de restrições OCL.

[Ackermann, 2006] propõe o uso de padrões para a geração automática de restrições OCL no intuito de simplificar a tarefa de definir tais restrições manualmente. Para isto, alguns padrões de especificação de restrições OCL foram identificados manualmente e descritos, enumerando as principais características de acordo com um *template* pré-estabelecido, bem como mostrando um exemplo de aplicação e a especificação formal de cada um. Adicionalmente, este trabalho estende o meta-modelo de OCL para incorporar funções responsáveis por criar os padrões de especificação OCL. [Ackermann, 2006] também propõe uma ferramenta de suporte para geração automática de restrições, a qual nos permite definir os parâmetros necessários para cada padrão e, dessa forma, gerar a restrição OCL desejada. Este trabalho estende [Ackermann, 2005] com um catálogo constituído por 18 padrões encontrados para especificação de restrições OCL, incluindo os padrões ilustrados em [Ackermann, 2005].

[Ackermann, 2006] possui uma certa relação com o nosso trabalho. Apesar de ambos não focarem no mesmo artefato da infra-estrutura de MDA, eles identificam padrões ou diretrizes que auxiliam em suas respectivas tarefas.

## 6.2 Padrões em Transformações

Assim como os padrões OCL, este outro escopo para identificação de padrões possui uma quantidade muito pequena de trabalhos produzidos.

[Bézivin et al., 2005] propõe uma coleção com um número de 23 padrões de projeto para transformações de modelos no contexto da abordagem MDE (*Model-Driven Engineering*) [Kent, 2002]. Esta coleção ainda está sendo expandida na medida em que os usuários de ATL reportam novas experiências. Como ATL é uma linguagem bastante usada por usuários diferentes, [Bézivin et al., 2005] conseguiu coletar uma grande biblioteca de transformações em ATL a partir destes usuários que, por sua vez, reportavam situações recorrentes que poderiam ser descritas por padrões MDE. Além disso, [Bézivin et al., 2005] está em busca de diferentes categorias de padrões MDE, algumas relacionadas à estrutura do

meta-modelo, descrição de transformação, padrões de projeto para mega-modelos e padrões de projeto para OCL.

[Iacob et al., 2008] reflete a experiência dos autores com a especificação e execução de transformações de modelos utilizando como linguagens QVT *Core* e *Relations*. Com o objetivo de contribuir para a engenharia de transformação, o trabalho propõe alguns padrões úteis para especificação de transformações, os quais são soluções reusáveis para problemas de transformação de modelos em geral. Da mesma forma, ele ilustra como tais padrões podem ser aplicados em especificações de transformações complexas. Além dos padrões, [Iacob et al., 2008] ainda propõe um método para documentação e especificação dos mesmos. Uma iniciativa interessante foi a criação de um catálogo *Wiki* com a intenção de divulgar os padrões já encontrados, assim como abrir espaço para que outras pessoas também possam contribuir com novos padrões. Desta forma, ampliando cada vez mais a base de conhecimento acerca da engenharia de transformação.

Em síntese, os dois trabalhos discutidos anteriormente exploram a identificação de padrões no escopo de transformações de modelos, de forma a aplicar soluções reutilizáveis em problemas de transformações em geral. Apenas [Bézivin et al., 2005] menciona alguns trabalhos iniciais sobre padrões em meta-modelos, porém, não apresenta exemplos de nenhum padrão nesta abordagem. [Bézivin et al., 2005] pretende expandir o conjunto de padrões, estendendo-o para outras categorias. Já [Iacob et al., 2008] pretende ampliar ainda mais o número de padrões por meio de um catálogo *Wiki*. Contudo, nenhum destes trabalhos mostrou diretrizes ou padrões identificados em meta-modelos, nem uma ferramenta de suporte para automatizar a aplicação.

## 6.3 Padrões em Meta-modelos

A identificação de padrões ou diretrizes no escopo de meta-modelos é justamente o foco do nosso trabalho. Como veremos, existem poucos trabalhos relacionados com esta idéia, os quais apresentam abordagens diferentes da nossa.

[Mili et al., 1998] identifica três padrões de meta-modelagem específicos para o meta-modelo de *SmallTalk* [Goldberg et al., 1989]. No entanto, eles não utilizam a infra-estrutura de MDA, nem seus formalismos, como MOF ou OCL. Além disso, os meta-modelos usados são pequenos se comparados aos da OMG.

[García et al., 2009] propõe uma diretriz com o objetivo de facilitar a definição de Linguagens de Modelagem (MLs) através de meta-modelos, tendo como foco as MLs baseadas em conexão. A diretriz define um *framework* que provê uma estrutura para organizar os elementos do meta-modelo de uma ML, assim como oferece diferentes alternativas para a representação Entidade-Relacionamento das características deste meta-modelo. A diretriz já foi aplicada em dois casos reais: nos meta-modelos de INGENIAS [Pavón et al., 2005] e SPEM 1.1.

Apesar de se referir à diretrizes para meta-modelos, a abordagem apresentada em [García et al., 2009] difere da que propomos em nosso trabalho. Ela define um processo decisório para a definição de meta-modelos composto por quatro atividades. Nestas atividades, o desenvolvedor especifica as características da linguagem que deseja modelar e, ao final do processo, a diretriz propõe um meta-modelo para esta linguagem com base nas informações passadas. Logo, a diretriz serve para orientar a meta-modelagem de cada meta-modelo, em específico, de acordo com as características da ML que o desenvolvedor especificar. Já a abordagem do nosso trabalho propõe um conjunto de diretrizes documentadas em forma de catálogo, podendo ser aplicadas em qualquer tipo de meta-modelo, sem restrições, e não dependem de nenhum processo para a utilização.

[Hu et al., 2001] apresenta uma abordagem orientada a padrão para auxiliar na construção de meta-modelos e definição de elementos básicos de um padrão de meta-modelo XML. Esta abordagem ajuda pessoas a adquirir a experiência de outros, reduzir custo de desenvolvimento e acelerar o processo de meta-modelagem. No intuito de melhor descrever a estrutura de cada padrão, o artigo os dividiu em três grupos: padrões arquiteturais, padrões de projeto e padrões de programação. Este trabalho tem como foco padrões para meta-modelos XML, ou seja, são regras que descrevem como criar, usar e compor os elementos XML para estabelecer meta-modelos XML. Para descrever os padrões, este trabalho segue um *template* formado por nome, problema e contexto, casos de uso adicionais, estrutura, regras, implementação, conseqüências e padrões relacionados. Alguns dos padrões propostos são: *Embedding Container, Self-Embedding Container, Link* e *Association*. Em todos os exemplos foi utilizado *MS XML schema* para a implementação do meta-modelo.

De fato, os três trabalhos apresentados anteriormente são referentes a padrões ou diretrizes no escopo de meta-modelos. Contudo, [Mili et al., 1998] e [Hu et al., 2001] propõem padrões para meta-modelos específicos, *SmallTalk* e XML, respectivamente. Já

[García et al., 2009] propõe uma diretriz para facilitar a definição de MLs através de meta-modelos. Nenhum destes trabalhos propõe diretrizes ou padrões independentes de domínio, que possam ser aplicados em qualquer meta-modelo. Além do mais, nenhum deles dispõe de uma ferramenta de suporte para automatizar a aplicação dos padrões oferecidos.

## 6.4 Aplicação de Padrões em UML

Nesta subseção serão apresentados alguns trabalhos que, apesar de não possuírem relação direta com o nosso, abordam aspectos interessantes da aplicação de padrões em UML que, de uma forma ou de outra, contribuíram para o desenvolvimento deste trabalho.

Os trabalhos desta subseção estão divididos em duas categorias: padrões de projeto e padrões de processo. Uma grande diferença em relação ao nosso trabalho é que nenhum deles apresenta a identificação de novos padrões ou diretrizes, ao invés disso, abordam a aplicação destes em UML.

### 6.4.1 Padrões de Projeto

[France et al., 2004] propõe uma extensão do meta-modelo de UML com a finalidade de criar uma nova sintaxe, baseada em UML, para especificação de padrões de projeto. O principal objetivo deste trabalho é a criação de uma técnica prática para a especificação de padrões, oferecendo suporte ao uso destes durante a modelagem do projeto, ou seja, a aplicação de padrões de projeto em modelos UML. Para isto, os autores desenvolveram uma linguagem para especificação de padrões que usa a sintaxe UML. Como resultado, eles desenvolveram especificações de padrões completas para os seguintes padrões de projeto: *Abstract Factory, Bridge, Decorator, Singleton, Observer, Composit* e *Visitor*.

Apesar de não sugerir nenhum padrão novo, [France et al., 2004] torna-se bastante interessante devido à técnica criada para oferecer suporte ao uso de padrões durante a modelagem de sistemas. Através de vários exemplos, o trabalho deixa bastante clara a aplicação dos padrões de projeto em UML. Contudo, esta técnica está restrita para descrições de estrutura e comportamento que podem ser expressas em UML.

[Dong et al., 2003] destaca que há uma certa dificuldade na identificação dos elementos do modelo – classes, operações e atributos – pertencentes à determinado padrão, uma vez que este padrão é modelado e documentado em UML. Isto ocorre porque a UML não

guarda informações sobre o padrão de projeto aplicado. Entretanto, ela possui mecanismos de extensão, como o perfil UML, que nos permite definir nomes e marcações (estereótipos, *tagged values* e restrições) apropriados para os elementos do modelo, de forma a facilitar a identificação dos padrões. Nesse sentido, [Dong et al., 2003] propõe um conjunto de estereótipos UML, *tagged values* e restrições para serem adicionados à diagramas UML no intuito de, explicitamente, identificar padrões de projeto nestes diagramas.

Para visualizarmos a aplicação dos padrões de projeto nos diagramas UML, [Dong et al., 2003] define um perfil UML com quatro estereótipos, mostrando um exemplo prático em que todos eles são aplicados.

[Pires et al.] propõe a extensão de um trabalho anterior [Pires et al., 2008], o qual apresenta uma técnica que permite verificar a conformidade entre implementações Java e diagramas de classes UML usando testes de *design.* Entretanto, [Pires et al., 2008] não era capaz de verificar completamente os padrões de projeto. A extensão [Pires et al.] tem a finalidade de criar uma biblioteca de padrões de projeto que possam ser verificáveis com códigos-fonte Java. A nova abordagem consiste em: um conjunto de perfis UML para incorporar padrões de projeto em diagramas de classe UML; um conjunto de *templates* de testes de *design* para verificar a implementação correta dos padrões de projeto com os códigos-fonte Java; e um conjunto de transformações MDA para gerar, automaticamente, estes testes de *design*.

A abordagem apresentada em [Pires et al.] mostra alguns pontos bem interessantes. Ela não apenas aplica padrões de projeto em UML, como feito nos demais trabalhos mencionados nesta subseção, mas também utiliza testes de *design* gerados, automaticamente, através de transformações MDA para verificar a implementação de tais padrões com os códigos-fonte Java.

Apesar de [France et al., 2004], [Dong et al., 2003] e [Pires et al.] possuírem a mesma idéia de aplicar os padrões de projeto em modelos UML, cada um adota uma abordagem diferente para este fim. O primeiro estende o meta-modelo de UML a fim de criar uma nova sintaxe para especificação de padrões de projeto. Já o segundo e o terceiro, usam mecanismos de extensão da UML para definir um perfil UML com o intuito de visualizar os padrões de projeto. Além destes, existem vários outros trabalhos que tratam da aplicação de padrões de projeto neste escopo, como por exemplo, [Mak et al., 2004] e [Sunyé et al., 2000].

### 6.4.2 Padrões de Processo

Segundo [Störrle, 2001], processos de *software* são descritos em uma variedade de formalismos e estilos, contudo, alguns requisitos essenciais não são adequadamente abordados por muitos deles, tais como entendimento, flexibilidade, precisão e estrutura fractal. Nesse sentido, [Störrle, 2001] acredita que os padrões de processo que utilizam UML são uma nova forma de lidar com estas questões. Já existem alguns trabalhos que mencionam o termo "padrão de processo", porém, estes são muito superficiais, apresentando poucos detalhes na descrição dos padrões. Comparada com as outras abordagens existentes, a proposta de [Störrle, 2001] é uma das poucas que tenta incorporar a UML.

De fato, [Störrle, 2001] apresenta uma abordagem interessante para o contexto de processos. Entretanto, este trabalho, além de não apresentar de forma clara os padrões de processo, não usa nenhum exemplo prático ou estudo de caso para demonstrar como seria a aplicação de tais padrões.

## 6.5 Outros Trabalhos Relacionados

Foram encontrados vários trabalhos que não se enquadram nas abordagens apresentadas anteriormente, mas alguns deles discutem questões interessantes que foram úteis para o desenvolvimento deste trabalho e outros discutem alguns pontos comparativos com a nossa abordagem.

[Cook et al., 2004] propõe a especificação de PIMs com a utilização de uma linguagem funcional, de maneira que sejam utilizados padrões de projeto nestes modelos. Dessa forma, permitindo que as transformações para os PSMs possuam a otimização de tais padrões. Em síntese, o trabalho não propõe nenhum padrão novo, ele apenas utiliza na especificação do PIM alguns padrões já existentes, chamados pelos autores de "*Design* PIM", com o intuito de gerar seus PSMs com tais otimizações.

[Herzner et al., 2006] propõe a aplicação do conceito de padrões em nível de definição dos requisitos, isto é, no PIM. A finalidade é amenizar os problemas existentes durante a especificação do PIM, a qual é uma tarefa realizada manualmente. Como problemas desta natureza podemos citar os possíveis erros que podem ser gerados devido ao número de elementos repetitivos a serem criados. Com esta finalidade, o trabalho apresenta como abordagem a utilização de padrões específicos de domínio para a modelagem da aplicação,

denominados “PIM *Patterns*”. Além disso, [Herzner et al., 2006] mostra um estudo de caso do projeto DECOS [Althammer, 2002], o qual se destina ao suporte de desenvolvimento para sistemas em tempo real embarcados e distribuídos de mais alta criticidade. A utilização do conceito de “PIM *Patterns*” pode auxiliar o trabalho do projetista a partir de padrões para criação de vários elementos PIM de uma só vez. Ao utilizarmos tais padrões no PIM como elementos estruturais será possível oferecer suporte ao rastreamento, como também reduzir sua complexidade.

