UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

ALEXANDRE TORRES

MD-JPA: Um perfil UML para modelagem do mapeamento objeto-relacional com JPA em uma abordagem dirigida por modelos

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

Profa. Dra. Renata Galante
Orientadora

Prof. Dr. Marcelo S. Pimenta
Co-orientador

Porto Alegre, junho de 2009.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Torres, Alexandre

MD-JPA: Um perfil UML para modelagem do mapeamento objeto-relacional com JPA em uma abordagem dirigida por modelos / Alexandre Torres – Porto Alegre: Programa de Pós-Graduação em Computação, 2009.

140 p.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2009. Orientador: Renata Galante; Co-orientador: Marcelo S. Pimenta.

1.Persistence. 2.UML 3.DDM 4.JPA 5.Database. 6.Java. I. Galante, Renata. II. Pimenta, Marcelo S. III. Título.

AGRADECIMENTOS

Gostaria de dedicar este trabalho às mulheres, em especial à Berenice, minha mãe, que se formou em filosofia depois dos sessenta anos; Aline, minha amada companheira, pelo amor e paciência que sempre demonstrou, principalmente nas fases mais conturbadas que passei nos últimos tempos; a minha tia-madrinha Sônia, que me incentivou e me ajudou muito, mesmo a distância; e a minha avó Lacínia, que não está mais presente entre nós. É seu amor a única coisa que realmente me motiva. Como diria Orson Wells, se não fossem as mulheres...

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

BD	Banco de Dados
DDM	Desenvolvimento Dirigido por Modelos
EMF	Eclipse Modeling Framework
EMOF	Essential MOF
JDT	Java Development Tools
JPA	Java Persistence API
MDA	Model Driven Architecture
MD-JPA	Model Driven JPA
MOF	Meta-Object Facility
MOR	Mapeamento Objeto-Relacional
OCL	Object Constraint Language
OMG	Object Management Group
SGBD	Sistema Gerenciador de Banco de Dados
XMI	XML Metadata Interchange

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

A abordagem de desenvolvimento dirigido por modelos (DDM) propõe que modelos (e transformações entre modelos) assumam o papel principal no desenvolvimento de sistemas. Entretanto, não há uma notação consensual para modelagem de persistência baseada em arcabouços de mapeamento objeto-relacional: enquanto a UML não possui recursos específicos para a modelagem de persistência, o modelo entidade-relacionamento não expressa os conceitos dinâmicos existentes na UML.

Este trabalho propõe o perfil UML MD-JPA (*Model Driven JPA*) para a modelagem de persistência baseada na já difundida API de persistência Java (JPA), buscando a modelagem dos elementos persistentes e transientes de forma mais coerente e integrada. São especificadas as principais características do perfil MD-JPA, assim como a maneira pela qual modelos que adotam este perfil podem ser transformados em implementação Java, através de transformações de modelos propostas em uma abordagem DDM. Por fim, uma ferramenta de código livre foi desenvolvida para disponibilizar para comunidade os resultados deste trabalho.

Palavras-Chave: Persistência, UML, DDM, JPA, Banco de dados, Java.

MD-JPA: A UML profile for object relational mapping with JPA in a model driven approach

ABSTRACT

The model driven development (MDD) approach proposes that models (and model-to-model transformations) play the main role on system development. However, there is not a consensual notation to model persistence based upon object relational mapping frameworks: while UML lacks specific resources for persistence modeling, the entity-relationship model does not make reference to the dynamic concepts existing in UML.

This work proposes MD-JPA, a UML profile for persistence modeling based on the well-known Java Persistence API (JPA), pursuing the modeling of transient and persistent elements in a more coherent and integrated way. This work describes the main characteristics of MD-JPA as well as the way that models that adopt such profile can them be used to generate a Java implementation by the application of the proposed model transformations on a MDD approach. Finally, an open source tool was developed to make the results of this work available to the community.

Keywords: Persistence, UML, MDD, JPA, Database, Java.

1 INTRODUÇÃO

A UML (*Unified Modeling Language*) originou-se do paradigma orientado a objetos (RUMBAUGH, JACOBSON e BOOCH, 1999) e rapidamente se tornou uma notação padrão para a modelagem de software, apesar de sua deficiência em recursos para tratar de alguns aspectos importantes de sistemas tais como a modelagem da persistência de dados (AMBLER, 2003) e a interação homem-máquina (PALANQUE e BASTIDE, 2003). No que tange à persistência relacional, o modelo entidade-relacionamento (ER) (CHEN, 1976) e seus derivados ainda são empregados como os principais artefatos para modelagem conceitual de bancos de dados, coexistindo com os modelos UML no processo de Engenharia de Software (AMBLER, 2003).

O desenvolvimento dirigido por modelos (DDM) propõe que os modelos assumam o papel principal no processo de desenvolvimento de sistemas, substituindo em parte ou totalmente a codificação de sistemas (BEYDEDA, BOOK e GRUHN, 2005). Ao utilizar a abordagem DDM a informação representada no conjunto de modelos que descreve o sistema deve ser coerente, integrada e computável, de forma que transformações automáticas possam tornar modelos em um sistema executável (MELLOR et al, 2004). Neste contexto, utilizar modelos separados em UML e ER tornou-se um problema para a construção das transformações, pois não há uma integração entre os elementos representados nos modelos, nem um metamodelo comum às duas notações.

Bancos de dados relacionais e linguagens de programação orientadas a objeto são baseados em paradigmas diferentes, apresentando incompatibilidades técnicas, conceituais e até mesmo culturais. O conjunto de dificuldades enfrentadas na integração dos dois paradigmas é comumente chamado de problema de incompatibilidade da impedância[1] objeto-relacional (AMBLER, 2003).

Os *frameworks* de mapeamento objeto-relacional (MOR) procuram solucionar (com ênfase nos aspectos técnicos) o problema de "incompatibilidade da impedância" no nível da implementação do sistema (AMBLER, 2003). A API de persistência Java (JPA, do inglês *Java Persistence API*) é uma especificação de *frameworks* MOR para plataforma Java bastante difundida, que permite a representação dos conceitos da orientação a objetos e de bancos de dados relacionais em programas Java anotados. Todavia, tanto no JPA, quanto no escopo do MOR, há uma carência de linguagens de modelagem para expressão de seus conceitos particulares.

Alguns trabalhos abordam a questão da persistência dentro da abordagem DDM, para bancos de dados orientados a objetos (GRANT, CHENNAMANENI e REZA, 2006) e *data warehouses* (LUJÁN-MORA, TRUJILLO e SONG, 2006) por exemplo,

1 *impedance mismatch*, termo em Inglês.

sem abordar as ferramentas MOR. Algumas ferramentas comerciais também abordam a modelagem de persistência em sistemas, gerando sistemas que utilizam MOR, mas vinculando a modelagem a um conjunto de *frameworks*, tecnologias e padrões utilizados por estas ferramentas, sem aprofundarem-se nos recursos do MOR ou na integração entre os conceitos relacionais e da orientação a objetos.

Este trabalho propõe o MD-JPA (*Model Driven JPA*), um perfil UML para modelagem de sistemas no padrão JPA. O MD-JPA permite a representação dos aspectos estruturais de bancos de dados e sistemas de informação em uma abordagem DDM. Através das extensões propostas à UML, modelos podem ser utilizados como artefatos no ciclo de desenvolvimento de software e como fonte para transformações visando a geração de partes do sistema.

A contribuição principal deste trabalho é o perfil UML, que pode ser aplicado em modelos criados por ferramentas padrão de modelagem UML, e uma ferramenta de desenvolvimento que viabiliza a utilização do perfil no contexto DDM. A ferramenta MD-JPA é um *plugin* da plataforma Eclipse (ECLIPSE FOUNDATION, 2009a) e complementa o perfil, permitindo avaliar e transformar modelos na implementação. A Figura 1.1 apresenta uma visão geral dos 3 componentes da ferramenta MD-JPA:

- O perfil UML, com extensões propostas à notação UML.
- As restrições que avaliam se um determinado modelo está, segundo a especificação JPA, bem formado, detectando e apresentando possíveis problemas ao usuário.
- O conjunto de transformações que levam modelos para uma implementação compatível com o padrão JPA.

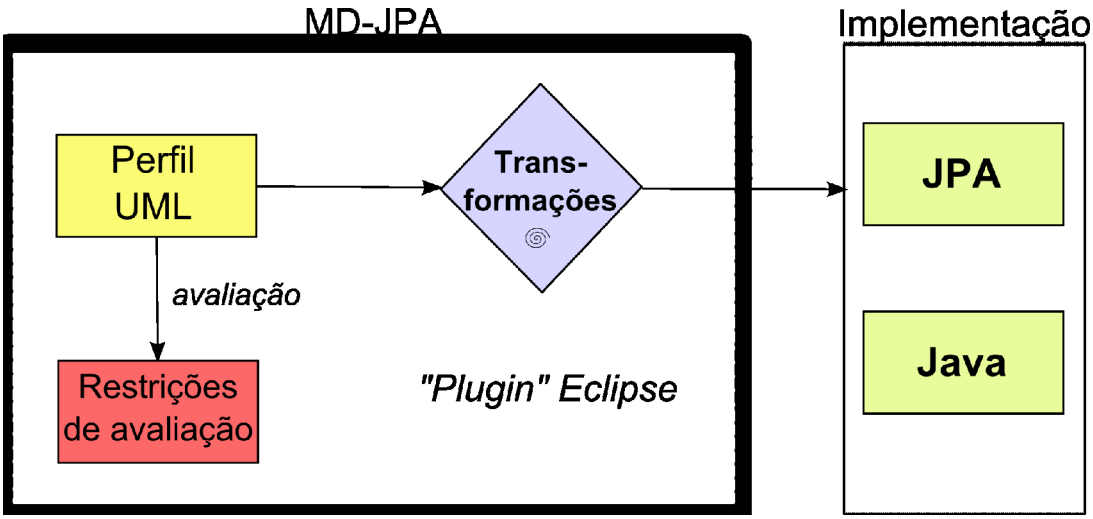


Figura 1.1: Visão geral da ferramenta MD-JPA.

O restante deste trabalho está organizado em seis capítulos. No capítulo 2 são apresentados a base conceitual e os trabalhos relacionados. O capítulo 3 especifica os elementos de modelagem do perfil MD-JPA, traçando um paralelo com a especificação JPA, através de uma série de diagramas criados a partir de uma aplicação exemplo. No capítulo 4 é apresentado um conjunto de restrições para verificar se os modelos MD-JPA estão bem formados e verificar, através de um modelo de testes, a presença de erros no perfil ou nas restrições. O capítulo 5 apresenta as transformações utilizadas para levar de modelos MD-JPA para a implementação, em Java utilizando anotações JPA, e

resultados comparativos entre os códigos fontes gerados por transformações e escritos de forma manual. O capítulo 6 faz um resumo geral da arquitetura da ferramenta desenvolvida para a utilização do MD-JPA. Por fim, o capítulo 7 traz as considerações finais sobre os resultados do trabalho.

Complementam a dissertação dois apêndices e um anexo. No Apêndice A é listado o código fonte completo das restrições OCL apresentadas no capítulo 4. No Apêndice B é listado o código fonte completo das transformações apresentadas no capítulo 5 para transformação de modelos em implementação. No Anexo é apresentado um diagrama do metamodelo da árvore sintática abstrata da linguagem Java, utilizada nas transformações do MD-JPA.

2 BASE CONCEITUAL E TRABALHOS RELACIONADOS

Este capítulo apresenta o contexto no qual a dissertação está inserida, compreendendo o mapeamento objeto-relacional, a abordagem dirigida por modelos, a modelagem de dados, a integração entre modelos e a implementação de ferramentas para modelagem e transformação de modelos. As áreas de pesquisa são apresentadas, identificando as principais necessidades e desafios, com o objetivo de delimitar o escopo do problema tratado.

Na primeira seção é apresentado o problema de incompatibilidade da impedância e o mapeamento objeto-relacional como uma das abordagens para solução deste problema, assim como sua utilização para a definição da estrutura de dados utilizada nos SGBDs, através de unidades de programa e anotações. Na segunda seção é apresentada a abordagem dirigida por modelos, seus conceitos e demandas por modelos mais completos e coerentes. Na terceira seção é apresentado, com ênfase nos bancos de dados relacionais, o estado da arte da modelagem de persistência de sistemas e trabalhos relacionados. Na quarta e última seção, são expostos os trabalhos relacionados à integração entre modelos e código, além de sua utilização nesta proposta para geração de sistemas.

2.1 Mapeamento Objeto-Relacional

O paradigma orientado a objetos é baseado em princípios da Engenharia de Software, enquanto que o paradigma relacional é baseado em princípios matemáticos. A diferença entre os dois paradigmas se manifesta principalmente quando se tenta trabalhar com ambas as tecnologias (AMBLER, 2003). O exemplo mais comum é a diferença na maneira preferencial de acesso às informações. No paradigma Orientado a Objetos (OO), o acesso aos objetos é realizado navegando-se entre seus relacionamentos, enquanto que no paradigma relacional o acesso é realizado pela junção de registros de dados em tabelas. O problema da persistência da informação quando se utiliza as duas tecnologias é frequentemente denominado de incompatibilidade da impedância objeto-relacional (AMBLER, 2003).

Uma das soluções para a incompatibilidade de impedância é a utilização de serviços de persistência, fornecidos por um *framework* de persistência reutilizável, responsável por traduzir objetos em registros para sua persistência, e traduzir registros em objetos ao realizar consultas (LARMAN, 2004). Este conjunto de serviços oferecidos é também conhecido como Mapeamento Objeto-Relacional (MOR).

O MOR busca a solução do problema de incompatibilidade da impedância no nível da implementação, permitindo ao desenvolvedor de sistemas utilizar as tecnologias

relacional e orientada a objetos em conjunto, abstraindo a estrutura física do banco de dados (AMBLER, 2003). Tipicamente, um conjunto de mapeamentos é fornecido pelo desenvolvedor a partir de arquivos de configuração ou extensões de linguagem (como anotações) que permitem ao *framework* realizar sua função.

Essas informações de mapeamento são utilizadas pelo *framework* MOR para traduzir classes em tabelas, atributos em colunas e identificadores em chaves primárias, por exemplo. Se o *framework* possuir um mapeamento com informações suficientes sobre a base de dados, pode também ser utilizado para gerar a estrutura do banco de dados, centralizando as informações sobre tabelas, colunas, relacionamentos, restrições e até índices (DEMICHIEL e KEITH, 2006).

A plataforma Java (GOSLING et al, 2005) possui uma especificação genérica para *frameworks* MOR, o JPA (DEMICHIEL e KEITH, 2006). O JPA define padrões para o comportamento e desenvolvimento de *frameworks* MOR e sistemas que utilizam seus serviços, tais como padrões para anotações, arquivos de configuração e regras para escrita de código. Além disso, o JPA pode ser utilizado tanto em aplicações independentes, quanto em aplicações integradas com a plataforma *Enterprise JavaBeans* (EJB), esta última voltada para sistemas executados em servidores.

Frameworks MOR também estão presentes em outras plataformas de desenvolvimento, alguns dos quais semelhantes ao padrão JPA. O *Hibernate* disponibiliza uma versão tanto para a plataforma Java quanto para *.NET* da Microsoft ®, que apesar de não seguir estritamente o padrão JPA, utiliza os mesmos conceitos básicos deste padrão (RED HAT MIDDLEWARE, 2009). O ADO.NET Entity Framework [2] é uma iniciativa recente para MOR independente do padrão JPA, mas que, por enquanto, ainda fornece um conjunto muito pequeno de recursos [3].

O desenvolvimento de classes utilizando MOR frequentemente impõe restrições ao desenvolvedor e uma série de novas implicações para as decisões tomadas na programação de classes, suas propriedades e métodos, tais como: (i) certos tipos de dados não podem ser utilizados em propriedades (ou trazem consequências inesperadas no armazenamento); (ii) métodos de acesso para estas propriedades devem ser escritos conforme um padrão de nomenclatura; (iii) necessidade de definir chaves primárias. Afora estas restrições semânticas, há também as restrições de performance no sistema, que dependem da estratégia de mapeamento adotada. Tanto os *frameworks* quanto as ferramentas de desenvolvimento atuais ajudam a identificar se estas restrições estão sendo consideradas, orientando o desenvolvedor a realizar as correções necessárias.

Nesta seção foram apresentados conceitos referentes aos *frameworks* MOR e o padrão JPA, que vêm sendo utilizados no desenvolvimento de sistemas com o objetivo de amenizar as diferenças entre o modelo relacional e o modelo orientado a objetos. Entretanto, quando se aborda a modelagem destes sistemas, pouca ou nenhuma informação sobre este mapeamento objeto-relacional está presente, nem tampouco pode ser verificada no contexto do padrão JPA. Nas próximas seções são apresentados os conceitos do desenvolvimento dirigido por modelos e trabalhos relacionados ao problema da modelagem da persistência nos sistemas de informação.

---

2 Disponível em http://msdn.microsoft.com/en-us/library/bb399572.aspx

3 Segundo petição online da comunidade de usuários em http://efvote.wufoo.com/forms/ado-net-entity-framework-vote-of-no-confidence/

2.2 Desenvolvimento Dirigido por Modelos

As metodologias modernas de desenvolvimento de sistemas de informação incluem a criação de modelos que descrevem suas características. O foco destes modelos é servir de ferramenta para compreensão dos objetivos e problemas levantados no sistema atual, bem como para a discussão, projeto de soluções e documentação do sistema (PRESSMAN, 2001).

Se modelos são utilizados para compreender o domínio do problema e de sua solução, a rede de relacionamento entre estes modelos registra o processo pelo qual esta solução foi desenvolvida e pode ser utilizada para avaliar as implicações de modificações em qualquer ponto do processo (BEYDEDA, BOOK e GRUHN, 2005). A abordagem DDM parte desta premissa ao colocar os modelos como artefatos centrais no desenvolvimento de sistemas.

A MDA (*Model Driven Architecture*) é um conjunto de padrões, ainda em desenvolvimento, propostos pela OMG (*Object Management Group*) com o objetivo de fomentar o desenvolvimento dirigido por modelos (OMG, 2001). A partir desta iniciativa, surgiram novas ferramentas e notações para especificar modelos, metamodelos e transformações entre modelos. O propósito desta iniciativa é viabilizar a construção de modelos computáveis, artefatos centrais do desenvolvimento de sistemas.

A linguagem de descrição de metamodelos MOF (*Meta-Object Facility*) permite a construção de novas notações de modelagem para o padrão MDA (OMG, 2006b). A notação UML foi redefinida na linguagem MOF, o que permitiu a criação de novos metamodelos derivados da UML, e introduziu o recurso de expansão através do perfil. Também fazem parte do esforço para aprimorar a UML a padronização do armazenamento de modelos em diferentes ferramentas através do XMI[4] e da representação gráfica destes modelos em diagramas (OMG, 2006a).

A linguagem OCL (*Object Constraint Language*) foi originalmente proposta para permitir a criação de restrições em modelos UML. A proposta MDA fez surgir a necessidade de se expandir a OCL para suportar os elementos de metamodelos definidos pelo usuário e permitir a avaliação da conformidade dos modelos em relação a seus metamodelos (OMG, 2006c).

A especificação QVT (*Query View Transform*) (OMG, 2008) propõe um padrão para transformação entre modelos no padrão MOF, estendendo a OCL para incluir o mapeamento entre modelos de diferentes domínios, ou seja, modelos com diferentes metamodelos. Transformações em QVT podem ser construídas em duas linguagens de transformação, a relacional (declarativa) e a operacional (imperativa). Na linguagem relacional, são definidas equivalências entre elementos do metamodelo de entrada e elementos de saída. A linguagem operacional permite a construção de procedimentos, como em um programa, que avaliam um determinado contexto e desencadeiam a relação de transformação.

Durante o período entre as primeiras versões do padrão MDA até a especificação da linguagem padrão de transformação QVT, foram propostas outras soluções para a construção de transformações entre modelos. A linguagem ATL (JOUAULT et al, 2006) é uma destas soluções e continua sendo utilizada enquanto ferramentas abertas e completas para QVT ainda não estão disponíveis.

---

4 XML Metadata Interchange, disponível em http://www.omg.org/spec/XMI/index.htm

A linguagem ATL é similar à QVT, possuindo os recursos de linguagem relacional e operacional. Sua sintaxe é semelhante porque também é baseada na linguagem OCL, mas sem incorporar o recurso de rastreabilidade previsto na QVT. O ATL possui uma ferramenta de código aberto madura e completa, em contraposição às poucas ferramentas que implementam o padrão QVT disponíveis. Ademais, segundo o quê pôde ser apurado no presente trabalho, nenhuma ferramenta ainda abrange as linguagens relacional e operacional ao mesmo tempo, tal como o ATL abrange.

Ao longo deste trabalho, a linguagem ATL é utilizada para descrever as transformações entre modelos. É relevante salientar que quando as ferramentas que implementam o padrão QVT estiverem maduras, essas transformações poderão ser portadas para o QVT. Eclipse QVTO[5] é uma das implementações em andamento que é aberta, já implementa a linguagem operacional e futuramente implementará a linguagem relacional do QVT.

2.3 Modelagem de dados para persistência

O modelo de entidades e relacionamentos (ER) foi proposto na década de 70 com o objetivo de unificar a modelagem de dados para sistemas de informação (CHEN, 1976). Para atingir este objetivo, procurou unificar os modelos de rede, relacional e de entidade, incorporando suas melhores características em uma única notação gráfica. Ainda hoje, os diagramas ER e seus derivados são amplamente utilizados no projeto de sistemas e bancos de dados relacionais, apesar da existência de novos padrões para modelagem de sistemas.

A partir da popularização do paradigma de desenvolvimento OO surgiram diversas notações de modelagem para suportar projetos OO, que levaram a criação da Linguagem de Modelagem Unificada (em inglês, UML). A UML se propõe a especificar, visualizar, construir e documentar os artefatos em um sistema de software. Sua abordagem é independente de domínio e método de desenvolvimento, dividindo a representação do sistema em diversas visões com diferentes pontos de vista, mas compartilhando dos mesmos conceitos (RUMBAUGH, JACOBSON e BOOCH, 1999).

A UML possui uma série de notações expressas em diagramas, divididas em diagramas estruturais e diagramas dinâmicos. O grupo dos diagramas estruturais tem como principal ferramenta o diagrama de classes, abordando a definição de dados e operações utilizados para a realização dos casos de uso. O grupo de diagramas dinâmicos tem como objetivo principal descrever como os objetos colaboram para realização de uma determinada funcionalidade, compreendendo diagramas de caso de uso, colaboração e estado, entre outros.

O foco de interesse deste estudo será nos diagramas estruturais, especialmente no diagrama de classes, pois a maior parte dos elementos propostos na JPA e na modelagem de dados tem um caráter estrutural. Conceitos como tabelas, colunas, classes, operações, relacionamentos e herança descrevem a organização de um sistema. Além disso, os diagramas dinâmicos da UML geralmente referenciam elementos definidos nos diagramas estruturais: um diagrama de interação apresenta instâncias e mensagens que referenciam classes e operações definidas no diagrama de classes.

---

5 Eclipse QVTO, disponível em http://www.eclipse.org/m2m/qvto/doc/M2M-QVTO.pdf

Assim como a notação ER no projeto de banco de dados, a UML foi amplamente adotada na Engenharia de Software. Como a UML não possui recursos específicos para representação da persistência, frequentemente são utilizados modelos nas duas notações. Aos poucos, a modelagem de sistemas e suas bases de dados vêm se tornando processos independentes, que utilizam ferramentas diferentes e por vezes sob responsabilidade de equipes distintas. No campo da modelagem, assim como no de desenvolvimento, também se manifesta o problema de incompatibilidade da impedância objeto-relacional.

A modelagem ágil de bancos de dados é uma das propostas mais difundidas para modelagem de bancos de dados com UML (AMBLER, 2003). Ela utiliza o diagrama de classes para representar os modelos de dados através do uso de estereótipos de modelo, classe e atributo. São criados modelos em três níveis de abstração: conceitual, lógico e físico.

Nos modelos conceitual e lógico são representadas entidades, relacionamentos e atributos. Entre os conceitos suportados estão a distinção dos tipos de relacionamento entre generalização, agregação e composição, e a distinção de tipos de atributos entre chaves candidatas e identificadores únicos. O modelo conceitual é construído com o ponto de vista independente da classe de banco de dados utilizada, enquanto que o modelo lógico pressupõe a utilização de um banco de dados relacional. Já no modelo físico são representadas tabelas, visões, relacionamentos e colunas do ponto de vista de um SGBD. Entre os conceitos suportados estão a distinção de chaves primárias, chaves estrangeiras, colunas obrigatórias e índices.

Como resultado, o perfil proposto traz para UML a especificação da persistência de sistemas, possibilitando modelar as bases de dados na mesma linguagem utilizada para modelar os componentes de software. A proposta de modelagem ágil é neutra em relação à solução para o problema do mapeamento objeto-relacional. Tanto pode ser utilizada em um sistema no qual todo o acesso aos bancos de dados é manualmente implementado, quanto em um sistema que utiliza *frameworks* MOR.

Contudo, a neutralidade quanto a solução utilizada para o mapeamento objeto-relacional fez com que o perfil ágil não abordasse a questão da integração entre os modelos de dados e de sistemas. Seu metamodelo não representa a relação entre os diagramas de banco de dados e os diagramas que representam os componentes de software do sistema. Ou seja, os modelos de dados representam apenas os aspectos de persistência, os modelos comuns representam apenas os componentes de software e se supõe que algum *processo* (manual ou automático) verifique a coerência entre esses modelos, de acordo com a tecnologia utilizada para busca e armazenamento de dados.

2.3.1 Trabalhos Relacionados

Mais recentemente, novas extensões vêm sendo propostas para a notação UML a fim de representar aspectos mais avançados da persistência. Um perfil para modelagem de data warehouses (LUJÁN-MORA, TRUJILLO e SONG, 2006) propõe a extensão dos diagramas de classe, a organização de pacotes da UML para modelagem multidimensional e utiliza recursos do DDM para geração dos esquemas de banco de dados.

Outro aspecto também abordado é a utilização dos diagramas de comportamento da UML para representar operações sobre bases de dados (SONG, YIN e RAY, 2007). Nesta proposta, diagramas de sequência são utilizados para representar consultas, inclusões, alterações e exclusões de registros. As tabelas e colunas utilizadas nos

diagramas de sequência referenciam classes e atributos que representam tabelas e colunas em um diagrama de classes semelhante ao proposto por Ambler (AMBLER, 2003).

Um perfil UML para modelagem de dados (AMBLER, 2009) é uma versão mais completa do perfil de modelagem ágil, mas igualmente focada na especificação de bancos de dados com UML. Estas propostas mostram o potencial de extensibilidade da UML para representar modelos de dados. Modelos estes que podem ser utilizados para representar bases de dados, mas que não abordam as relações entre os modelos de dados e os modelos de aplicação orientados a objetos.

A modelagem de bancos de dados orientados a objeto (GRANT, CHENNAMANENI e REZA, 2006) é um exemplo de padrão que utiliza a abordagem DDM, permitindo a construção de modelos que podem ser transformados em código. No entanto, seu foco é na representação da estrutura de banco de dados OO com a notação UML, deixando de lado a integração entre os elementos relacionais e orientados a objetos - um problema central da incompatibilidade de impedância objeto-relacional. Além disso, o MOR não se restringe aos bancos de dados OO, na verdade nem utiliza os recursos de orientação a objetos destes SGBDs.

Em Mercator (WITTHAWASKUL e JOHNSON, 2003) os autores propõem uma abordagem para modelagem de persistência com modelos independentes de plataforma. Esta abordagem utiliza modelos UML anotados com estereótipos genéricos, que são então transformados para modelos específicos nas plataformas de persistência EJB ou XML. Os modelos específicos podem ser alterados pelo usuário, empregando estereótipos específicos. Por fim, estes modelos são transformados em código.

No caso do Mercator, apesar de ser uma iniciativa de metamodelo independente de plataforma, percebe-se que é visivelmente influenciado pela especificação EJB utilizada na época: era necessário, por exemplo, definir operações para criação e busca por chave primária para cada classe, que são características específicas da plataforma EJB. Este tipo de operação em um sistema JPA não faz sentido, já que a busca por chave é realizada pelo gerenciador de entidades (*EntityManager*) e o método de criação é o próprio construtor, cabendo ao gerenciador de entidades realizar a persistência, quando for o caso.

Um estudo sobre a modelagem de persistência com ferramentas DDM (CALIARI e SILVA, 2007) realiza um comparativo entre três ferramentas, verificando como estas lidam com a modelagem de um sistema persistente. O estudo mostra que as ferramentas DDM apresentadas forçam o usuário a utilizar certos padrões de desenvolvimento e tecnologias que não têm relação direta com a persistência que se quer modelar.

Segundo este estudo, a ferramenta AndroMDA utiliza padrões tais como objetos para transporte de dados (DTO - *Data Transfer Objects*), e a definição de classes pelo padrão *Model-View-Controller*. Na ferramenta ArcStyler, além de ter de especificar DTOs, modelos que representam páginas *web* devem ser criados para se obter algum resultado na geração. Por fim, a ferramenta OptimalJ utiliza um modelo exclusivo para persistência que não trabalha com o elemento *Interface* da UML.

No que tange à modelagem do mapeamento objeto-relacional dentro do padrão JPA, os modelos estudados não apresentam uma solução completa. Alguns modelos separam a especificação de objetos e persistência, outros apresentam traços de tecnologias específicas ou apresentam uma solução amarrada a uma plataforma restrita enquanto

outros estão focados em campos específicos da persistência, como bancos de dados orientados a objetos. Ainda que alguns destes modelos possam ser utilizados para representar os conceitos mais gerais do JPA, tais como entidades e relacionamentos persistentes, nenhum aborda todos os recursos de mapeamento disponibilizados pelo JPA.

A possibilidade de representar todas as informações do mapeamento no modelo permite a especificação do modelo de dados e modelo de objetos em um único documento central. Sem as informações sobre o mapeamento, fica difícil a avaliação de modelos quanto a sua aplicabilidade com ferramentas MOR. Consequentemente, problemas de modelagem passam para a implementação, que frequentemente assume uma estrutura diferente da proposta na modelagem.

2.4 Modelos e Código

No que tange ao DDM, os mecanismos de transformação entre modelos são o foco principal, deixando de abordar a relação entre estes modelos e o código fonte. Normalmente a transformação de código é realizada por um programa transformador baseado em fragmentos de texto. A abordagem adotada neste trabalho é de que o código fonte é também um modelo, que segue a estrutura da árvore sintática da linguagem utilizada.

A utilização da árvore sintática, como metamodelo que representa o código fonte, é um tema de pesquisa atual fomentado pela OMG, com o objetivo de definir árvores sintáticas que representem conceitos transformáveis entre linguagens diferentes em um mesmo paradigma, ou até em diferentes paradigmas (FISCHER, LUSIARDI e VON GUDENBERG, 2007). Porém, como a pesquisa de árvores sintáticas mais genéricas ainda não resultou em padrões, tampouco em ferramentas práticas, optou-se pela utilização de uma árvore sintática da linguagem Java. Algumas ferramentas foram empregadas para integrar o perfil MD-JPA e suas transformações em uma ferramenta de desenvolvimento dirigido por modelos

O metamodelo JAS (INRIA ATLANMOD, 2009) é um mapeamento completo entre a árvore sintática abstrata da linguagem de programação Java para o formato compatível MOF, armazenável em XMI. O JAS já foi utilizado com sucesso na transformação de código Java em modelos que possam ser utilizados em transformações (PIRES, BRUNET e RAMALHO, 2008).

Neste trabalho o metamodelo JAS é utilizado para transformar modelos na implementação, em código fonte da linguagem Java. Para isto é utilizado o módulo JDT (*Java Development Tools*) do Eclipse (ECLIPSE FOUNDATION, 2009b), uma API para a construção de programas Java com representação da árvore sintática em memória e facilidades para sua transformação em texto, com formatação e organização do código.

2.5 Resumo e discussão

Este capítulo apresentou a base conceitual e os trabalhos relacionados que delimitam o problema estudado neste trabalho. Primeiramente apresentou-se o problema geral da incompatibilidade de impedância entre os paradigmas da orientação a objetos e dos bancos de dados relacionais. Uma das soluções para este problema é a utilização de *frameworks* MOR na implementação dos sistemas. No entanto, no contexto do

desenvolvimento dirigido por modelos, estes passam a ocupar um papel relevante na produção dos sistemas e não há uma notação de modelagem padrão para sistemas que utilizam *frameworks* MOR.

Trabalhos relacionados à modelagem da persistência tem abordado outros aspectos da representação da persistência, sem no entanto satisfazer as necessidades específicas da maior parte dos *frameworks* atuais para MOR, especialmente os que seguem o padrão JPA. As ferramentas comerciais pesquisadas suportam a geração de aplicações que utilizam MOR, mas misturam a questão da representação da persistência com outras tecnologias, sem apresentar uma solução completa para o mapeamento objeto-relacional.

No próximo capítulo será apresentado um perfil para modelagem de sistemas que implementam a persistência através de *frameworks* MOR que seguem o padrão JPA. Este perfil parte da linguagem UML e da especificação JPA para definir um conjunto de recursos de modelagem que exploram a sinergia entre os conceitos relacionais e de orientação a objetos, permitindo ao modelador a construção de modelos com persistência dentro da abordagem DDM.

3 MD-JPA: UM PERFIL PRA MODELAGEM DE PERSISTÊNCIA

Este capítulo descreve o perfil MD-JPA (*Model Driven JPA*) para a modelagem de sistemas que utilizam o JPA para a persistência de dados. Um perfil UML é um conjunto de extensões (estereótipos) de elementos existentes na UML, que podem definir novas propriedades e relacionamentos entre outros elementos novos ou pré-existentes. O MD-JPA é composto por estereótipos, que estendem os elementos presentes no modelo de classes, e complementado por classes que definem propriedades estruturadas e enumerações.

Uma das vantagens da utilização do JPA é que ele permite a escrita de programas com poucos detalhes sobre a persistência, facilitando a prototipação, como também com um mapeamento detalhado que pode, mais tarde, ser utilizado para a criação da estrutura do banco de dados definitiva. O perfil MD-JPA preserva estas características, disponibilizando um conjunto de mapeamentos detalhados que pode ser utilizado pelo usuário. Cabe ressaltar que o usuário pode optar por utilizar apenas alguns destes estereótipos para projetar seu modelo persistente.

Além disso, o MD-JPA, sendo um perfil UML, incorpora a capacidade de permitir a construção de vários diagramas sobre o mesmo modelo. Desta forma, podem-se definir diagramas com um nível de abstração maior, com poucos detalhes sobre a persistência, e diagramas contendo uma visão completa do sistema, todos utilizando o mesmo modelo. À medida que o modelo incorpora detalhes sobre a sua implementação, os diagramas de alto nível mostram esses detalhes conforme o desejo do usuário.

Este capítulo está dividido em seções que apresentam ao mesmo tempo os principais elementos do perfil e um exemplo motivador realizado sobre o tema da avaliação da produção científica. Primeiramente é introduzido o perfil MD-JPA em uma visão geral de seus principais conceitos, seguido da introdução do exemplo motivador. As seções seguintes apresentam os recursos do perfil, com exemplos baseados na aplicação motivadora construídos na ferramenta MD-JPA, com o editor de diagramas Papyrus[6].

3.1 Visão Geral

O perfil MD-JPA propõe um conjunto de recursos para modelagem de sistemas que implementam sua persistência utilizando o padrão JPA. Os recursos apresentados no restante do capítulo referenciam as regras, anotações e outros elementos que fazem parte da especificação JPA (DEMICHIEL e KEITH, 2006). A Figura 3.1 apresenta o diagrama principal de estereótipos do MD-JPA.

6 Ferramenta Papyrus, http://www.papyrusuml.org/

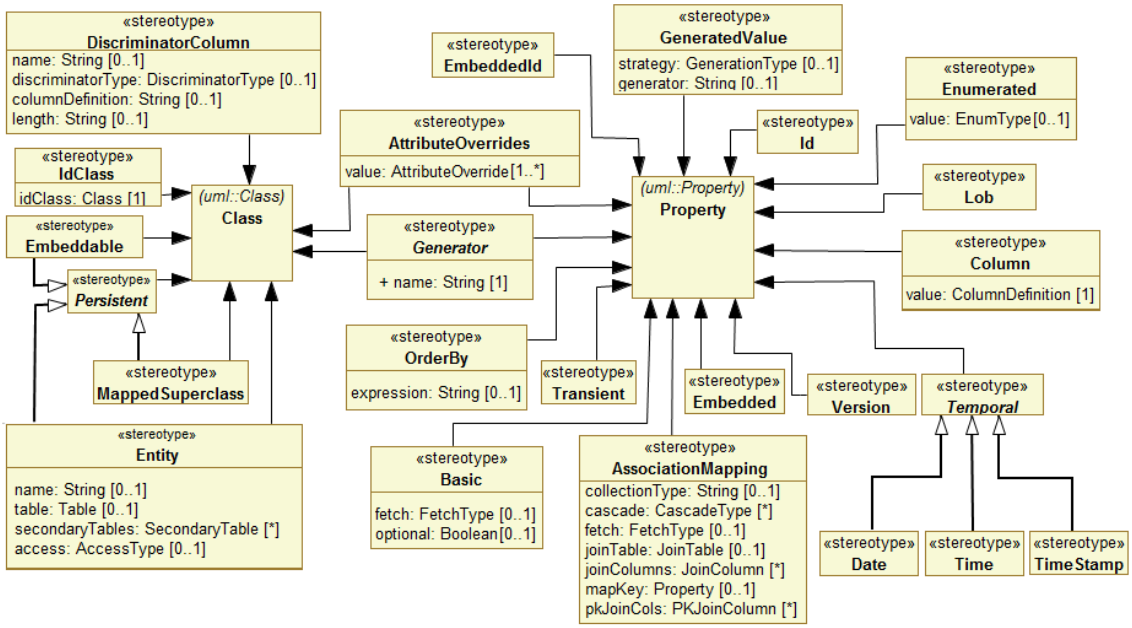


Figura 3.1: Diagrama principal de estereótipos do perfil MD-JPA.

A maior parte dos estereótipos propostos estende os elementos que representam as classes e propriedades do modelo UML. Os estereótipos receberam os nomes das anotações presentes na especificação, facilitando a identificação dos conceitos que representam.

O perfil abrange todas as anotações disponíveis na especificação JPA. A maior parte das anotações foi mapeada para estereótipos específicos. A Tabela 3.1 sumariza estas anotações e os estereótipos equivalentes no perfil MD-JPA, indicando em qual seção do trabalho a anotação é tratada.

Tabela 3.1: Anotações JPA e seus estereótipos equivalentes no MD-JPA.

JPA	*Estereótipo(s)*	*Elemento UML*	*Seção*
@Entity	Entity	Class	3.1
@Inheritance (Strategy)	SingleTable, Joined, TablePerClass	Generalization	3.6
@(One/Many) To (One/Many)	AssociationMapping	Property	3.5
@Embeddable	Embeddable	Class	3.3
@Embedded	Embedded	Property	3.3
@Transient	Transient	Property	3.2
@Id	Id	Property	3.4
@IdClass	IdClass	Class	3.4
@EmbeddedId	EmbeddedId	Property	3.4
@Column	Column	Property	3.1
@Version	Version	Property	3.1
@Enumerated	Enumerated	Property	3.3
@MappedSuperclass	MappedSuperclass	Class	3.7
@GeneratedValue	GeneratedValue	Property	3.4
@Lob	Lob	Property	3.1
@Temporal	Date, Time, TimeStamp	Property	3.1

JPA	*Estereótipo(s)*	*Elemento UML*	*Seção*
@AttributeOverride(s)	AttributeOverrides	Class,Property	3.8
@OrderBy	OrderBy	Property	3.5
@DiscriminatorColumn	DiscriminatorColumn	Class	3.6
@SequenceGenerator	SequenceGenerator	Property/Class	3.4
@TableGenerator	TableGenerator	Property/Class	3.4
@AssociationOverride	AssociationOverrides	Generalization	3.8

Algumas anotações foram simplificadas e incluídas como propriedades em outros estereótipos. Estas anotações são apresentadas na seção 3.5. Antes de introduzir os recursos disponibilizados nesta proposta, é apresentado um sistema de exemplo sobre a aplicação do MD-JPA.

3.1.1 Exemplo de aplicação: Sistema de avaliação para produção científica

Com o objetivo de demonstrar o emprego do perfil MD-JPA é apresentado um exemplo de aplicação para o desenvolvimento de um sistema para avaliação da produção científica. Primeiramente são introduzidos os objetivos do sistema e seus requisitos principais, juntamente com um diagrama de domínio representando as principais classes e relacionamentos. Detalhes do sistema serão apresentados como exemplos, na medida em que os recursos do perfil forem descritos.

O sistema de avaliação tem dois objetivos principais:

- Especificar uma estrutura única de informações, permitindo a implementação de um processo único para avaliar pesquisadores, publicações científicas, eventos e instituições de pesquisa.
- Especificar um sistema de avaliação aberta para documentos científicos (OLIVEIRA et al, 2005).

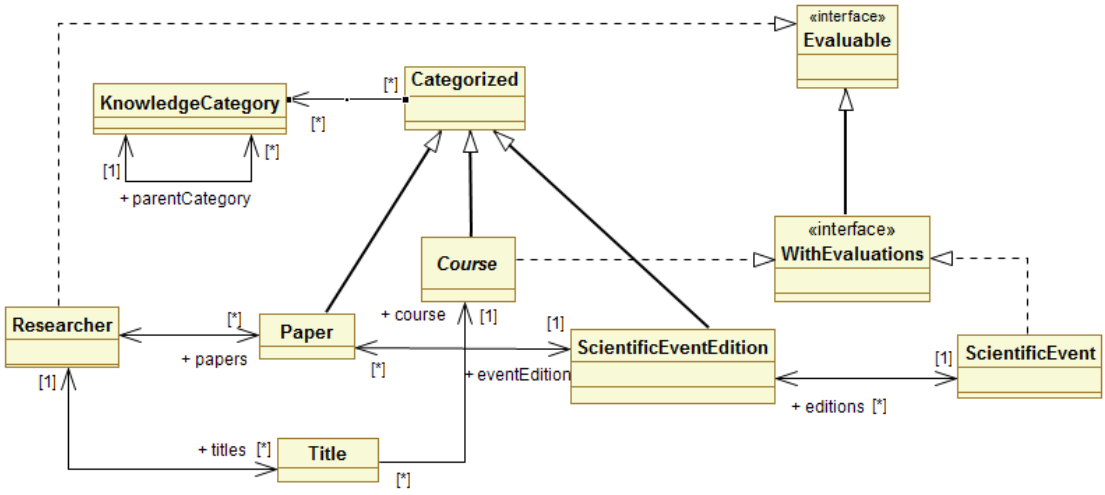


Figura 3.2: Diagrama de domínio nos primeiros estágios do projeto do sistema[7].

Na Figura 3.2 é apresentado um diagrama de classes com a relação dos principais conceitos utilizados na fase inicial do projeto. Neste diagrama são identificados os

7 Os exemplos serão apresentados em inglês para manter a coerência com os artigos.

conceitos de Artigo (*Paper*); Curso (*Course*), abrangendo os cursos, como graduação ou mestrado); *EventoCientífico* (*ScientificEvent*) e *EdiçãoDeEventoCientífico* (*ScientificEventEdition*), abrangendo eventos científicos, periódicos ou outros eventos que levam a publicação de artigos. Estes conceitos são especializações do conceito mais amplo chamado Categorizado (*Categorized*), que abrange todos os elementos que podem ser categorizados em uma árvore do conhecimento.

O conceito categoria de conhecimento (*KnowledgeCategory)* representa as áreas do conhecimento em categorias, as quais estão relacionadas entre si em uma estrutura de árvore através do auto-relacionamento categoria-pai (*parentCategory*). A ideia da árvore do conhecimento é similar à utilizada pela CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) na tabela de áreas do conhecimento[8], ou no caso da Ciência da Computação a ACM (*Association for Computing Machinery*), que também propõe uma classificação em forma de árvore[9]. Cada elemento categorizável (instâncias de *Categorized*) pode estar associado a mais de um elemento na árvore do conhecimento.

A interface Avaliável (*Evaluable*) representa os serviços de avaliação disponibilizados, implementados pelas classes que são avaliáveis. A interface *ComAvaliações* (*WithEvaluations*) estende *Evaluable* permitindo o armazenamento de um histórico de avaliações. Desta forma, o Pesquisador (*Researcher*) implementa a interface *Evaluable,* pois sua avaliação é realizada no momento presente. Já o curso implementa a interface *WithEvaluations* porque sua avaliação depende do momento avaliado. Finalmente, a classe Título (*Title)* representa os títulos adquiridos pelo pesquisador na conclusão de cursos acadêmicos.

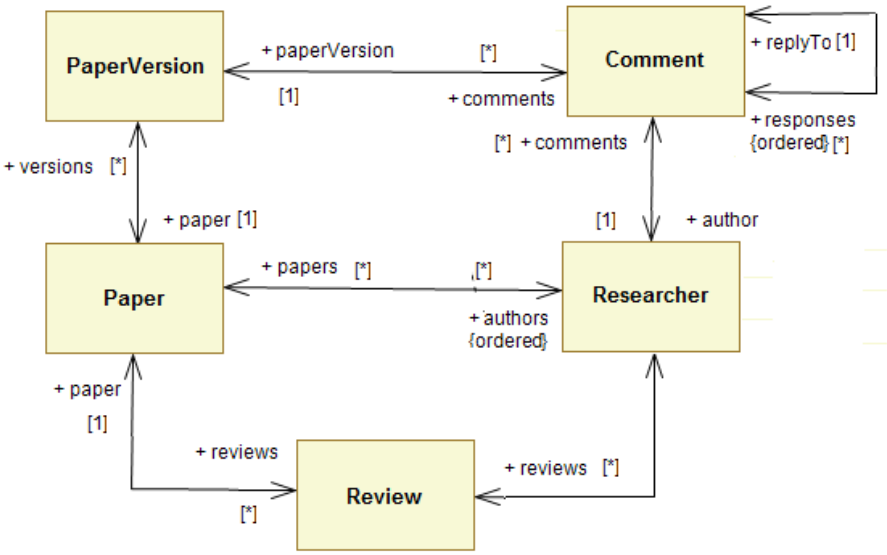


Figura 3.3: Avaliação aberta de artigos científicos.

A Figura 3.3 apresenta um diagrama com as principais classes e relacionamentos utilizados na avaliação aberta de artigos científicos. Cada artigo (*Paper*) pode ter várias versões de seu texto (*PaperVersion*). Quando o artigo é submetido, são criadas revisões (*Review*), atribuindo quais pesquisadores serão responsáveis por avaliar o trabalho.

8 Disponível em http://www.capes.gov.br/avaliacao/tabela-de-areas-de-conhecimento

9 Disponível em http://www.acm.org/about/class/1998/

Se o trabalho for considerado adequado pelos pesquisadores, ele passa para a fase de avaliação pública, na qual a comunidade pode realizar comentários (*Comment*) sobre o texto. Os comentários também podem ser adicionados em resposta a outros comentários (relação *replyTo*). Ao mesmo tempo, novas versões do texto podem ser submetidas pelos autores, cabendo ao sistema apresentar os comentários discernindo a qual versão do documento se referem.

Após um tempo determinado, os responsáveis reavaliam a evolução do trabalho, registrando sua aceitação final. Esses dois modelos representam os principais conceitos do sistema de exemplo, sem uma preocupação com a questão da persistência. Nas próximas seções serão apresentados os recursos disponibilizados pelo MD-JPA para modelagem de sistemas e seu uso na aplicação apresentada.

3.2 Entidades e propriedades

O conceito de entidade do JPA é representado pelo estereótipo *Entity*, aplicável às classes UML. Uma entidade representa uma classe na qual cada instância é persistente e tem um identificador único. Uma entidade pode ter um conjunto de propriedades persistentes que devem ter tipos segundo as limitações da especificação JPA.

O perfil MD-JPA, quando aplicado a um pacote ou diagrama, não atribui automaticamente nenhuma característica a seus componentes. Cada classe deve ser anotada individualmente como persistente, de forma que um mesmo diagrama possa conter classes persistentes e transientes, bem como seus relacionamentos.

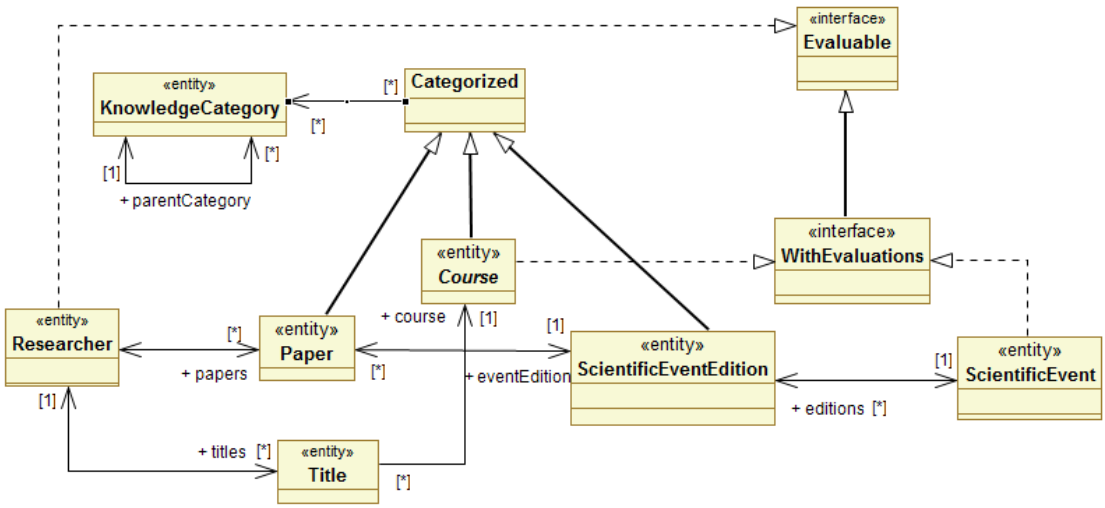


Figura 3.4. Diagrama de classes geral com persistência anotada.

A Figura 3.4 apresenta o diagrama de classes principal do sistema de avaliação da produção acadêmica, contendo as anotações que identificam as classes que representam entidades. Este diagrama também contém a classe não persistente *Categorized,* que se relaciona com outras classes persistentes que representam *Paper*, *Course* ou *ScientificEventEdition*.

Os conceitos representados por *Paper*, *Researcher*, *Course, ScientificEvent* e *ScientificEventEdition* são candidatos a classes persistentes, pois especificam informações que devem ser armazenadas sobre objetos da realidade. Da mesma forma, a árvore de categorias formada por *KnowledgeCategory* e seu auto-relacionamento também representam dados persistentes. No entanto, o elemento categorizável

(*Categorized*) mostra-se muito mais um artifício lógico do sistema do que um objeto com informações próprias, como será aprofundado mais adiante.

As interfaces *WithEvaluations* e *Evaluable* não podem ser persistentes no padrão JPA, mas são implementadas por classes persistentes. Completa o modelo a classe persistente *Title* que representa as titulações dos pesquisadores.

As propriedades de cada entidade são automaticamente consideradas persistentes, exceto quando marcadas com o estereótipo *Transient*. A mesma regra se aplica a todos os descendentes do estereótipo *Persistent*, do qual as entidades, classes embutidas e superclasses mapeadas fazem parte. Esta característica se baseia na especificação JPA, a qual também assume todas as propriedades destes elementos persistentes.

A especificação JPA define dois tipos de acesso ao conteúdo das propriedades de cada entidade: através de campos (*field*) ou de métodos (*method*). Cada hierarquia de entidades deve implementar o mesmo tipo de acesso, isto é, se uma determinada entidade possui acesso por campos, todas as entidades que a generalizam ou especificam devem ter também acesso por campos.

Utilizando anotações JPA, o tipo de acesso é determinado pela posição da primeira anotação da classe. Se a anotação estiver em um método, o acesso é via método, caso contrário, o acesso será via campo. Entretanto, se existirem anotações tanto nos métodos quanto nos campos dentro de uma mesma hierarquia, o mapeamento é inválido pela especificação. É fácil para um desenvolvedor desatento esquecer que a posição da primeira anotação na classe influencia no mecanismo de acesso utilizado pelo mapeamento objeto-relacional, introduzindo erros na aplicação.

No perfil MD-JPA o tipo de acesso às propriedades é definido no estereótipo de entidade, deixando mais claro que tipo de acesso será utilizado. Além disso, a utilização de tipos diferentes de acesso numa mesma hierarquia é verificada automaticamente, como será abordado no capítulo 5.

Independentemente do tipo de acesso, no JPA é esperado que cada propriedade apresente um par de métodos de acesso ("*getter*"e "*setter*"), responsáveis por ler e escrever o valor das variáveis, respectivamente. No MD-JPA, com o objetivo de permitir a construção de modelos mais simples, cada propriedade UML é considerada como uma propriedade no padrão *JavaBean* (SUN MICROSYSTEMS, 1997), que consiste em uma variável de instância privada com métodos de acesso públicos. Desta forma, não é necessário especificar no modelo, para cada propriedade, os métodos de acesso.

A Figura 3.5 apresenta um diagrama das classes focadas na estrutura utilizada para representar a revisão de artigos científicos. A entidade *Researcher* engloba tanto os autores, quanto os revisores dos artigos submetidos. Os revisores de um evento são representados pelo relacionamento entre *ScientificEventEdition* e *Researcher*. O sistema atribui artigos aos revisores criando entidades *Review,* que são atualizadas à medida que os revisores trabalham sobre os artigos. As classes *Paper*, *Title*, *ScientificEvent* e *Course* são as mesmas presentes no diagrama da Figura 3.4.

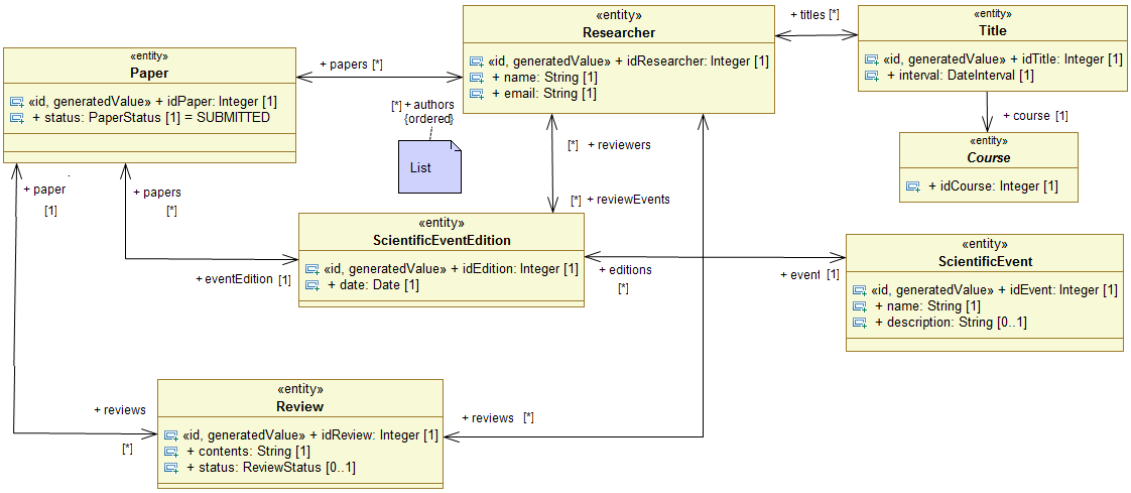


Figura 3.5: Diagrama para o registro da revisão de artigos.

O sistema pode utilizar as informações sobre artigos publicados e a lista de títulos vinculados a cursos para escolher, ou indicar, os revisores de cada artigo. Como os cursos e publicações são categorizados nas áreas de conhecimento, estes podem ser utilizados para definir quais pesquisadores são especialistas nas áreas de conhecimento abordadas no artigo em avaliação.

3.2.1 Detalhando a especificação de colunas e tabelas secundárias

Cada propriedade persistente pode ter um mapeamento de coluna marcado pelo estereótipo *Column*, habilitando uma descrição mais precisa para o mapeamento do tipo da coluna na base de dados. O objeto *ColumnDefinition* detalha o nome da coluna, precisão, escala e se valores nulos são permitidos, entre outras informações.

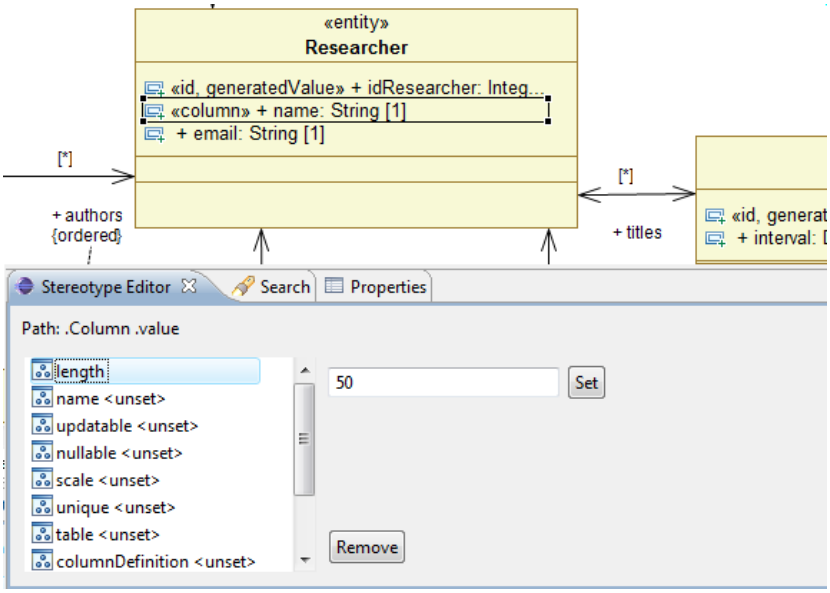


Figura 3.6. Exemplo do estereótipo *Column* no sistema de avaliação.

Na Figura 3.6, a classe que representa o pesquisador tem a propriedade nome com o estereótipo *Column*. Na sua definição de coluna é especificado o comprimento desejado do campo que representará a *string* na base de dados para 50 caracteres.

Entidades podem ter uma definição de tabela principal e tantas definições de tabelas secundárias (*SecondaryTable*) quantas forem necessárias. Desta forma, as entidades

podem ser usadas para definir informações sobre o mapeamento e geração do esquema de dados, tais como: nome, catálogo, esquema e restrições de unicidade. No caso de entidades representadas em mais de uma tabela, é possível definir a tabela de cada propriedade no estereótipo *ColumnDefinition*.

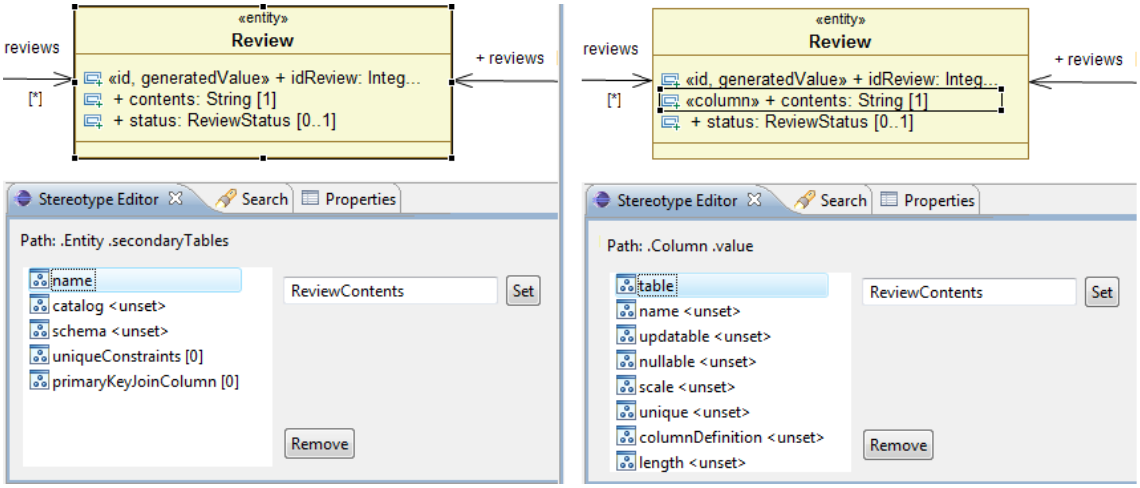


Figura 3.7: Definição de uma tabela secundária, e sua utilização por uma propriedade na classe *Review* do sistema de avaliação.

Na Figura 3.7 são apresentadas respectivamente a definição de uma tabela secundária e sua utilização por uma das propriedades. Instâncias da classe *Review*, que representam cada revisão de um artigo, terão as propriedades armazenadas na tabela principal (que representa a classe), com exceção da propriedade *contents* que é armazenada na tabela secundária *ReviewContents*. No MD-JPA, a definição da tabela secundária é realizada no estereótipo de entidade, definindo (no mínimo) o nome da tabela a se adicionar, como apresentado no lado esquerdo da figura. Em seguida, atribui-se o estereótipo *Column* na propriedade *contents*, especificando a tabela utilizada, como apresentado no lado direito da figura.

3.2.2 Estereótipos complementares para propriedades

O estereótipo *Basic* permite detalhar uma propriedade comum com informações específicas do padrão JPA. Pode-se definir a estratégia de busca (*fetch)* do banco de dados ou se o *framework* deve rejeitar valores nulos para a propriedade. Para propriedades que representam tipos binários ou textos longos, o estereótipo *Lob* representa a anotação de mesmo nome. O estereótipo *Version* identifica uma propriedade utilizada para o controle de versão otimista, equivalente à anotação de mesmo nome na especificação JPA.

Para mapear tipos que representam o tempo, foram definidos três estereótipos aplicáveis às propriedades. Em Java as classes mais comuns para representar o tempo (*Date* e *Calendar*) representam tanto a informação de data, como a de horário; já nos bancos de dados o padrão é especificar o tipo de informação temporal armazenada. Os estereótipos *Date*, *Time* e *TimeStamp* definem o mapeamento temporal de uma propriedade na base de dados.

No exemplo da Figura 3.8 são apresentadas duas classes que representam intervalos de tempo. A classe *DateTimeInterval* representa intervalos de tempo com granularidade de data e horário, utilizando o estereótipo *TimeStamp* para garantir que as propriedades de início e fim preservem as informações sobre a hora ao serem persistidas. Já a classe

DateInterval utiliza o estereótipo *Date* para que somente as informações sobre os dias de início e fim do intervalo sejam persistidas.