[Frank et al., 2005] faz uma visão geral de uma das ferramentas mais utilizadas no meio acadêmico para a execução de tarefas de mineração de dados, a Weka. Esta ferramenta encontra-se disponível em quase todas as plataformas e foi projetada de forma que o usuário possa testar, rapidamente, vários algoritmos em novos conjuntos de dados com muita flexibilidade. [Frank et al., 2005] mostra suas principais funcionalidades, aplicações, alguns dos algoritmos implementados e tarefas de mineração suportadas, bem como as principais vantagens e desvantagens de sua utilização. Este trabalho foi bastante útil para analisarmos a ferramenta Weka e compararmos com as existentes no mercado.

[Michail, 2000] tem a finalidade de mostrar como podemos usar a mineração de dados para a descoberta de uma biblioteca de padrões que podem ser reutilizáveis em aplicações já existentes. Para isto, ele propõe a aplicação de regras de associação generalizadas, as quais apresentam uma melhoria em relação às regras de associação tradicionais devido à aplicação de um novo elemento denominado taxonomia. De maneira geral, a taxonomia utiliza o conceito de especialização para formular as regras, dessa forma, nos permite realizar a mineração em diferentes níveis de abstração.

Embora não seja muito recente e não utilize uma abordagem inovadora para a mineração de dados, [Michail, 2000] constitui-se um trabalho interessante pela aplicação que ele faz da mineração de dados para descobrir uma biblioteca de padrões reutilizáveis em aplicações já existentes. Este foi um dos trabalhos que nos serviu como base para melhor entendermos como se dá a aplicação da mineração de dados em contextos diferentes.

# Capítulo 7

# Conclusões

Neste trabalho, identificamos um conjunto de diretrizes para meta-modelos da infra-estrutura de MDA no sentido de melhorar o desenvolvimento destes artefatos. A idéia inicial era que todas as diretrizes fossem identificadas de forma automática. Por este motivo, exploramos uma área multidisciplinar para descoberta de padrões, chamada mineração de dados e, dentre suas tarefas, utilizamos a análise de associação acompanhada do algoritmo *Apriori*. Além disso, desenvolvemos uma ferramenta de suporte a fim de nos auxiliar na realização de algumas atividades para esta descoberta.

Contudo, os resultados obtidos não foram os esperados: conseguimos descobrir algumas regras de associação que serviram para a identificação de uma única diretriz e várias outras regras que serviram para a validação de meta-modelos. Em contrapartida, no intuito de identificarmos mais diretrizes, realizamos também um procedimento manual por meio de uma análise detalhada em meta-modelos variados.

Como resultado final do processo de identificação, manual e automático, propomos um catálogo com 13 diretrizes. Neste catálogo, cada uma delas foi documentada de forma padronizada, conforme um *template* pré-estabelecido. Estas diretrizes propõem oferecer, dentre outras vantagens, uma maior facilidade de construção, manutenção, entendimento, evolução e reuso dos meta-modelos. Adicionalmente, desenvolvemos uma ferramenta com o objetivo de aplicar, automaticamente, as diretrizes em meta-modelos já existentes.

Para a avaliação deste trabalho, selecionamos seis meta-modelos com a finalidade de aplicarmos as diretrizes e verificarmos a aplicabilidade de cada uma delas. Para isto, escolhemos meta-modelos bem consolidados, dentre os quais alguns são da OMG. Portanto, são meta-modelos consistentes e confiáveis, construídos e validados por especialistas da área.

Como resultado, em relação à aplicabilidade, observamos uma média de 5,8 diretrizes aplicadas, considerando todos os meta-modelos analisados. Apesar de não ter ocorrido a aplicação de todas, o número de diretrizes aplicadas nestes meta-modelos foi satisfatória, tendo em vista que quase todas elas puderam ser aplicadas diversas vezes. Além do mais, vale ressaltar que estas diretrizes não são úteis apenas para a aplicação em meta-modelos já existentes, mas, principalmente, para auxiliar pessoas no desenvolvimento destes artefatos. Com a análise dos resultados, observamos também que uma maior quantidade de diretrizes foi aplicada nos meta-modelos mais extensos. Contudo, isto não pode ser considerado sempre como verdade, pois pode haver meta-modelos menores com grande aplicabilidade das diretrizes, dependendo de quão bem eles foram construídos.

Considerando a qualidade alcançada pelos meta-modelos, notamos que, de fato, houve uma melhor compreensão, manutenção, reuso e, conseqüentemente, maior capacidade de evolução destes artefatos. Além do mais, é importante destacarmos o impacto das diretrizes em relação à produtividade na tarefa de meta-modelagem. Provavelmente, um desenvolvedor construirá um novo meta-modelo de forma mais rápida se ele conhecer diretrizes que o orientem com boas práticas. Desta forma, concluímos que a aplicação das diretrizes tende a atribuir cada vez mais qualidade aos meta-modelos. Vale salientar que a análise qualitativa realizada neste trabalho apresentou uma opinião nossa em relação aos resultados obtidos com a aplicação das diretrizes, não houve nenhuma avaliação qualitativa.

## 7.1 Contribuições

Este trabalho oferece diversas contribuições para a abordagem de MDA, conforme ressaltamos a seguir:

**Catálogo de diretrizes para meta-modelagem**. Tendo em vista que o desenvolvimento de meta-modelos não é uma tarefa trivial na abordagem de MDA, este trabalho propõe 13 diretrizes no intuito de auxiliar as pessoas na realização desta tarefa. Estas diretrizes são independentes de domínio, desta forma, podendo ser utilizadas em qualquer

meta-modelo. Além do mais, o catálogo proposto pode ser estendido, futuramente, ao passo que surge a necessidade de novos artifícios que auxiliem na meta-modelagem;

**Exploração da mineração de dados**. Para automatizar o processo de identificação das diretrizes em meta-modelos exploramos a mineração de dados, seguida de suas técnicas, metodologias e ferramentas. Vale ressaltar que o nosso trabalho é o primeiro a explorar a mineração de dados no contexto de MDA. Apesar de não termos obtido os resultados esperados, todo o legado que deixamos sobre as etapas e os procedimentos necessários para a utilização da mineração de dados, desde o tratamento dos dados até a implantação do conhecimento, pode ser aproveitado para a descoberta de conhecimento em outras áreas;

**Validação de meta-modelos**. Dentre os resultados obtidos com a mineração de dados dentro do contexto de MDA, encontramos muitas regras de associação que não refletem conhecimento inovador sobre meta-modelagem. De acordo com os fundamentos da mineração de dados, uma regra também pode ser considerada interessante se ela validar alguma hipótese que o usuário pretende confirmar [Han et al., 2001]. Portanto, estas regras que encontramos podem ser usadas como um mecanismo para a validação de meta-modelos;

**Ferramenta de suporte para a mineração de dados em MDA**. Para nos auxiliar na realização de algumas atividades da mineração de dados, implementamos uma ferramenta com a colaboração de um aluno de iniciação científica. Em futuros trabalhos que apliquem a mineração de dados, esta ferramenta pode servir como exemplo para ilustrar como construir uma base de dados no formato ARFF, já com a eliminação de possíveis ruídos ou inconsistência nos dados. Além disso, ela pode servir para executar algoritmos de análise de associação, reutilizados da Weka, a partir de um arquivo ARFF informado pelo usuário;

**Ferramenta de suporte para a aplicação das diretrizes**. Além de auxiliar as pessoas na construção de novos meta-modelos, este trabalho oferece uma ferramenta de suporte desenvolvida com a finalidade de aplicar, automaticamente, as diretrizes em meta-modelos já construídos. Desta forma, aumentando a qualidade dos mesmos.

## 7.2 Trabalhos Futuros

Com a conclusão deste trabalho, algumas idéias surgiram no sentido tanto de dar continuidade ao que já foi desenvolvido quanto de propor outras direções de pesquisa. A seguir, vejamos alguns destes trabalhos que propomos como futuros:

- **Extensão do catálogo de diretrizes**. O catálogo proposto neste trabalho pode ser estendido para melhor servir à comunidade de MDA. Nesse sentido, novas diretrizes podem ser identificadas no escopo de meta-modelagem e documentadas seguindo o mesmo *template* utilizado neste trabalho;

- **Criação de um catálogo *Wiki***. Construir um catálogo *Wiki*, tal como o apresentado em [Iacob et al., 2008]. Este catálogo poderá servir como veículo de divulgação das diretrizes identificadas, de modo que a comunidade possa contribuir utilizando-as, adicionando novas e relatando os pontos positivos e negativos da experiência;

- **Conclusão da ferramenta de suporte para aplicação das diretrizes**. Complementar a ferramenta que desenvolvemos para que a mesma ofereça suporte à aplicação de todas as diretrizes em meta-modelos, pois neste trabalho foi implementado um total de cinco delas como contribuição inicial;

- **Identificação de diretrizes em outros contextos**. Seguindo a mesma idéia apresentada neste trabalho, é possível descobrir diretrizes em contextos diferentes dentro da infra-estrutura de MDA, como por exemplo, em transformações de modelos, modelos e OCL. Depois de identificadas, as novas diretrizes podem ser documentadas de acordo com o mesmo *template* que utilizamos para as diretrizes em meta-modelos;

- **Verificação semântica após a aplicação das diretrizes**. Em nosso trabalho, depois que as diretrizes forem aplicadas nos meta-modelos por meio da ferramenta de suporte, nenhuma verificação é realizada a fim de descobrir se houve alteração na semântica. Nesse sentido, pode-se investigar maneiras de checar formalmente a consistência dos meta-modelos, seja de forma manual ou automática. Contudo, vale ressaltar que cada uma das diretrizes propostas foi pensada e analisada de forma que a sua aplicação não ocasione alteração na semântica dos meta-modelos. Além do mais, o catálogo de diretrizes faz ressalvas quando a aplicação de alguma delas necessita de uma prévia análise por parte do usuário para verificar questões semânticas.

# Bibliografia


[Ackermann, 2005] Ackermann, J. Formal Description of OCL Specification Patterns for Behavioral Specification of Software Components. Em *Proceedings of the Workshop on Tool Support for OCL and Related Formalisms.* Relatório Técnico, LGL-REPORT-2005-001, 2005.

[Ackermann, 2006] Ackermann, J. *Detailed Description of OCL Specification Patterns*. Relatório Técnico. Universidade de Augsburg, Alemanha, 2006.

[Agrawal et al., 1993] Agrawal, R.; Imiclinski, T.; Swami, A. Mining Association Rules Between Sets of Items in Large Databases. Em *Proceedings of the ACM SIGMOD conference, Washington*, D.C., Maio, 1993.

[Agrawal et al., 1994] Agrawal, R., Srikant, R. Fast Algorithms for Mining Association Rules. Em *Proceedings of the 20th International Conference on Very Large Databases*, páginas 487-499, São Francisco. Morgan Kaufmann, 1994.

[Alexander et al., 1977] Alexander, C.; Ishikawa, S.; Silverstein, M. *A Pattern Language: Towns, Buildings, Construction*. New York, NY. Oxford University Press, 1977.

[Althammer, 2002] Althammer, E; Weienbacher, G; Herzner, W; Schoitsch, E. From Requirements to Deployment Verify that the Right Things are Done Correctly - the decos test bench. Em *ITSC'05*: *Proceedings of the 8th IEEE Int. Conf. On Intelligent Transportation Systems*, páginas 7–12, IEEE, 2002.

[Ant Project, 2010] *The Apache Ant Project*, 2010. http://ant.apache.org

[Ant, 2005] Ruchaud, J.; Brunel, P. *Meta-modelo de Ant*. AtlanMod, 2005. http://www.emn.fr/z-info/atlanmod/index.php/Ecore#Ant_0.3

[ARFF, 2008] *Attribute-Relation File Format* (ARFF), 2008. http://www.cs.waikato.ac.nz/~ml/weka/arff.html

[Atkinson et al., 2009] Atkison, C.; Robin, J.; Stoll, D. *Meta-modelo de KobrA2*, 2009. http://wakameemf.appspot.com/statics/KobrA2MM.pdf

[Atkinson et al., 2002] Atkinson, C.; et al. *Component-based Product Line Engineering with UML*, Addison-Wesley, 2002.

[ATL, 2008] *ATL - ATLAS Transformation Language, 2008*. http://www.eclipse.org/m2m/atl

[AtlanMod, 1990] *AtlanMod team (Atlantic Modeling)*, 1990. http://www.emn.fr/z-info/atlanmod/index.php/Main_Page

[ATLtoProblem, 2005] ATLAS group, LINA & INRIA, Nantes. *ATL Transformation Examples: ATL to Problem*, Versão 0.1, 2005.

[Balena et al., 2005] Balena, F.; Dimauro, G. *Practical Guidelines and Best Practices for Microsoft Visual Basic and Visual C# Developers*. (Pro-Developer). 1ª ed. Microsoft Press, 2005.

[Barroso, 1994] Barroso, M. E. G. V. *Dicionário Aurélio Eletrônico*. Versão 1. 3, Editora Nova Fronteira, 1994.

[Basili et al., 2001] Basili, V. R.; Caldiera, G.; Rombach, H. D. *The Goal Question Metric Approach*. 2º ed., Wiley-Interscience, Enciclopédia de Engenharia de *Software*, 2001.