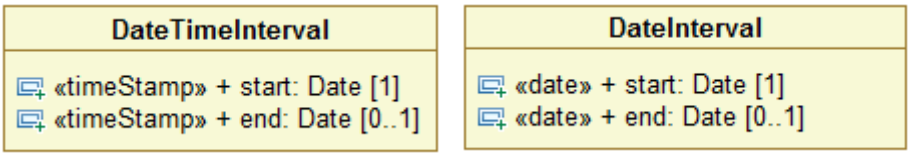


Figura 3.8. Estereótipos de tempo no sistema de avaliação.

3.3 Classes embutidas

Classes embutidas são utilizadas para representar parte do estado persistente de entidades. Elas não possuem uma identidade como as entidades e cada instância não pode pertencer a mais do que uma instância de entidade por vez. Classes embutidas são úteis para representar partes da informação que se repetem em vários componentes, mas que, por si só, não constituem uma entidade independente, permitindo o encapsulamento de uma lógica comum. Além disso, uma instância de classe embutida só é persistida quando atribuída a uma propriedade em uma entidade.

O estereótipo *Embeddable* identifica no modelo uma classe que pode ser embutida ("embutível") em entidades. Cada uma de suas propriedades pode ter o estereótipo *Column* descrevendo como elas podem ser mapeadas para colunas no banco de dados. As propriedades de entidades que referenciam classes "embutíveis" (ou seja, tem o tipo referenciando tal classe) devem ser marcadas com o estereótipo *Embedded*.

O diagrama da Figura 3.9 exemplifica o uso de entidades e classes embutidas, com um fragmento do diagrama de classes que representa um sistema de avaliação acadêmico. A classe *DateInterval* estende a classe *Interval* que representa um intervalo de tempo para a granularidade de um dia. Cada instância de *Evaluation* representa uma avaliação para um determinado período de tempo expresso em dias, período este que por sua vez é uma instância embutida de *DateInterval*.

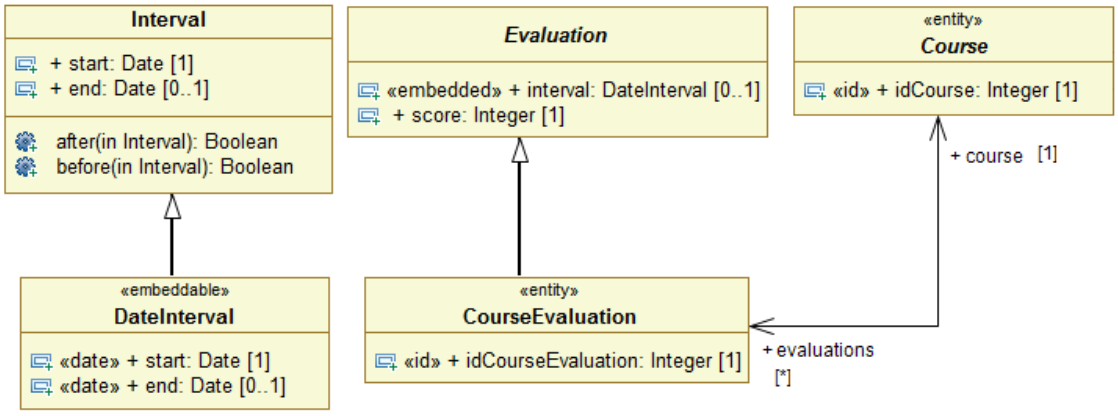


Figura 3.9: Diagrama exemplo com entidades e classes embutidas.

A entidade *CourseEvaluation* representa uma avaliação de curso (*Course*) implementando a classe abstrata *Evaluation* e herdando a propriedade *interval*. O mapeamento objeto-relacional correspondente de avaliação de curso teria duas colunas do tipo data (Date em SQL) em sua tabela: *start* e *end*.

3.4 Identificadores compostos

Toda hierarquia de entidade deve ter uma chave primária identificadora, definida na entidade raiz (ou topo) da árvore de hierarquia. Segundo a especificação JPA, chaves primárias simples são representadas como atributos anotados, enquanto que chaves compostas devem ter uma classe separada, na qual cada propriedade representa uma coluna da chave. Neste caso, é possível especificar uma propriedade que faz referência à classe embutida ou um conjunto de propriedades que faz referência à cada uma das propriedades da classe embutida que representa a chave primária.

Os seguintes estereótipos são utilizados para definir chaves primárias:

- *Id*: é utilizado em uma propriedade da entidade para identificar uma chave simples, ou em um conjunto de propriedades para identificar chaves compostas. Quando utilizada para identificar chaves compostas, a classe deve ter o estereótipo *IdClass*.
- *IdClass*: especifica que a entidade possui uma chave composta, e que um conjunto de propriedades marcadas referencia as propriedades de uma classe de chave primária identificada no estereótipo.
- *EmbeddedId*: é utilizado em uma única propriedade da entidade para identificá-la como uma chave composta que referencia uma classe embutida.
- *GeneratedValue*: define uma propriedade como tendo valor gerado automaticamente. É possível definir uma estratégia de geração ou deixar esta decisão para o *framework*.
- *TableGenerator* e *SequenceGenerator*: são dois estereótipos aplicáveis a classes que detalham uma tabela ou sequência utilizada para a estratégia de geração de valores. Seu uso também é opcional.

No sistema de avaliação optou-se pela utilização de chaves simples para identificar todas as entidades do sistema, como exemplificado na Figura 3.10. Para demonstrar o mapeamento de chaves compostas será utilizado um fragmento do sistema que mostra a relação entre eventos científicos (*ScientificEvent*) e edições de eventos científicos (*ScientificEventEdition*).

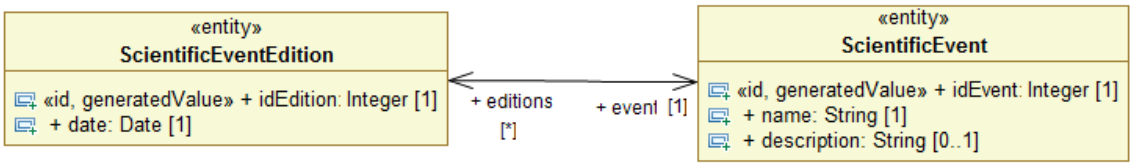


Figura 3.10: Eventos científicos mapeados com chave simples.

O mapeamento de chave composta requer a definição de uma classe que represente a chave primária da entidade. Esta classe deve ser relacionada com a entidade, através da utilização do estereótipo *IdClass*, ou da definição de uma propriedade embutida *EmbeddedId*.

Na Figura 3.11, a classe *ScientificEventEdition* possui uma chave composta pelas propriedades *idEdition* e *idScientificEvent*, marcadas pelo estereótipo *Id*. A classe *ScientificEventEditionPK* representa a instância de cada chave primária, e deve ter as mesmas propriedades especificadas como parte da identidade da entidade. Finalmente, o estereótipo *IdClass* é aplicado na entidade para identificar a classe *ScientificEventEditionPK* como sua chave primária.

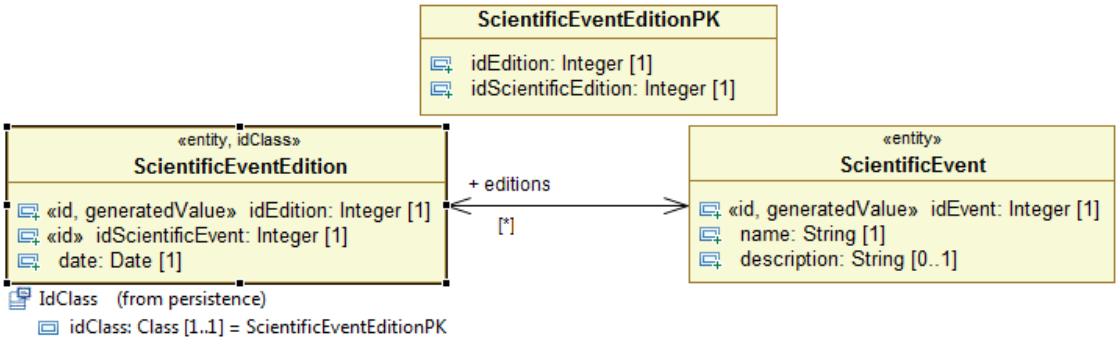


Figura 3.11: Eventos científicos, se usassem chave composta mapeada com *IdClass*.

Na Figura 3.12, a mesma chave composta é representada através de uma propriedade que tem uma classe embutível como tipo. O estereótipo *embeddedId* é utilizado na propriedade *idEdition*, que declara como seu tipo a classe *ScientificEventEditionPK*.

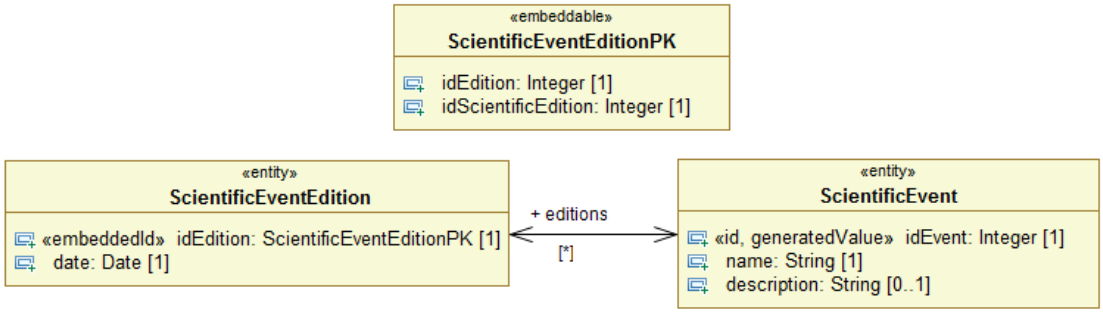


Figura 3.12: Eventos científicos, se usassem chave composta mapeada com *EmbeddedId*.

A utilização de chave embutida tem como principal diferencial estabelecer o acesso dos valores que compõe a chave através de uma instância da chave primária, enquanto que ao se utilizar *IdClass*, as propriedades que fazem parte da chave estão diretamente representadas na classe. Como consequência, não é suportada a utilização da estratégia de geração de valores.

3.5 Relacionamentos

No padrão JPA, relacionamentos entre entidades são persistentes e devem receber uma das seguintes anotações, de acordo com a cardinalidade da relação: *um para um*, *muitos para um*, *um para muitos* ou *muitos para muitos*. As anotações são atribuídas nas propriedades que referenciam a entidade relacionada, ou uma coleção de entidades relacionadas. Os relacionamentos também podem ser unidirecionais ou bidirecionais, dependendo da existência de uma propriedade declarada como inversa em uma das entidades relacionadas.

Na UML, o relacionamento é representado pelo elemento associação (*Association*), que referencia uma propriedade de cada classe relacionada. A cardinalidade da relação é armazenada em cada propriedade, através dos valores superior e inferior (*upperValue* e *lowerValue*) atribuídos a cada propriedade. O tipo de cada propriedade referencia a classe oposta da associação.

Na modelagem MD-JPA, os relacionamentos entre classes marcadas como entidades são persistentes. As anotações de cardinalidade JPA não precisam ser explícitos, pois a informação de cardinalidade já é expressa pela UML. Entretanto, a UML permite a

definição de valores arbitrários para cardinalidade, enquanto que os bancos de dados relacionais somente reconhecem as cardinalidades zero, um ou muitos. Por este motivo, todas as cardinalidades acima de *um* representam, no escopo do MD-JPA, um relacionamento de "para muitos".

Outro aspecto que foi abordado pela MD-JPA é o tipo a ser utilizado pela propriedade que representa este relacionamento "para muitos". O padrão JPA oferece duas alternativas, a utilização de coleções com tipo parametrizável ou a utilização de coleções sem tipo parametrizável, mas que obriga a declaração do tipo na anotação de relacionamento. Porém, a UML tem uma abordagem diferente para representar tipos parametrizáveis, através da declaração de *templates*.

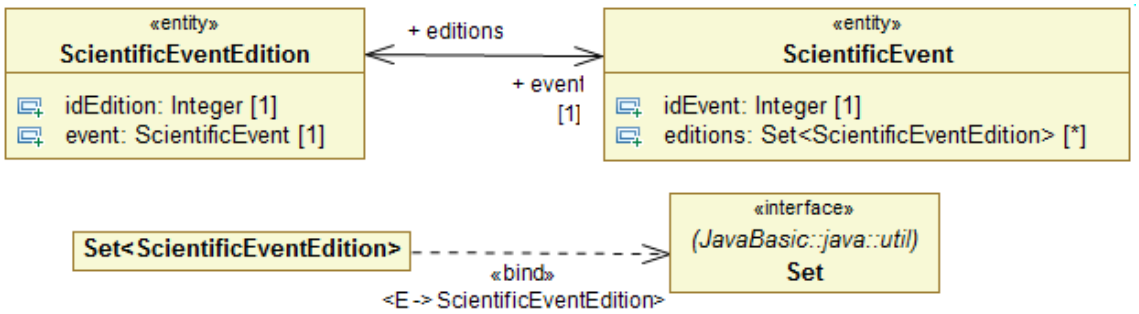


Figura 3.13: Exemplo de relacionamento, utilizando *templates*.

A Figura 3.13 apresenta um exemplo do que seria a modelagem de relacionamento utilizando apenas *templates*. Primeiramente é definida uma classe de ligação que implementa o tipo de conjunto escolhido (*Set*) para a classe parâmetro (*ScientificEventEdition*), através do recurso de ligação (*bind*) da UML. Esta classe de ligação é então utilizada como tipo da propriedade que representará a coleção de edições de evento. Esta abordagem, apesar de correta no ponto de vista de modelagem, leva a criação de uma classe de ligação que geralmente não é importante para a representação do sistema.

Com o objetivo de simplificar a modelagem, o MD-JPA assume que todos os relacionamentos "para muitos" utilizarão coleções parametrizáveis. Desta forma, elimina-se a necessidade de se criar classes que representam as coleções parametrizáveis ligadas a um determinado tipo. O parâmetro para esta coleção é o tipo da classe relacionada, e o tipo da coleção é especificado no estereótipo *AssociationMapping*, associado à propriedade. Este estereótipo é opcional, e sua omissão significa que a coleção é comum (interface *Collection* do Java).

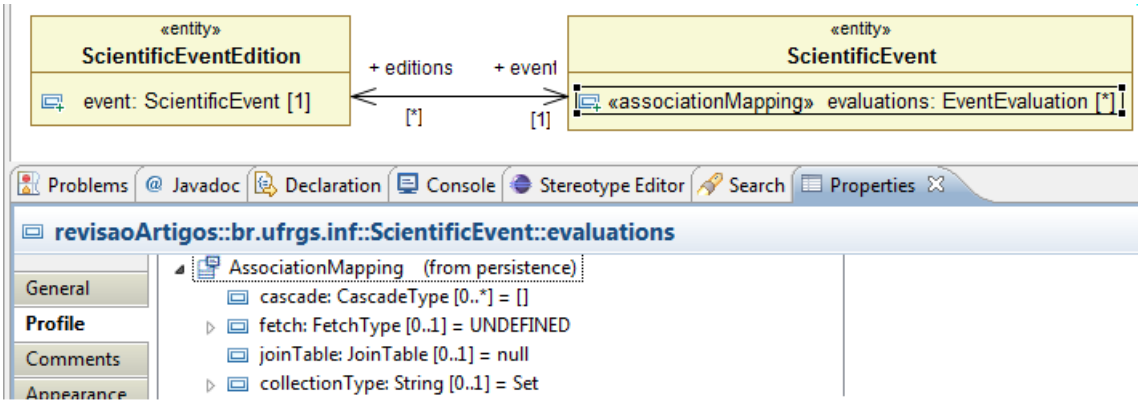


Figura 3.14: Exemplo de relacionamento persistente, utilizando o MD-JPA.

A Figura 3.14 mostra o mesmo relacionamento anterior, mas utilizando o estereótipo *AssociationMapping* do MD-JPA. O diagrama é focado no relacionamento entre as entidades e a classe de ligação não precisa ser definida, já que o tipo de coleção é especificado diretamente no estereótipo.

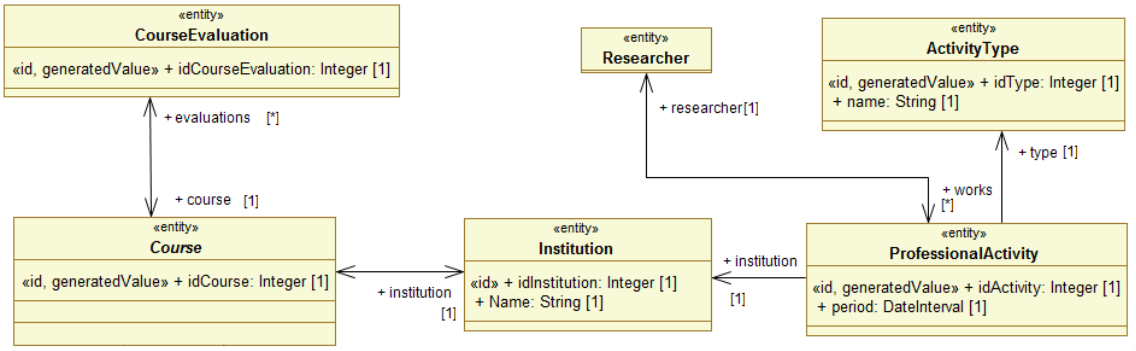


Figura 3.15: Exemplo com relacionamento unidirecional.

A Figura 3.15 introduz as classes ligadas as atividades profissionais de um pesquisador, exemplificando o mapeamento de relacionamentos. A entidade *Institution* representa instituições ligadas à pesquisa e ensino, podendo oferecer cursos, que são categorizados pelas áreas de conhecimento. A relação de trabalho entre um pesquisador e uma instituição é representada por *AtividadeProfissional* (*ProfessionalActitivy*), identificando um período de tempo (*period*) e um tipo de atividade, como por exemplo *professor titular*.

A partir de uma instância de atividade profissional pode-se navegar para as instâncias de *Researcher*, *ActivityType* e *Institution* através das propriedades *researcher*, *type* e *institution,* que dão nome às pontas dos relacionamentos. A partir de uma instância de pesquisador, pode-se navegar, no sentido inverso da propriedade *researcher*, para o conjunto de atividades profissionais já exercidas pelo pesquisador através da propriedade *works*.

Os relacionamentos de atividade profissional para tipo de atividade e instituição são unidirecionais. Na modelagem de exemplo não há uma propriedade que realize a navegação entre uma instância de instituição e os profissionais que exercem alguma atividade na instituição. Porém, o modelo relacional permite realizar consultas que busquem estas instâncias, já que o conceito de direção de relacionamento é exclusivo do paradigma OO.

3.5.1 Estereótipos para o mapeamento de relacionamentos

Os tipos de coleção definidos no estereótipo *AssociationMapping* podem ser de conjunto (*Set*), lista (*List*) ou mapeamento (*Map*), tal como definido no padrão JPA. No caso de coleções de mapeamento, a propriedade representa um conjunto de pares de chaves únicas e valores que referenciam uma instância da classe alvo. A chave da coleção pode ser uma propriedade especificada no valor *mapKey* do *AssociationMapping,* ou, quando nenhum valor é atribuído a *mapKey*, o identificador definido para a classe.

Como cada relacionamento possui duas extremidades, o estereótipo *AssociationMapping* é aplicável às propriedades que participam do relacionamento, e não ao elemento de relacionamento que faz a ligação entre as propriedades. Os motivos para isto são que o JPA utiliza anotações associadas às propriedades e a maior parte das

informações sobre o relacionamento são particulares de cada uma destas propriedades. Neste caso, a utilização dos estereótipos diretamente nas pontas do relacionamento simplifica a visualização de suas informações e o mapeamento para classes e anotações Java.

O estereótipo *AssociationMapping* (Figura 3.1) também permite definir o tipo de cascateamento, colunas de junção, estratégia de busca (*fetch*) e tabelas de junção. As tabelas de junção são aplicadas principalmente para implementar relacionamentos de muitos para muitos no banco de dados. Elas também são utilizadas para relacionamentos unidirecionais com cardinalidade de muitos, conforme a especificação JPA. A Figura 3.16 mostra a estrutura das classes relacionadas a definição de tabelas no perfil MD-JPA.

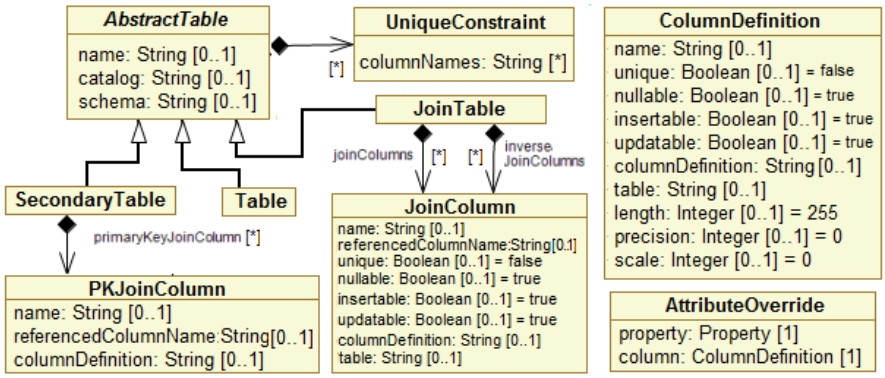


Figura 3.16: Meta-classes que detalham informações sobre tabelas no perfil MD-JPA.

A propriedade colunas de junção (*joinColumns*) de *AssociationMapping* pode conter um ou mais mapeamentos que detalham como a chave estrangeira do relacionamento deve ser gerada. Cada mapeamento é uma instância da meta-classe *JoinColumn* utilizada também nas tabelas de junção, quando necessárias. Suas propriedades correspondem às da anotação de mesmo nome na especificação JPA. Se a chave estrangeira faz parte da chave primária, a propriedade *pkJoinCols* deve ser utilizada para especificar esses mapeamentos. Neste caso, cada mapeamento é uma instância de *PKJoinColumn*, equivalente à anotação *PrimaryKeyJoinColumn*.

A tabela de junção do relacionamento pode ser especificada no modelo, representado com uma instância da meta-classe *JoinTable*. Os três tipos de especificação de tabelas do JPA foram implementados em uma mesma hierarquia, cuja raiz é a classe abstrata *AbstractTable*, que contém as propriedades comuns a todas as tabelas: nome, catálogo e esquema. Além disso, pode agregar um conjunto de restrições de unicidade. *JoinTable* é uma especialização desta classe para tabelas que representam relacionamentos, especificando detalhes para as colunas da tabela que possuem o relacionamento (isto é, a ponta onde está definida a instância de *JoinTable*) e para as colunas inversas que estão na outra ponta do relacionamento.

O estereótipo *OrderBy* pode ser aplicado às propriedades que fazem parte de relacionamentos persistentes e tem cardinalidade maior que um, para se especificar que a coleção de elementos deve ser ordenada. Tal qual a anotação de mesmo nome na especificação do JPA, pode-se definir uma expressão de ordenação. Se esta expressão não for informada, a ordenação será pela chave primária da entidade associada.

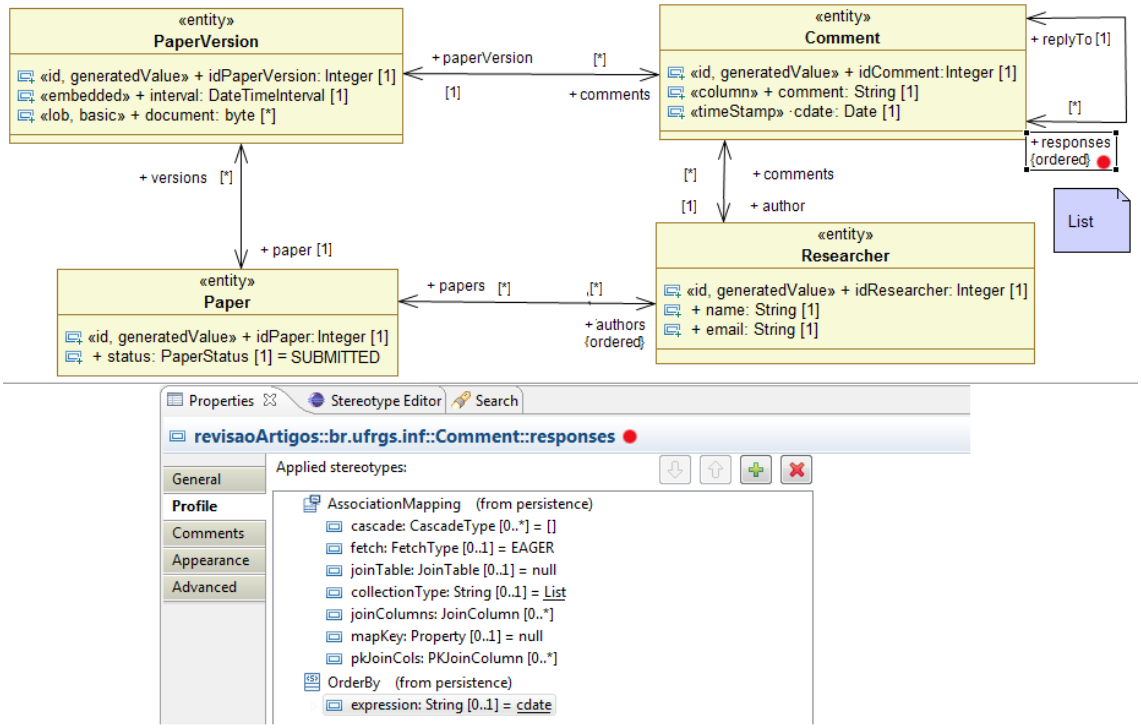


Figura 3.17: Exemplo de utilização de *AssociationMapping* para definir uma lista.

A Figura 3.17 mostra a aplicação dos estereótipos *AssociationMapping* e *OrderBy* no exemplo do sistema de avaliação acadêmico. A classe comentário tem um auto-relacionamento no qual a propriedade *responses* é uma lista persistente de comentários de resposta, ordenada pela coluna *cdate*. A coleção *responses* é definida como lista ordenada através da propriedade *collectionType,* quando atribuída com o valor *List*. Já a ordenação é definida pela propriedade *expression* do estereótipo *OrderBy*.

Tabela 3.2: Anotações JPA e classes derivadas utilizadas no MD-JPA.

JPA	***Classe do perfil***	***Atributo de estereótipo***	***Seção***
@JoinColumn	JoinColumn	AssociationOverride.joinColumn, AssociationMapping.joinColumns, JoinTable.joinColumns/inverseJoinColumns	3.5
@Column	ColumnDefinition	Column.value, AttributeOverride.column	3.1
@Table	Table	Entity.table	3.1
@SecondaryTable	SecondaryTable	Entity.secondaryTables	3.1
@JoinTable	JoinTable	AssociationMapping.joinTable	3.5
@AttributeOverride	AttributeOverride	AttributeOverrides.value	3.8
@UniqueConstraint	UniqueConstraint	AbstractTable.uniqueConstraints, TableGenerator.uniqueConstraints	3.5
@MapKey	UML::Property	AssociationMapping.mapKey	3.5
@DiscriminatorValue	DiscriminatorValue	DiscriminableGeneralization.discriminatorValue	3.6
@PrimaryKeyJoinColumn	PKJoinColumn	AssociationMapping.pkJoinCols, SecondaryTable.primaryKeyJoinColumn	3.5
@AssociationOverride	AssociationOverride	AssociationOverride.overrides	3.8

A Tabela 3.2 lista as anotações que não são mapeadas diretamente como estereótipos no perfil MD-JPA, tais como tabelas de junção e suas colunas. Essas anotações estão

definidas em meta-classes, instanciadas como propriedades de estereótipos ou propriedades de outras meta-classes do perfil. Quando são instâncias atribuídas a propriedades de meta-classes, sempre há um caminho que liga as instâncias a algum estereótipo. Por exemplo, uma restrição de unicidade é uma instância da meta-classe *UniqueConstraint*, e pode estar associada a uma tabela de junção. Esta tabela de junção está então associada ao estereótipo *AssociationMapping* aplicado a uma propriedade. Os possíveis caminhos para se chegar às instâncias que representam uma determinada anotação estão representados na coluna 3 em "Atributo de estereótipo".

3.5.2 Agregação e Composição

Na especificação JPA, agregações e composições seguem as mesmas regras das associações, não havendo nenhuma anotação específica para sua representação. Já na UML, cada propriedade pode ser marcada como agregação ou composição, através do atributo *aggregation*. Caso a propriedade faça parte de uma associação, esta recebe um sinal gráfico em forma de losango vazado para agregações ou preenchido, para composição.

Todavia, a agregação na notação UML contém pontos de variação semântica, ou seja, não há uma definição precisa sobre como uma agregação deve se comportar, pois ela varia conforme a área de aplicação. A composição, por outro lado, requer que a classe composta seja responsável por instanciar e armazenar suas partes, mas mantém como ponto de variação semântica a questão da ordem de criação das partes.

No perfil MD-JPA escolheu-se por não atribuir nenhuma semântica adicional às agregações e composições utilizadas em associações, tratando-as como relacionamentos normais. Da mesma forma que no relacionamento comum, pode-se utilizar os estereótipos *AssociationMapping*, *OrderBy* e seus derivados para detalhar o mapeamento objeto-relacional utilizado.

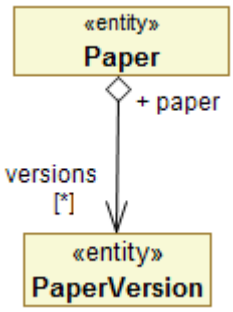


Figura 3.18: Exemplo de relação de agregação.

A Figura 3.18 exemplifica o uso da agregação no modelo de avaliação. A relação entre artigo (*Paper*) e suas versões (*PaperVersion*) é uma agregação, pois as versões não podem existir independentemente do artigo. Na prática, o modelo equivale a um relacionamento um para muitos (um artigo para muitas verões).

3.6 Herança

Uma entidade pode generalizar ou especializar outra classe. Quando a superclasse é também uma entidade, existem três estratégias possíveis para o mapeamento objeto-relacional no JPA. Para cada estratégia, o perfil MD-JPA oferece um estereótipo que estende o elemento de generalização (*Generalization*) da UML, como mostra o

diagrama da Figura 3.19. Se nenhum estereótipo é associado a uma relação de generalização, a estratégia adotada é a de tabela única, especificada como padrão no JPA.

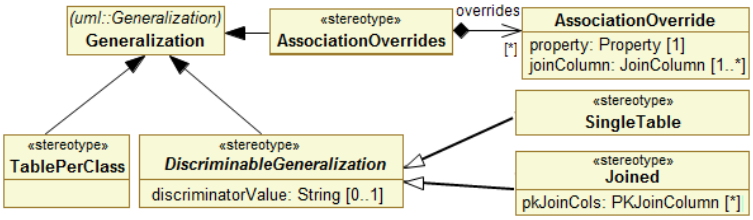


Figura 3.19: Estereótipos relacionados à herança no MD-JPA.

O estereótipo *SingleTable* representa o mapeamento de uma hierarquia de classes persistentes para uma única tabela, com tantas colunas quantas forem a soma de todas as propriedades persistentes, associações persistentes entre classes e o discriminador. O discriminador é uma coluna especial que identificará o tipo de cada instância armazenada na tabela, mas que não será mapeado como propriedade de uma classe.

Cada classe especializada pode ter definido seu valor discriminador no estereótipo, ou a decisão pode ser deixada para o *framework* MOR. A solução de tabela única tem como vantagens evitar a realização de junções entre tabelas para ler instâncias de classes e propiciar um bom suporte para consultas polimórficas. Em contrapartida, esta abordagem pode levar a tabelas esparsas, com número excessivo de colunas e muitos valores nulos, o que pode prejudicar a performance em certos bancos de dados.

O estereótipo *TablePerClass* representa um mapeamento no qual cada classe concreta tem uma tabela separada. As propriedades comuns da superclasse são repetidas em cada tabela, de forma que o *framework* de persistência não necessite realizar junções para recuperar uma instância da base de dados. Entretanto, para traduzir consultas polimórficas no ambiente relacional, são utilizadas operações de união (*union*), que podem consumir mais recursos de processamento pelo SGBD.

O estereótipo *Joined* também representa uma estratégia na qual existe uma tabela para cada classe, mas as propriedades da superclasse são armazenadas na tabela que a representa. Para recuperar uma instância na hierarquia, o *framework* precisa realizar junções entre as tabelas, com a exceção de consultas realizadas na classe raiz desta árvore. Estas junções podem ter um impacto negativo na performance do banco de dados, mas por outro lado as consultas polimórficas são facilitadas sem incorrer em tabelas esparsas.

Na especificação JPA, pode-se definir uma coluna discriminadora através da anotação *DiscriminatorColumn*, no caso das estratégias de herança *TablePerClass* e *SingleTable.* Quando a estratégia de tabela única é utilizada, sempre haverá uma coluna discriminadora, mesmo que o usuário deixe de especifica-la. Já na estratégia de tabela por classe, a coluna discriminadora estará presente conforme opção do usuário.

No perfil MD-JPA foi definido o estereótipo *DiscriminatorColumn* para representar a coluna discriminadora. O estereótipo pode ser aplicado na superclasse, e no caso da estratégia *TablePerClass* indica que o sistema deve utilizar uma coluna discriminadora. As propriedades de *DiscriminatorColumn* não são obrigatórias, permitindo ao usuário especificar o nome e o tamanho da coluna, o tipo de discriminador ou a definição em SQL para sua criação, segundo a especificação da anotação homônima em JPA. Para o

tipo de discriminador, foi criada uma enumeração que representa as mesmas opções (*STRING, CHAR* e *INTEGER*) previstas na especificação.

A hierarquia de estereótipos, definida na Figura 3.19, propõe o estereótipo abstrato *DiscriminableGeneralization,* que generaliza as estratégias *Joined* e *SingleTable*. A propriedade *DiscriminatorValue* permite especificar os valores discriminadores atribuídos à coluna discriminadora em cada especialização. Por exemplo, a classe *Graduation* (Figura 3.20) pode ter como valor discriminador a letra G.

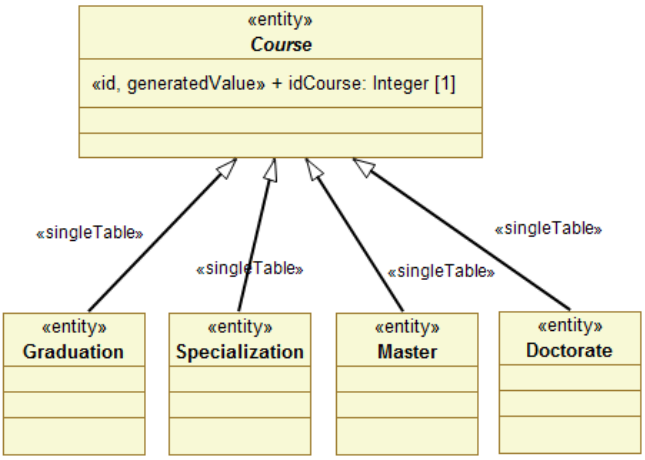


Figura 3.20: Especializações de curso, utilizando estratégia de tabela única.

A Figura 3.20 ilustra um fragmento de diagrama que representa as classes concretas que implementam cursos (*Course*) no sistema de avaliação, tais como graduação, especialização e doutorado (*Graduation*, *Specialization* e *Doctorate*). Ao utilizar a estratégia *singleTable* uma única tabela de curso deve armazenar todas as informações sobre todas as especializações de curso. Além disso, a tabela deve conter uma coluna discriminadora que identifica a qual entidade cada tupla pertence.

3.7 Enumerações

O conceito de enumeração da UML representa um conjunto finito de valores definidos pelo usuário (OMG, 2009). No JPA, os valores enumerados devem ser mapeados para valores ordinais ou literais, que possam ser armazenados em um dos tipos de dados padrão dos bancos de dados.

O MD-JPA especifica o estereótipo *Enumerated* para ser utilizado em propriedades que referenciam uma enumeração pelo tipo, permitindo especificar o tipo de mapeamento utilizado para cada entidade que referencia a enumeração. Os tipos de mapeamento são definidos em uma enumeração UML, que contém os valores *ordinal* e *string*. A Figura 3.21 apresenta o estereótipo *Enumerated* e a enumeração de tipos de mapeamento *EnumType*.

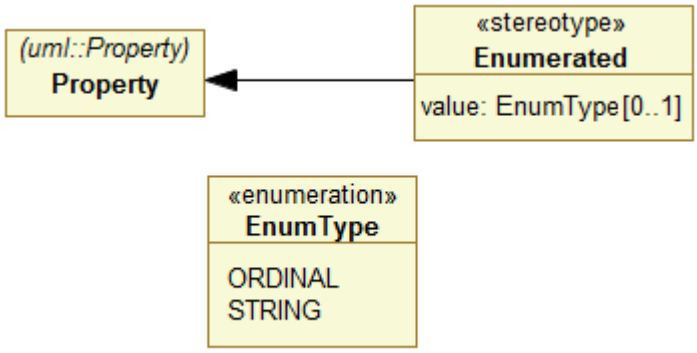


Figura 3.21: Definição de enumeração no perfil MD-JPA.

O recurso de enumeração é bastante utilizado na identificação dos estados pelos quais um determinado objeto pode passar. No exemplo da avaliação acadêmica, os artigos passam por vários estados que determinam quais funções estão disponíveis no sistema. A Figura 3.22 apresenta a propriedade *status* da classe *Paper* que referencia o tipo enumerado *PaperStatus*.

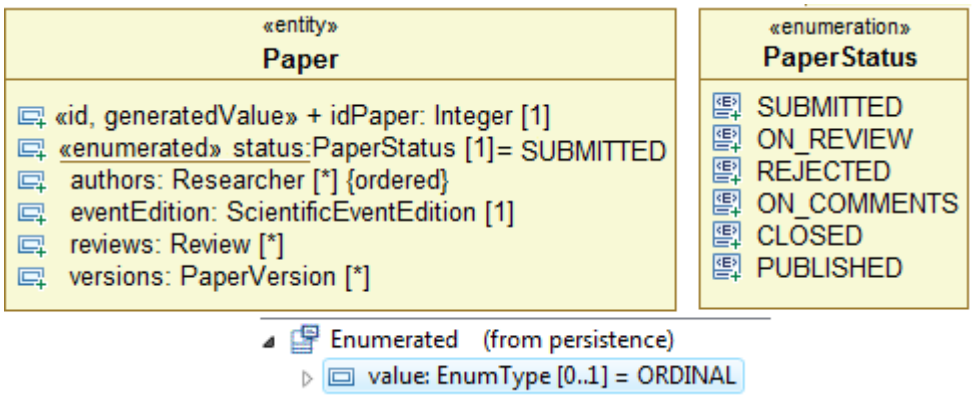


Figura 3.22: Exemplo de enumeração no sistema de avaliação.

O status representa o estado do artigo, sendo inicializado para o estado submetido (*SUBMITTED*). A estratégia definida no estereótipo é ordinal, o que significa que cada literal da enumeração *PaperStatus* será representado por um número inteiro conforme a ordem representada no modelo. O estado *SUBMITTED* será representado pelo valor zero, o estado em revisão (*ON_REVIEW)* pelo valor um, e assim sucessivamente.

A Figura 3.23 mostra, através de uma máquina de estados UML, a atribuição dos valores à propriedade persistente *status* da classe *Paper*. No estado submetido (*SUBMITTED*), os dados e texto do artigo podem ser alterados pelos autores, sem registro de histórico. Quando são iniciadas as revisões, o artigo passa para o estado *On Review* no qual não pode ser alterado pelos autores. Cada revisor faz sua avaliação, que é registrada na propriedade status da classe *Review*.

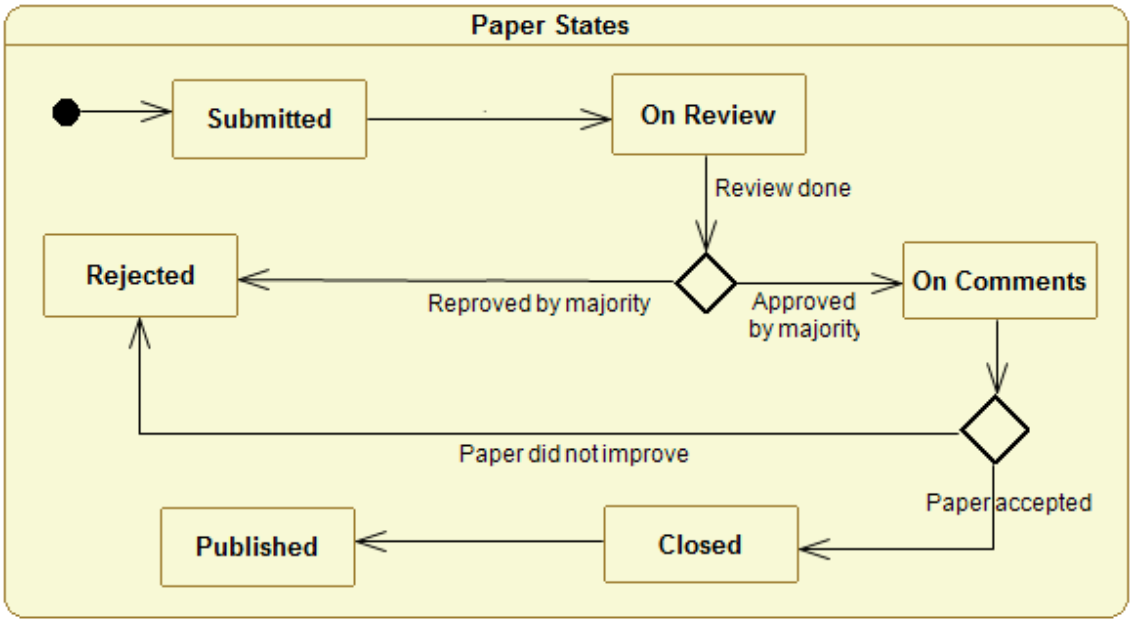


Figura 3.23: Estados possíveis para propriedade status.

Se ao fim da revisão a maioria dos revisores aprovar o artigo, ele passa para o estado Sob-comentários (*ON COMMENTS*). Neste estado, os autores podem realizar mudanças no artigo, registradas em entidades *PaperVersion* (apresentadas na Figura 3.17), e se comunicar com a comunidade acadêmica por um determinado período de tempo. Se o artigo atender aos requisitos de publicação, segundo a opinião dos revisores, ele passa para o estado fechado (*CLOSED*). Finalmente, após a publicação do artigo em anais de conferência ou periódico, o artigo passa ao estado final Publicado (*PUBLISHED*).

O diagrama de estados não utiliza nenhum recurso do perfil MD-JPA, mas complementa a especificação da enumeração, documentando o propósito desta enumeração no contexto da entidade *Paper*. Este tipo de informação pode ser difícil de se obter diretamente do código fonte da aplicação, mesmo quando bem documentado.

3.8 Mapeamento de superclasse não persistente

Pelo padrão JPA, propriedades de uma entidade que são herdadas de uma classe transiente (isto é, uma classe que não é uma *entidade*) não são persistidas. Para permitir a persistência de propriedades nas entidades que especializam esta superclasse, o JPA oferece o recurso de mapeamento de superclasse.

No perfil MD-JPA este recurso é representado pelo estereótipo *MappedSuperclass*. Este estereótipo pode ser aplicado a uma classe transiente para definir como será feito o mapeamento de suas propriedades em entidades que venham a especializar esta classe. Assim sendo, instâncias de uma *MappedSuperclass* não são persistidas, mas instâncias de entidades que a especializam, terão as propriedades herdadas da superclasse mapeada persistidas.

A Figura 3.24 ilustra a utilização de uma superclasse mapeada no contexto do sistema de avaliação científica, exemplificando as relações de herança entre classes persistentes e transientes. A classe *Categorized* representa os objetos que podem ser categorizados dentro da hierarquia de áreas de conhecimento da ciência, ou seja, uma instância de categorizado é um conjunto nomeado de áreas do conhecimento. As classes

Paper e *ScientificEventEdition* são entidades que representam artigos e edições de eventos científicos, que herdam a habilidade de serem categorizados. Como a superclasse é mapeada, a informação do relacionamento com as áreas de conhecimento, bem como o atributo nome, serão persistentes nas entidades que representam artigos e eventos, apesar de não serem persistentes em instâncias da própria classe *Categorized*.

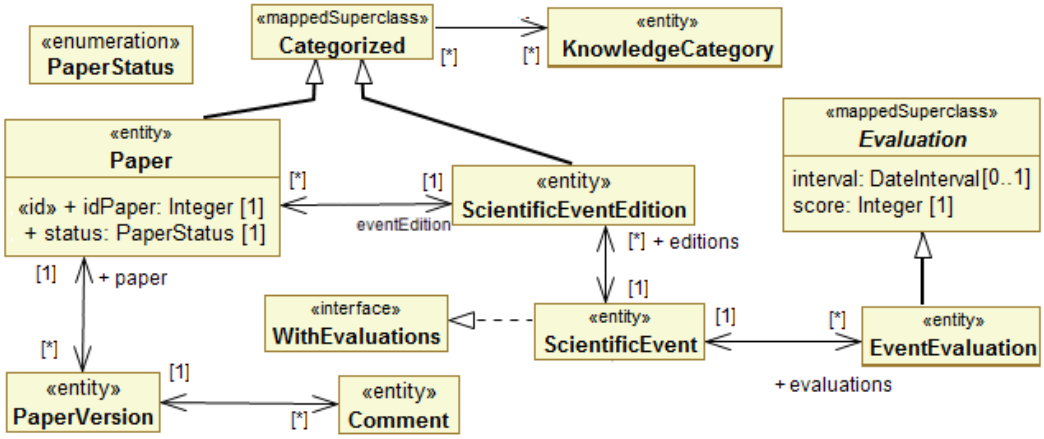


Figura 3.24: Diagrama do sistema de avaliação utilizando superclasse mapeada.

O banco de dados gerado pelo *framework* JPA terá uma tabela representando os artigos e outra para as edições de eventos científicos, mas não haverá nenhuma relação entre as duas estruturas. O conceito de categorizado, da classe *Categorized*, não existirá no banco de dados, e cada uma das duas tabelas (artigos e edições de evento) terá um relacionamento com a tabela que representa as áreas de conhecimento, implementado através de uma tabela de junção para artigos e outra para edições (uma vez que trata-se de um relacionamento de muitos para muitos).

Sob o ponto de vista do sistema, a classe *Categorized* é concreta e pode ser instanciada para utilização em partes transientes do sistema. Por exemplo, pode-se criar uma instância de *categorizado* para representar um conjunto de áreas do conhecimento. Esta instância representa uma seleção de áreas realizada por um usuário, que não deve ser armazenada na base de dados. Ainda assim, esta categorização pode estar referenciando áreas do conhecimento persistentes. A vantagem, neste caso, está na possibilidade de centralizar todos os serviços comuns para conjuntos de categorias na classe principal, tais como busca ou checagem de inconsistências. Tais serviços, implementados como métodos, por exemplo, serão herdados pelas classes persistentes *Paper* e *ScientificEvent*.

A avaliação (*Evaluation*) é outro exemplo de superclasse com mapeamentos, neste caso uma classe abstrata. A avaliação define os elementos comuns das avaliações, o intervalo de tempo expresso em dias e uma nota (score). Entretanto, fica ao encargo das especializações a implementação da relação com as entidades avaliadas. Por exemplo, a entidade *EventEvaluation* representa uma avaliação de um evento, e tem um relacionamento com a entidade evento científico. Outra entidade que implementa a avaliação é a *CourseEvaluation*.

3.9 Sobrescrita de mapeamento

A sobrescrita de mapeamentos é a redefinição de um mapeamento especificado em uma superclasse ou classe embutível. Desta forma, é possível alterar o mapeamento pré-definido em uma classe transiente, atendendo a alguma especificidade da entidade que

realiza a persistência dos elementos mapeados. Além disso, a sobrescrita pode ser utilizada para definir o mapeamento de atributos que não foram mapeados nas superclasses.

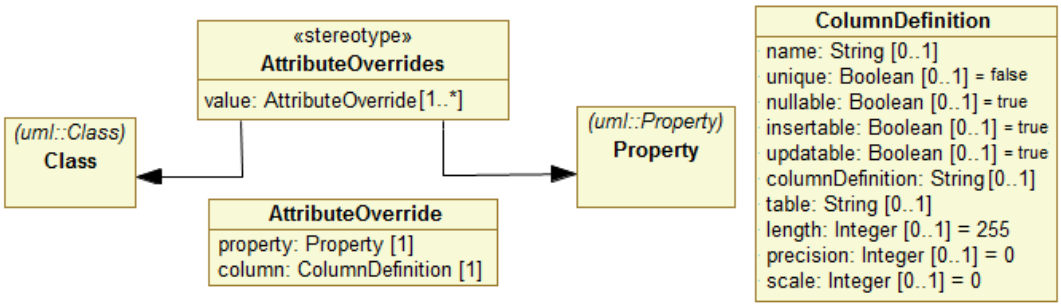


Figura 3.25: Recursos do MD-JPA utilizados na sobrescrita de mapeamento.

Na Figura 3.25, o estereótipo *AttributeOverrides* pode ser aplicado às classes ou propriedades, permitindo a sobrescrita de um conjunto de mapeamentos de uma ou mais propriedades. Cada mapeamento de propriedade é um objeto *AttributeOverride* que relaciona uma propriedade com uma definição de coluna (uma instância da meta-classe *ColumnDefinition*). Ainda que nenhuma definição explícita tenha sido feita na superclasse, os estereótipos de sobrescrita podem ser utilizados para alterar o mapeamento padrão.

Nos exemplos das Figuras 3.26 e 3.27 é apresentada uma classe embutível para intervalos de datas (*DateInterval*), que estende uma classe comum de intervalo de tempo. As figuras apresentam duas maneiras de redefinir as propriedades *start* e *end* para que suas informações sejam persistidas na colunas *startDate* e *endDate*. Na Figura 3.26, o estereótipo *AttributeOverrides* é aplicado na classe *DateInterval*. Na Figura 3.27 o estereótipo é utilizado em cada propriedade sobrescrita.

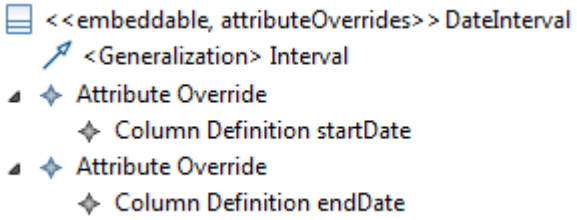


Figura 3.26: Sobrescrita do mapeamento de propriedade herdada, na classe.

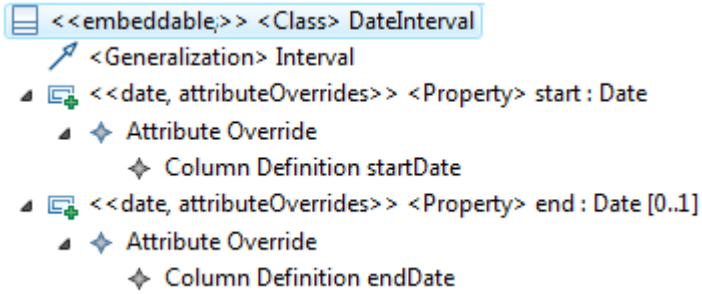


Figura 3.27: Sobrescrita do mapeamento de propriedade herdada, nos atributos.

A utilização do estereótipo *AttributeOverrides* na classe permite redefinir o mapeamento de propriedades que foram definidas apenas na superclasse. Por exemplo, na Figura 3.26 a classe de especialização não possui nem sobrescreve nenhuma

propriedade. Já no caso em que a classe sobrescreve as propriedades, o estereótipo deve ser aplicado em cada uma dessas propriedades.

Outro tipo de sobrescrita de mapeamento é a redefinição de uma associação feita em uma superclasse. O estereótipo *AssociationOverrides* estende a relação de generalização da UML, permitindo esta sobrescrita pela especificação de um conjunto de objetos da meta-classe *AssociationOverride*, que representa a sobrescrita de uma propriedade associativa. Cada instância de *AssociationOverride* referencia uma propriedade associativa em uma superclasse da hierarquia, atribuindo um conjunto de instâncias da classe *JoinColumn* que redefinem as colunas de junção.

3.10 Tipos Parametrizáveis em relacionamentos

Na Figura 3.13 foi apresentado um exemplo de como são modeladas coleções parametrizadas na UML, através do recurso de *template*. No caso dos relacionamentos persistentes, o perfil MD-JPA dispensa a utilização deste recurso ao assumir que os relacionamentos são mapeados como coleções parametrizadas. Porém, existem casos nos quais os tipos parametrizáveis são ferramentas importantes para tirar melhor proveito do polimorfismo no sistema. Para exemplificar a utilização dos tipos parametrizáveis, será apresentado um exemplo aplicado na avaliação de eventos e cursos.

A avaliação dos cursos e eventos científicos é realizada em períodos de tempo. As avaliações de eventos e cursos especializam a classe avaliação, como já apresentado nas Figuras 3.9 e 3.24, apresentando um relacionamento com as classes de evento e curso respectivamente. Foi também apresentada na seção 3.2 a interface *WithEvaluations* que especifica o contrato da relação entre as avaliações e o quê é avaliado. Na Figura 3.28 é detalhado como *WithEvaluations* utiliza o recurso de *Template* para especificar uma relação detalhada apenas em suas especializações.

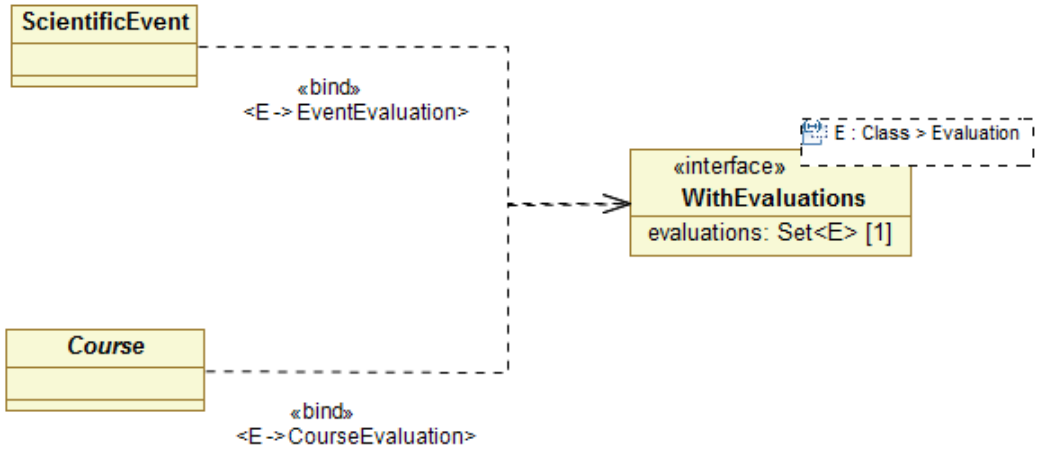


Figura 3.28: Implementação da interface genérica *WithEvaluations*.

A propriedade *evaluations* é definida na interface *WithEvaluations* (interfaces em UML podem ter propriedades) como um tipo parametrizado *Set<E>*, passando como parâmetro a variável de ligação *E*. A interface *WithEvaluations* define a assinatura de *E* como uma classe restrita pela classe *Evaluation*, ou seja, *evaluations* é um conjunto definido para um tipo *E*, desde que este tipo seja uma avaliação.

As classes que implementam a interface *WithEvaluations* devem especificar um tipo de ligação para a variável *E*. O evento científico define *E* como avaliação de evento (*EventEvaluation*), enquanto que o curso define *E* como uma avaliação de curso (*CourseEvaluation*). A propriedade *evaluations* também é sobrescrita para trabalhar com conjuntos de eventos e cursos – e passa também a compor um relacionamento bidirecional com a entidade de avaliação, como pode se ver no exemplo da Figura 3.15 no caso dos cursos.

Esta solução facilita a utilização dos métodos de acesso de cada classe da hierarquia. Quando se realiza uma consulta sobre cursos, a propriedade *evaluations* de cada elemento lista um conjunto de avaliações de curso. No entanto, se o objeto de operação é um conjunto de *objetos com avaliação* (por exemplo, contendo cursos e eventos), podemos utilizar a mesma propriedade para acessar o conjunto de avaliações. Do ponto de vista do banco de dados, não existem *objetos com avaliação* (*WithEvaluation*), apenas as tabelas para curso e avaliação de curso; evento científico e sua avaliação.

3.11 Operações e construtores

As operações no diagrama de classes representam a estrutura das mensagens que cada classe pode receber. As consequências do acionamento de cada operação são especificadas nos diagramas dinâmicos, tais como de colaboração, atividades ou estado. O padrão MD-JPA permite a especificação das operações nos modelos de classe, utilizando o estereótipo *create* da própria UML para diferenciar os construtores.

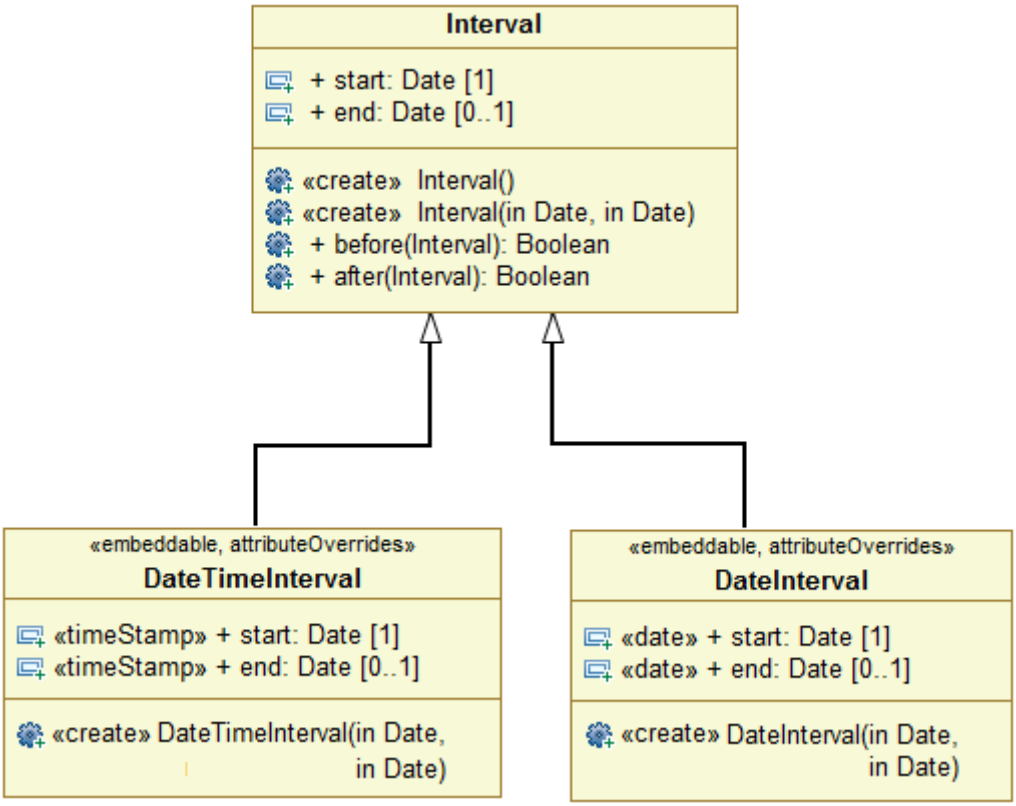


Figura 3.29: Construtores e operações da hierarquia de intervalo de tempo.

A Figura 3.29 apresenta o modelo de intervalos de tempo, detalhando as classes introduzidas na Figura 3.9. Neste diagrama estão exemplificadas a definição de operações e construtores. O estereótipo *create* é aplicado nas operações para identificar os construtores, que utilizam o tipo genérico de tempo *java.util.Date* para sua inicialização.

Cada especialização de intervalo sobrescreve o construtor para armazenar a informação de tempo segundo sua necessidade. Um intervalo do tipo *DateTime* deve armazenar apenas a informação de tempo referente a dias, garantindo que informações sobre horas e minutos sejam removidas na inicialização do intervalo.

As operações *before* e *after* implementadas em *Interval* não são sobrescritas nas suas especializações. *Before* verifica se um intervalo termina antes que o outro comece, enquanto *after* é a operação inversa. Outras operações podem ser definidas para comparação ou aritmética de intervalos.

3.12 Avaliação da adequação do perfil MD-JPA

Esta seção apresenta uma avaliação de adequação dos recursos do perfil MD-JPA como um conjunto. Primeiramente serão apresentadas as relações entre diagramas e o modelo UML, com o objetivo de mostrar quais (e como) informações modeladas são apresentadas nos diagramas. Depois será apresentada uma avaliação de adequação, baseada nas dimensões cognitivas (GREEN e PETRE, 1996), com o objetivo de avaliar a utilização do perfil MD-JPA em comparação a notação UML e implementação em código fonte.

Com a abordagem MDD, as novas versões da UML incorporaram características que possibilitassem o armazenamento de dados mais precisos sobre os sistemas modelados. Na UML 2 a especificação do sistema é separada de sua apresentação visual em diagramas. O conceito de modelo na UML corresponde a especificação de um conjunto de elementos, dentre os quais os diagramas são os que apresentam graficamente os demais elementos. Desta forma, os diagramas correspondem a determinadas visões gráficas dos elementos especificados no modelo (OMG, 2009).

As extensões propostas neste trabalho são realizadas sobre os elementos da UML que são utilizados em diagramas de classes. Cada elemento pode ser descrito em mais de um diagrama, com diferentes características gráficas, desde que não haja uma incoerência nos dados apresentados. Cada elemento pode participar de mais de um diagrama do mesmo tipo.

Por exemplo, no diagrama 3.23 a interface *WithEvaluations* é representada por um caixa pequena apenas com o nome da interface, enquanto que na Figura 3.28 a mesma interface é apresentada com parâmetros de *template* UML e uma propriedade chamada *evaluations*, em uma caixa maior. É possível construir outros diagramas que representam *WithEvaluations,* desde que a consistência seja mantida. Por exemplo, seria incoerente que em um diagrama a propriedade *evaluations* seja do tipo *Set<E>* e em outro diagrama seja do tipo *List<E>*.

Os diagramas são representações das informações contidas no modelo, sendo assim, a interface *WithEvaluations* é a mesma em todos os diagramas. Em uma ferramenta UML, ao alterar o tipo da propriedade *evaluations* para *List<E>*, automaticamente esta alteração deve ser atualizada em todos os diagramas que apresentam esta informação. Porém, ao se alterar características visuais do diagrama, como posição e tamanho da

caixa que representa a interface, estas informações não são atualizadas nos demais diagramas.

As informações gráficas que compõem os diagramas são especificadas separadamente pela OMG, e frequentemente armazenadas de maneira proprietária pelas ferramentas de modelagem. A especificação de intercâmbio de diagramas[10] (OMG, 2006a) detalha como informações gráficas dos diagramas podem ser armazenadas de maneira intercambiável entre ferramentas.