[Bauer et al., 2007] Bauer, B.; Lautenbacher, F.; Roser, S. *AgilPro - Agile Processes in the Context of ERP*, 2007. http://wiki.eclipse.org/images/2/2f/AgilPro_MetamodelDescription.pdf

[Bézivin et al., 2005] Bézivin, J.; Jouault, F.; Paliès, J. Towards Model Transformation Design Patterns. ATLAS Group (INRIA & LINA, Universidade de Nantes). Em *EWMT'05*: *Proceedings of the First European Workshop on Model Transformations (EWMT),* 2005.

[Bossa, 2006] Youssef, S. *Meta-modelo de Bossa*. AtlanMod, 2006. http://www.emn.fr/z-info/atlanmod/index.php/Ecore#Bossa_1.0_1.1

[BPEL, 2006] Jouault, F. *Meta-modelo de BPEL*. AtlanMod, 2006. http://www.emn.fr/z-info/atlanmod/index.php/Ecore#BPEL_1.0

[Briand et al., 1997] Briand, L., C.; Differding, C., M.; Rombach, H. D. *Practical Guidelines for Measurement-Based Process Improvement*. Software Process Improvement and Practice, vol. 2, 1997.

[Carr, 2005] Carr, D. *Practical Guidelines and Improvements in Agile Software Process Development*, 2005. http://www.ironspeed.com/company/Practical%20Guidelines%20Article.aspx

[Cobol, 2005] Bruneliere, H.; Guyard, P. *Meta-modelo de Cobol*. AtlanMod, 2005. http://www.emn.fr/z-info/atlanmod/index.php/Ecore#COBOL_1.0

[Cook et al., 2004] Cook, W.; Nedunuri, S. Transforming Declarative Models Using Patterns in MDA. Em *Proceedings of the OOPSLA Workshop on Best Practices for Model Driven Software Development*, 2004.

[CWM, 2003] *Common Warehouse Metamodel (CWM) Specification*. Object Management Group (OMG). Versão 1.1 – Março, 2003. http://www.omg.org/spec/CWM/1.1/PDF

[Dong et al, 2003] Dong, J.; Yang, S. Visualizing Design Patterns With A UML Profile. Em *Proceedings of the IEEE Symposium on Visual/Multimedia Languages (VL),* 2003.

[DTD, 2005] Guyard, P. *Meta-modelo de DTD*. AtlanMod, 2005. http://www.emn.fr/z-info/atlanmod/index.php/Ecore#DTD_1.0

[Ecore, 2009] *Eclipse Modeling Framework Project (EMF),* 2009. http://www.eclipse.org/modeling/emf

[Express, 2007] Allilaire, F. *Meta-modelo de Express*. AtlanMod, 2007. http://www.emn.fr/z-info/atlanmod/index.php/Ecore#EXPRESS_0.1

[Fayyad et al., 1996] Fayyad, U.; Piatesky-Shapiro, G.; Smyth, P. From Data Mining to Knowledge Discovery: An Overview. Em: *Advances in Knowledge Discovery and Data Mining*, AAAI Press, 1996.

[FilteredAssociator, 2010] *Algoritmo FilteredAssociator*, 2010. http://wiki.pentaho.com/display/DATAMINING/FilteredAssociator

[France et al., 2004] France, R. B.; Kim, D.-K.; Ghosh, S.; Song, E. *A UML-Based Pattern Specification Technique.* IEEE Trans. Software Eng., vol. 30, no. 3, Março, 2004.

[Frank et al., 2005] Frank, E.; Hall, M.; Holmes, G.; Kirkby, R.; Pfahringer, B. WEKA - A Machine Learning Workbench for Data Mining. Em: *O. Maimon and L. Rokach*, editors, The Data Mining and Knowledge Discovery Handbook, páginas 1305-1314. Springer, 2005.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R.; Vlissides, J. *Design patterns: Elements of reusable object-oriented software*. Addison Wesley, 1995.

[García et al., 2009] García-Magariño, I.; Fuentes-Fernández, R.; Gómez-Sanz, J. *Guideline for the Definition of EMF Metamodels Using an Entity-Relationship Approach*. Information and Software Technology, Elsevier, 2009.

[GNU, 2008] *GNU - General Public License*, 2008. http://www.gnu.org

[Goldberg et al., 1989] Goldberg, A.; Robson, D. *Smalltalk-80 The Language*. Addison-Wesley Publishing Company, 1989.

[Grafcet, 2005] Guyard, P. *Meta-modelo de Grafcet*. AtlanMod, 2005. http://www.emn.fr/z-info/atlanmod/index.php/Ecore#GRAFCET_1.0

[Han et al., 2001] Han, J.; Kamber, M. *Data Mining: Concepts and Techniques*. Academic Press, 2001.

[Herzner et al., 2006] Herzner, W.; Csertán, G.; Balogh, A. *Design Patterns for Domain-specific Application Modelling*. 2° ERCIM/DECOS Workshop em Euromicro (SEAA, DSD), Dubrovnik, Croácia, 2006.

[HTML, 2005] Allilaire, F. *Meta-modelo de HTML 1.0*. AtlanMod, 2005. http://www.emn.fr/z-info/atlanmod/index.php/Ecore#HTML_1.0

[Hu et al., 2001] Hu, Z.; Vollmar, G. Towards XML Metamodel Patterns for XML Data Modeling. Dexa, páginas 0071-0075. Em *Proceedings in 12th International Workshop on Database and Expert Systems Applications*, 2001.

[Iacob et al., 2008] Iacob, M.-E.; Steen, M. W. A.; Heerink, L. Reusable Model Transformation Patterns. Freeband. Em *3M4EC'08*: *Proceedings of the Workshop on Models and Model-driven Methods for Enterprise Computing (3M4EC)*, 2008.

[IRL, 2006] Abouzahra, A. *Meta-modelo de IRL 0.1*. AtlanMod, 2006. http://www.emn.fr/z-info/atlanmod/index.php/Ecore#IRL_0.1

[J2SE5, 2007] Barbero, M. *Meta-modelo de J2SE 1.0*. AtlanMod, 2007. http://www.emn.fr/z-info/atlanmod/index.php/Ecore#J2SE5_1.0

[Jackowski, 2003] Jackowski, Z. *Metamodel of a System Development Method*. Agile Alliance Articles, Setembro, 2003.

[Java, 2006] Barbero, M. *Meta-modelo de Java 1.0*. AtlanMod, 2006. http://www.emn.fr/z-info/atlanmod/index.php/Ecore#JavaAbstractSyntax_1.0

[KDB2000, 2009] *KDB2000, a System for Knowledge Discovery in Databases*, 2009. http://www.di.uniba.it/~malerba/software/kdb2000

[KDM, 2007] ATLAS group (INRIA & LINA). *Meta-modelo de KDM 1.0*. AtlanMod, 2007. http://www.emn.fr/z-info/atlanmod/index.php/Ecore#KDM_1.0

[Kent, 2002] Kent, S. Model Driven Engineering. Em *IFM'02*: *Proceedings of the Third International Conference on Integrated Formal Methods - IFM2002*, LNCS 2335, Springer, páginas 286-298, 2002.

[Kitchenham et al., 2002] Kitchenham, B. A.; Pfleeger, S. L.; Pickard, L. M.; Jones, P. W.; Hoaglin, D. C.; El-Emam, K.; Rosenberg, J. *Guidelines for Empirical Research in Software Engineering*, IEEE Transactions on Software Engineering. Volume 28, páginas 721 – 734. IEEE Press Piscataway, NJ, USA, 2002.

[Kleppe et al., 2003] Kleppe, A.; Warmer, J.; Bast, W. *MDA explained: The model-driven architecture: practice and promise*. Object-Technology Series. Addison-Wesley, 2003.

[Leal e Ramalho, 2009] Leal, F. S.; Ramalho, F. S. *Investigação e Implementação de Técnicas de Mineração de Dados para Descoberta de Padrões em MDA*. VI Congresso de Iniciação Científica da Universidade Federal de Campina Grande, 2009. http://www.gmf.ufcg.edu.br/~fabiosl/pibic2008/Relatorio_Final_Pibic.pdf

[Mak et al., 2004] Mak, J.; Choy, C.; Lun, D. Precise Modeling of Design Patterns in UML. Em *Proceedings of the 26th International Conference on Software Engineering*. IEEE Computer Society, páginas 252-261, 2004.

[Maven, 2005] Brunel, P. *Meta-modelo de Maven*. AtlanMod, 2005. http://www.emn.fr/z-info/atlanmod/index.php/Ecore#Maven_.28maven.xml.29_0.3

[MDA, 2010] *OMG Model-Driven Architecture*. Object Management Group (OMG), 2010. http://www.omg.org/mda

[Mellor et al, 2004] Mellor, S.; Scott, K.; Uhl, A.; Weise, D. *MDA Distilled: Principles of Model-Driven Architecture.* Addison-Wesley, 2004.

[Michail, 2000] Michail, A. Data Mining Library Reuse Patterns using Generalized Association Rules. Em *Proceedings of the 22nd International Conference on Software Engineering*, páginas 167-176, 2000.

[Mili et al., 1998] Mili, H.; Pachet, F. *Patterns for Metamodeling*, 1998. http://citeseer.nj.nec.com/44259.html

[MOF, 2002] *OMG's MetaObject Facility*. Object Management Group (OMG). Versão 1.4 – Abril, 2002.

[MOF, 2006] *OMG's MetaObject Facility*. Object Management Group (OMG). Versão 2.0 – Janeiro, 2006. http://www.omg.org/mof

[MtoTTL, 2008] *MOF Model to Text Transformation Language*. Object Management Group (OMG). Versão 1.0, 2008. http://www.omg.org/spec/MOFM2T/1.0/PDF

[OCL, 2006] *Object Constraint Language*. Object Management Group (OMG). Versão 2.0 – Maio, 2006. http://www.omg.org/spec/OCL/2.0/PDF

[Oliveira et al., 2005] Oliveira, R. F.; Blois, A. P.; Vasconcelos, A.; Werner, C. *Metamodelo de Características da Notação Odyssey-FEX: Descrição de Classes*. Publicações Técnicas, COPPE/UFRJ, 2005.

[OMG, 2010] *OMG Specifications*, 2010. http://www.omg.org

[Orange, 2010] *Orange Data Mining Framework*, 2010. http://ailab.si/orange

[Pavón et al., 2005] J. Pavón, J.J. Gómez-Sanz, R. Fuentes. *The INGENIAS methodology and tool*. Em B. Henderson-Sellers, P. Giorgini (Eds.), Agent-Oriented Methodologies, Idea Group Publishing, London, UK, páginas 236–276, 2005.

[Pires et al., 2008] Pires W.; Brunet J.; Ramalho, F.; Serey D. *Uml-based Design Test Generation*. Em *SAC'08: Proceedings of the 2008 ACM symposium on Applied computing*, Track: Software Engineering. New York, NY, USA, ACM, páginas 735-740, 2008.

[Pires et al.] Pires W.; Ramalho, F.; Serey D.; Ledo, A. *Checking UML Design Patterns in Java Implementations*. Artigo não publicado ainda.

[PNML, 2005] Guyard, P. *Meta-modelo de PNML*. AtlanMod, 2005. http://gforge.inria.fr/plugins/scmsvn/viewcvs.php/*checkout*/AtlantEcore/PNML_basic.ecore?root=atlantic-zoos

[PredictiveApriori, 2010] *Algoritmo PredictiveApriori*, 2010. http://wiki.pentaho.com/display/DATAMINING/PredictiveApriori

[QVT, 2008] *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*. Object Management Group (OMG). Versão 1.0 – Abril, 2008. http://www.omg.org/docs/formal/08-04-03.pdf

[Ramalho, 2007] Ramalho, F. S. *MODELOG: Model-Oriented Development with Executable Logic Object Generation*, Tese de Doutorado, CIn-UFPE, Recife-PE, Brasil, fev/2007.

[Ramalho, 2008] Ramalho, F. S. *Curso de Desenvolvimento Dirigido por Modelos*, 2008. http://www.dsc.ufcg.edu.br/~franklin/disciplinas/2008-1/DDM/index.php/Main/HomePage.

[Rational, 1998] Rational Software. *Rational Unified Process: Best Practices for Software Development Teams*. Rational Software Corporation White Paper TP026B, 1998. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.27.439

[RUP, 2010] *Rational Unified Process (RUP)*, 2010. http://www-01.ibm.com/software/awdtools/rup

[SCADE, 2006] Eric, S. *Meta-modelo de Scade*. AtlanMod, 2006. http://www.emn.fr/z-info/atlanmod/index.php/Ecore#SCADE_1.0

[Šilingas, 2006] Šilingas, D. *Best Practices for Applying UML, Part I*. NoMagic Inc., 2006.http://www.magicdraw.com/files/whitepapers/Best_Practices_for_Applying_UML_Part1.pdf

[Silva e Paula, 2001] Silva, L. F.; Paula, V. C. C. Um Meta-Modelo para Especificação de Arquiteturas de Software em Camadas. Em SBES'01: *XV Brazilian Symposium on Software Engineering* (SBES2001). Rio de Janeiro-RJ, 2001.

[Silva, 2009] Silva, M. A. A.: *CHORD: Constraint Handling Object-Oriented Rules with Disjunctions*. Dissertação de Mestrado em Ciência da Computação. Universidade Federal de Pernambuco, 2009.

[Solingen et al., 1999] Solingen, R.; Berghout, E. *The Goal/Question/Metric Method: A Practical Guide for Quality Improvement of Software Development*. McGraw-Hill Publishers, 1999.

[SPEM, 2008] *Software and Systems Process Engineering Meta-Model Specification*. Object Management Group (OMG). Versão 2.0 – Abril, 2008. http://www.omg.org

[Stahl et al., 2006] Stahl, T.; Voelter, M. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley, 2006.

[Störrle, 2001] Störrle, H. Describing Process Pattern with UML. Em *Proceedings of the 8th European Workshop on Software Process Technology*, LNCS 2077, Springer, páginas 173-181, 2001.

[Sunyé et al., 2000] Sunyé, G.; Guennec, A.; Jézéquel, J-M. Design Pattern Application in UML. Em *Proceedings of the 14th European conference on Object Oriented programming*, LNCS 1850, Springer, páginas 44-62, 2000.

[Tanagra, 2005] Rakotomalala, R. *TANAGRA: Un Logiciel Gratuit Pour L'enseignement et la Recherche*. Em: Actes de EGC'2005, RNTI-E-3, vol. 2, páginas 697-702, 2005.

[Tertius, 2010] *Algoritmo Tertius*, 2010. http://wiki.pentaho.com/display/DATAMINING/Tertius

[Travassos et al., 2002] Travassos, G.H.; Gurov, D.; Amaral, E.A.G.G. *Introdução à Engenharia de Software Experimental*. Em: Relatório Técnico ES-590/02-Abril, Programa de Engenharia de Sistemas e Computação, COPPE/UFRJ, 2002.

[U2TP, 2005] *UML Testing Profile.* Object Management Group (OMG). Versão 1.0 – Julho, 2005. http://www.omg.org/cgi-bin/doc?formal/05-07-07

[UML, 2007] *Unified Modeling Language.* Object Management Group (OMG). Versão 2.1.2 – Novembro, 2007. http://www.uml.org

[Vieira e Ramalho, 2009] Vieira, A.; Ramalho, F. *Ferramenta de Suporte para Aplicação Automática das Diretrizes em Meta-modelos*, 2009. http://code.google.com/p/toolsupport

[Vieira et al., 2009] Vieira, A.; Leal, F; Ramalho, F. *Ferramenta de Suporte para Mineração de Dados em Meta-modelos*, 2009. http://code.google.com/p/toolsupport

[Vieira et al.] Vieira, A.; Ramalho, F.; Machado, P.; Leal, F. *Applying Data Mining Techniques to Semi-Automatically Discover Guidelines for Metamodels*. Em *Proceedings of the 1st Brazilian Workshop on Model-Driven Development*, 2010.

[Warmer et al., 2003] Warmer, J.; Kleppe, A. *The Object Constraint Language*. 2ª Ed. (Getting your models ready for MDA). Object-Technology Series. Addison-Wesley, 2003.

[Weka, 2009] Hall, M.; Frank, E.; Holmes, G.; Pfahringer, B.; Reutemann, P.; Witten, I. H.; *The WEKA Data Mining Software: An Update*. SIGKDD Explorations, Volume 11, Issue 1, 2009.

[XHTML, 2005] Guyard, P. *Meta-modelo de XHTML 1.0*. AtlanMod, 2005. http://www.emn.fr/z-info/atlanmod/index.php/Ecore#XHTML_1.0

[XSchema, 2005] Guyard, P.; Piers, W.; Kurtev, I. *Meta-modelo de XSchema 1.0*. AtlanMod, 2005. http://www.emn.fr/z-info/atlanmod/index.php/Ecore#XSchema_1.0

# Apêndice A

# Tabela Única com Todos os Campos para a Mineração de Dados

Neste apêndice, serão enumerados todos os campos extraídos do meta-modelo de MOF e utilizados na nossa tabela única, o arquivo ARFF, para a mineração dos dados. Vale ressaltar que além dos campos obtidos a partir da aglutinação das tabelas relacionais, foram criados campos adicionais (derivados de outros e até mesmo a partir da união de dois ou mais campos), a fim de obtermos regras mais diversificadas. Estes campos adicionais são apresentados a seguir com um asterisco (*) para diferenciá-los dos demais.

| Campo | Descrição |
|---|---|
| *class.name* | Especifica o nome de uma meta-classe. |
| *class.isAbstract* | Especifica se uma meta-classe é abstrata. |
| *class.isNavigable* | Especifica se uma meta-classe é navegável a partir de outra em um relacionamento. |
| *class.lower* | Especifica o limite mínimo de multiplicidade de uma meta-classe. |
| *class.upper* | Especifica o limite máximo de multiplicidade de uma meta-classe. |
| *class.composition* | Especifica se uma meta-classe: (i) representa "o todo" (*Owner*) de uma associação de composição; (ii) representa "a parte" (*Owned*) desta associação; ou (iii) não possui associação de composição (*None*). |
| *class.opposite.name* | No caso em que as duas meta-classes de um relacionamento são navegáveis, uma a partir da outra, este campo especifica o nome da meta-classe do lado oposto do relacionamento. |
| *class.isReadOnly* | Especifica se uma meta-classe pode ser escrita depois de sua inicialização. |
| * *class.depth* | Especifica o nível de profundidade, em valor inteiro, em que a meta-classe se encontra, considerando a primeira meta-classe da hierarquia como a raiz. |
| * *class.numSuperClasses* | Especifica o número de superclasses de uma meta-classe. |
| * *class.numSubClasses* | Especifica o número de subclasses de uma meta-classe. |
| *class.isNone* | Especifica se uma meta-classe possui alguma associação, que não seja de composição. |
| *class.isOwner* | Especifica se uma meta-classe é o todo de alguma associação de composição. |
| *class.isOwned* | Especifica se uma meta-classe é a parte de alguma associação de composição. |
| * *class.isCompositionOwnerLower1* | Especifica se uma meta-classe é o todo de alguma associação de composição com o limite mínimo de multiplicidade igual a `1`. |
| * *class.isCompositionOwnerLower0* | Especifica se uma meta-classe é o todo de alguma associação de composição com o limite mínimo de multiplicidade igual a `0`. |
| * *class.isCompositionOwnedLower1* | Especifica se uma meta-classe é a parte de alguma associação de composição com o limite mínimo de multiplicidade igual a `1`. |

| | |
|---|---|
| * *class.isCompositionOwnedLower0* | Especifica se uma meta-classe é a parte de alguma associação de composição com o limite mínimo de multiplicidade igual a `0`. |
| * *class.isCompositionNoneLower1* | Especifica se uma meta-classe possui algum relacionamento de associação, que não seja de composição, com o limite mínimo de multiplicidade igual a `1`. |
| * *class.isCompositionNoneLower0* | Especifica se uma meta-classe possui algum relacionamento de associação, que não seja de composição, com o limite mínimo de multiplicidade igual a `0`. |
| * *class.isCompositionOwnerUpper1* | Especifica se uma meta-classe é o todo de alguma associação de composição com o limite máximo de multiplicidade igual a `1`. |
| * *class.isCompositionOwnerUpper2* | Especifica se uma meta-classe é o todo de alguma associação de composição com o limite máximo de multiplicidade igual a `2`. |
| * *class.isCompositionOwnerUpperN* | Especifica se uma meta-classe é o todo de alguma associação de composição com o limite máximo de multiplicidade igual a `*`. |
| * *class.isCompositionOwnedUpper1* | Especifica se uma meta-classe é a parte de alguma associação de composição com o limite máximo de multiplicidade igual a `1`. |
| * *class.isCompositionOwnedUpper2* | Especifica se uma meta-classe é a parte de alguma associação de composição com o limite máximo de multiplicidade igual a `2`. |
| * *class.isCompositionOwnedUpperN* | Especifica se uma meta-classe é a parte de alguma associação de composição com o limite máximo de multiplicidade igual a `*`. |
| * *class.isCompositionNoneUpper1* | Especifica se uma meta-classe possui algum relacionamento de associação, que não seja de composição, com o limite máximo de multiplicidade igual a `1`. |
| * *class.isCompositionNoneUpper2* | Especifica se uma meta-classe possui algum relacionamento de associação, que não seja de composição, com o limite máximo de multiplicidade igual a `2`. |
| * *class.isCompositionNoneUpperN* | Especifica se uma meta-classe possui algum relacionamento de associação, que não seja de composição, com o limite máximo de multiplicidade igual a `*`. |
| *class.superClass.name* | Especifica o nome da superclasse de uma meta-classe. |

| | |
|---|---|
| *class.superClass.isAbstract* | Especifica se a superclasse de uma meta-classe é abstrata. |
| *class.superClass.isNavigable* | Especifica se a superclasse de uma meta-classe é navegável a partir de outra. |
| *class.superClass.composition* | Especifica se a superclasse de uma meta-classe representa: (i) o todo, (ii) a parte de uma associação de composição; ou (iii) se ela não possui tal associação. |
| *class.superClass.lower* | Especifica o limite mínimo de multiplicidade da superclasse de uma meta-classe. |
| *class.superClass.upper* | Especifica o limite máximo de multiplicidade da superclasse de uma meta-classe. |
| *class.superClass.association.name* | Especifica o nome da associação da superclasse de uma meta-classe. |
| * *superClass.isCompositionOwnerLower1* | Especifica se a superclasse de uma meta-classe é o todo de alguma associação de composição com o limite mínimo de multiplicidade igual a `1`. |
| * *superClass.isCompositionOwnerLower0* | Especifica se a superclasse de uma meta-classe é o todo de alguma associação de composição com o limite mínimo de multiplicidade igual a `0`. |
| * *superClass.isCompositionOwnedLower1* | Especifica se a superclasse de uma meta-classe é a parte de alguma associação de composição com o limite mínimo de multiplicidade igual a `1`. |
| * *superClass.isCompositionOwnedLower0* | Especifica se a superclasse de uma meta-classe é a parte de alguma associação de composição com o limite mínimo de multiplicidade igual a `0`. |
| * *superClass.isCompositionNoneLower1* | Especifica se a superclasse de uma meta-classe possui algum relacionamento de associação, que não seja de composição, com o limite mínimo de multiplicidade igual a `1`. |
| * *superClass.isCompositionNoneLower0* | Especifica se a superclasse de uma meta-classe possui algum relacionamento de associação, que não seja de composição, com o limite mínimo de multiplicidade igual a `0`. |
| * *superClass.isCompositionOwnerUpper1* | Especifica se a superclasse de uma meta-classe é o todo de alguma associação de composição com o limite máximo de multiplicidade igual a `1`. |
| * *superClass.isCompositionOwnerUpper2* | Especifica se a superclasse de uma meta-classe é o todo de alguma associação de composição com o limite máximo de multiplicidade igual a `2`. |
| * *superClass.isCompositionOwnerUpperN* | Especifica se a superclasse de uma meta-classe é o todo de alguma associação de composição com o limite máximo de multiplicidade igual a `*`. |

| | |
|---|---|
| * *superClass.isCompositionOwnedUpper1* | Especifica se a superclasse de uma meta-classe é a parte de alguma associação de composição com o limite máximo de multiplicidade igual a `1`. |
| * *superClass.isCompositionOwnedUpper2* | Especifica se a superclasse de uma meta-classe é a parte de alguma associação de composição com o limite máximo de multiplicidade igual a `2`. |
| * *superClass.isCompositionOwnedUpperN* | Especifica se a superclasse de uma meta-classe é a parte de alguma associação de composição com o limite máximo de multiplicidade igual a `*`. |
| * *superClass.isCompositionNoneUpper1* | Especifica se a superclasse de uma meta-classe possui algum relacionamento de associação, que não seja de composição, com o limite máximo de multiplicidade igual a `1`. |
| * *superClass.isCompositionNoneUpper2* | Especifica se a superclasse de uma meta-classe possui algum relacionamento de associação, que não seja de composição, com o limite máximo de multiplicidade igual a `2`. |
| * *superClass.isCompositionNoneUpperN* | Especifica se a superclasse de uma meta-classe possui algum relacionamento de associação, que não seja de composição, com o limite máximo de multiplicidade igual a `*`. |
| *class.subClass.name* | Especifica o nome da subclasse de uma meta-classe. |
| *class.subClass.isAbstract* | Especifica se a subclasse de uma meta-classe é abstrata. |
| *class.subClass.isNavigable* | Especifica se a subclasse de uma meta-classe é navegável a partir de outra. |
| *class.subClass.composition* | Especifica se a subclasse de uma meta-classe representa: (i) o todo, (ii) a parte de uma associação de composição; ou (iii) se ela não possui tal associação. |
| *class.subClass.lower* | Especifica o limite mínimo de multiplicidade da subclasse de uma meta-classe. |
| *class.subClass.upper* | Especifica o limite máximo de multiplicidade da subclasse de uma meta-classe. |
| *class.subClass.association.name* | Especifica o nome da associação da subclasse de uma meta-classe. |
| * *subClass.isCompositionOwnerLower1* | Especifica se a subclasse de uma meta-classe é o todo de alguma associação de composição com o limite mínimo de multiplicidade igual a `1`. |
| * *subClass.isCompositionOwnerLower0* | Especifica se a subclasse de uma meta-classe é o todo de alguma associação de composição com o limite mínimo de multiplicidade igual a `0`. |
| * *subClass.isCompositionOwnedLower1* | Especifica se a subclasse de uma meta-classe é a parte de alguma associação de composição |