Ao se construir um diagrama, o usuário fica livre para incluir ou omitir informações do modelo, tais como propriedades e operações de uma classe, relacionamentos, herança e estereótipos. Esta característica dá uma grande flexibilidade para a construção de diagramas com diferentes propósitos, que são automaticamente atualizados quando ocorrem pequenas modificações, como no nome de uma classe ou tipo de propriedade.

Na aplicação do perfil MD-JPA, a utilização de vários diagramas de classe pode ajudar na organização da documentação do sistema. Os diferentes diagramas apresentados ao longo do capítulo 3 fazem parte do mesmo modelo, ainda que tenham graus de detalhamento bem diferentes.

O diagrama da Figura 3.2 apresenta as classes do sistema sem nenhuma informação sobre persistência, propriedades ou métodos. Sua função é apresentar os conceitos principais do sistema, tal qual o diagrama da Figura 3.3. Já o diagrama da Figura 3.5 tem como objetivo detalhar as classes, propriedades e relacionamentos com os recursos do perfil MD-JPA.

Nos diagramas UML, o usuário decide para cada elemento quais estereótipos, dos que estão aplicados no elemento, vão ser visualizados. Por outro lado, as informações correspondentes às propriedades dos estereótipos não são representadas nos diagramas, assim como muitas das propriedades definidas pela UML. Para visualizar os detalhes dos elementos que estão ocultos nos diagramas, as ferramentas UML oferecem editores que permitem inspecionar estes valores.

A Figura 3.6 apresentou um diagrama capturado em uma ferramenta de edição UML, selecionando a propriedade *name* da classe *Researcher*. A janela de inspeção de propriedades mostra que a propriedade *length* tem o valor de 50. Esta propriedade é uma informação introduzida pelo MD-JPA que não é apresentada no diagrama, mas pode ser consultada no editor. Desta forma, apenas as informações mais importantes sobre o mapeamento objeto-relacional são apresentadas no diagrama, conforme a Tabela 3.3.

Com base nestas características de representação da UML, foi realizada uma análise de avaliação de adequação baseada em dimensões cognitivas. Para cada dimensão cognitiva é realizada uma avaliação do perfil MD-JPA em comparação com a implementação direta em código e com as características cognitivas da própria UML.

As dimensões cognitivas foram escolhidas para avaliar a utilização do perfil MD-JPA porque apresentam um vocabulário comum que ajuda a explorar os pontos fortes e fracos de cada notação, e que tipo de abordagem ou ferramenta auxiliar podem amenizar estes pontos fracos. Além disso, é uma análise que consome pouco recursos e pode ser aplicada antes que uma ferramenta esteja completa.

---

10 Diagram Interchange Specification, termo em Inglês.

Tabela 3.3: Elementos visíveis nos diagramas.

Elemento	*Visibilidade*
Entity	Classe persistente (Entidade).
SingleTable	Generalização implementada com estratégia *SingleTable.*
Joined	Generalização implementada com estratégia *Joined.*
TablePerClass	Generalização implementada com estratégia *TablePerClass.*
AssociationMapping	Propriedade que define seu mapeamento de associação. Visível apenas quando as propriedades conectoras são apresentadas no diagrama.
Embeddable	Classe que que pode ser embutida em entidades.
Embedded	Propriedade que referencia classe embutida.
Transient	Propriedade que não deve ser persistida.
Id	Propriedade que é / faz parte de chave primária.
IdClass	Classe que tem especificada uma chave composta através de duas ou mais propriedades.
EmbeddedId	Propriedade (única) que especifica chave primária composta através de classe embutível.
Column	Propriedade que tem mapeamento para coluna especificado.
Version	Propriedade utilizada para controle de versão
Enumerated	Propriedade que especifica sua enumeração
MappedSuperclass	Classes transientes cujas propriedades são mapeadas, quando estendidas por entidades
GeneratedValue	Mapeamento de propriedades com valores gerados automaticamente.
Lob	Mapeamento de propriedades para tipos binários largos.
Date, Time, TimeStamp	Mapeamento de propriedades com tipo Data.
AttributeOverrides	Elemento que sobrescreve o mapeamento definido em uma superclasse/classe embutida. Quais/como atributos são sobrescritos não é visualizado no diagrama.
OrderBy	Propriedade que representa coleção ordenada.
DiscriminatorColumn	Classe possui uma coluna discriminadora especificada.
SequenceGenerator	Elemento define um gerador de identificador através de *Sequence.*
TableGenerator	Elemento define um gerador de identificador através de uma tabela auxiliar.
AssociationOverrides	Generalização sobrescreve o mapeamento de um relacionamento.

3.12.1 Quanto ao gradiente de abstração

O gradiente de abstração avalia a capacidade de encapsulamento e extensão dos conceitos em uma notação. Os diagramas da UML podem ser considerados tolerantes à abstração, ou seja, o usuário pode, mas não necessita, construir novos elementos através dos mecanismos de extensão, como o perfil. A utilização do perfil MD-JPA não interfere na utilização de outros perfis e extensões da UML.

Diagramas de classe podem ser construídos de forma a agrupar elementos do modelo conforme uma temática, abstraindo certos detalhes conforme o desejo do usuário. Porém, estes modelos não introduzem novos elementos semânticos na notação UML.

Outro recurso da UML é a habilidade de referenciar elementos definidos em outros modelos, que podem participar nos diagramas do modelo cliente. Geralmente os

modelos importados correspondem a módulos básicos ou desenvolvidos por terceiros. Um exemplo de módulo básico é o modelo UML da API da linguagem Java. Este recurso pode ser utilizado em conjunto com o perfil MD-JPA.

3.12.2 Quanto às dependências escondidas

Dependências escondidas são causadas por elementos que se relacionam de forma unidirecional, dificultando ao usuário prever os impactos de uma alteração realizada sobre um determinado elemento. Por exemplo, ao se alterarem os parâmetros de uma função em um módulo, quaisquer outros módulos que utilizem esta função precisam alterar suas chamadas, mas não há nenhuma informação na função sobre quais são estes módulos que necessitam ser alterados.

O MD-JPA introduz alguns elementos com dependências escondidas nas classes embutíveis e relações de herança. Ao se alterar uma classe embutível ou superclasse mapeada, esta alteração pode invalidar o mapeamento nas classes que fazem referencia, mas apenas quando estas fazem sobrescrita de mapeamento.

As dependências escondidas se tornam um problema na medida da dificuldade de se descobri-las (GREEN e PETRE, 1996). No caso do MD-JPA, as regras de avaliação detectam quando ocorrem incongruências, alertando o usuário sobre quais mapeamentos deixaram de ser válidos após uma alteração no modelo.

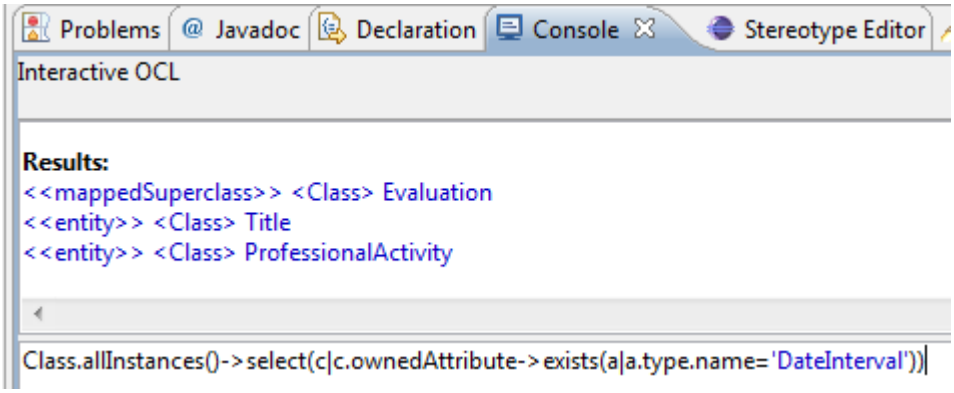


Figura 3.30: Uso de OCL para verificar dependências escondidas do MD-JPA.

Além disso, é fácil detectar as relações de dependência utilizando-se expressões OCL, as quais permitem consultar o modelo para determinar quais classes fazem uso de objetos embutidos ou sobrescrevem o mapeamento definido em uma superclasse. Por exemplo, se a classe embutida *DateInterval* é modificada, o usuário pode consultar quais classes do modelo utilizam *DateInterval,* como exemplificado na Figura 3.30. Estas consultas OCL podem ser incorporadas na ferramenta de modelagem para facilitar a descoberta das dependências escondidas.

3.12.3 Quanto às decisões prematuras

A dimensão das decisões prematuras [11] avalia o quanto uma notação obriga o usuário a tomar decisões em uma ordem pré-estabelecida, quando o usuário ainda não tem todos os dados para tomar tais decisões. Poucos elementos introduzidos no perfil MD-JPA necessitam ser criados em uma ordem específica.

O usuário pode começar definido as classes, propriedades, operações e relacionamentos. Em um segundo momento decidir quais classes são entidades, classes

11 Premature Commitment, termo em Inglês.

embutidas ou superclasses mapeadas. Depois pode definir os identificadores, e por fim o detalhamento do mapeamento através de estereótipos como *Column* e propriedades de estereótipos, como *Table* em *Entity*.

Não obstante, o usuário pode optar por definir as classes já detalhando propriedades, mapeamento e identificadores. Para cada classe ele cria, quando for o caso, as classes embutíveis utilizadas pela entidade. Depois passa para a criação das classes relacionadas ou da hierarquia de herança, atribuindo a sobrescrita de mapeamento se for necessário.

A única ordem imposta pelas ferramentas UML é a criação dos elementos antes que estes sejam referenciados no modelo. Por exemplo, para definir uma propriedade como chave de um mapa (*MapKey*), ela deve primeiro existir. Assim como para se estabelecer um relacionamento entre duas classes, estas classes devem ser criadas no modelo em primeiro lugar.

Na linguagem de programação escrita é permitido referenciar um elemento que ainda não foi codificado. Da mesma forma, é permitido parar a qualquer momento de especificar um elemento para se especificar outro. Entretanto a ordem de criação dos elementos em diagramas de classe UML não causa, em geral, decisões prematuras, já que a definição das propriedades de cada elemento pode ser postergada.

3.12.4 Quanto à viscosidade

A viscosidade mede o tamanho do esforço necessário para realizar uma pequena alteração em um modelo. Ao se introduzir um novo elemento em um diagrama, vários elementos precisam ser reposicionados e redimensionados para manter o diagrama agradável para leitura. Este processo geralmente onera o desenvolvedor, que em uma linguagem de texto tem muito menos esforço para reformatar o código fonte.

No caso da UML, a viscosidade está muito ligada à ferramenta que realiza a modelagem. Como a alteração em um diagrama pode causar efeitos colaterais em outros diagramas que apresentam o mesmo elemento, um esforço extra precisa ser despendido para verificar se as caixas utilizadas para representar uma determinada classe não ficaram muito pequenas para o número de propriedades, ou se não há figuras sobrepostas após as alterações.

O perfil MD-JPA não introduz novos problemas quanto à viscosidade da notação UML, mas ao introduzir mais informações, aumenta o leque de situações que levam à necessidade de corrigir a disposição visual dos elementos em diagramas. Porém, como o perfil MD-JPA não introduz novos elementos visuais, ele pode ser aplicado em diferentes ferramentas de modelagem UML, que implementam algoritmos para automaticamente reposicionar os elementos gráficos.

3.12.5 Quanto à avaliação progressiva

A avaliação progressiva é a habilidade do usuário de verificar a correção de um modelo incompleto. No caso dos modelos MD-JPA, um usuário pode avaliar seu modelo através das restrições OCL ou do resultado da transformação realizada no modelo.

O usuário pode submeter seus modelos MD-JPA incompletos para verificação de erros a qualquer momento. Modelos incompletos podem ser transformados em código fonte, mas apenas se estiverem bem formados e forem válidos. Caso o modelo esteja

mal formado, o usuário precisa corrigir os erros indicados antes de poder submeter o modelo para transformação.

3.12.6 Quanto à notação secundária

A notação secundária compreende os recursos não formais de uma linguagem para expressar informações para seu usuário, tais como identação de código, posicionamento dos elementos em modelos e inclusão de elementos visuais com objetivo explicativo. O metamodelo da UML permite a definição de comentários que podem ser relacionados a elementos nos diagramas, bem como dá liberdade para o usuário representar os elementos nos modelos, dentro das regras gráficas propostas pela notação.

O perfil MD-JPA não acrescenta nem altera nenhum dos recursos já existentes na notação gráfica da UML. Como nem todas as informações especificadas são visíveis nos diagramas, a utilização de comentários pode facilitar na distinção de características especiais do mapeamento, tais como tipo de ordenamento de uma associação, ou o nome das tabelas secundárias de uma entidade.

3.12.7 Quanto à visibilidade e justaposição

A visibilidade é o quanto da informação sobre o sistema pode ser visualizada "ao mesmo tempo", e a justaposição verifica se esta informação visual pode ser colocada "lado a lado" para comparação. A UML é muito flexível quanto a visibilidade e justaposição, pois permite a criação de vários diagramas sobre os mesmos elementos, e permite que as ferramentas apresentem uma visualização simultânea destes diagramas.

O perfil MD-JPA mantém esta flexibilidade, permitindo ao usuário selecionar quais informações são visualizadas em cada diagrama. Porém, as informações contidas nas propriedades de estereótipos não são apresentadas em diagramas UML, exceto através de comentários feitos pelo usuário. Por exemplo, a informação sobre o comprimento de uma propriedade no mapeamento objeto-relacional não fica visível no diagrama de classes, mas pode ser acessado no inspetor de propriedades. Mesmo assim, fica difícil comparar o comprimento de várias propriedades ao mesmo tempo, utilizando apenas os recursos de uma ferramenta de modelagem padrão.

Através de consultas OCL sobre o modelo, pode-se determinar quais elementos tem determinadas características. As consultas OCL amenizam o problema da visibilidade dos detalhes de mapeamento, ainda que estas informações não sejam apresentadas como um diagrama UML.

3.12.8 Quanto à complexidade das operações mentais

A complexidade das operações mentais avalia o quanto uma notação pode ser facilmente compreendida. Um dos indicadores para se avaliar a complexidade das operações mentais é a utilização de anotações separadas para a compreensão do diagrama. Outro indicador são as construções inválidas, e o tempo necessário para corrigi-las.

Ao se comparar a utilização do modelo de classes com o perfil MD-JPA, com a utilização de Java com anotações, é perceptível a vantagem oferecida pelos modelos de classe ao permitir visualizar o mapeamento de várias classes simultaneamente. A representação das classes como texto ocupa muito espaço, permitindo a visualização de poucas classes simultaneamente. Isto dificulta o entendimento das relações entre as

classes, e seu papel no sistema. É uma vantagem da notação UML sobre o código fonte, que é aproveitada nos modelos MD-JPA.

A Figura 3.31 mostra uma tela capturada em uma ferramenta de edição de código, apresentando os métodos de acesso que implementam os relacionamentos entre *Paper*, *ScientificEventEdition* e *ScientificEvent*. A compreensão da relação entre as três entidades é difícil, uma vez que é preciso encontrar cada método de acesso, interpretar o tipo de mapeamento utilizado e os tipos de retorno dos métodos de acesso (ou tipo utilizado na variável de instância). Já no diagrama MD-JPA da Figura 3.32, os relacionamentos são elementos visuais explícitos.

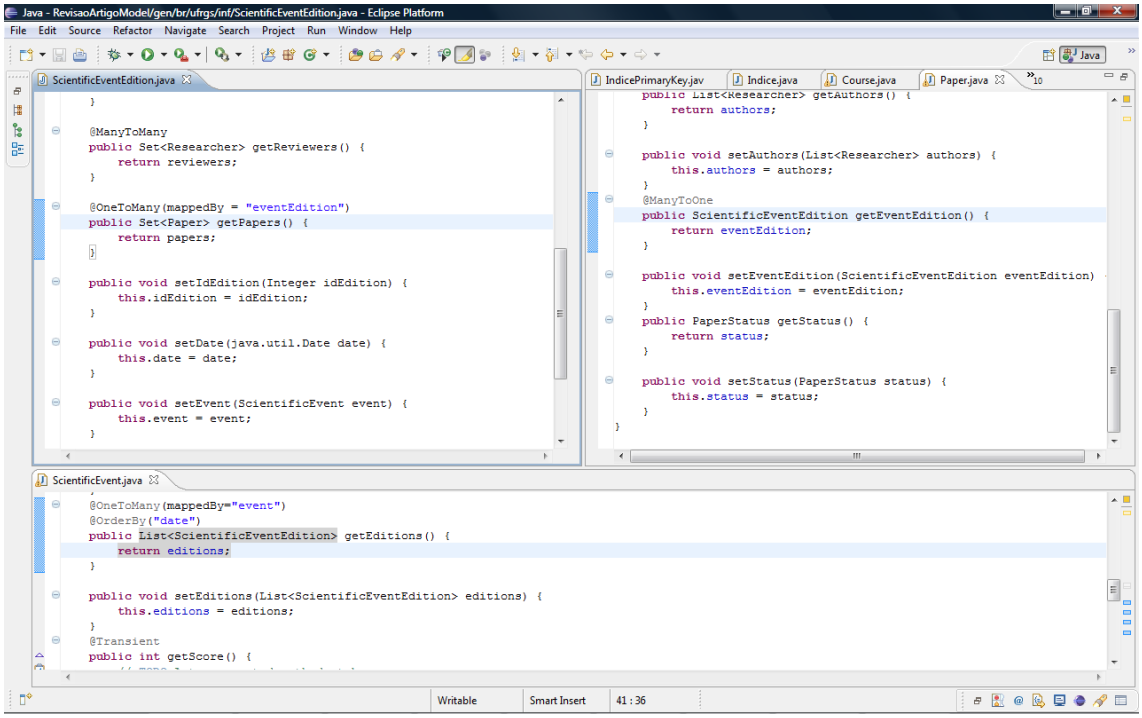


Figura 3.31: Apenas com o código, relacionamentos não são claros.

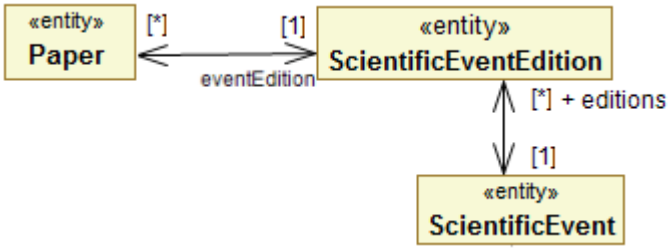


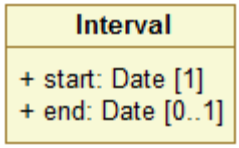
Figura 3.32: O diagrama de classes facilita a visualização dos relacionamentos.

3.12.9 Quanto à compactabilidade (Diffuseness)

A compactabilidade procura avaliar uma notação quanto ao número de símbolos utilizados para representar um determinado conceito. Os diagramas com o perfil MD-JPA, quando comparados a implementação em Java, empregam, em geral, menos elementos visuais para representar a mesma informação. O caso mais visível é o da

representação das propriedades *JavaBean* nos modelos. Cada propriedade na linguagem Java compreende a declaração de uma variável de instância e dois métodos de acesso. Já nos modelos, os métodos de acesso não são visualizados.

A Figura 3.33 compara a definição de uma classe no diagrama de classes, e na implementação. Como cada propriedade corresponde a três elementos (declaração de variável, método de leitura e método de escrita), a notação gráfica é claramente mais concisa neste caso, mesmo desconsiderando o corpo de implementação dos métodos.




```
public class Interval {
    private java.util.Date start;
    private java.util.Date end;
    public Date getEnd() {
        return end;
    }
    public void setEnd(Date end) {
        this.end = end;
    }
    public Date getStart() {
        return start;
    }
    public void setStart(Date start) {
        this.start = start;
    }
```

Figura 3.33: Propriedades em um diagrama de classe e no código fonte.

3.12.10 Quanto à inclusão de erros

Por ser uma notação gráfica, uma parte considerável dos erros são verificados pela própria ferramenta de modelagem utilizada na UML. As referencias entre diferentes elementos são verificadas no momento da edição, eliminando a maior parte dos erros de sintaxe.

O perfil MD-JPA é complementado pelas regras de avaliação OCL, com o objetivo de detectar os erros nos modelos. A definição de coluna, por exemplo, permite a digitação do nome da tabela utilizada, que pode ser escrita de maneira errada. Este tipo de avaliação é realizado em OCL, verificando se a tabela é uma das tabelas definidas para a entidade.

3.12.11 Quanto à consistência

O quesito consistência avalia se existem maneiras diferentes de se realizar tarefas semelhantes em uma notação. O objetivo é verificar se os recursos de uma linguagem são harmoniosos, verificando se há casos especiais como por exemplo a diferenciação entre tipos primitivos e classes que existe em linguagens OO e na UML.

O MD-JPA foi projetado a partir da especificação JPA, mas objetivando utilizar alguns recursos da UML para melhorar a consistência da representação do mapeamento. A definição de tabelas é um exemplo, já que as classes que representam tabelas secundárias, tabelas de junção e a tabela principal (*SecondaryTable*, *JoinTable* e *Table*) tem como superclasse comum a classe *AbstractTable*. Da mesma forma, os estereótipos *Embedded*, *Entity* e *Mappedsuperclass* são especializações do estereótipo *Persistent*, que identifica todas as classes que possuem mapeamento objeto-relacional. Estas superclasses podem ser utilizadas em critério de busca OCL, obtendo relatórios com dados comuns entre as classes com maior facilidade, como mostrado na captura de tela da Figura 3.34.

Figura 3.34: Relatório de classes persistentes, incluindo entidades, superclasses mapeáveis e classes embutíveis.

Também para incrementar a consistência dos modelos em relação as anotações JPA, foram eliminadas as anotações duplicadas para representar um ou muitos elementos. Por exemplo, no JPA a anotação *AttributeOverride* é utilizada para sobrescrever um atributo e a anotação *AttributeOverrides* é utilizada para sobrescrever um ou mais atributos. Já no perfil MD-JPA, sempre se utiliza o estereótipo *AttributeOverrides*, que contém uma ou mais declarações de sobrescrita, armazenadas em instâncias de *AttributeOverride*. Desta forma, o usuário modela de maneira consistente, já que o número de atributos sobrescritos não interfere na construção utilizada.

3.12.12 Quanto à proximidade do mapeamento

A proximidade de mapeamento representa o quanto a notação se aproxima do domínio do problema abordado. Neste caso, verifica-se quantos elementos específicos da notação são utilizados para representar o elemento a ser modelado. Na UML, por exemplo, o conceito de nota fiscal pode ser representado por duas classes relacionadas, uma representando a nota e outra representando cada item de compra.

Ao permitir representar no mesmo elemento a unidade de programa (classe) e a estrutura de persistência (quando em uma única tabela), o perfil MD-JPA diminui o número de elementos utilizados para representar um conceito real, quando comparado a outras abordagens que necessitam representar separadamente tabelas e classes. Porém, os novos recursos introduzidos pelo perfil podem incrementar o número de elementos utilizados no modelo final, dependendo do tipo de construção pretendida pelo usuário.

Quando se especificam chaves compostas, por exemplo, é necessária a definição de uma classe embutível que represente as propriedades da chave. A utilização de chaves compostas, ao introduzir estas classes, afasta o modelo do domínio representado. Ainda assim, o usuário pode optar por não utilizar, ou minimizar a utilização de chaves compostas.

3.13 Resumo e discussão

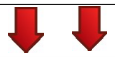
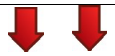
Este capítulo apresentou os recursos do perfil MD-JPA, uma extensão da notação UML para a modelagem de sistemas que realizam persistência através de *frameworks* MOR, segundo o padrão JPA. Cada recurso modelável foi apresentado enfatizando sua

relação com a especificação JPA, com exemplos baseados em uma aplicação motivadora.

O MD-JPA oferece um conjunto completo de recursos para a modelagem da persistência, sem no entanto obrigar ao usuário a utilização de todos estes recursos. Ao aproveitar os recursos da UML, para construção de diagramas com diferentes visões do mesmo sistema, permite o emprego gradual dos recursos de modelagem e a construção de visões especificas para cada situação descrita.

O perfil MD-JPA permite a modelagem da persistência relacional em conjunto com os conceitos da orientação a objetos. Recursos como o de superclasse mapeada permitem explorar características importantes da orientação a objetos, como a herança e o polimorfismo, respeitando alguns limites inerentes ao paradigma relacional.

Tabela 3.4: Resumo da avaliação utilizando dimensões cognitivas.

Dimensão cognitiva	*Código*	*UML*	*MD-JPA*
Gradiente de abstração			
Dependências escondidas			
Decisões prematuras			
Viscosidade			
Avaliação progressiva			
Notação secundária			
Visibilidade e justaposição			
Complexidade op. mentais			
Compactabilidade			
Inclusão de erros			
Consistência			
Proximidade do mapeamento			

Uma avaliação de adequação segundo as dimensões cognitivas foi realizada comparando a utilização do perfil MD-JPA com código fonte e modelos UML. A avaliação mostrou que a utilização do perfil MD-JPA apresenta algumas vantagens sobre a utilização de código fonte simples ou modelos UML padrão, no contexto do desenvolvimento de aplicações com MOR. A Tabela 3.4 resume os pontos fortes e fracos de cada abordagem, quanto às dimensões apresentadas neste capítulo.

Algumas simplificações propostas no MD-JPA ainda podem ser estendidas, com o objetivo de permitir configurações especiais para o usuário. Extensões ao perfil podem ser criadas para configurar os casos de relacionamentos com valores discretos entre um e muitos (como “dois para um”), especificando como a composição e agregação devem afetar o sistema modelado, ou ainda como nomear as variáveis de instância que representarão as propriedades *JavaBean* na implementação.

No próximo capítulo é apresentado o conjunto de restrições definidas para verificar se os modelos MD-JPA estão bem formados. A partir das informações centralizadas em

um único modelo, um conjunto de restrições pode verificar se este modelo é compatível com as restrições impostas pelo mapeamento objeto-relacional do JPA. Finalmente, a partir de modelos bem formados, será possível gerar partes da implementação da aplicação.

4 AVALIAÇÃO DE MODELOS NO MD-JPA

Uma das principais vantagens de se empregar uma notação para representação do MOR está na avaliação dos modelos quanto à representação de sistemas válidos, segundo um conjunto de princípios pré-definidos. Este capítulo especifica um conjunto de regras utilizadas para verificar se modelos construídos com o MD-JPA estão bem formados e válidos, ou seja, representam um conjunto de elementos que podem ser mapeados para um sistema segundo a especificação JPA. Este conjunto de regras é implementado de forma que se possa detectar quais são os problemas e quais elementos do modelo são afetados por estes problemas, orientando, portanto, o usuário para a correção dos mesmos.

Este capítulo está dividido em três seções. Na primeira seção é apresentada a metodologia utilizada para construção das regras, que inclui: os objetivos das regras propostas; como estão organizadas; e quais são suas origens. Na segunda seção, são especificadas as regras de avaliação para cada um dos recursos definidos no Capítulo 3. A terceira seção finaliza sumarizando as contribuições do capítulo.

4.1 Metodologia para a avaliação de modelos MD-JPA

Para avaliar modelos MD-JPA é proposto um conjunto de regras de restrição escritas na linguagem OCL. O OCL permite atribuir restrições, chamadas na OCL de invariantes, aos elementos UML ou extensões definidas no perfil. Para avaliar um modelo construído pelo usuário, cada elemento deste modelo é testado contra as invariantes definidas para o seu contexto. Esta seção apresenta como a linguagem OCL é utilizada para avaliação de modelos e como foi definido o conjunto de invariantes a serem testadas.

Cada modelo UML é composto por um conjunto de elementos e cada elemento corresponde a um (“é uma instância de”) meta-elemento da linguagem UML, tal como uma classe, propriedade ou caso de uso. Quando se aplica um estereótipo sobre um elemento, este continua correspondendo ao seu meta-elemento original. Porém, ele também passa a referenciar um objeto de extensão. Este objeto de extensão é uma instância do estereótipo, que contém as informações definidas nas propriedades e relações deste estereótipo.

A Figura 4.1 apresenta um exemplo de como a especificação de um elemento UML se relaciona com um estereótipo associado. *Course* é um elemento do tipo *Class* da UML, com propriedades que representam suas características tais como o nome (*name*), a visibilidade (*visibility*) e se é ou não uma classe abstrata (*abstract*). A extensão correspondente ao estereótipo *DiscriminatorColumn* do MD-JPA é uma instância deste estereótipo, relacionada através da propriedade *extension_DiscriminatorColumn* que,

por sua vez, especifica os valores das propriedades da coluna discriminadora, tais como o tipo de mapeamento (*discriminatorType*) e comprimento do atributo (*length*). A classe *Course* é pública (visibilidade *public*) e abstrata (*abstract* é *true*); tem associada a extensão *DiscriminatorColumn,* que define o tipo de mapeamento como inteiro (*integer*) e comprimento do atributo com valor dois.

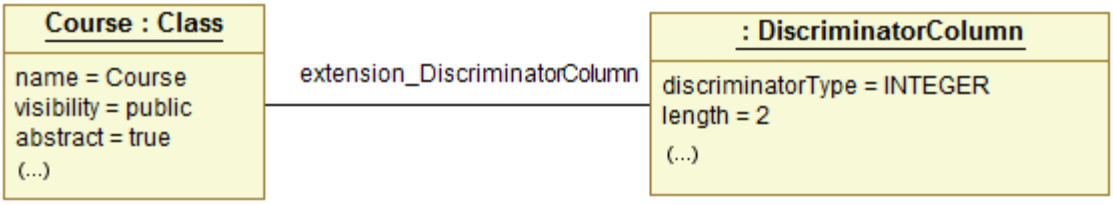


Figura 4.1: Elemento *Course* é uma instância de *Class* e *DiscriminatorColumn.*

A linguagem OCL permite avaliar modelos através da definição de invariantes. As invariantes são definidas sempre sobre um contexto, isto é, um tipo de elemento a ser testado. O contexto pode ser tanto um elemento da UML quanto um elemento definido em um perfil de extensão, tal qual o MD-JPA. Testar um modelo contra uma invariante consiste em selecionar os elementos contidos neste modelo que são do tipo definido em seu contexto e, para cada um destes elementos, verificar se a invariante é verdadeira. Considere, por exemplo:

```
context uml::Class

inv example1 : name<>'int'
```

Neste caso, é apresentada uma invariante que verifica se o nome de uma classe é diferente da palavra reservada *int*. Ao aplicar esta invariante no modelo, cada elemento classe (contexto *uml::Class*) é avaliado, resultando em *verdadeiro* ou *falso*. Considere agora outro exemplo de avaliação de palavra reservada, sendo aplicado no contexto apenas das entidades: prevenir a definição de entidades com a palavra reservada *Table* do SQL.

```
context persistence::Entity

inv example2 : base_Class.name<>'Table'
```

Neste caso os objetos selecionados são as extensões de classe do tipo entidade. A relação 0..1 → 1 entre a extensão e sua classe é definida na propriedade *base_Class*. Como o nome de uma entidade é definido em sua classe, o teste é feito sobre a navegação <*base_Class.name*>. A propriedade inversa de *base_Class* é *extension_Entity*. No perfil MD-JPA as relações seguem o seguinte padrão de nomenclatura: *base_*<*meta-classe*> nos estereótipos e *extension_*<*estereótipo*> para a direção inversa.