| | |
|---|---|
| | com o limite mínimo de multiplicidade igual a `1`. |
| * *subClass.isCompositionOwnedLower0* | Especifica se a subclasse de uma meta-classe é a parte de alguma associação de composição com o limite mínimo de multiplicidade igual a `0`. |
| * *subClass.isCompositionNoneLower1* | Especifica se a subclasse de uma meta-classe possui algum relacionamento de associação, que não seja de composição, com o limite mínimo de multiplicidade igual a `1`. |
| * *subClass.isCompositionNoneLower0* | Especifica se a subclasse de uma meta-classe possui algum relacionamento de associação, que não seja de composição, com o limite mínimo de multiplicidade igual a `0`. |
| * *subClass.isCompositionOwnerUpper1* | Especifica se a subclasse de uma meta-classe é o todo de alguma associação de composição com o limite máximo de multiplicidade igual a `1`. |
| * *subClass.isCompositionOwnerUpper2* | Especifica se a subclasse de uma meta-classe é o todo de alguma associação de composição com o limite máximo de multiplicidade igual a `2`. |
| * *subClass.isCompositionOwnerUpperN* | Especifica se a subclasse de uma meta-classe é o todo de alguma associação de composição com o limite máximo de multiplicidade igual a `*`. |
| * *subClass.isCompositionOwnedUpper1* | Especifica se a subclasse de uma meta-classe é a parte de alguma associação de composição com o limite máximo de multiplicidade igual a `1`. |
| * *subClass.isCompositionOwnedUpper2* | Especifica se a subclasse de uma meta-classe é a parte de alguma associação de composição com o limite máximo de multiplicidade igual a `2`. |
| * *subClass.isCompositionOwnedUpperN* | Especifica se a subclasse de uma meta-classe é a parte de alguma associação de composição com o limite máximo de multiplicidade igual a `*`. |
| * *subClass.isCompositionNoneUpper1* | Especifica se a subclasse de uma meta-classe possui algum relacionamento de associação, que não seja de composição, com o limite máximo de multiplicidade igual a `1`. |
| * *subClass.isCompositionNoneUpper2* | Especifica se a subclasse de uma meta-classe possui algum relacionamento de associação, que não seja de composição, com o limite máximo de multiplicidade igual a `2`. |
| * *subClass.isCompositionNoneUpperN* | Especifica se a subclasse de uma meta-classe possui algum relacionamento de associação, que não seja de composição, com o limite máximo de multiplicidade igual a `*`. |

| | |
|---|---|
| *class.package* | Especifica o nome do pacote que contém determinada meta-classe. |
| *class.package.packageImport.importedPackage* | Especifica os pacotes importados pelo pacote que possui determinada meta-classe. |
| *class.package.packageMerge.mergedPackage* | Especifica os pacotes que fazem *merge* com o pacote que possui determinada meta-classe. |
| *class.property.name* | Especifica o nome do atributo de uma meta-classe. |
| *class.property.isDerivedUnion* | Especifica se uma propriedade é derivada da união de todas as propriedades que estão limitadas a um subconjunto. |
| *class.property.isDerived* | Especifica se o valor do atributo de uma meta-classe é derivado de outra informação. |
| *class.property.isOrdered* | Para uma multiplicidade multivalorada, este campo especifica se os valores do atributo de uma meta-classe estão seqüencialmente ordenados. |
| *class.property.type* | Especifica o tipo do atributo de uma meta-classe. |
| *class.property.default* | Especifica um *String* que representa o valor que será usado quando nenhum outro for informado para o atributo de uma meta-classe. |
| *class.association.name* | Especifica o nome da associação de uma meta-classe. |
| *class.association.isDerived* | Especifica se a associação de uma meta-classe é derivada de outros elementos do modelo, tais como outras associações ou restrições. |
| *class.association.isComposition* | Especifica se a associação de uma meta-classe é de composição. |
| *class.rule.name* | Especifica o nome da restrição para uma meta-classe. |
| *class.rule.opaqueExpression.body* | Especifica o corpo da expressão de uma restrição para uma meta-classe. |
| *class.rule.constraintKind* | Especifica o tipo da restrição para uma meta-classe. |
| *class.operation.isQuery* | Especifica se uma operação é de consulta. |
| *class.operation.name* | Especifica o nome da operação de uma meta-classe. |
| *class.operation.rule.name* | Especifica o nome da restrição para uma operação. |
| *class.operation.rule.constraintKind* | Especifica o tipo da restrição para uma operação, que pode ser: *bodyCondition*, *postCondition*, ou *preCondition*. |
| *class.operation.rule.opaqueExpression.body* | Especifica o corpo da expressão de uma restrição para uma operação da meta-classe. |

| | |
|---|---|
| *class.operation.parameter.name* | Especifica o nome do parâmetro de uma operação. |
| *class.operation.parameters.direction* | Especifica a direção dos parâmetros de uma operação, que pode ser: (i) *in* – de entrada; (ii) *inout* – de entrada/saída; (iii) *out* – de saída; ou (iv) *return* – de retorno. |
| *class.operation.parameters.type* | Especifica o tipo dos parâmetros de uma operação. |

# Apêndice B

# Regras de Associação Úteis para Validação de Meta-modelos

Neste apêndice, serão apresentadas todas as regras de associação que não apresentaram nenhum conhecimento inovador, as quais foram geradas a partir da execução do algoritmo *Apriori* na nossa base de dados. Tais regras são úteis para a validação de meta-modelos, uma vez que são totalmente confiáveis, isto é, possuem valor máximo para a métrica da confiança. Vale ressaltar que, como o valor de confiança é 100% em todas estas regras, mostraremos apenas o valor de suporte para cada uma. A seguir, vejamos cada uma das regras.

- Com 71% de suporte, a regra R1 especifica que operações definidas por meio de uma restrição OCL `body` são operações de consulta.

  *[R1] class.operation.rule.constraintKind=body 40378* → *class.operation.isQuery=true 40378*

- Com 61% de suporte, a regra R2 especifica que operações com parâmetros de retorno são operações de consulta.

  *[R2] class.operation.parameters.direction=return 34510* → *class.operation.isQuery=true 34510*

- Com 53% de suporte, a regra R3 especifica que operações definidas por meio de uma restrição OCL `body` e que possuem parâmetros de retorno são operações de consulta.

*[R3] class.operation.rule.constraintKind=body*
*class.operation.parameters.direction=return 30015 →*
*class.operation.isQuery=true 30015*

- Com 19% de suporte, a regra R4 especifica que uma meta-classe que representa a parte de uma associação de composição sempre deve ser navegável.

*[R4] class.composition=owned 11046 →*
*class.isNavigable=true 11046*

- Com 17% de suporte, a regra R5 especifica que uma meta-classe concreta que representa a parte de uma associação de composição deve ser sempre navegável.

*[R5] class.isAbstract=false class.composition=owned 9570 →*
*class.isNavigable=true 9570*

- Com 11% de suporte, a regra R6 especifica que uma meta-classe que representa o todo de uma associação de composição deve possuir o limite máximo de multiplicidade igual a `1`.

*[R6] class.composition=owner 6160 →*
*class.upper=1 6160*

- Com 35% de suporte, a regra R7 especifica que operações com parâmetros do tipo *Boolean* são operações com parâmetros de retorno.

*[R7] class.operation.parameters.type=Boolean 19716 →*
*class.operation.parameters.direction=return 19716*

- Com 54% de suporte, a regra R8 especifica que uma meta-classe que pode ser escrita depois de inicializada não possui associações derivadas.

*[R8] class.isReadOnly=false 31042 →*
*class.association.isDerived=false 31042*

- Com 48% de suporte, a regra R9 especifica que uma meta-classe que não possui atributos derivados pode ser escrita depois de inicializada.

*[R9] class.property.isDerived=false 27373 →*
*class.isReadOnly=false 27373*

- Com 48% de suporte, a regra R10 especifica que uma meta-classe que não possui atributos derivados e que também não possui associações derivadas pode ser escrita depois de inicializada.

*[R10] class.property.isDerived=false*
*class.association.isDerived=false 27373 →*
*class.isReadOnly=false 27373*

- Com 33% de suporte, a regra R11 especifica que operações definidas por meio de uma restrição OCL `body` e com parâmetros do tipo *Boolean* são operações com parâmetros de retorno.

*[R11] class.operation.rule.constraintKind=body*
*class.operation.parameters.type=Boolean 18568 →*
*class.operation.parameters.direction=return 18568*

- Com 31% de suporte, a regra R12 especifica que operações com parâmetros do tipo *Boolean* de uma meta-classe concreta são operações com parâmetros de retorno.

*[R12] class.isAbstract=false*
*class.operation.parameters.type=Boolean 17782 →*
*class.operation.parameters.direction=return 17782*

- Com 29% de suporte, a regra R13 especifica que operações com parâmetros de retorno definidas por meio de uma restrição OCL `body` de uma meta-classe concreta são operações de consulta.

*[R13] class.isAbstract=false*
*class.operation.rule.constraintKind=body*
*class.operation.parameters.direction=return 16579 →*
*class.operation.isQuery=true 16579*

- Com 31% de suporte, a regra R14 especifica que operações com parâmetros de retorno de uma meta-classe concreta são operações de consulta.

*[R14] class.isAbstract=false*
*class.operation.parameters.direction=return 17949* →
*class.operation.isQuery=true 17949*

- Com 38% de suporte, a regra R15 especifica que operações definidas por meio de uma restrição OCL `body` de uma meta-classe concreta são operações de consulta.

*[R15] class.isAbstract=false*
*class.operation.rule.constraintKind=body 21548* →
*class.operation.isQuery=true 21548*

# Apêndice C

# Outros Meta-modelos Analisados na Avaliação Experimental

Neste apêndice, será apresentada a avaliação experimental do nosso trabalho considerando os meta-modelos de Ant, OCL e SPEM. Cada meta-modelo será apresentado, juntamente com um quantitativo dos seus elementos e, em seguida, será ilustrada a aplicação de todas as diretrizes que são aplicáveis nele.

## C.1 Meta-modelo de Ant

Nesta seção, é avaliada a aplicabilidade das diretrizes no meta-modelo de Ant. Trata-se de uma ferramenta escrita em Java para automatizar a construção (*build*) de *software*, permitindo operações como atualização do *classpath*, geração do *javadoc* do projeto, configuração e execução da aplicação [Ant Project, 2010]. Este meta-modelo foi retirado de um repositório de meta-modelos disponibilizado por um grupo de pesquisas chamado AtlanMod. Na seqüência, a Tabela 13 apresenta alguns elementos que quantificamos no meta-modelo de Ant, os quais foram alvo para a aplicação das diretrizes. Todos estes elementos estão presentes em um único pacote, chamado `Ant`.

**Tabela 13: Quantitativo de elementos do meta-modelo de Ant.**

| Elemento | Quantidade |
|---|---|
| Meta-classes | 48 |
| Atributos | 93 |
| *Enumerations* | 0 |
| Associações | 27 |

A seguir, a Tabela 14 relaciona cada diretriz com o número de vezes que ela pode ser aplicada no meta-modelo de Ant. Ao final desta tabela, é informado o número total de aplicações de todas as diretrizes neste meta-modelo, bem como o número de diretrizes que foram aplicadas.

**Tabela 14: Quantitativo de aplicações das diretrizes no meta-modelo de Ant.**

| Diretriz | Núm. Aplicações |
|---|---|
| D1 | 5 |
| D2 | 4 |
| D5 | 1 |
| D11 | 27 |
| *Número de Aplicações Total* | 37 |
| *Número de Diretrizes Aplicadas*: 4 | |

Como podemos observar, a diretriz D6 não foi aplicada, pois o meta-modelo de Ant já possui um pacote chamado `PrimitiveTypes`, o qual contém todos os tipos primitivos do projeto. Da mesma forma, a diretriz D7 não foi aplicada, uma vez que este meta-modelo já possui um pacote principal chamado `Ant`. Por outro lado, é importante destacar que o meta-modelo de Ant não possui valor *default* para nenhum dos seus atributos. Contudo, como todos eles são do tipo *String*, não havendo nenhum *boolean* ou *enumeration*, a definição de valores *default* não foi necessária. Portanto, as diretrizes D9 e D10 também não precisaram ser aplicadas. As demais diretrizes – D3, D4, D8, D12 e D13 – não foram aplicadas devido ao fato de o meta-modelo de Ant já se apresentar em conformidade com todas elas.

### C.1.1 D1 - *Abstracting Common Attributes* (Ant)

A diretriz D1 possui seis aplicações em todo o meta-modelo de Ant. A aplicação que será ilustrada a seguir envolve seis meta-classes: `TaskDef`, *`PropertyName`*, `Attribut`, *`InExcludes`*, `Target` e `Project`. Algumas delas são apresentadas na Figura 76 e possuem uma característica em comum: o atributo `name`.

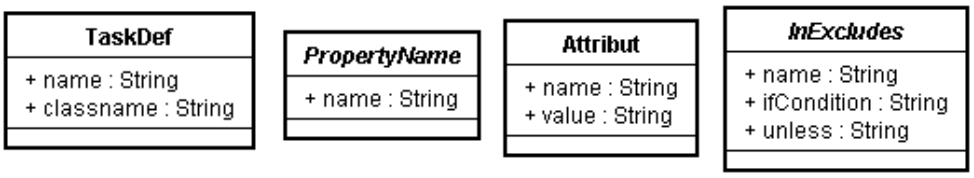



**Figura 76: Meta-classes do meta-modelo de Ant com um atributo em comum.**

Portanto, para aplicarmos a diretriz D1 criamos uma meta-classe abstrata (no pacote `Abstractions`) chamada *`NamedElement`* com o atributo `name`. Então, as seis meta-classes que possuem o atributo `name` em comum estendem *`NamedElement`* para ter acesso ao atributo, como podemos ver na Figura 77.

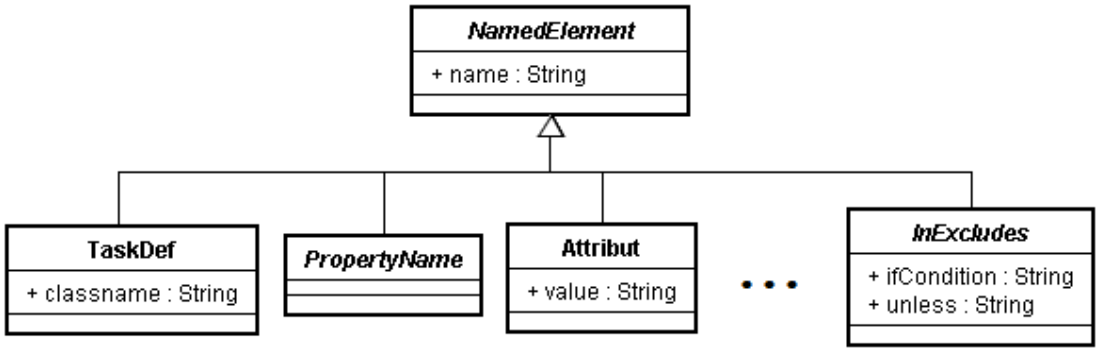



**Figura 77: Meta-modelo de Ant com a aplicação da diretriz D1.**

Outras aplicações da diretriz D1 no meta-modelo de Ant podem ser conferidas na subseção D.1.1 do Apêndice D.