A avaliação é, portanto, realizada por um conjunto de regras invariantes associadas a um contexto. A determinação do contexto de cada regra afeta a forma como a regra é escrita e a granularidade do erro a ser encontrado. Por exemplo, ao escrever esta mesma regra no contexto do pacote (*package*) UML, a avaliação pode identificar quais pacotes tem classes com nomes inválidos, mas não identificará quais classes dos pacotes têm este problema.

A Figura 4.2 apresenta uma visão geral da avaliação de um modelo MD-JPA. O modelo com o perfil MD-JPA aplicado é submetido ao conjunto de regras OCL. As regras foram construídas acessando as estruturas de dados definidas no MD-JPA,

verificando cada um dos objetos presentes no modelo. Cada vez que um objeto falha para alguma regra, ele é colocado na lista de erros da regra em avaliação.

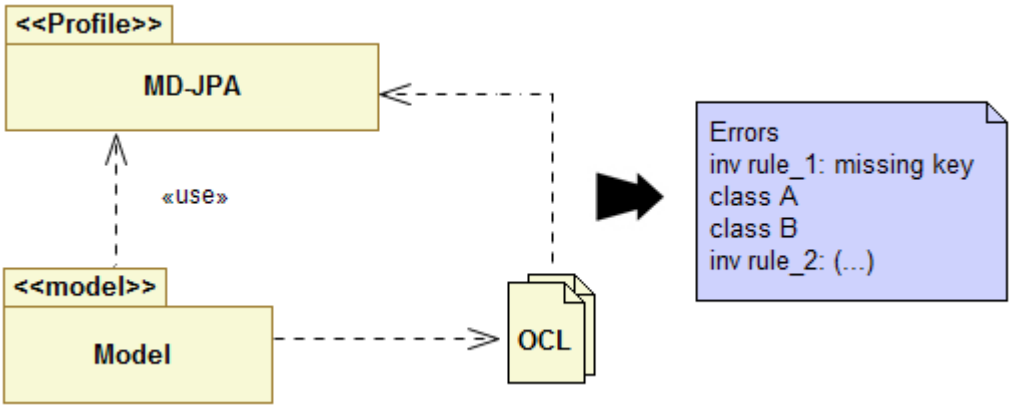


Figura 4.2: Avaliação de um modelo MD-JPA.

Após determinar a forma de avaliação de inconsistência, determinam-se os critérios para decidir se um modelo é consistente ou não. O ponto de partida para a construção das regras de avaliação é a especificação do JPA (DEMICHIEL e KEITH, 2006), especialmente em seu capítulo dois, que estabelece normas para a definição de entidades, relacionamentos, chaves primárias e outros elementos do mapeamento objeto-relacional.

As regras de avaliação propostas neste trabalho procuram em primeiro lugar garantir que o modelo esteja de acordo com a especificação JPA. Não obstante, algumas regras foram definidas para complementar a avaliação, melhorar a usabilidade e garantir que o modelo expresse um sistema que possa ser traduzido para uma implementação na plataforma Java.

4.2 Restrições

As restrições aqui apresentadas formam um conjunto de invariantes e definições auxiliares em OCL que são executadas no pacote *persistent,* ou seja, referenciam o pacote de estereótipos do perfil MD-JPA. Com o objetivo de organizar a seção, as regras estão divididas em subseções de acordo com o contexto que avaliam.

Cada regra apresentada cita a parte da especificação referente a sua função [12]. No fim da seção é apresentada uma lista de restrições adicionais que não referenciam a especificação, mas que verificam alguns aspectos úteis da integridade dos modelos.

Com o objetivo de facilitar posterior referência, as invariantes foram desenvolvidas com nomes numerados segundo o capítulo, seção e parágrafo da especificação JPA. Desta forma, a regra *rule21_3* referencia o capítulo 2, seção 1 e parágrafo 3. As restrições adicionais, que não referenciam diretamente a especificação, utilizam uma numeração sequencial com o nome *ruleProfile*.

4.2.1 Requisitos para *Persistent*

O estereótipo *Persistent* definido no perfil MD-JPA é especializado pelos estereótipos que representam entidades (*Entity*), classes embutíveis (*Embeddable*) e superclasses mapeadas (*MappedSuperclass*), tal qual é apresentado no diagrama da

12. As citações literais nesta seção são traduções minhas.

Figura 4.3. O estereótipo *Persistent* é abstrato, portanto não é utilizado pelo usuário na construção de modelos. Sua finalidade é permitir a construção de expressões OCL com um contexto que abranja todos os três tipos de mapeamento.

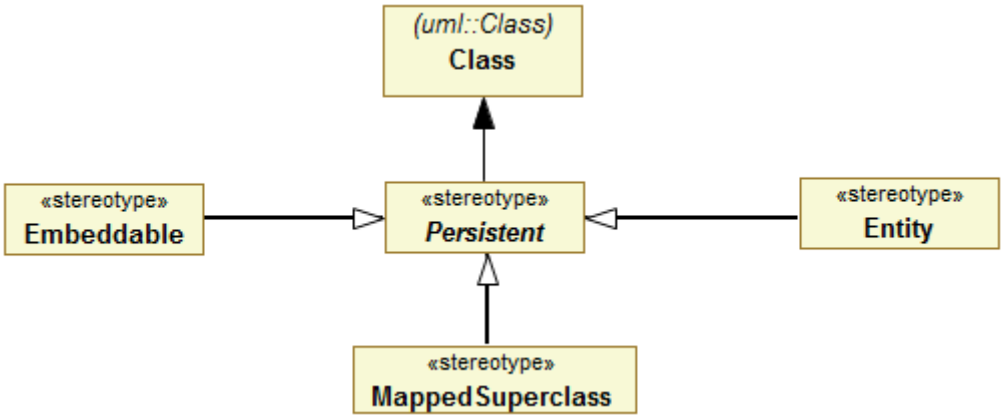


Figura 4.3: Estereótipos do MD-JPA que derivam de *Persistent*.

As restrições seguintes foram definidas para o contexto de todas as classes persistentes, independente de serem embutíveis, entidades ou superclasses mapeadas:

4.2.1.1 Construtor sem argumentos

"O construtor sem argumentos deve ser público ou protegido." (DEMICHIEL e KEITH, 2006, p.17).

A definição de construtores sem argumentos é opcional em Java e também não é obrigatória no MD-JPA. Porém, classes marcadas com um dos estereótipos derivados de *Persistent* não podem ter um construtor sem argumentos que não seja público ou protegido, pois as implementações dos *frameworks* JPA precisam acessar um construtor sem argumentos para instanciar objetos destas classes. A seguinte invariante testa os construtores especificados em extensões de *Persistent*:

```
context Persistent
inv rule21_p2:
base_Class.ownedOperation->forAll(
        (isConstructor() and ownedParameter->isEmpty()) implies
                visibility=uml::VisibilityKind::public
                or
                visibility=uml::VisibilityKind::protected
)
```

Para cada aplicação de *Persistent* no modelo, a classe representada pela propriedade *base_Class* referencia o conjunto de operações a ser verificado. Caso seja encontrado, neste conjunto, um construtor sem parâmetros, testa-se através do operador de implicação (*implies*) se a visibilidade é pública ou privada.

4.2.1.2 Classes "top level"

"A entidade deve ser uma classe *top-level*. Enumerações e interfaces não podem ser entidades." (DEMICHIEL e KEITH, 2006, p.17).

Uma classe é dita *top-level* quando satisfaz as seguintes condições (GOSLING et al, 2005):

- Não é contida em outra classe.
- Não é protegida.
- Não é privada.
- Não é estática.

No perfil MD-JPA, o estereótipo de entidade não pode ser aplicado às interfaces e enumerações. Além disso, na UML não existe o conceito de classe estática e o perfil MD-JPA não aborda sua definição.

Todavia, a UML contém especializações do elemento *Class* que não são utilizadas nos modelos de classe, tais como atividades (elementos do diagrama de atividades), componentes (diagramas de componente) e outros estereótipos. A invariante deve garantir que os estereótipos do tipo *Persistent* sejam aplicados apenas aos elementos que representam classes, como é especificado na invariante a seguir:

```
context Persistent

inv rule21_p3:

(base_Class.owner.oclIsKindOf(uml::Package) or base_Class.owner.oclIsKindOf(uml::Model))

and not(base_Class.visibility=uml::VisibilityKind::protected or
       base_Class.visibility=uml::VisibilityKind::private)

and base_Class.oclIsTypeOf(Class)
```

Primeiramente, a invariante verifica se a classe especificada está vinculada a um pacote ou modelo, filtrando as classes internas, tais como as definidas dentro de outras classes. Em seguida, verifica se a visibilidade da classe não é protegida ou privada. Por fim, verifica se a classe base é uma classe, e não uma das extensões de classe presentes na UML.

4.2.1.3 Classes finais

"As classes de entidades não devem ser finais. As propriedades persistentes de uma entidade não devem ser finais." (DEMICHIEL e KEITH, 2006, p.17).

Classes finais (*final*) são aquelas que não permitem extensão, ou seja, não podem ser especializadas por outras classes. As propriedades finais são aquelas que só podem ter valor atribuído no construtor da classe, e apenas uma única vez. Os *frameworks* JPA utilizam especializações anônimas das classes, com sobrescrita de suas propriedades, para implementar o mapeamento objeto-relacional.

O conceito de *final* na UML é representado pela propriedade *isLeaf*, que controla a redefinição dos elementos redefiníveis (*RedefinableElement* é a meta-classe ancestral de classes e propriedades na UML). No perfil MD-JPA se verifica a especificação de classes e propriedades finais, exceto aquelas propriedades marcadas como transientes, pois podem ser finais. A invariante a seguir realiza estas verificações:

```
inv rule21_p4 :

not(base_Class.isLeaf) and

base_Class.attribute->forAll(not(isLeaf and extension_Transient.oclIsUndefined()))
```

Em primeiro lugar, a invariante verifica se a classe marcada como persistente não está marcada como final pela propriedade *isLeaf*. Em segundo lugar, para cada propriedade definida na classe, verifica se ela é final e não transiente ao mesmo tempo.

4.2.2 Propriedades

As propriedades, na UML, formam o conjunto de características que definem uma classe. Cada propriedade é de um tipo, que pode ser definido em uma classe ou como um tipo primitivo. Na linguagem Java não há construções específicas para propriedades, que são definidas segundo um padrão de código definido na especificação *JavaBean* (SUN MICROSYSTEMS, 1997). A avaliação das propriedades é focada na utilização de tipos adequados a persistência e a correta marcação de propriedades transientes.

4.2.2.1 Tipos válidos para propriedades persistentes

As propriedades persistentes podem ter os seguintes tipos atribuídos: primitivos, *String*, enumerações, entidades, coleções de entidades, classes embutíveis e tipos Java serializáveis (incluindo os *wrappers* de primitivos, *BigInteger, BigDecimal, java.util.Date, Calendar, java.sql.Date, java.sql.Time, java.sql.Timestamp*; os tipos *byte[], Byte[], char[], Character[]*; e tipos definidos pelo usuário também podem ser empregados, sendo mapeados como objetos binários) (DEMICHIEL e KEITH, 2006).

As propriedades persistentes em JPA precisam ser de um tipo que possa ser mapeado para um banco de dados relacional. O padrão JPA é bastante flexível quanto aos tipos que podem ser mapeados, pois além dos tipos básicos, entidades e classes embutidas, pode armazenar tipos definidos pelo usuário, desde que estes sejam serializáveis (ou seja, que implementam ou descendem de classes que implementam a interface *java.io.Serializable*). Entretanto, classes não serializáveis não podem ser utilizadas na definição de uma propriedade persistente, o que é verificado pela seguinte restrição implementada para os modelos MD-JPA:

```
context Property
inv rule211_p10:
not(class.extension_Persistent.oclIsUndefined() or isTransient()) implies
       let fullName:String=type.qualifiedName in
              type.hasStereotype('Entity') or
              type.hasStereotype('Embeddable') or
              type.oclIsKindOf(uml::Enumeration) or
              ( type.oclIsKindOf(uml::PrimitiveType) and
                     primitiveTypes->includes(type.name) ) or
              ( type.oclIsKindOf(uml::Classifier) and
                     javaSerializableStdClasses->includes(fullName) ) or
              type.oclAsType(uml::BehavioredClassifier).interfaceRealization
                     ->exists(contract.qualifiedName=serializableInterface)
```

Para esta regra foram definidas duas coleções auxiliares que listam os tipos primitivos e classes serializáveis padrão do Java. Cada propriedade que pertence a uma classe persistente, mas não é transiente, tem seu tipo avaliado quanto aos quesitos da especificação.

A primeira parte da invariante seleciona apenas as propriedades que pertencem à classes persistentes e não são marcadas como transientes. A seguir, verifica-se se o tipo da propriedade (*type*) é uma entidade, uma classe embutível, uma enumeração, um tipo primitivo permitido ou uma classe serializável.

4.2.2.2 Propriedades transientes não devem ser mapeadas

"Anotações de mapeamento não podem ser aplicadas em campos ou propriedades transientes" (DEMICHIEL e KEITH, 2006, p.18).

No MD-JPA as propriedades transientes de uma entidade são marcadas pelo estereótipo *Transient*. A invariante verifica, para estes casos, se nenhuma das outras extensões do pacote está definida, como apresentado a seguir:

```
context Transient
inv rule211_p2C:
        base_Property.extension_Column.oclIsUndefined() and
        base_Property.extension_AssociationMapping.oclIsUndefined() and
        base_Property.extension_Id.oclIsUndefined() and
        base_Property.extension_Lob.oclIsUndefined() and
        base_Property.extension_Temporal.oclIsUndefined() and
        base_Property.extension_Embedded.oclIsUndefined() and
        base_Property.extension_EmbeddedId.oclIsUndefined() and
        base_Property.extension_Basic.oclIsUndefined() and
        base_Property.extension_OrderBy.oclIsUndefined() and
        base_Property.extension_Version.oclIsUndefined() and
        base_Property.extension_AttributeOverrides.oclIsUndefined()
```

A invariante tem como contexto as extensões *Transient,* verificando através da propriedade *base_Property* se algum estereótipo de mapeamento foi definido para a propriedade verificada.

4.2.3 Chaves primárias e identidade de entidades

No perfil MD-JPA, assim como na especificação JPA, as entidades devem ter uma ou mais propriedades definidas como identificadoras. Estas propriedades identificadoras serão mapeadas como chaves primárias na base de dados relacional. As restrições apresentadas nesta seção verificam se os identificadores foram utilizados de forma correta e transformável para uma implementação JPA.

4.2.3.1 Obrigatoriedade da chave primária

"Toda entidade deve ter uma chave primária" (DEMICHIEL e KEITH, 2006, p.22).

No padrão JPA, e também no perfil MD-JPA, toda entidade deve ter uma chave primária definida. Porém, quando uma entidade especializa outra entidade, a chave primária deve ser definida na entidade mais geral da hierarquia. Esta regra tem como objetivo verificar se a chave primária foi definida em algum ponto da hierarquia de cada entidade, como apresentado a seguir:

```
context Entity
inv rule214_p1:
base_Class.getAllAttributes()->exists(
        (class.hasStereotype('Entity') or class.hasStereotype('MappedSuperclass'))
        and
        (hasStereotype('Id') or hasStereotype('EmbeddedId') )
)
```

Para verificar se uma entidade tem chave primária é preciso verificar todo o conjunto de propriedades herdadas, já que a chave primária pode estar em uma classe especializada pela entidade – desde que ela seja uma entidade ou classe mapeada. A operação *getAllAttributes* lista todas as propriedades da classe, incluindo as herdadas de qualquer classe ancestral. A invariante verifica se nesta lista existe alguma propriedade que esteja definida em uma entidade ou superclasse mapeada e que tenha aplicado o estereótipo *Id* ou *EmbeddedId*.

A Figura 4.4 exemplifica a avaliação do identificador na classe *Graduation*. A avaliação da operação *getAllAttributes()* lista todas as propriedades da hierarquia da classe. As propriedades *name e knowledgeCategories* são herdadas da classe *Categorized* e as propriedades *idCourse*, *institution* e *evaluations* são *heradas* de *Course*. Como a propriedade *idCourse* é marcada como *Id*, e a classe *Course* é uma entidade, a condição da invariante é satisfeita.

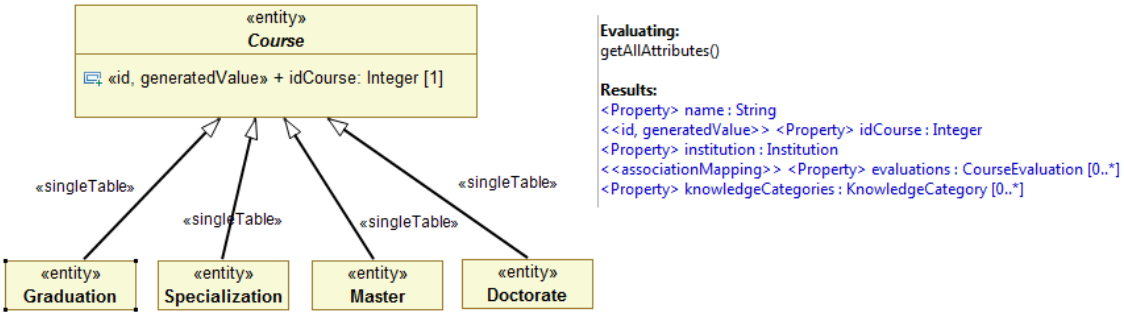


Figura 4.4: Identificador em entidade que especializa outra entidade.

4.2.3.2 Localização da chave primária

"A chave primária deve ser definida na entidade que é a raiz da hierarquia de entidades, ou em uma superclasse mapeada desta hierarquia. A chave primária só pode ser definida uma vez nesta hierarquia" (DEMICHIEL e KEITH, 2006, p.22).

Esta invariante foi escrita para garantir que se uma entidade tem uma chave primária definida, esta entidade deve ser a raiz da hierarquia. Uma entidade com algum atributo chave implica que nenhuma de suas superclasses sejam entidades e que, caso alguma superclasse seja mapeada, ela não tenha definida uma chave.

```
context Entity
inv rule214_p2:
base_Class.attribute->exists(
        not(extension_Id.oclIsUndefined() and extension_EmbeddedId.oclIsUndefined() )
) implies
        base_Class.superClass->forAll(
                not(hasStereotype('Entity'))
                and (
                        not(hasStereotype('MappedSuperclass')) or
                        getAllAttributes()->forAll(
                                extension_Id.oclIsUndefined() and
                                extension_EmbeddedId.oclIsUndefined()
                        )
                )
        )
```

O contexto da invariante é o estereótipo *Entity*, verificando as propriedades especificadas na classe base, e não em toda a hierarquia como na regra anterior. Se existir alguma propriedade identificadora, então nenhuma superclasse pode ser uma entidade ou uma superclasse que já defina uma propriedade identificadora.

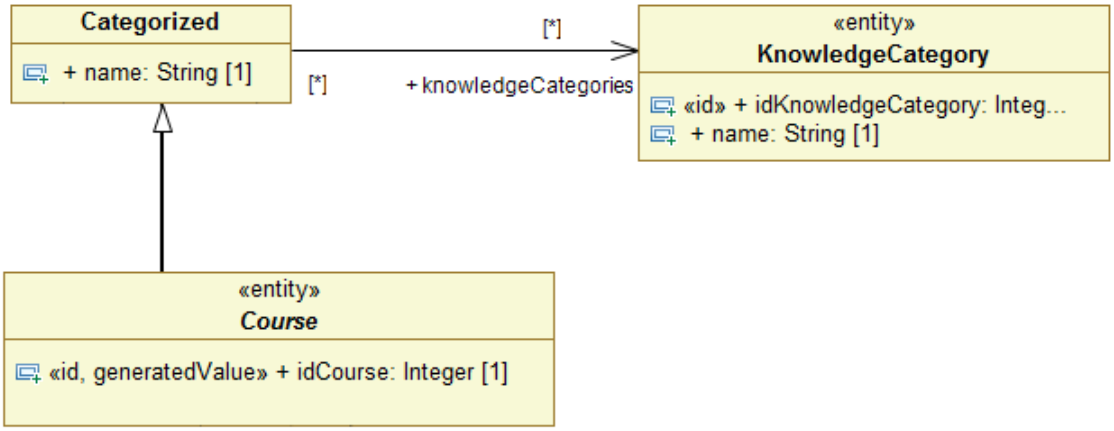


Figura 4.5: Identificador em entidade que especializa outra classe.

Na Figura 4.5 são apresentadas as classes *Course*, *Categorized* e *KnowledgeCategory*. A classe *Course* possui uma especificação de chave primária pois sua superclasse não é uma entidade.

4.2.3.3 Tipos permitidos para chaves primárias

Os campos que compõe a chave primária podem ser dos seguintes tipos: tipos primitivos Java, *wrappers* de tipos primitivos, *String*, *java.util.Date*, *java.sql.Date* (DEMICHIEL e KEITH, 2006).

Esta invariante analisa cada propriedade das entidades para verificar se ela faz parte de uma chave primária. Neste caso, a propriedade é testada na regra secundária *verifyPKType* quanto a seu tipo (muito semelhante ao que se faz na *rule211_p10*).

```
context Entity
inv rule214_p5:
let embeddedId : Sequence(uml::Property)=
        base_Class.attribute->select(hasStereotype('EmbeddedId')) in
        if (embeddedId->isEmpty()) then
                base_Class.attribute->forAll(not(hasStereotype('Id')) or verifyPKType())
        else
                let ebtyp: uml::Type=embeddedId->first().type in
                        ebtyp.oclAsType(uml::Class).attribute->forAll(verifyPKType())
        endif
```

A invariante verifica as definições de entidade, primeiramente definindo uma variável que contém todas as propriedades que utilizam o estereótipo *EmbeddedId*. Caso não existam propriedades marcadas, seleciona-se as propriedades marcadas com o estereótipo *Id* e verifica-se se todas estas propriedades possuem tipos válidos. Caso exista uma propriedade *EmbeddedId*, a variável *ebtyp* recebe o tipo desta propriedade. Como este tipo deve ser uma classe embutida, todas os atributos desta classe são testados para verificar se seu tipo é válido.

4.2.3.4 Mapeamento de chaves compostas

"Uma chave composta deve ser representada e mapeada como uma classe embutível, ou ser representada e mapeada com múltiplos campos ou propriedades na classe entidade" (DEMICHIEL e KEITH, 2006, p.23).

Esta invariante tem como função assegurar que somente um método de mapeamento seja utilizado em uma mesma entidade. Os métodos de mapeamento permitidos são: nenhum mapeamento, chave simples, chave composta com tipo embutido (*EmbeddedId*) ou chave composta mapeada para várias propriedades (*IdClass* aplicado à classe e duas ou mais propriedades *Id*).

```
context Entity
inv rule214_p7E:
(base_Class.attribute->forAll(extension_Id.oclIsUndefined()) and
 base_Class.attribute->select(hasStereotype('EmbeddedId'))->size()=1
) or
(base_Class.attribute->select(hasStereotype('Id'))->size()>1 and
 base_Class.hasStereotype('IdClass')
) or
(base_Class.attribute->select(hasStereotype('Id'))->size()=1 and
 base_Class.attribute->forAll(extension_EmbeddedId.oclIsUndefined()) and
 base_Class.extension_IdClass.oclIsUndefined()
) or  --NOPK
(base_Class.attribute->forAll(extension_Id.oclIsUndefined()) and
 base_Class.attribute->forAll(extension_EmbeddedId.oclIsUndefined()) and
 base_Class.extension_IdClass.oclIsUndefined()
)
```

Esta invariante verifica se as entidades se enquadram e uma das quatro situações descritas abaixo:

1. Chave composta com tipo embutido: Verifica se a classe base possui uma (e apenas uma) propriedade marcada como *EmbeddedId* e não possui nenhuma propriedade marcada como *Id*.
2. Chave composta com várias propriedades mapeadas: Verifica se a classe base tem aplicado o estereótipo *IdClass* e se existe mais do que uma propriedade marcada com o estereótipo *Id*.
3. Chave simples: Verifica se a classe base tem uma (e apenas uma) propriedade marcada como estereótipo *Id*, nenhuma propriedade marcada como *EmbeddedId* e não tem aplicado o estereótipo *IdClass*.
4. Sem chave definida: Verifica se a classe base não tem aplicado o estereótipo *IdClass* e se seus atributos não tem aplicados os estereótipo *Id* ou *EmbeddedId*.

4.2.3.5 Mapeamento entre chave composta e classe que a representa

Se a chave primária composta é mapeada para várias propriedades na entidade, os nomes e tipos destas propriedades devem corresponder aos nomes e tipos das propriedades na classe da chave primária (DEMICHIEL e KEITH, 2006).

Para classes com a extensão *IdClass*, é preciso testar se todas as propriedades na classe de chave primária estão presentes na entidade, marcadas com *Id* e têm o mesmo tipo – e que somente estas propriedades estejam marcadas com o estereótipo *Id*.

```
context Entity
inv rule214_p7F:
base_Class.hasStereotype('IdClass') implies
        let idprops : Sequence(uml::Property)=
                base_Class.extension_IdClass.idClass.attribute in
        let clprops : Sequence(uml::Property)=
                base_Class.attribute->select(hasStereotype('Id')) in
        clprops->forAll(p|idprops->exists(name=p.name and type=p.type)) and
        idprops->forAll(p|clprops->exists(name=p.name and type=p.type))
```

A invariante verifica se a entidade possui o estereótipo *IdClass*. Em caso afirmativo, declara duas listas de propriedades, uma contendo as propriedades da classe declarada como identificadora (*idprops*) e outra contendo as propriedades da classe base da entidade anotadas com *Id (clprops)*. Para cada propriedade em *idprops* deve haver uma propriedade em *clprops* com mesmo tipo e nome e vice versa.

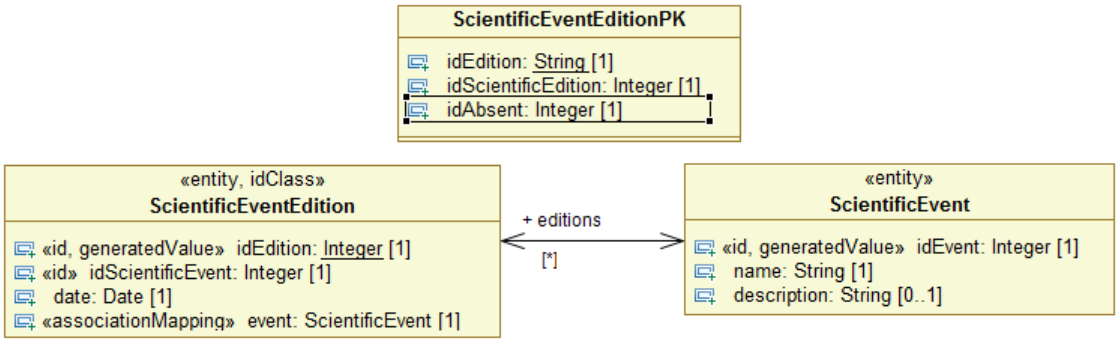


Figura 4.6: Mapeamento incorreto entre classe que representa o identificador e propriedades do identificador.

Na Figura 4.6 a classe *ScientificEventEdition* apresenta dois problemas em sua classe de chave primária. O primeiro problema é o atributo *idEdition* que está definido como um *Integer* na entidade e como um *String* na classe *ScientificEventEditionPK*. O segundo problema é a propriedade *isAbsent* que está definida apenas na classe de identificador e não está definida na entidade.

4.2.3.6 Outras regras para identificadores

Regras ainda foram definidas para as seguintes restrições de classes que representam as chaves primárias compostas:

- A classe de chave primária deve ser pública.
- A classe de chave primária deve ter um um construtor público sem argumentos.
- As propriedades da classe de chave primária devem ser públicas ou protegidas.

4.2.4 Relacionamentos

Relacionamentos nos modelos UML são elementos de primeira classe, sendo considerados persistentes quando envolvem duas entidades. A avaliação dos modelos

está focada na correta utilização dos recursos mais avançados de relacionamentos, tais como especificação de tabelas e chaves primárias de junção.

4.2.4.1 Uso de tabela para junção em relacionamento

"A anotação *JoinTable* é especificada no lado proprietário de associações *muitos para muitos*, ou em uma associação *um para muitos* unidirecional" (DEMICHIEL e KEITH, 2006, p.186).

A tabela de junção em relacionamento é criada em dois casos do mapeamento objeto-relacional, nos quais relacionamentos não podem ser representados no SGBD: no caso de relacionamentos muitos para muitos, que sempre necessitam de uma tabela intermediária; e no caso de relacionamentos unidirecionais um para muitos, pois não há como armazenar a informação de um relacionamento em uma tabela "mestre" e se esta informação for armazenada na tabela detalhe, então o relacionamento passaria a ser bidirecional.

A tabela de junção é criada automaticamente pelo *framework* JPA, mas o usuário pode especificar como esta tabela será criada através da anotação *JoinTable.* No perfil MD-JPA, a tabela de junção pode ser especificada na propriedade *joinTable* do estereótipo *AssociationMapping* aplicado à propriedade que participa de um relacionamento. A invariante a seguir verifica a correta utilização da tabela de junção:

```
context AssociationMapping
inv rule91_25_p1:
not(self.joinTable.oclIsUndefined()) implies
        ( base_Property.oneToMany() and
          not(base_Property.getOtherEnd().isNavigable())
        ) or
        base_Property.manyToMany()
```

As aplicações do estereótipo *AssociationMapping* são selecionadas no contexto da regra. Caso a propriedade *joinTable* tenha sido definida, é verificado, pela expressão *implies*, se a propriedade base faz parte de um relacionamento um para muitos unidirecional (propriedade *isNavigable* com valor falso), ou se faz parte de um relacionamento muitos para muitos.

4.2.4.2 Especificação de chave primária de junção

"A anotação *PrimaryKeyJoinColumn* […] pode ser utilizada em mapeamentos um para um nos quais a chave primária da entidade referenciante é utilizada como chave estrangeira para a entidade referenciada." (DEMICHIEL e KEITH, 2006, p.194).

O mapeamento de chave primária de junção é especificado no MD-JPA em objetos do tipo *PKJoinColumn* em relacionamentos, na herança ou em uma definição de tabela secundária. Porém, quando utilizada em relacionamentos, estes devem obrigatoriamente ser de mapeamento um para um, o que é verificado na invariante a seguir:

```
context AssociationMapping
inv rule91_32_p2:
not(pkJoinCols->isEmpty()) implies base_Property.oneToOne()
```

A invariante verifica as especificações de *AssociationMapping.* Se houver uma especificação de chave na propriedade *pkJoinCols,* então a propriedade base deve participar de um relacionamento do tipo um para um.

4.2.5 Herança

No MD-JPA a herança entre entidades é representada na relação de especialização, e não nas classes que participam da herança, como no JPA. A principal verificação realizada é quanto à utilização correta do valor discriminador das especializações, quando este é especificado.

4.2.5.1 Valor discriminador

O valor discriminador deve ter seu tipo consistente com o tipo da coluna discriminadora, seja ela especificada ou padrão (DEMICHIEL e KEITH, 2006).

A coluna discriminadora é utilizada na implementação do mapeamento de herança para determinar a qual classe cada conjunto de informações pertence. O usuário pode determinar o tipo de dados que a coluna discriminadora utiliza, bem como os valores que cada classe deve atribuir a esta coluna. Por exemplo, na hierarquia de *Course*, que inclui classes como *Graduation* e *Doctorate*, o usuário pode definir a coluna discriminadora como sendo um caractere, e atribuir o valor discriminador *G* para *Graduation* e *D* para *Doctorate*. Porém, se o usuário definir a coluna discriminadora como um inteiro, os valores discriminadores G e D serão inválidos.