### C.1.2 D2 - *Abstracting Common Associations* (Ant)

Esta diretriz possui quatro aplicações no meta-modelo de Ant. Como exemplo de sua aplicação vejamos a Figura 78. Ela apresenta um trecho do meta-modelo em que as meta-classes `Java` e `Javac` possuem uma associação de composição com a meta-classe `ClassPath`, ambas com a mesma multiplicidade e o mesmo papel.

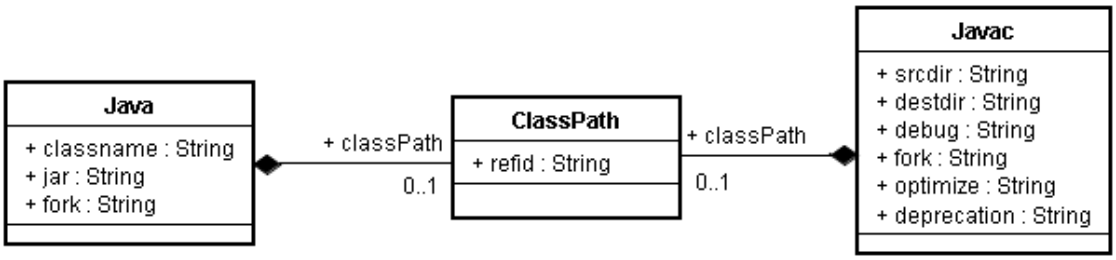



**Figura 78: Trecho do meta-modelo de Ant com duas associações em comum.**

Podemos verificar a aplicação da diretriz D2 na Figura 79. Criamos uma meta-classe abstrata (no pacote `Abstractions`) chamada *`OwnerClassPath`* para generalizar as meta-classes que especificam a associação com `ClassPath`. O nome desta meta-classe é apenas uma convenção que adotamos, o desenvolvedor pode defini-lo de acordo com a semântica do meta-modelo. Logo, as meta-classes `Java` e `Javac` estendem a meta-classe criada, assim, herdando a associação.

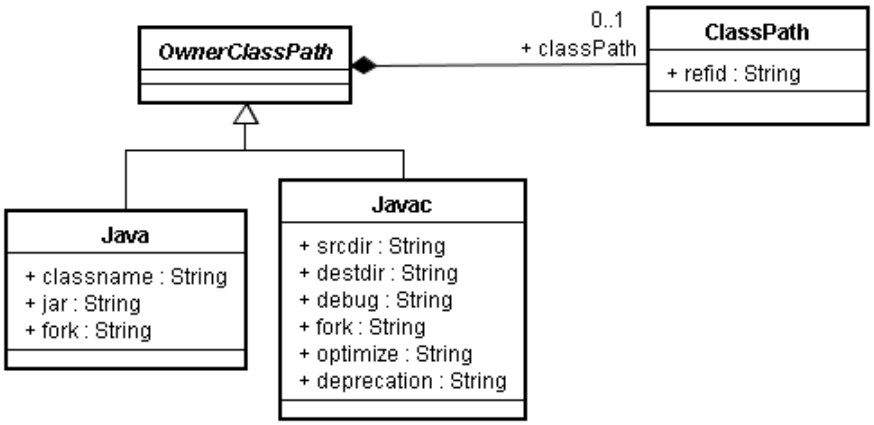



**Figura 79: Meta-modelo de Ant com a aplicação da diretriz D2.**

Outra aplicação da diretriz D2 no meta-modelo de Ant pode ser conferida na subseção D.2.2 do Apêndice D.

### C.1.3 D5 - *Adding Abstractions Package* (Ant)

O meta-modelo de Ant não possui nenhum pacote que contenha todas as meta-classes reutilizáveis do projeto. Portanto, ao aplicarmos esta diretriz definimos um pacote chamado `Abstractions`, como ilustrado na Figura 80.

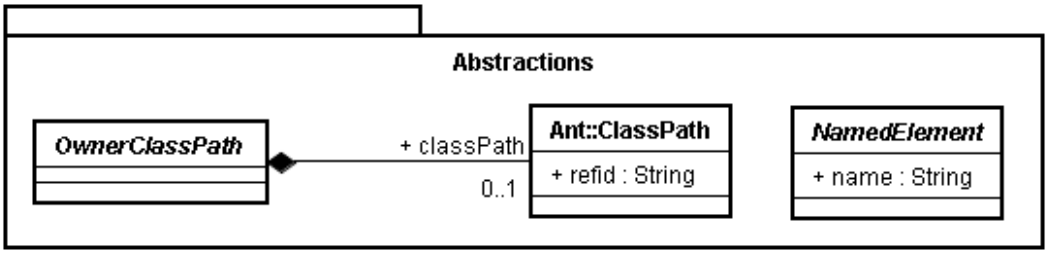



**Figura 80: Pacote Abstractions para o meta-modelo de Ant.**

O pacote `Abstractions` contém as meta-classes que podem ser reutilizadas em todo o meta-modelo de Ant, como por exemplo, *`NamedElement`* e *`OwnerClassPath`*, as quais foram oriundas da aplicação das diretrizes D1 e D2, respectivamente.

### C.1.4 D11 - *Defining Association Member Ends Features* (Ant)

Esta diretriz pode ser aplicada em todas as associações de composição presentes no meta-modelo de Ant. Contudo, vale observar que antes de aplicarmos a diretriz D11 nestas associações, devemos analisar a semântica em cada caso para, de fato, sabermos se a diretriz é ou não aplicável. Temos que verificar se o relacionamento realmente exige que a parte não pertença a mais de um todo ao mesmo tempo. Como exemplo, vejamos a sua aplicação nas associações de composição entre as meta-classes `Path-FileSet` e `ClassPath-FileSet`, ilustradas na Figura 81.

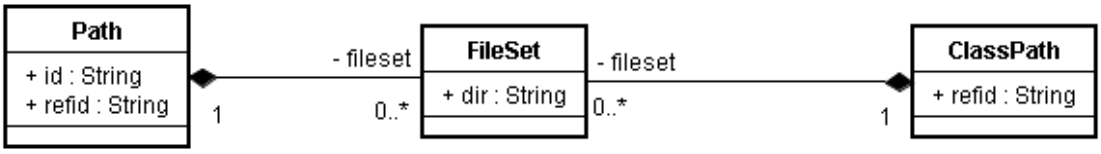



**Figura 81: Trecho do meta-modelo de Ant com duas associações de composição.**

Como podemos observar, algumas características destas duas associações de composição já estão em conformidade com a diretriz D11: (i) a parte (`FileSet`) é uma meta-classe concreta, navegável a partir das duas meta-classes que representam o todo (`Path` e `ClassPath`) e possui o limite mínimo de multiplicidade igual a `0` nas duas associações de composição; (ii) tanto `FileSet` quanto `ClassPath` e `Path` possuem atributos que podem ser escritos depois de inicializados (eles não são *readOnly*); e (iii) `ClassPath` e `Path` não possuem atributos derivados. Por outro lado, a multiplicidade das duas meta-classes que representam o todo foi definida como `1`. Isto significa dizer que um mesmo `FileSet` está

presente tanto em um `Path` quanto em um `ClassPath` ao mesmo tempo. Para deixarmos as duas associações em conformidade com a diretriz D11, apenas definimos tais multiplicidades como sendo `0..1`. Podemos conferir o resultado na Figura 82.

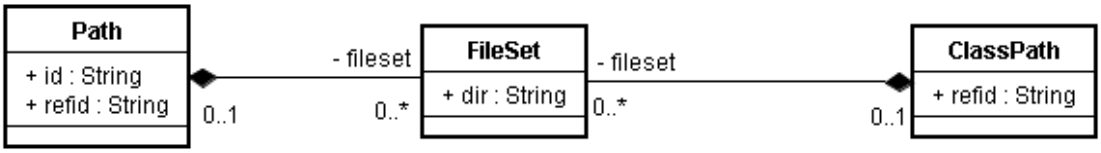



**Figura 82: Meta-modelo de Ant com a aplicação da diretriz D11.**

Outras aplicações da diretriz D11 no meta-modelo de Ant podem ser conferidas na subseção D.5.2 do Apêndice D.

## C.2 Meta-modelo de OCL

Nesta seção, é avaliada a aplicabilidade das diretrizes no meta-modelo de OCL, que é um padrão definido pela OMG. Em seguida, a Tabela 15 apresenta alguns elementos que quantificamos neste meta-modelo, os quais foram alvo para a aplicação das diretrizes. A contagem destes elementos está agrupada de acordo com os dois pacotes que o meta-modelo de OCL possui: `Types` e `Expressions`. Ao final da tabela, é informado o somatório de cada elemento em todos os pacotes.

**Tabela 15: Quantitativo de elementos do meta-modelo de OCL.**

| **Pacote / Elemento** | *Meta-classes* | *Atributos* | *Enumerations* | *Associações* |
|---|---|---|---|---|
| *Types* | 12 | 0 | 0 | 3 |
| *Expressions* | 43 | 6 | 1 | 30 |
| *Total* | 55 | 6 | 1 | 33 |

A seguir, a Tabela 16 relaciona cada diretriz com o número de vezes que ela pode ser aplicada no meta-modelo de OCL. Ao final desta tabela, é informado o número total de aplicações de todas as diretrizes neste meta-modelo, bem como o número de diretrizes que foram aplicadas.

**Tabela 16: Quantitativo de aplicações das diretrizes no meta-modelo de OCL.**

| **Diretriz** | **Núm. Aplicações** |
|---|---|
| D9 | 1 |
| D10 | 1 |
| D11 | 1 |
| *Número de Aplicações Total* | 3 |
| *Número de Diretrizes Aplicadas:* 3 | |

Como podemos observar, a diretriz D5 não foi aplicada no meta-modelo de OCL, pois, mesmo não existindo um pacote `Abstractions` neste meta-modelo, não houve a necessidade de criá-lo, uma vez que nenhuma meta-classe reutilizável foi criada para o propósito deste pacote. A diretriz D6 também não foi aplicada, pois, embora não possua um pacote `PrimitiveTypes`, o meta-modelo de OCL já possui um pacote que contém todos os tipos primitivos do projeto, chamado `Types`. Da mesma forma ocorre com a diretriz D7, pois, apesar de não possuir um pacote chamado `Core`, o meta-modelo de OCL já possui um pacote principal chamado `Ocl-AbstractSyntax`.

O meta-modelo de OCL possui apenas um atributo do tipo *boolean*, chamado `booleanSymbol`, o qual não se encontra de acordo com a diretriz D12. Como podemos notar, ele não se inicia com o prefixo *is* ou *has*. Entretanto, neste caso específico não é preciso aplicar a diretriz, uma vez que o nome do atributo já condiz exatamente com o seu significado: ele informa se o símbolo booleano é *true* ou *false*. As demais diretrizes – D1, D2, D3, D4, D8 e D13 – não foram aplicadas devido ao fato de o meta-modelo de OCL já se apresentar em conformidade com todas elas.

### C.2.1 D9 - *Defining Boolean Attributes Default Value* (OCL)

O meta-modelo de OCL não possui nenhum valor *default* para o único atributo do tipo *boolean* que ele possui. Portanto, ao aplicarmos esta diretriz devemos escolher um valor *default* para o atributo apresentado na Figura 83.

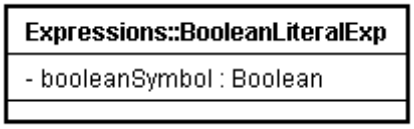



**Figura 83: Atributo booelano do meta-modelo de OCL sem valor *default*.**

### C.2.2 D10 - *Defining Enum Default Value* (OCL)

O único *enumeration* que existe no meta-modelo de OCL não especifica nenhum valor *default*. Portanto, a Figura 84 ilustra este *enumeration* para o qual um valor *default* deve ser especificado.

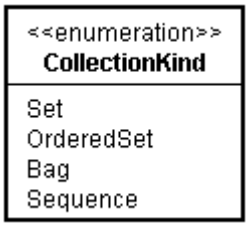



**Figura 84: *Enumeration* do meta-modelo de OCL sem valor *default*.**

### C.2.3 D11 - *Defining Association Member Ends Features Value* (OCL)

A diretriz D11 pode ser aplicada em apenas uma associação de composição do meta-modelo de OCL. Vejamos esta aplicação na associação entre as meta-classes `CollectionLiteralExp` e *`CollectionLiteralPart`*, ilustrada na Figura 85.

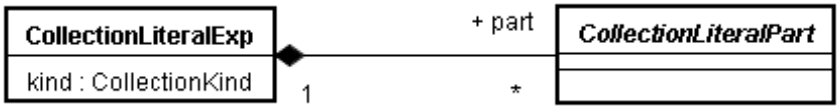



**Figura 85: Trecho do meta-modelo de OCL com uma associação de composição.**

Como podemos observar, a meta-classe parte da associação de composição não é concreta, como sugere a diretriz. Além disso, a multiplicidade do todo desta associação é `1`. Embora um *`CollectionLiteralPart`* pertença apenas a uma meta-classe em todo o meta-modelo, a `CollectionLiteralExp`, a aplicação desta diretriz é válida para garantirmos uma melhor evolução do meta-modelo. Portanto, para aplicá-la definimos a multiplicidade do todo para `0..1` e definimos a meta-classe que representa a parte da associação como sendo concreta. Vejamos o resultado na Figura 86.

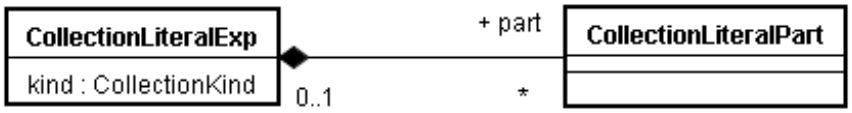



**Figura 86: Meta-modelo de OCL com a aplicação da diretriz D11.**

## C.3 Meta-modelo de SPEM

Nesta seção, é avaliada a aplicabilidade das diretrizes no meta-modelo de SPEM. Este, por sua vez, é um padrão da OMG "usado para definir processos de desenvolvimento de sistemas e *softwares* e seus componentes" [SPEM, 2008]. Com o escopo em desenvolvimento de

projetos, SPEM tem a finalidade de tratar um grande número de processos e métodos de desenvolvimento de diferentes estilos, níveis de formalismo, modelos de ciclo de vida e comunidades. Na seqüência, a Tabela 17 apresenta alguns elementos que quantificamos no meta-modelo de SPEM, os quais foram alvo para a aplicação das diretrizes. A contagem destes elementos está agrupada de acordo com os sete pacotes que o meta-modelo de SPEM possui: `Core`, `ProcessStructure`, `ProcessBehavior`, `ManagedContent`, `MethodContent`, `ProcessWithMethods` e `MethodPlugin`. Ao final da tabela, é informado o somatório de cada elemento em todos os pacotes.

**Tabela 17: Quantitativo de elementos do meta-modelo de SPEM.**

| **Pacote / Elemento** | *Meta-classes* | *Atributos* | *Enumerations* | *Associações* |
|---|---|---|---|---|
| *Core* | 5 | 1 | 1 | 6 |
| *ProcessStructure* | 12 | 7 | 2 | 19 |
| *ProcessBehavior* | 6 | 0 | 0 | 12 |
| *ManagedContent* | 6 | 6 | 0 | 9 |
| *MethodContent* | 11 | 1 | 1 | 13 |
| *ProcessWithMethods* | 16 | 6 | 0 | 28 |
| *MethodPlugin* | 12 | 4 | 1 | 23 |
| *Total* | 68 | 25 | 5 | 110 |

A seguir, a Tabela 18 relaciona cada diretriz com o número de vezes que ela pode ser aplicada no meta-modelo de SPEM. Ao final desta tabela, é informado o número total de aplicações de todas as diretrizes neste meta-modelo, bem como o número de diretrizes que foram aplicadas.

**Tabela 18: Quantitativo de aplicações das diretrizes no meta-modelo de SPEM.**

| **Diretriz** | **Núm. Aplicações** |
|---|---|
| D2 | 2 |
| D5 | 1 |
| D9 | 2 |
| D10 | 2 |
| D11 | 14 |
| D12 | 1 |
| D13 | 1 |
| *Número de Aplicações Total* | 23 |
| *Número de Diretrizes Aplicadas*: 7 | |

Como podemos observar, a diretriz D6 não foi aplicada, pois o meta-modelo de SPEM reusa o meta-modelo da UML2 *Infrastructure*, o qual possui um pacote chamado

`PrimitiveTypes` contendo todos os tipos primitivos do meta-modelo. Da mesma forma ocorre com a diretriz D7, pois o meta-modelo de SPEM já possui um pacote chamado `Core`, o qual contém as meta-classes que constroem a base para as meta-classes de outros pacotes do meta-modelo. As demais diretrizes – D1, D3, D4 e D8 – não foram aplicadas devido ao fato de o meta-modelo de SPEM já se apresentar em conformidade com todas elas.

É importante destacar que, mesmo se tratando de um processo e não de uma linguagem, as diretrizes foram bastante úteis no meta-modelo de SPEM. Como podemos observar na Tabela 18, um total de sete diretrizes foram aplicadas.

### C.3.1 D2 - *Abstracting Common Associations* (SPEM)

A diretriz D2 pode ter duas aplicações no meta-modelo de SPEM. Vejamos um exemplo na Figura 87, a qual apresenta um trecho deste meta-modelo com associações envolvendo as seguintes meta-classes: `ProcessPerformer` e `ProcessResponsibilityAssignment` (ambas do pacote `ProcessStructure`), `DefaultTaskDefinitionPerformer` (do pacote `MethodContent`) e `RoleUse` (do pacote `ProcessWithMethods`). Como podemos perceber, as três associações possuem a mesma multiplicidade e o mesmo papel.

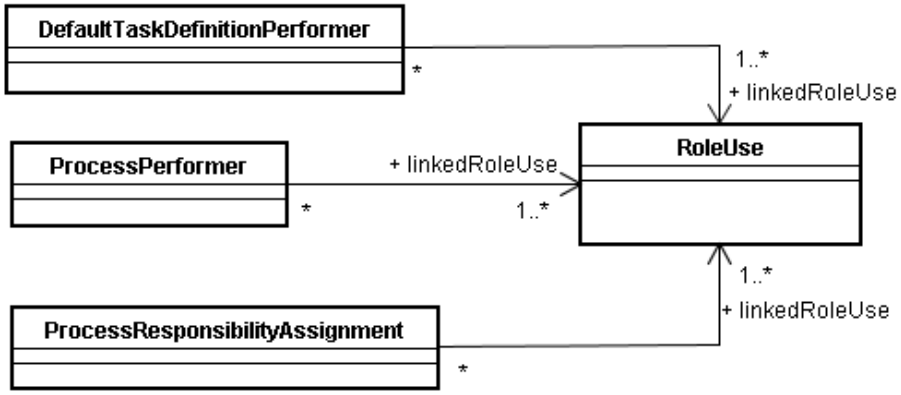



**Figura 87: Trecho do meta-modelo de SPEM com associações em comum.**

A aplicação da diretriz pode ser vista na Figura 88. Como podemos observar, foi necessário criarmos uma meta-classe abstrata (no pacote `Abstractions`), a qual chamamos de *`OwnerLinkedRoleUse`*, para generalizar as meta-classes que especificam a associação com `RoleUse`. O nome da meta-classe abstrata é apenas uma convenção que adotamos, o desenvolvedor pode defini-lo de acordo com a semântica do meta-modelo. Logo,

as meta-classes chamadas `DefaultTaskDefinitionPerformer`, `ProcessPerformer` e `ProcessResponsibilityAssignment` estendem a nova meta-classe criada, assim, herdando a associação.

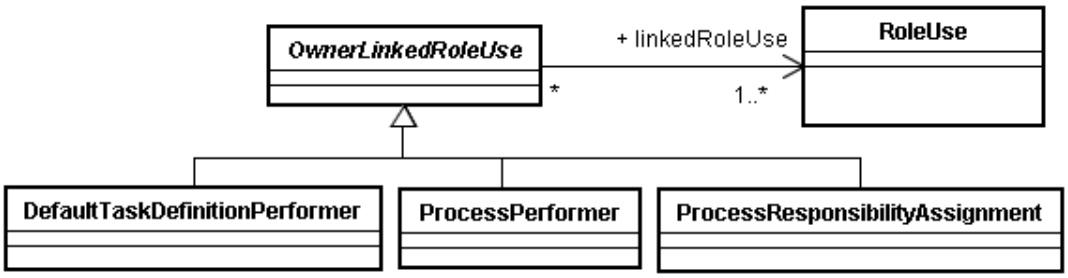



**Figura 88: Trecho de SPEM com a aplicação da diretriz D2.**

A outra aplicação da diretriz D2 no meta-modelo de SPEM pode ser conferida na subseção D.2.5 do Apêndice D.

### C.3.2 D5 - *Adding Abstractions Package* (SPEM)

O meta-modelo de SPEM não possui nenhum pacote que melhor organize todas as meta-classes reutilizáveis do projeto. Portanto, ao aplicarmos esta diretriz temos um pacote chamado `Abstractions`, como ilustrado na Figura 89.

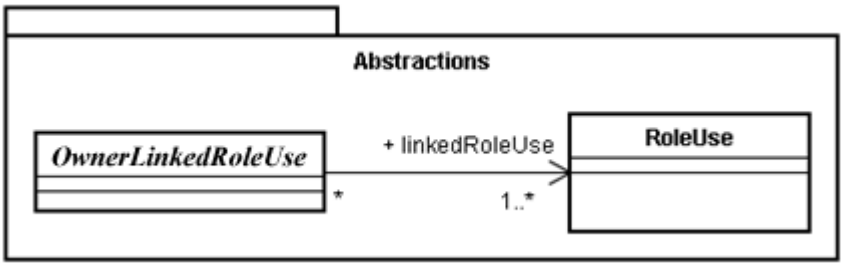



**Figura 89: Pacote Abstractions para o meta-modelo de SPEM.**

Como podemos ver na Figura 89, inicialmente, o pacote `Abstractions` possui apenas a meta-classe *`OwnerLinkedRoleUse`*. Contudo, este pacote vai crescendo ao passo que as diretrizes D1 e D2 são aplicadas no meta-modelo, criando meta-classes reutilizáveis para todo o projeto.

### C.3.3 D9 - *Defining Boolean Attributes Default Value* (SPEM)

Apenas dois dos atributos booleanos presentes no meta-modelo de SPEM não possuem valor *default* especificado. Na Figura 90, vejamos quais são estes atributos para os quais devemos definir valores *default* ao aplicarmos a diretriz D9.

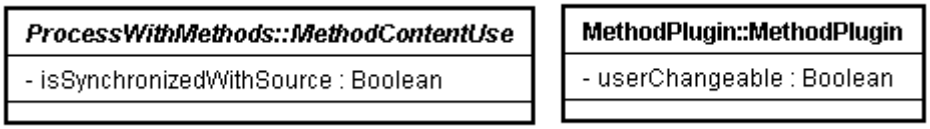



**Figura 90: Atributos do meta-modelo de SPEM sem valores *default*.**

### C.3.4 D10 - *Defining Enum Default Value* (SPEM)

No meta-modelo de SPEM existem dois *enumerations* que não especificam valores *default*. A Figura 91 ilustra estes dois *enumerations* que, ao aplicarmos a diretriz D10, precisam definir valores *default*.

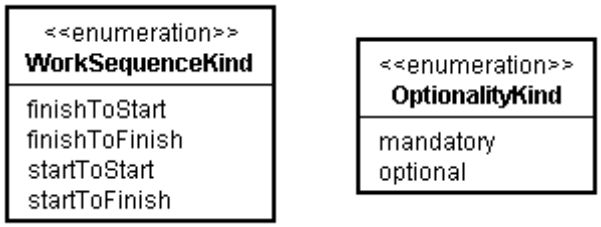



**Figura 91: *Enumerations* do meta-modelo de SPEM sem valores *default*.**

### C.3.5 D11 - *Defining Association Member Ends Features* (SPEM)

Esta diretriz pode ser aplicada em 14 associações de composição do meta-modelo de SPEM. Contudo, vale observar que antes de aplicarmos esta diretriz nestas associações, devemos analisar a semântica em cada caso para, de fato, sabermos se a diretriz é ou não aplicável. Temos que verificar se a associação realmente exige que a parte não pertença a mais de um todo ao mesmo tempo. Como exemplo, vejamos a sua aplicação nas associações de composição entre as meta-classes `Activity-ProcessParameter` e `TaskUse-ProcessParameter`, ilustradas na Figura 92:

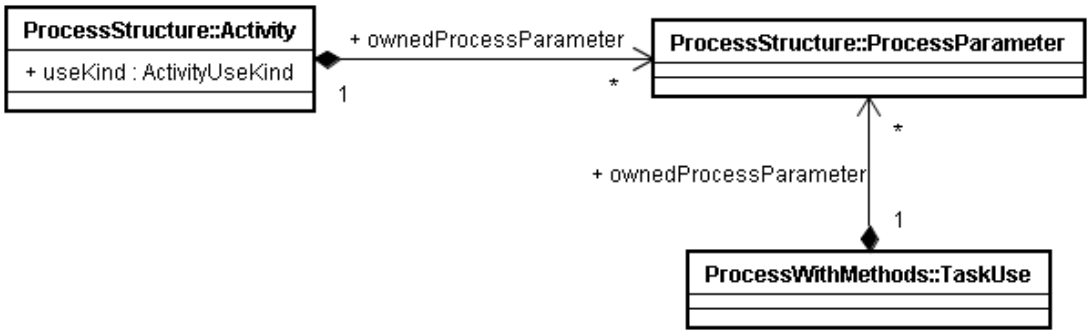



**Figura 92: Trecho do meta-modelo de SPEM com duas associações de composição.**

Como podemos observar, as multiplicidades das meta-classes que representam o todo destas associações foram definidas como `1`. Isto significa dizer que um mesmo `ProcessParameter` está presente tanto em um `Activity` quanto em um `TaskUse` ao mesmo tempo. Para deixarmos as associações em conformidade com a diretriz D11, definimos estas multiplicidades para `0..1`, como mostra a Figura 93.

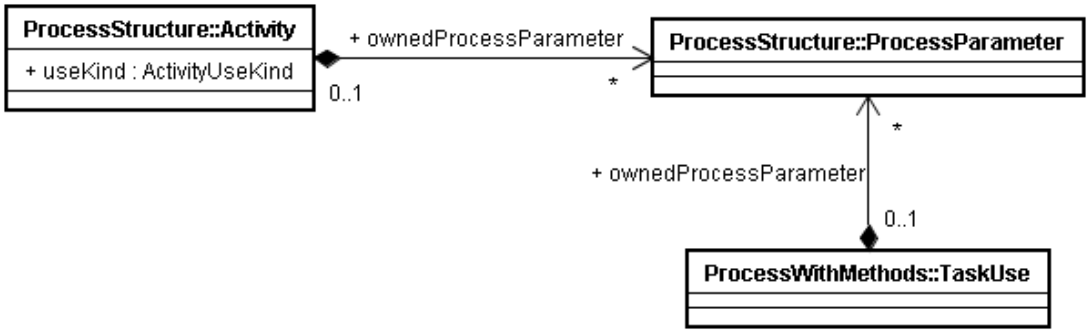



**Figura 93: Meta-modelo de SPEM com a aplicação da diretriz D11.**

Outras aplicações da diretriz D11 no meta-modelo de SPEM podem ser conferidas na subseção D.5.5 do Apêndice D.