O valor discriminador pode ser especificado em dois estereótipos diferentes no MD-JPA, no *DiscriminableGeneralization* que identifica o discriminador para uma especialização ou no *DiscriminatorColumn* que identifica o discriminador da superclasse. Foram especificadas duas regras, uma para cada estereótipo, para verificar se o valor fornecido está de acordo com o tipo definido no estereótipo *DiscriminatorColumn* da superclasse:

```
context DiscriminatorColumn
inv rule91_31_p4A:
not(self.discriminatorValue.oclIsUndefined()) implies
        (
                discriminatorType=DiscriminatorType::INTEGER and
                not(discriminatorValue.toInteger().oclIsUndefined())
        ) or (
                discriminatorType=DiscriminatorType::CHAR and
                discriminatorValue.size()=1
        ) or discriminatorType=DiscriminatorType::STRING
```

Esta invariante avalia as extensões *DiscriminatorColumn* definidas pelo usuário. Se o usuário definir um valor discriminador (*implies*), então ele é testado para cada um dos três tipos permitidos: em caso de ser um inteiro (INTEGER), avalia se o valor pode ser convertido para um inteiro; em caso de ser um caractere (CHAR), verifica se o comprimento do valor é de apenas um caractere; caso seja um *String*, nenhuma verificação precisa ser realizada. Se o usuário não define o valor discriminador, o *framework* JPA gerará um valor adequado.

```
context DiscriminableGeneralization
inv rule91_31_p4B:
let dc:persistence::DiscriminatorColumn=
base_Generalization.general.oclAsType(uml::Class).extension_DiscriminatorColumn in
(
        dc.discriminatorType.oclIsUndefined() or
```

```
        discriminatorValue.oclIsUndefined() or
        dc.discriminatorType=DiscriminatorType::STRING or
        (
                dc.discriminatorType=DiscriminatorType::INTEGER
                and not(discriminatorValue.toInteger().oclIsUndefined())
        ) or (
                dc.discriminatorType=DiscriminatorType::CHAR
                and discriminatorValue.size()=1)
        )
```

A segunda invariante avalia as extensões *DiscriminableGeneralization*, que podem ser aplicadas nas relações de generalização. Primeiramente é declarada a variável *dc* com a especificação da coluna discriminadora realizada pelo usuário na classe especializada (através da extensão *extension_DiscriminatorColumn,* na propriedade *general* da generalização base do estereótipo). Se o tipo de discriminador em *dc* não foi definido, se o valor do discriminador não foi definido ou se o tipo de discriminador em *dc* for *string*, a invariante estará satisfeita. Caso o tipo de discriminador seja inteiro ou caractere, a compatibilidade é verificada de forma análoga à invariante anterior.

4.2.6 Sobrescrita de mapeamento

A sobrescrita de mapeamento prepara uma classe não persistente para que suas informações possam ser persistidas em entidades que a especializam. As verificações apresentadas estão focadas na sobrescrita de atributos e associações, quando são especificadas nas especializações.

4.2.6.1 Sobrescrita de associação

"A anotação de sobrescrita de associação pode ser aplicada em entidades que estendem superclasses mapeadas para sobrescrever um mapeamento muitos-para-um ou um-para-um definidos nesta superclasse mapeada." (DEMICHIEL e KEITH, 2006, p.174).

A sobrescrita de associação é um recurso que permite sobrescrever um relacionamento quando ele é definido em uma superclasse mapeada. No MD-JPA, esta sobrescrita é realizada pelo estereótipo *AssociationOverrides* que tem como elemento base a relação de generalização.

Por ser aplicada na relação de generalização, a sobrescrita de associação deve referenciar um relacionamento herdado através deste relacionamento, mas que pode ter se originado em algum ancestral da superclasse. Para verificar se a sobrescrita está correta, a invariante a seguir utiliza uma função recursiva (*recurseMapped*), que verifica se a classe da associação sobrescrita é uma superclasse mapeada através da generalização base.

```
context AssociationOverrides
inv rule91_12_p2:
overrides->collect(property)->forAll(p|
        p.isAssociationProp() and
        base_Generalization.recurseMapped(p.class) and
        (p.manyToOne() or
         p.oneToOne()
```

```
        )
)
```

Cada extensão *AssociationOverrides* pode sobrescrever uma ou mais associações, armazenadas na propriedade *overrides*. Cada objeto na coleção de *overrides* é uma instância de *AssociationOverride*, que referencia uma propriedade do modelo que é sobrescrita. A invariante verifica se para todas as sobrescritas em *overrides*, a propriedade sobrescrita (*property*) faz parte de uma associação persistente (*isAssociationProp*), faz parte de uma superclasse mapeada da hierarquia e tem uma cardinalidade adequada.

4.2.6.2 Correta Sobrescrita de atributos

A anotação de sobrescrita de atributo (*AttributeOverride*) pode ser aplicada em uma entidade que estende uma superclasse mapeada, em uma propriedade embutida para sobrescrever o mapeamento básico definido por uma superclasse mapeada ou classe embutível (DEMICHIEL e KEITH, 2006).

O estereótipo *AttributeOverride* do MD-JPA pode ser utilizado tanto em classes, quanto em propriedades. Caso esteja associado a uma classe, os atributos sobrescritos podem referenciar propriedades em uma uma superclasse mapeável, desde que estas propriedades não sejam associações persistentes (caso tratado com *AssociationOverride*); ou referenciar uma propriedade em uma superclasse de uma hierarquia embutível. Caso esteja associado a uma propriedade, o atributo sobrescrito pode tanto referenciar propriedades em uma superclasse embutível, quanto propriedades de um mapeamento embutido na propriedade - isto é, ou o tipo da propriedade é embutível, ou a classe é embutível e a propriedade sobrescreve um mapeamento.

```
context AttributeOverrides
inv rule91_10_p2:
if (not(base_Class.oclIsUndefined())) then
        value->collect(property)->forAll(p|
                (       not(p.isAssociationProp()) and
                        base_Class.generalization->exists(g|g.recurseMapped(p.class))
                ) or (
                        base_Class.generalization->exists(g|g.isAncestral(p.class)) and
                        base_Class.hasStereotype('Embeddable')
                )
        )
else
        value->collect(property)->forAll(p|
                (       base_Property.class.generalization
                                ->exists(g|g.isAncestral(p.class)) and
                        base_Property.class.hasStereotype('Embeddable')
                ) or (
                        base_Property.type.hasStereotype('Embeddable') and
                        base_Property.type = p.class
                )
        )
endif
```

A invariante verifica as definições de sobrescritas de atributos, separada em duas verificações distintas: a primeira para a sobrescrita de atributo na classe e a segunda para sobrescrita de atributo na propriedade. Esta separação é feita verificando se a propriedade de classe base (*base_Class*) é nula. Todas as propriedades sobrescritas são testadas para que não façam parte de associações, mas que façam parte de uma superclasse mapeada (chamando a função recursiva *recurseMapped*) ou superclasse embutível (com o estereótipo *Embedded*). No caso de estender uma propriedade, ela pode sobrescrever o mapeamento da classe embutível representada na propriedade (denotado na operação *base_property.type*).

4.2.7 Restrições fora da especificação JPA

As regras apresentadas na seções anteriores cobrem todas as restrições levantadas a partir da especificação do JPA. Entretanto, existem alguns problemas que se mostraram comuns durante a construção dos modelos e que fogem do escopo da especificação JPA. Com o objetivo de detectar estes problemas, esta seção apresenta um conjunto de restrições adicionais utilizadas para verificar modelos MD-JPA.

O primeiro tipo de problema fora da especificação JPA é decorrente de diferenças entre o perfil MD-JPA e o mecanismo de anotações utilizado na implementação JPA. Por exemplo, no MD-JPA a estratégia de herança é registrada na relação de generalização de cada classe de especialização, enquanto que nas anotações JPA a estratégia de herança é registrada diretamente na superclasse. Esta diferença pode levar o usuário a definir estratégias diferentes para cada especialização, o que não ocorre quando se utiliza o recurso de herança do JPA. Este problema pode ser verificado por uma restrição adicional.

Outra classe de problemas diz respeito ao mapeamento dos elementos do modelo para uma linguagem de programação, ou para o banco de dados. As ferramentas de modelagem UML permitem definições incompletas para as classes, tais como classes sem nome, ou propriedades sem tipo. Outro problema comum é a duplicidade de elementos com mesmo nome em contextos que não permitem elementos de mesmo nome em uma linguagem de programação, como por exemplo classes com mesmo nome em um mesmo pacote, ou propriedades com um mesmo nome em uma mesma classe.

Por fim, existem os problemas comuns que dizem respeito ao mapeamento específico de uma linguagem, como por exemplo a limitação de herança múltipla imposta na plataforma Java. Na UML é permitido construir modelos onde classes especializam mais que uma superclasse, o que também é permitido em certas linguagens de programação como o C++. Com o objetivo de facilitar a avaliação dos modelos, invariantes foram também especificadas para verificar se os modelos estão de acordo com as limitações da plataforma Java.

Estas regras adicionais foram criadas para complementar a detecção de erros nos modelos e projetadas para formar um conjunto extensível, ou seja, que permita ao usuário incorporar novas regras para verificar aspectos do modelo que considere inadequados. O conjunto completo da implementação das restrições e suas funções auxiliares pode ser consultado no Apêndice A. A Tabela 4.1 traz uma lista com as invariantes adicionais, seu contexto e uma breve descrição de sua função.

Tabela 4.1: Restrições adicionais do MD-JPA.

Invariante	***Contexto***	***Descrição***
ruleProfile_1	AssociationMapping	O mapeamento de associações só é permitido em associações entre entidades.
ruleProfile_13	AssociationMapping	*JoinTable* só pode ser definido em um lado da relação.
ruleProfile_11	AssociationMapping	*MapKey* deve referenciar uma propriedade na entidade referenciada que é a chave da coleção mapa e o tipo de coleção deve ser *Map*.
ruleProfile_5	Entity	Cada entidade que é uma superclasse deve ter o mesmo tipo de estratégia de herança definida para todos os seus descendentes.
ruleProfile_6	Generalization	Cada relação de especialização só pode ter um estereótipo definindo a estratégia de herança.
ruleProfile_7	Association Overrides	O estereótipo deve ser aplicado apenas na generalização que referencia uma superclasse mapeada.
ruleProfile_9	AttributeOverrides	Pelo menos uma propriedade deve ser especificada.
ruleProfile_14	uml::Property	Cada propriedade deve ter um nome único na classe.
ruleProfile_15	Column	Se duas colunas têm o mesmo nome em uma entidade, então suas tabelas devem ser diferentes.
ruleProfile_16	Column	Se uma tabela é definida na coluna, ela deve fazer parte do mapeamento da entidade (tabela principal ou uma das tabelas secundárias)
ruleProfile_10	uml::Class	Uma classe só pode especializar no máximo uma outra classe.
ruleProfile_18	uml::Class	Classes não persistentes não devem ter estereótipos de mapeamento.
ruleProfile_2	Persistent	Toda classe persistente deve ter um nome.
ruleProfile_3	Persistent	Toda propriedade de classe persistente deve ter um nome.
ruleProfile_4	Persistent	Toda operação de classe persistente deve ter um nome.
ruleProfile_8	Persistent	Toda a propriedade de classe persistente deve ter um tipo definido.

4.3 Resumo e discussão

Ao longo deste capítulo foram apresentadas regras para avaliar se modelos utilizando o perfil MD-JPA estão bem formados. Estas regras verificam os modelos, tanto pelo alinhamento às principais regras da especificação JPA, quanto pela capacidade de representar um sistema que possa ser transformado em uma implementação.

O conjunto de invariantes apresentado é abrangente em relação à especificação JPA, mas é extensível. É abrangente porque aborda todas as restrições previstas na especificação JPA e todos os principais componentes especificados no perfil. Todavia, é extensível porque o usuário do perfil pode definir novas regras, utilizando a linguagem OCL, que verifiquem seus modelos para situações específicas além das previstas no conjunto proposto. Independente disso, a linguagem OCL pode ser utilizada para realizar consultas e obter métricas sobre os modelos.

Com a checagem dos modelos realizada pelas regras OCL, as transformações puderam ser implementadas sem a preocupação de tratar modelos mal-formados ou inválidos. No próximo capítulo será abordada a transformação dos modelos na implementação, dentro da abordagem DDM.

5 TRANSFORMAÇÕES

Na abordagem DDM, sistemas são desenvolvidos pela transformação de modelos com um nível mais alto de abstração em modelos com nível mais baixo de abstração. Este capítulo tem como objetivo avaliar o perfil MD-JPA neste contexto, demonstrando como modelos com o perfil MD-JPA são utilizados no desenvolvimento de sistemas através de sua transformação em código e roteiros (*scripts)* de criação de banco de dados.

Na primeira parte do capítulo é apresentado o processo que transforma modelos de classe MD-JPA, bem formados e válidos, na implementação destas classes em Java segundo o padrão JPA. A segunda parte do capítulo mostra a utilização do perfil para modelar um sistema já existente, avaliando comparativamente os resultados obtidos na geração com a implementação original deste sistema.

5.1 O processo de transformação

O processo adotado para a transformação de modelos em código é dividido em duas etapas: a transformação do modelo de classes em modelo de programa e a tradução do modelo de programa em código fonte Java. A primeira etapa é implementada em um conjunto de regras de transformação que levam dos elementos da UML e MD-JPA para elementos definidos no metamodelo da árvore sintática abstrata da linguagem Java (*Java Abstract Syntax* ou JAS) (INRIA ATLANMOD, 2009). A segunda etapa é implementada pela transformação de modelos do JAS em código fonte anotado, utilizando o módulo de desenvolvimento Java da ferramenta Eclipse (ECLIPSE FOUNDATION, 2009b).

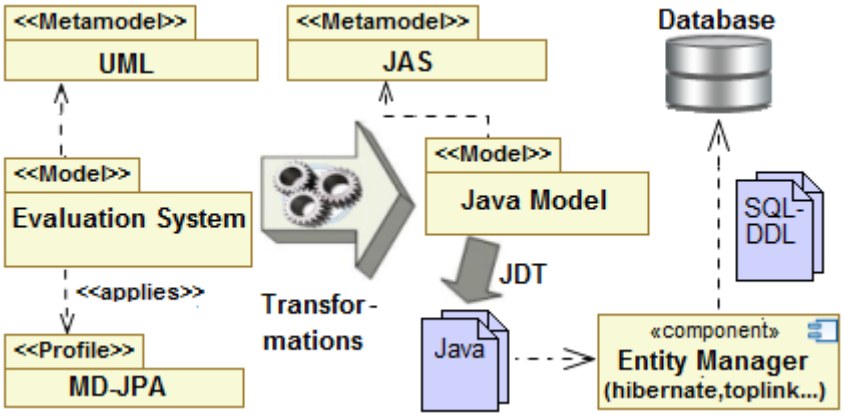


Figura 5.1: Visão geral da transformação (sistema de avaliação).

Na Figura 5.1 é apresentada uma visão geral das duas etapas de transformação, aplicadas sobre o sistema de avaliação acadêmico (*Evaluation System*). Um modelo

UML, com o perfil MD-JPA e bem formado, é a entrada para as regras de transformação que selecionam elementos do modelo de origem e produzem elementos no modelo de saída. O resultado é um modelo no metamodelo JAS, que representa exatamente o conjunto de programas a ser gerado.

A segunda etapa é executada por um programa tradutor que cria os objetos no *framework* JDT a partir do modelo JAS, gerando os arquivos fonte Java com anotações JPA. Estes fontes podem ser transformados pela maior parte dos *frameworks* MOR em instruções SQL para a definição de bancos de dados, ou podem ser utilizados em conjunto com uma base de dados pré-existente.

5.1.1 Etapa 1: Transformação do modelo de classes em modelo de código

A etapa de transformação de modelos tem como objetivo criar um modelo de código a partir do modelo MD-JPA. As transformações foram implementadas utilizando a linguagem de transformação ATL (JOUAULT et al, 2006). A ATL possui dois recursos para a transformação de elementos, as regras de casamento (*match rules*) e as regras imperativas (ou chamadas). As regras de casamento realizam as transformações estabelecendo relações entre elementos selecionados no modelo de origem e elementos produzidos no modelo de destino, enquanto que as regras imperativas funcionam como funções, gerando elementos a partir de parâmetros fornecidos pelo usuário.

No caso do perfil MD-JPA, as regras de casamento são o recurso preferencial utilizado para selecionar elementos de origem da UML (tais como classes, propriedades e operações), produzindo elementos correspondentes no metamodelo Java. As regras imperativas são utilizadas com o objetivo de complementar a transformação, nos casos em que é difícil estabelecer um relacionamento entre elementos de origem e destino.

5.1.1.1 Regras de casamento

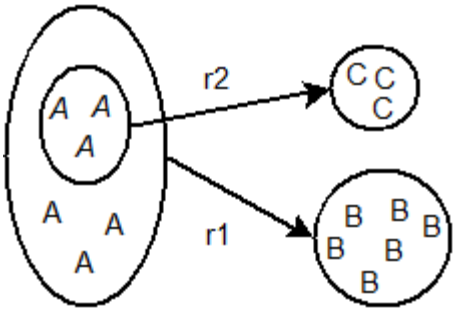
Na linguagem ATL, as regras de casamento obedecem ao seguinte formato:

```
rule <nome> {
from
  <tipo da origem> (<condição de seleção opcional>)
to
  <tipos de destino> (
    (lista de atribuição de propriedades)
  )
```

Cada regra de casamento ou declarativa estabelece o tipo de elemento que seleciona e uma cláusula de seleção opcional, na seção declarativa de origem (*from)*. Porém, cada elemento existente no modelo de origem pode ser selecionado apenas por uma regra de casamento. Tomando como exemplo uma transformação com duas regras, que tratam respectivamente de classes abstratas e classes públicas: se um modelo, contendo classes que são ao mesmo tempo abstratas e públicas, for transformado, as classes abstratas e públicas serão selecionadas pelas duas regras, levando a transformação a um estado de erro.

Os elementos produzidos em uma regra são definidos pela seção declarativa *to*, que permite a definição de um ou mais elementos de *destino* para cada elemento selecionado. Entretanto, a seção *to* não permite a definição de uma cláusula de seleção: se uma regra declara a geração dos elementos *B* e *C* a partir do elemento *A*, não é possível acrescentar uma condição nesta mesma regra para geração apenas de *C* em alguma situação especial.

A solução declarativa para este problema passa pela criação de duas regras para os elementos *A,* que geram ou não geram *C*. Para remover a definição redundante da geração de *B*, a linguagem ATL oferece o recurso de extensão de regra. Supondo-se uma regra *r1* que define a geração de *B* para todos os elementos *A*, pode-se escrever uma regra *r2* que estende *r1*. A regra *r2* deve definir como origem um subconjunto de *A*, e só necessita declarar na seção de *destino* a geração de *C*.




```
rule   r1 {
from   a : A
to     b : B }

rule   r2 extends r1 {
from   a : A   (a.isItalic)
to     c : C }
```

Figura 5.2: Extensão entre regras de casamento.

A figura 5.2 exemplifica a definição de extensão de regra para se definir duas regras sobre um mesmo domínio. No exemplo, deseja-se que todo elemento *A* gere um elemento *B*, mas que os elementos *A* em itálico gerem também um elemento *C*. A regra *r1* produz elementos *B* a partir de elementos *A* quaisquer, enquanto a regra *r2* estende a definição da regra *r1* produzindo elementos *C* a partir de *A*, quando *A* é itálico.

Para a transformação de modelos MD-JPA em modelos de código, foi definido um conjunto de regras declarativas que originam-se nos elementos da UML. Um fragmento da regra *genClasses* é apresentada como exemplo, a seguir.

```
rule genClass {
from
  cl : UML!Class  (not(cl.isEGenericType()))
to
  typ: JAST!TypeDeclaration (
    superclassType<-
      if (cl.superClass->isEmpty()) then
        OclUndefined
      else
        cl.superClass->first().genAnyType(cl)
      endif,
    superInterfaceTypes<- (...),
    bodyDeclarations<- (...),
    name<-thisModule.genSimpleName(cl)
  ),
  cun: JAST!CompilationUnit (
    package <- thisModule.genPackageDcl(cl)
  )
```

O conjunto de origem desta regra é formado por todos os elementos que são classes, mas não definem parâmetro de *template* (a função *isEGenericType* faz esta verificação). As classes utilizadas como parâmetro de *template* existem apenas para definição de tipos genéricos e, por esse motivo, não são selecionadas nesta regra.

A seção destino (*to*) define dois objetos resultantes: uma declaração de tipo (*JAST! TypeDeclaration*) e uma unidade de compilação (*JAST!CompilationUnit*). Uma lista de atribuições é definida para cada objeto, declarando para cada propriedade do objeto destino seu valor, através de expressões OCL a partir dos objetos selecionados. No caso

da declaração de tipo, que corresponde a uma classe Java, são definidas as propriedades *superclassType, superInterfaceTypes, bodyDeclarations* e *name*.

A propriedade *superclassType* informa qual é a superclasse em Java. Cada classe UML pode ter um conjunto de superclasses definido. Se este conjunto for vazio, a propriedade recebe o valor nulo. Se o conjunto não for vazio, a primeira classe será utilizada como superclasse. A superclasse ainda precisa ser transformada em uma "referência ao tipo" pela função *genAnyType*, dentro do contexto da unidade de programa da classe. A referência ao tipo pode incluir o pacote (na forma "*extends com.acme.Magnet*") ou apenas o nome da classe referenciada ("*extends Magnet*"), dependendo deste contexto.

As demais propriedades referenciam outros componentes de uma classe, tais como nome, interfaces implementadas, e as declarações de corpo. As declarações de corpo são importantes na árvore sintática da linguagem, pois relacionam todos os elementos que fazem parte do corpo da classe tais como métodos e variáveis de instância – elementos estes que são gerados por outras regras de casamento.

Tabela 5.1: Regras Relacionais.

Regra	***Origem/condição***	***Destino(s)***
genClass	UML~Class Não ser variável para tipo genérico	JAST~TypeDeclaration, JAST~CompilationUnit
genMethod	UML~Operation Operação de classe	JAST~MethodDeclaration
genProperty	UML~Property Propriedade de classe	JAST~FieldDeclaration, JAST~MethodDeclaration
genEnum	UML~Enumeration	JAST~EnumDeclaration, JAST~CompilationUnit
genEnumLiterals	UML~EnumerationLiteral	JAST~EnumConstantDeclaration
genInterface	UML~Interface	JAST~TypeDeclaration, JAST~CompilationUnit
genInterfaceAttrs	UML~Property Propriedade de interface	JAST~MethodDeclaration
genInterfaceMethods	UML~Operation Método de interface	JAST~MethodDeclaration
genGeneralClass **extends** genClass	UML~Class Ter alguma especialização com mapeamento	JAST~NormalAnnotation (tipo de herança)
genTypeParameters	UML~ClassifierTemplateParameter Ter assinatura	JAST~TypeParameter

A regra *genProperty,* por exemplo, seleciona cada propriedade para gerar uma variável de instância privada e um par de métodos de acesso (*getter* para leitura, *setter* para escrita). Da mesma forma, a regra *genMethod* seleciona as operações, produzindo métodos. A Tabela 5.1 lista as regras declarativas do MD-JPA, determinando os elementos de origem, suas condições (quando existentes) e os elementos de destino.

A execução da regra *genClasses,* na geração do modelo de código, é exemplificada utilizado um fragmento do modelo de avaliação que mostra as relações entre curso, avaliação de curso, avaliação e intervalos de datas. Cada classe deste diagrama é selecionada pela regra *genClasses*, com exceção da classe *Course,* que tem especializações com mapeamento, sendo selecionada pela regra *genGeneralClass* (que estende *genClasses*). A Figura 5.3 ilustra o diagrama de classes e um fragmento do modelo de código resultante.

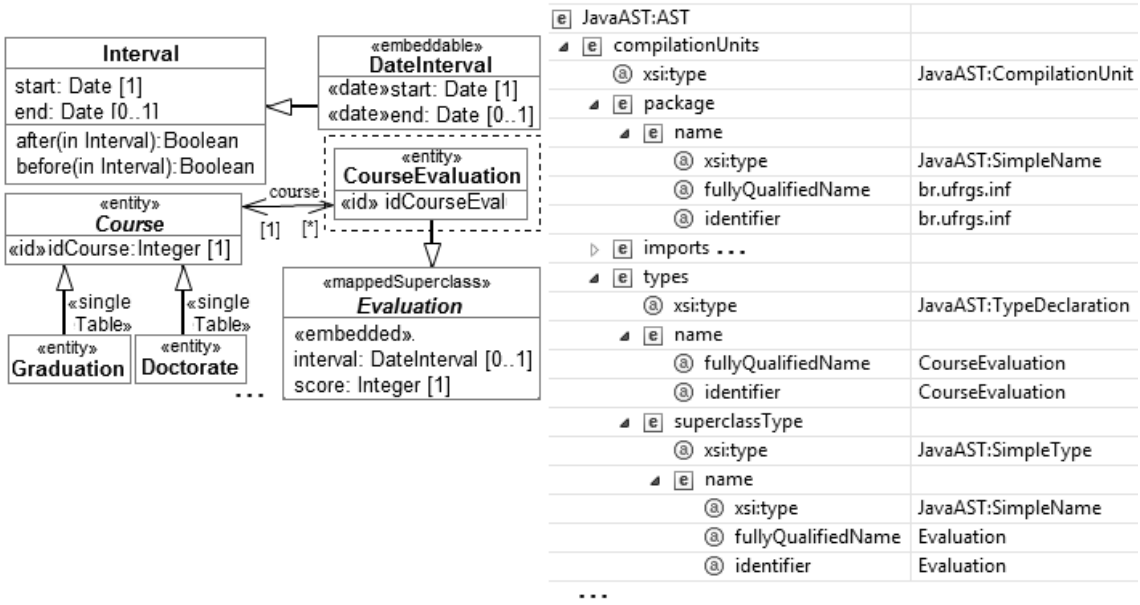


Figura 5.3. Diagrama de classes e modelo de código gerado para *CourseEvaluation.*

O modelo de código, gerado a partir de *CourseEvaluation,* começa com uma unidade de compilação contendo a identificação de pacote, importação de outros pacotes e uma lista de tipos declarados (*types*). Esta lista de tipos contém apenas a classe principal (em *TypeDeclaration*), que é uma declaração de tipo no metamodelo JAS.

Uma das propriedades do elemento *TypeDeclaration* é a *superclassType*, que consiste em uma referência a um tipo. Como *CourseEvaluation* e *Evaluation* foram declaradas no mesmo pacote, a declaração de tipo é simples (*SimpleType*), contendo apenas o nome da classe relacionada. Porém, a definição deste nome é também um objeto no modelo, da classe *SimpleName,* totalizando a criação de dois objetos para representar a extensão de classe. Este tipo de detalhe não é tratado pelas regras de casamento, mas através de funções e regras auxiliares, como a *genAnyType*, que têm um caráter imperativo.

5.1.1.2 Regras imperativas

As regras de casamento permitem também a definição de uma seção imperativa, na qual é possível programar a chamada de outras regras de produção parametrizadas. Neste caso, pode-se gerar elementos a partir da avaliação de um conjunto de propriedades do elemento de origem e suas relações com outros elementos do modelo. Sua principal utilidade está na geração de elementos que não têm uma origem óbvia no modelo de entrada. As regras imperativas são uma alternativa para evitar a criação de regras de casamento para todas as combinações possíveis de elementos produzidos por diferentes circunstâncias de um mesmo elemento (JOUAULT et al, 2006).

Um exemplo de regra imperativa é utilizado com os modificadores de visibilidade e de elemento abstrato aplicados às classes, propriedades ou operações. Os modificadores no modelo de destino Java são elementos de primeira ordem, mas sua geração é determinada por combinações entre as propriedades de visibilidade e abstração nos elementos da UML. Outro uso das regras imperativas está na geração de elementos derivados das classes do perfil (meta-classes), tais como *Table* ou *ColumnDefinition* , já que as regras de casamento ATL não selecionam elementos de um perfil.

A regra *genMethod* é um exemplo de utilização da cláusula imperativa ("*do*"). Esta regra seleciona operações UML, gerando declarações de métodos (*MethodDeclaration*). Os modificadores do método são gerados em uma "regra chamada" (*called rule*) auxiliar. A seguir são apresentadas a regra de geração de método e a regra de geração de modificadores.

```
rule genMethod {
from
  op : UML!Operation (...)
to
  method: JAST!MethodDeclaration (...)
do {
  if (op.visibility=#public)
    thisModule.genBodyModifiers(method,'public');
  else if (op.visibility=#private)
    thisModule.genBodyModifiers(method,'private');
  else if (op.visibility=#protected)
    thisModule.genBodyModifiers(method,'protected');
  if (op.isAbstract())
    thisModule.genBodyModifiers(method,'abstract');
  if (op.isStatic())
    thisModule.genBodyModifiers(method,'static');
  }
}
rule genBodyModifiers(typ: JAST!BodyDeclaration, pmod:String) {
to modifier: JAST!Modifier
do {
  if (pmod='public')
    modifier.public<-true;
  if (pmod='abstract')
    modifier.abstract<-true;
  if (pmod='protected')
    modifier.protected<-true;
  if (pmod='private')
    modifier.private<-true;
```

```
  if (pmod='static')
    modifier.static<-true;
  typ.modifiers<-typ.modifiers->append(modifier);
}
}
```

A cláusula *"do"* da regra *genMethod* contém a seção imperativa da regra, executada no contexto dos elementos criados pela cláusula *"to"*. Na *seção imperativa* são verificados os modificadores de visibilidade, as propriedades de abstração e se o método é estático. Para cada caso, é gerado um modificador no modelo de código correspondente, através da regra chamada *genBodyModifiers,* passando como parâmetros o método gerado e a *string* correspondente ao modificador.

A regra chamada *genBodyModifiers* não possui uma seção de origem, apenas a seção de destino e uma nova seção imperativa. É criado um objeto *Modifier* no modelo, inicializado de acordo com o tipo de modificador e, por fim, atribuído na lista de modificadores do objeto de declaração de corpo. Esta lista contém os modificadores aplicados sobre o elemento, tais como *Public, Protected* ou *Abstract*. Neste caso, a regra pode ser utilizada para qualquer declaração de corpo, tal como de propriedades e classes, além de métodos.

A Tabela 5.2 apresenta as principais regras relacionais, chamadas no contexto de regras de casamento, ou outras regras quaisquer. Ao invés de uma seção de origem, as regras chamadas possuem uma lista de parâmetros formais. Além disso, pode-se definir objetos de destino na seção declarativa, utilizando estes parâmetros para definir suas propriedades. Algumas regras chamadas podem ter apenas uma seção imperativa, responsável por chamar outras regras.

Tabela 5.2: Regras relacionais chamadas.