### C.3.6 D12 - *Redefining Boolean Attribute Names* (SPEM)

De todos os atributos booleanos que o meta-modelo de SPEM possui, apenas um não está em conformidade com a diretriz D12. Este atributo pertence à meta-classe `MethodPlugin` do pacote `MethodPlugin`, como ilustra a Figura 94. Após a aplicação da diretriz D12, o nome do atributo passa a ser `isUserChangeable` para a meta-classe `MethodPlugin`.

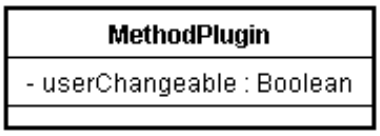



**Figura 94: Meta-classe do meta-modelo de SPEM com um atributo.**

### C.3.7 D13 - *Redefining Enum Names* (SPEM)

O meta-modelo de SPEM possui apenas um *enumeration* que não está em conformidade com a diretriz D13, chamado `VariabilityType`, como ilustrado na Figura 95. Aplicando esta diretriz, o nome do *enumeration* `VariabilityType` passa a ser `VariabilityTypeKind`.

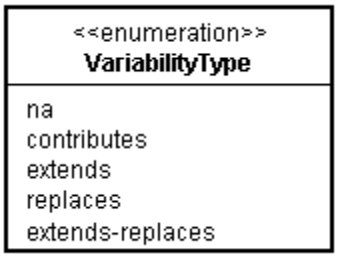



**Figura 95: *Enumeration* do meta-modelo de SPEM.**

# Apêndice D

# Exemplos de Aplicações das Diretrizes

Neste apêndice, serão citados diversos outros exemplos de aplicações das diretrizes nos meta-modelos participantes da avaliação: KobrA2, Ant, Java, OCL, QVT e SPEM. Ele está organizado de forma a apresentar cada diretriz seguida dos meta-modelos para os quais ela se aplica.

## D.1 *Abstracting Common Attributes* - D1

### D.1.1 Meta-modelo de Ant

A diretriz D1 pode ser aplicada em diversas meta-classes do meta-modelo de Ant, as quais possuem atributos com características em comum. A seguir, vejamos algumas destas meta-classes e seus atributos:

(i) `Target`, `Project` e *`PreDefinedTask`*, as quais são descritíveis, ou seja, possuem um atributo `description` do tipo *String*;

(ii) `FileList`, `FileSet`, `Mkdir`, `Exec` e `Delete`, as quais definem um atributo `dir` do tipo *String*;

(iii) `Echo`, `Delete`, `Copy`, `PropertyFile` e `FiltersFile`, as quais possuem um mesmo atributo do tipo *String* em comum, chamado `file`.

## D.2 *Abstracting Common Associations* - D2

### D.2.1 Meta-modelo de KobrA2

A diretriz D2 possui mais outra aplicação no meta-modelo de KobrA2, envolvendo duas associações de composição entre as meta-classes:

(i) `Constraint` e `ExpressionInOCL`;
(ii) `Constraint` e `ExpressionInOCL`.

As meta-classes em (i) e (ii) pertencem ao mesmo pacote, chamado `KobrA2::SUM::Constraint::Common`, com exceção da meta-classe `ExpressionInOCL` mostrada em (ii), que pertence ao pacote `KobrA2::SUM::Constraint::Behavioral`.

Neste exemplo, há duas associações de composição diferentes, porém, com os mesmos papéis, multiplicidades e a mesma meta-classe que representa o "todo": `Constraint`.

### D.2.2 Meta-modelo de Ant

A diretriz D2 pode ter mais três aplicações no meta-modelo de Ant, como por exemplo, nas associações de composição entre as seguintes meta-classes:

(i) `ClassPath` e `FileSet`;
(ii) `Path` e `FileSet`.

Em (i) e (ii) temos duas associações de composição diferentes, porém, com os mesmos papéis, multiplicidades e a mesma meta-classe que representa a "parte", chamada `FileSet`. Todas estas meta-classes apresentadas pertencem ao pacote `Ant`.

### D.2.3 Meta-modelo de Java

A diretriz D2 pode ter mais 17 aplicações no meta-modelo de Java, como por exemplo, nas associações de composição entre as meta-classes a seguir:

(i) `WhileStatement` e *`Statement`*;

(ii) `LabeledStatemant` e *`Statement`*;

(iii) `DoStatement` e *`Statement`*;

(iv) `EnhancedForStatement` e *`Statement`*;

(v) `ForStatement` e *`Statement`*.

Trata-se de cinco associações de composição diferentes, porém, com os mesmos papéis, multiplicidades e a mesma meta-classe que representa a "parte": *`Statement`*. Todas estas meta-classes apresentadas pertencem ao pacote `JavaAbstractSyntax`.

### D.2.4 Meta-modelo de QVT

A diretriz D2 pode ter mais uma aplicação no meta-modelo de QVT, envolvendo as associações entre as seguintes meta-classes:

(i) `OrderedTupleType` e *`Type`*;

(ii) `AnonymousTupleType` e *`Type`*.

Trata-se de duas associações diferentes, porém, com os mesmos papéis, multiplicidades e uma meta-classe em comum: *`Type`*. As meta-classes `OrderedTupleType` e `AnonymousTupleType` apresentadas em (i) e (ii) pertencem ao pacote `ImperativeOCL`. Já a meta-classe *`Type`* pertence ao pacote `EMOF`.

### D.2.5 Meta-modelo de SPEM

A diretriz D2 pode ter mais uma aplicação no meta-modelo de SPEM, envolvendo duas associações de composição entre as seguintes meta-classes:

(i) `Activity` e `ProcessParameter`;

(ii) `TaskUse` e `ProcessParameter`.

Trata-se de duas associações de composição diferentes, porém, com os mesmos papéis, multiplicidades e a mesma meta-classe que representa a "parte": `ProcessParameter`. As meta-classes `Activity` e `ProcessParameter` pertencem

ao pacote `ProcessStructure`. Já a meta-classe `TaskUse` pertence ao pacote `ProcessWithMethods`.

## D.3 *Adding Utility Operations* - D8

### D.3.1 Meta-modelo de KobrA2

A diretriz D8 pode ter mais duas aplicações no meta-modelo de KobrA2. Como exemplo de uma destas aplicações, vejamos três invariantes que possuem expressões repetidas. Para aplicarmos a diretriz, devemos definir uma nova operação, de forma que ela seja invocada dentro das invariantes para evitar a repetição:

```
[1] context KobrA2::Specification::Structural::Instance::Service
inv: packagedElements->select(oclIsKindOf(ComponentObject) and
hasStereotypes->includes('subject'))->size() = 1

[2] context KobrA2::Specification::Structural::Instance::Type
inv: packagedElements->select(oclIsKindOf(ComponentObject) and
hasStereotypes->includes('subject'))->size() = 1

[3] context KobrA2::Realization::Structural::Instance::Service
inv: packagedElements->select(oclIsKindOf(ComponentObject) and
hasStereotypes->includes('subject'))->size() = 1
```

## D.4 *Defining Boolean Attributes Default Value* - D9

### D.4.1 Meta-modelo de KobrA2

Os demais atributos do meta-modelo de KobrA2 que não possuem valor *default* especificado são apresentados na Tabela 19.

**Tabela 19: Exemplos de aplicações da diretriz D9 no meta-modelo de KobrA2.**

| Atributo | Meta-classe |
|---|---|
| atPre | KobrA2::SUM::Constraint::<br>OclExpressions::FeatureCallExp |
| boolean | KobrA2::SUM::Constraint::<br>Common::Constraint |
| query | |
| boolean | KobrA2::SUM::Constraint::<br>Common::ExpressionInOcl |
| query | |
| boolean | KobrA2::SUM::Constraint::<br>Common::OclExpression |
| query | |

## D.4.2 Meta-modelo de Java

O restante dos atributos booleanos presentes no meta-modelo de Java também não possui valor *default* especificado, vejamos como exemplo alguns deles na Tabela 20.

**Tabela 20: Exemplos de aplicações da diretriz D9 no meta-modelo de Java.**

| Atributo | Meta-classe |
|---|---|
| abstract | JavaAbstractSyntax::Modifier |
| final | |
| native | |
| none | |
| private | |
| protected | |
| public | |
| onDemand | JavaAbstractSyntax::ImportDeclaration |
| static | |
| nested | JavaAbstractSyntax::TagElement |
| booleanValue | JavaAbstractSyntax::BooleanLiteral |

## D.4.3 Meta-modelo de QVT

Os demais atributos do meta-modelo de QVT que não possuem valor *default* especificado são apresentados na Tabela 21.

**Tabela 21: Exemplos de aplicações da diretriz D9 no meta-modelo de QVT.**

| Atributo | Meta-classe |
|---|---|
| isBlackbox | QVTOperational::ImperativeOperation |
| isQuery | QVTOperational::Helper |
| isStrict | QVTOperational::MappingCallExp |
| isReset | ImperativeOCL::AssignExp |
| isDefault | QVTCore::Assignment |

# D.5 *Defining Association Member Ends Features* - D11

## D.5.1 Meta-modelo de KobrA2

A diretriz D11 pode ter mais 13 aplicações no meta-modelo de KobrA2, como por exemplo, nas associações de composição entre as meta-classes apresentadas na Tabela 22, a seguir.

**Tabela 22: Exemplos de aplicações da diretriz D11 no meta-modelo de KobrA2.**

| Meta-classes | Pacote |
|---|---|
| `EffectOperation` e `Post` | `KobrA2::SUM::` `Constraint::Behavioral` |
| `QueryOperation` e `Body` | |
| `ComponentClass` e *`View`* | `KobrA2::Views::Subject` |

## D.5.2 Meta-modelo de Ant

Esta diretriz pode ser aplicada nas demais associações de composição existentes no meta-modelo de Ant, como por exemplo, nas associações entre as meta-classes apresentadas na Tabela 23, a seguir.

**Tabela 23: Exemplos de aplicações da diretriz D11 no meta-modelo de Ant.**

| Meta-classes | Pacote |
|---|---|
| `Path` e `PathElement` | `Ant` |
| `Project` e `TaskDef` | |
| `ComponentClass` e *`View`* | |
| `FilterSet` e `Filter` | |

## D.5.3 Meta-modelo de Java

A diretriz D11 pode ser aplicada em mais 102 associações de composição do meta-modelo de Java, como por exemplo, nas existentes entre as meta-classes apresentadas na Tabela 24, a seguir.

**Tabela 24: Exemplos de aplicações da diretriz D11 no meta-modelo de Java.**

| Meta-classes | Pacote |
|---|---|
| *`VariableDeclaration`* e `SimpleName` | `JavaAbstractSyntax` |
| `Initializer` e `Block` | |
| `CatchClause` e `Block` | |
| `ArrayCreation` e `ArrayType` | |
| `FieldAccess` e *`Expression`* | |

### D.5.4 Meta-modelo de QVT

A diretriz D11 pode ter mais 16 aplicações no meta-modelo de QVT, como por exemplo, nas associações de composição entre as meta-classes apresentadas na Tabela 25, a seguir.

**Tabela 25: Exemplos de aplicações da diretriz D11 no meta-modelo de QVT.**

| Meta-classes | Pacote |
|---|---|
| `ObjectTemplateExp` e `PropertyTemplateItem` | `QVTTemplate` |
| `RelationDomain` e `DomainPattern` | `QVTRelational` |
| `OperationalTransformation` e `ModelParameter` | `QVTOperational` |
| `DictLiteralExp` e `DictLiteralPart` | `ImperativeOCL` |

### D.5.5 Meta-modelo de SPEM

A diretriz D11 pode ter mais 13 aplicações no meta-modelo de SPEM, como por exemplo, nas associações de composição entre as meta-classes apresentadas na Tabela 26, a seguir.

**Tabela 26: Exemplos de aplicações da diretriz D11 no meta-modelo de SPEM.**

| Meta-classes | Pacote |
|---|---|
| *`WorkDefinition`* e *`WorkDefinitionParameter`* | `Core` |
| *`DescribableElement`* e `ContentDescription` | `ManagedContent` |
| `MethodLibrary` e `MethodPlugin` | `MethodPlugin` |
| `MethodPlugin` e `ProcessPackage` | |

## D.6 *Redefining Boolean Attribute Names* - D12

### D.6.1 Meta-modelo de KobrA2

A diretriz D12 também pode ser aplicada nos seguintes atributos booleanos do meta-modelo de KobrA2, vejamos cada um deles na Tabela 27.

**Tabela 27: Exemplos de aplicações da diretriz D12 no meta-modelo de KobrA2.**

| Atributo | Meta-classe |
|---|---|
| query | KobrA2::SUM::Constraint::Common |
| boolean | |

### D.6.2 Meta-modelo de Java

A diretriz D12 também pode ser aplicada nos demais atributos booleanos presentes no meta-modelo de Java, vejamos como exemplo alguns deles na Tabela 28.

**Tabela 28: Exemplos de aplicações da diretriz D12 no meta-modelo de Java.**

| Atributo | Meta-classe |
|---|---|
| constructor | JavaAbstractSyntax::MethodDeclaration |
| varargs | |
| nested | JavaAbstractSyntax::TagElement |
| booleanValue | JavaAbstractSyntax::BooleanLiteral |
| declaration | JavaAbstractSyntax::SimpleName |