Regra	***Parâmetros***	***Destino(s)***	***Objetivo***
addClassAnnotations	UML~Classifier, JAST~TypeDeclaration, JAST~CompilationUnit	-	Adicionar anotações de classe baseado nos estereótipos aplicados; atualizar seção *import*
addImport	JAST~CompilationUnit, String	-	Chamar *genImport* se expressão *import* não existe na unidade de compilação
addPropAnnotations	UML~Property, JAST~TypeDeclaration	-	Adicionar anotações de propriedade baseado nos estereótipos aplicados; atualizar seção import; verificar relacionamentos e chamar regras one/many - one/many
doGenRules	UML~Classifier, JAST~TypeDeclaration, JAST~CompilationUnit	-	Seção imperativa compartilhada por genClass e genGeneralClass, chama addClassAnnotations
genBodyModifiers	JAST~BodyDeclaration, String	JAST~Modifier	Gera modificador de corpo de programa (abstrato, público, privado, estático, etc)
genImport	JAST~CompilationUnit, String	JAST~ImportDeclaration	Gera declaração de import
genMethodReturn	UML~Operation, JAST~Block	JAST~ReturnStatement	Gera comando de retorno de função
genPrimaryKeyJoinC	UML~Element,	-	Decide pela geração de anotação

Regra	*Parâmetros*	*Destino(s)*	*Objetivo*
olumn	JAST~TypeDeclaration, JAST~CompilationUnit, Sequence(UML~Element)		`PrimaryKeyJoinColumn (uma chave) ou PrimaryKeyJoinColumns (várias chaves)`
manyToMany	JAST~BodyDeclaration, UML~Property	-	Adiciona as anotações de “muitos para muitos”
manyToOne	JAST~BodyDeclaration, UML~Property	-	Adiciona as anotações de “muitos para um”
oneToMany	JAST~BodyDeclaration, UML~Property	-	Adiciona as anotações de “um para muitos”
oneToOne	JAST~BodyDeclaration, UML~Property	-	Adiciona as anotações de “um para um”
oneToOne	JAST~BodyDeclaration, UML~Property	-	Adiciona as anotações de “um para um”

Outro tipo de regra imperativa disponibilizada pelo ATL são as regras *lazy*. A diferença de uma regra *lazy* sobre as regras de casamento normais é que elas não possuem nem condição de seleção de origem nem tampouco parâmetros, e só são acionadas quando chamadas. Entretanto, as regras *lazy* funcionam como funções, retornando o objeto criado na seção de destino, permitindo sua utilização em expressões dentro da seção de destino de outras regras, como exemplificado nos seguintes fragmentos da transformação:

```
lazy rule genMethodParameter {
from
  p : UML!Parameter
to
  res : JAST!SingleVariableDeclaration (
      name <-thisModule.genSimpleNameStr(p.name),
      type <-p.type.genAnyType(p.operation)
  )
}
rule genMethod { (...)
to
  method: JAST!MethodDeclaration (
  (...)
    parameters<-method.parameters.append(op.ownedParameter
      ->select(p|not(p.direction=#return))
      ->collect(p|thisModule.genMethodParameter(p))
  ), (...)
```

O exemplo mostra a geração de parâmetros de método (*genMethodParameter*) utilizada dentro da seção destino da regra *genMethod*. Os parâmetros UML primeiramente são filtrados para remoção do retorno de função, pela função de seleção. Posteriormente, cada parâmetro é convertido para o modelo de código chamando a regra, que retorna o parâmetro transformado.

Tabela 5.3: Regras relacionais “lazy” (principais).

Regra	***Origem***	***Destino(s)***	***Objetivo***
genQualifiedNameStr	String	JAST~QualifiedName	Criar um nome qualificado a partir de uma *string* contendo separadores de pacote (“.”)
addMarkerAnnotation	JAST~TypeDeclaration, String	JAST~MarkerAnnotation	Gera uma anotação simples a partir da string de origem, no tipo declarado de origem.
addSingleMemberAnnotation	JAST~ASTNode, String, JAST~Expression	JAST~SingleMemberAnnotation	Gera uma anotação de um parâmetro a partir da *string* e da expressão, no nodo de origem
addNormalAnnotation	JAST~ASTNode, String, *pares* [String, JAST~Expression]	JAST~NormalAnnotation	Gera uma anotação a partir de uma lista de parâmetros e de uma *string*, no nodo de origem
genMethodBlock	UML~Operation	JAST~Block	Gera um bloco de programa a partir de uma operação
genMethodParameter	UML~Parameter	JAST~SingleVariableDeclaration	Gera declaração de variável de parâmetro a partir de parâmetro de método
genSetterVariable	UML~Property	JAST~SingleVariableDeclaration	Gera declaração de variável de parâmetro a partir de tipo de propriedade para método *setter*
genVariableDcl	UML~Property	JAST~VariableDeclarationFragment	Gera declaração de variável de instância para propriedade
genSetterBlock	UML~Property	JAST~Block, JAST~ThisExpression, JAST~FieldAccess, JAST~Assignment, JAST~ExpressionStatement	Gera a expressão de atribuição em um método setter a partir da variável, na forma: `this.<nome prop.> = <nome prop.>;`
genGetterReturnBlock	UML~Property	JAST~Block	Gera expressão de retorno para método de acesso à propriedade
genPackageDcl	UML~Classifier	JAST~PackageDeclaration	Gera declaração de pacote para unidade de programa
genDataType	String	JAST~SimpleType	Gera declaração de tipo simples a partir do nome do tipo.
genPrimitiveType	String	JAST~PrimitiveType	Gera declaração de tipo primitivo a partir do nome do tipo.
genGenericType	UML~TemplateBinding	JAST~ParameterizedType	Gera tipo parametrizado (genérico) a partir de ligação de *template.*
genParameterizedType	UML~TypedElement	JAST~ParameterizedType	Gera tipo parametrizado (genérico) a partir de associação com multiplicidade superior a um
genArrayType	UML~TypedElement	JAST~ArrayType	Gera *Array* a partir de elemento com tipo

A Tabela 5.3 lista as principais regras *lazy* utilizadas no MD-JPA. Na coluna origem está o tipo de entrada, que em alguns casos pode ser um “tipo composto” (tupla) ou um

conjunto de elementos. Na coluna destino estão os objetos criados, dos quais o primeiro é retornado pela regra.

O último recurso disponibilizado pela ATL é a função de ajuda (*helper*), que é útil para encapsular expressões repetitivas, como seleções de conjuntos, pequenas expressões condicionais ou manipulação de texto. Um exemplo de função de ajuda é a que cria os nomes dos métodos de acesso a partir das propriedades (função *asProperty*), concatenando os fragmentos de texto "*get*" e "*set*" ao nome da propriedade e corrigindo a primeira letra da propriedade para minúsculo, conforme o padrão *JavaBean*. Por exemplo, a propriedade "*salário*" deve ter um método de acesso declarado como "*getSalário*".

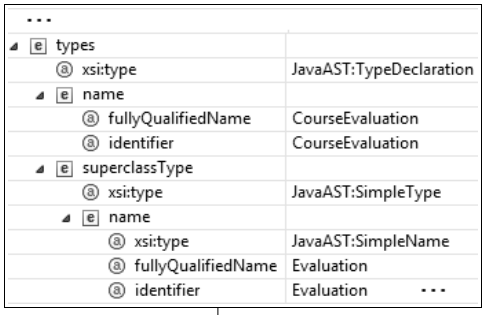
Até este ponto foram apresentadas as regras de transformação, exemplificadas no contexto do exemplo motivador, com o objetivo de transformar modelos MD-JPA em modelos da linguagem Java. O conjunto completo da implementação das transformações apresentadas nas tabelas está listado no Apêndice B para referência.

5.1.2 Etapa 2: Tradução do modelo de código em implementação

O resultado da transformação ATL é um modelo de código Java, no metamodelo JAS, armazenado em memória ou em arquivo no formato padrão XMI. O último passo para a obtenção dos arquivos de código fonte Java é a tradução dos elementos deste modelo em texto.

O metamodelo JAS (detalhado no Anexo) foi construído a partir do *framework* JDT (ECLIPSE FOUNDATION, 2009b). Assim sendo, ambos possuem a mesma estrutura, mesmos nomes de objetos e organização de dados. O JDT pode tanto ser utilizado para a leitura de código fonte e representação de sua estrutura em memória (na forma de uma árvore sintática concreta), quanto na transformação desta estrutura para código fonte, simplificando a tarefa de implementação da tradução entre modelos e código.

O metamodelo JAS já possui um módulo que traduz de código para modelo. A transformação almejada neste estudo é o processo contrário, ou seja, implementar a tradução de modelo para código: os objetos do modelo JAS para os elementos JDT, uma conversão simples de um para um. A partir desta conversão, o próprio *framework* JDT se encarrega de gerar os fontes, com opções de formatação como identação e organização de código. O modelo de código da classe *CourseEvaluation*, com fragmentos apresentados na Figura 5.3, é traduzido para a classe apresentada na Figura 5.4.




```java
package br.ufrgs.inf;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.ManyToOne;

@Entity
public class CourseEvaluation extends Evaluation {
    private Course course;
    private Integer idCourseEvaluation;

    @ManyToOne(optional = false)
    public Course getCourse() {
        return course;
    }

    @Id
    @GeneratedValue(strategy = GenerationType.TABLE)
    public Integer getIdCourseEvaluation() {
        return idCourseEvaluation;
    }

    public void setCourse(Course course) {
        this.course = course;
    }

    public void setIdCourseEvaluation(Integer idCourseEvaluation) {
        this.idCourseEvaluation = idCourseEvaluation;
    }
}
```

Figura 5.4: Classe gerada a partir do modelo de código.

As transformações entre modelo e código foram testadas utilizando o modelo de exemplo do sistema de avaliações acadêmicas e o modelo de referência da especificação JPA.

5.2 Resultados obtidos nas transformações

Esta seção avalia a adequação do perfil MD-JPA e de suas transformações na representação de um sistema previamente construído por terceiros. Para atingir este objetivo, foi construído um modelo MD-JPA a partir de uma implementação selecionada, e este modelo foi submetido às transformações, obtendo-se uma nova implementação do sistema. As duas implementações foram, então, comparadas para a avaliação do perfil.

Para a seleção do sistema à modelar, procurou-se um sistema aberto, já implementado e abrangente quanto ao uso dos recursos MOR. O sistema escolhido foi o de exemplo de referência, proposto na própria especificação JPA, chamado de "*A more complex example*" (DEMICHIEL e KEITH, 2006). O motivo para tal escolha é por trata-se de um sistema pequeno, mas que abrange as principais construções possíveis do padrão e tem seu código fonte publicado.

O modelo de referência foi construído manualmente e faz uso de extensões avançadas do padrão como tabelas secundárias, definição de junções, chaves compostas, classes embutidas com sobrescrita de atributos e herança. A Figura 5.5 mostra o modelo do pacote "*com.acme*" construído para verificar os resultados das transformações contra a implementação fornecida na especificação.

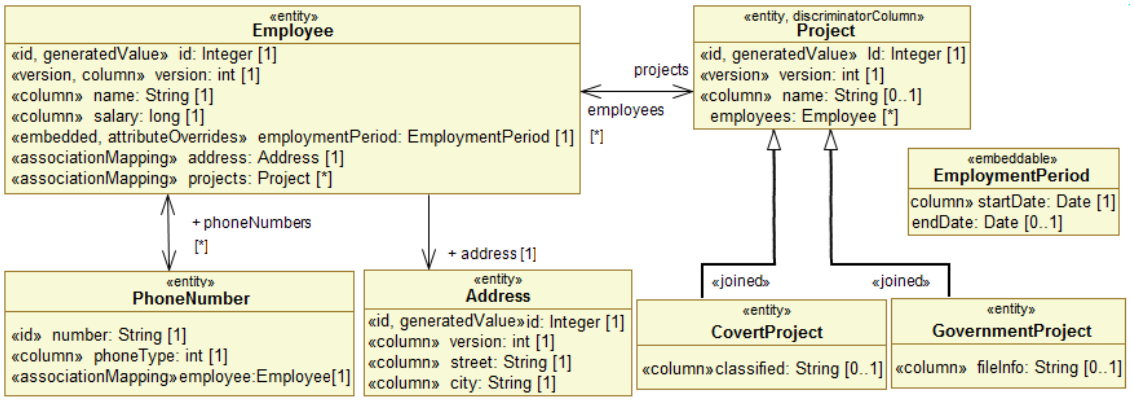


Figura 5.5: O modelo do sistema "*acme complex example*" da especificação JPA.

As únicas diferenças semânticas, entre o código fonte gerado pela transformação e o código original, foram encontradas nos nomes de variáveis de instância, quando eram diferentes do nome da propriedade, e no uso de tipos genéricos do Java. As diferenças já eram esperadas, devendo-se a decisões tomadas para facilitar a construção dos modelos e simplificação de elementos da linguagem Java. Ainda assim, as anotações geradas são equivalentes às do código original.

Ao permitir ao usuário especificar as propriedades e gerar automaticamente os métodos de acesso, o nome utilizado para os métodos e sua propriedade é o mesmo. Apesar desta diferença não influenciar a estrutura da aplicação no caso analisado, ela limita a modelagem de classes quanto a como uma variável de instância (privada) é utilizada para armazenar a informação de determinada propriedade.

Quanto aos tipos genéricos, a opção por sempre gera-los em relacionamentos, também objetiva simplificar a modelagem, especialmente por causa das dificuldades inerentes a expressão de tipos genéricos na própria UML. A UML aborda tipos parametrizáveis com o uso de *templates*, que são distintos dos tipos genéricos do Java, ainda que seja possível representar algumas construções *generics* com *templates* UML, e já existam extensões para atender o que não pode ser representado de forma padrão (BRUCK, 2007).

O próximo passo foi a comparação entre os modelos de dados, obtidos a partir do código fonte gerado do modelo e a partir do código fonte original da especificação. Para realizar a comparação foram utilizados dois *frameworks* de código aberto que implementam o JPA, e geram códigos SQL para criação de banco de dados: o *Hibernate* (RED HAT MIDDLEWARE, 2009) e o *Open-JPA* (APACHE FOUNDATION, 2009). Para cada um dos dialetos de bancos de dados, a comparação entre as bases geradas pelos dois fontes não apresentou nenhuma diferença.

O modelo *acme* MD-JPA, construído a partir da engenharia reversa do sistema de referência, foi transformado em uma implementação equivalente à original. Assim sendo, pode-se afirmar que o perfil MD-JPA é suficientemente abrangente para representar o sistema de exemplo. Também pode-se afirmar que o processo de

transformação apresentado possibilita a transformação de modelos MD-JPA bem formados em uma implementação válida.

5.3 Resumo e discussão

Este capítulo mostrou como modelos construídos com o perfil MD-JPA são utilizados no contexto do desenvolvimento dirigido por modelos. O conjunto de transformações implementado realiza a transformação dos modelos MD-JPA em implementação Java com anotações JPA. Para avaliar a eficácia das transformações, um modelo MD-JPA foi construído, a partir de uma implementação fornecida pela especificação JPA, a fim de comparar os resultados das transformações com uma implementação original.

A transformação do modelo em implementação é realizada em duas etapas. Na primeira etapa, regras de transformação levam elementos da UML e do perfil MD-JPA para elementos em um modelo de árvore sintática da linguagem Java. Estas regras foram escritas na linguagem ATL de transformação entre modelos. Na segunda etapa, o modelo de árvore sintática é traduzido em código fonte.

O sistema de exemplo apresentado para avaliar as transformações foi modelado e transformado em implementação. Esta implementação foi primeiramente comparada com os fontes originais e, posteriormente, quanto ao banco de dados gerado por diferentes *frameworks* JPA, obtendo resultados satisfatórios em ambas as comparações. O próximo capítulo descreve as ferramentas desenvolvidas para a utilização prática do MD-JPA.

6 IMPLEMENTAÇÃO

Este capítulo descreve a implementação de um *plugin* para ferramenta Eclipse, que centraliza os metamodelos, transformações e restrições, integrando, através de menus e editores, esses recursos com outras ferramentas de modelagem. O seu desenvolvimento levou em consideração todas as características definidas para o perfil MD-JPA e as regras utilizadas para transformação e avaliação no contexto da abordagem DDM. O principal objetivo de tal implementação é disponibilizar o perfil apresentado e seus recursos para a comunidade, viabilizando a utilização prática da abordagem proposta neste trabalho.

A ferramenta MD-JPA[13] é um módulo *plugin* com licença de código aberto que pode ser utilizado em conjunto com o editor UML padrão da ferramenta Eclipse, ou com outros editores compatíveis com a implementação UML feita pelo Eclipse. A ferramenta é composta pelo modelo do perfil MD-JPA, um modelo com os elementos básicos do kit de desenvolvimento Java, uma extensão de menu para avaliação de modelos via OCL, uma extensão de menu para geração de código a partir de modelos e um editor auxiliar de estereótipos.

A Figura 6.1 apresenta a arquitetura da ferramenta MD-JPA, construída para a plataforma Eclipse versão 3.5. O módulo MDT-UML2 é a implementação padrão da UML para o ambiente Eclipse, construída sobre os módulos ECore e EMF (*Eclipse Modeling Framework*). O ECore é uma variação da linguagem EMOF (*Essential MOF*) proposta pela OMG para definição de metamodelos que descrevem linguagens de modelagem e o EMF é um *framework* que implementa a gerência e construção de modelos. Porém o MDT-UML2 não é uma ferramenta de modelagem completa pois não permite a construção de diagramas, mas é utilizada por outras ferramentas de modelagem da plataforma eclipse. As regras de avaliação são executadas pelo interpretador OCL, também construído sobre os módulos MDT-UML2, ECore e EMF. As versões dos componentes utilizados são as compatíveis com a versão 3.5 da plataforma Eclipse.

O componente ATL é utilizado para executar as transformações definidas na ferramenta. Além do metamodelo da UML, é utilizado o metamodelo JAS e o componente JDT, já introduzidos no capítulo 5. O armazenamento em arquivo dos modelos é realizado no formato XMI, pelo componente de mesmo nome.

A implementação gerada em Java com anotações JPA pode ser compilada e executada dentro da ferramenta Eclipse, que pode ser configurada para utilizar uma das implementações disponíveis do padrão JPA. O roteiro SQL para geração ou manutenção

13 Disponível em http://wiki.inf.ufrgs.br/mediawiki/index.php/MD-JPA

do banco de dados é gerado a partir da implementação, utilizando *plugins* específicos para cada *framework*.

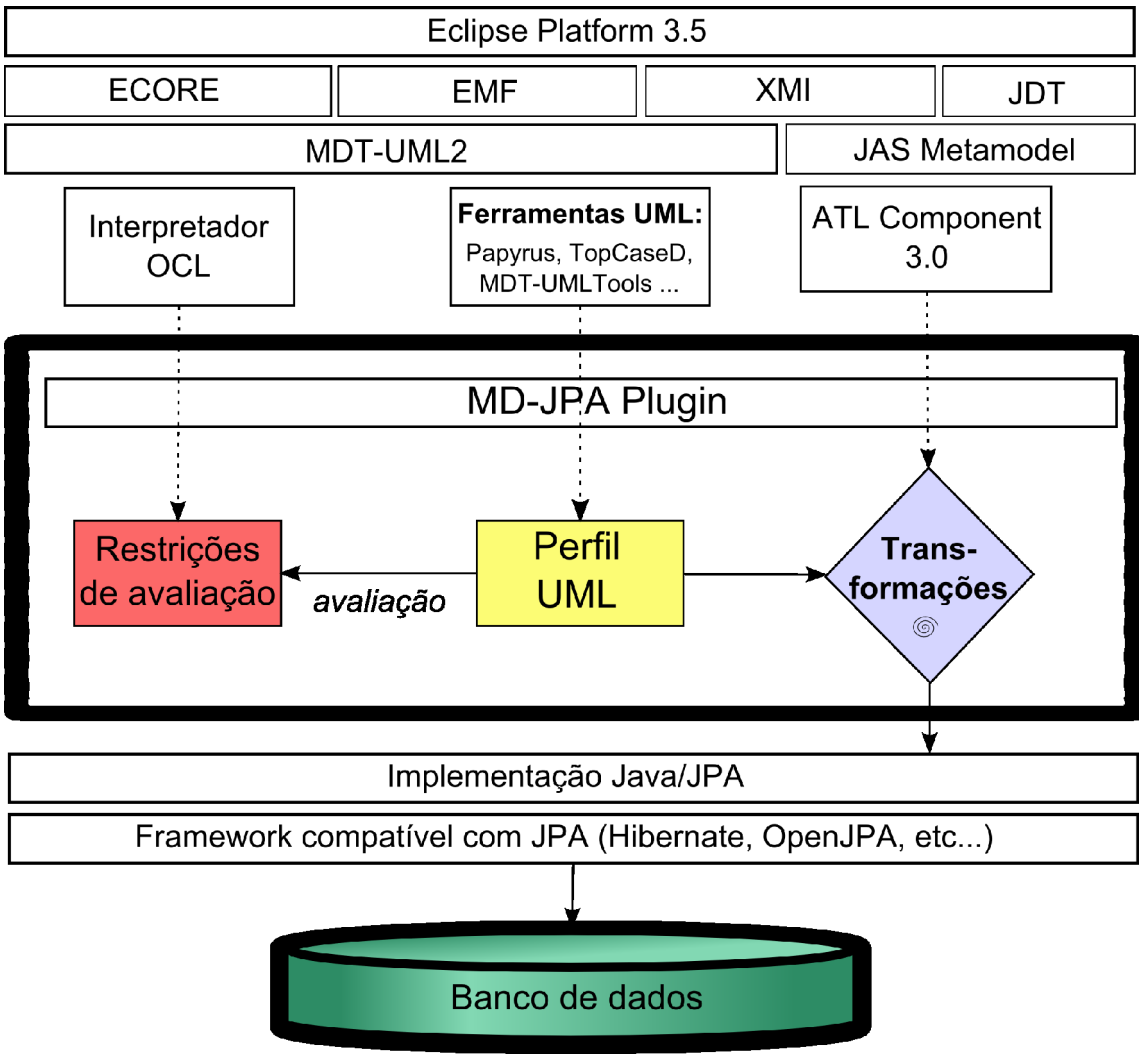


Figura 6.1: Arquitetura da ferramenta MD-JPA.

O modelo do perfil MD-JPA foi construído dentro da plataforma Eclipse, seguindo os padrões da UML 2, tendo como resultado final um arquivo no formato XMI. A plataforma Eclipse disponibiliza um *framework* para edição de modelos UML, utilizado por várias ferramentas para criação de diagramas, que via de regra são armazenados em um arquivo separado. Desta forma, a parte gráfica que compreende elementos como posição, tamanho e cor dos elementos (e que via de regra seguem extensões proprietárias de cada ferramenta) fica separada do modelo principal. O modelo MD-JPA aproveita-se desta característica, podendo ser utilizado em ferramentas que utilizam o Eclipse ou que importam modelos no formato XMI. A Figura 6.2 apresenta uma captura da tela do Eclipse durante a edição do modelo de avaliação no formato padrão, sem editor gráfico de diagramas.

A extensão de menu para avaliação de modelos disponibiliza uma opção para verificar modelos UML dentro da ferramenta Eclipse. Ela é composta por um pacote de classes Java que implementa o menu e um arquivo OCL contendo as regras de avaliação (apresentadas no capítulo 4). A execução das regras OCL no modelo é realizada por um

plugin disponibilizado pelo Eclipse, cabendo à implementação do menu apenas utilizar os serviços do Eclipse e organizar o resultado de forma legível ao usuário. O arquivo de regras OCL pode ser complementado pelo usuário com novas regras de avaliação. A Figura 6.3 apresenta o resultado da avaliação sobre o modelo de testes, contendo a lista de elementos que violam cada regra avaliada.

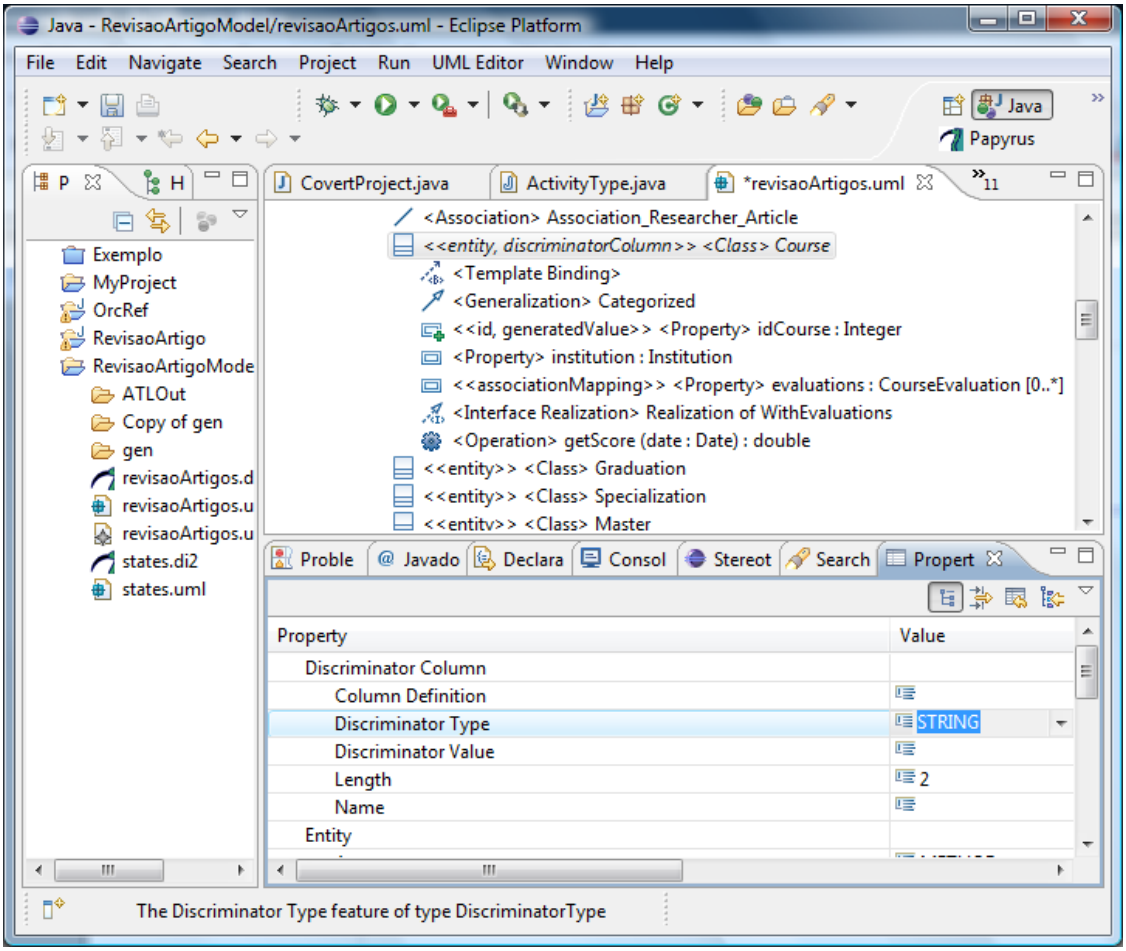


Figura 6.2: Modelo de curso no editor não gráfico do Eclipse.

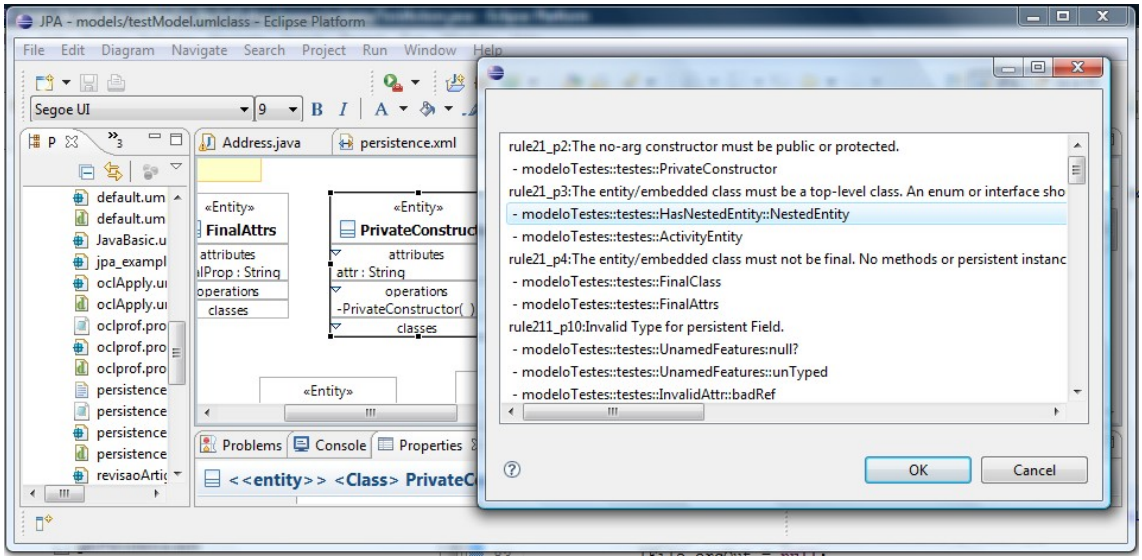


Figura 6.3: Avaliação do modelo de testes.

A extensão de menu para geração de código disponibiliza uma opção de menu e um módulo que realiza a tradução do modelo JAS para objetos do *framework* JDT. Primeiramente o arquivo de regras ATL (apresentadas no capítulo 5) é compilado pelo *plugin* fornecido pelo Eclipse e executado no modelo selecionado no menu, produzindo um modelo no formato JAS. Este modelo é então percorrido, instanciando os objetos JDT correspondentes a cada elemento JAS. Quando todos os elementos de cada unidade de programa estiverem criados, o JDT é utilizado para gerar os arquivos de código fonte. A Figura 6.4 é uma captura de tela da chamada do menu de geração de código sobre um modelo UML. A transformação do modelo UML em JAS e posteriormente em código é realizada, sob o ponto de vista do usuário, como um único passo na chamada do menu.

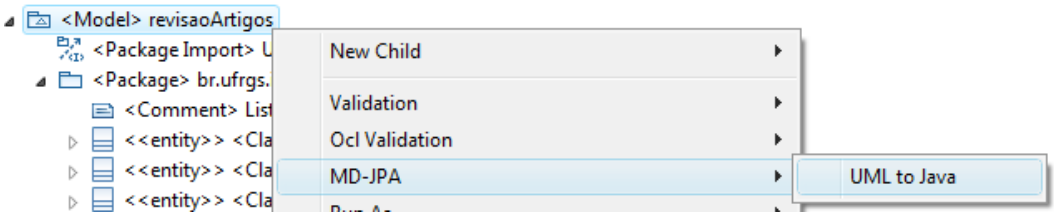


Figura 6.4: Transformando o modelo em Java.

O editor de estereótipos é um componente auxiliar desenvolvido para edição de propriedades estruturadas, ou seja, propriedades que referenciam objetos compostos por uma ou mais propriedades. Na ferramenta Eclipse este tipo de edição só pode ser realizado no editor de modelo, fora do editor gráfico de diagramas. O editor de estereótipos possibilita alterar as extensões de um elemento no diagrama sem precisar mudar de aba de edição. As figuras 6.5 e 6.6 apresentam telas capturadas de editores de diagramas UML, com a utilização do editor de estereótipos.

O *plugin* MD-JPA é disponibilizado como uma parte dependente da plataforma Eclipse. Entretanto, a maior parte de seus componentes e componentes dependentes podem ter sua implementação migrada para outras plataformas. Os módulos de OCL, ATL e UML (e suas dependências) utilizados no *plugin* podem ser utilizados fora do Eclipse, tanto em uma ferramenta independente, como em *plugins* para outras ferramentas ou plataformas de desenvolvimento.

A escolha de componentes de código livre, independentes da plataforma escolhida, facilita a evolução da ferramenta, ou sua migração para diferentes plataformas. Com essas características, espera-se disponibilizar um conjunto de ferramentas que facilite o trabalho da modelagem de sistemas com persistência, sua avaliação e utilização como artefato de desenvolvimento.

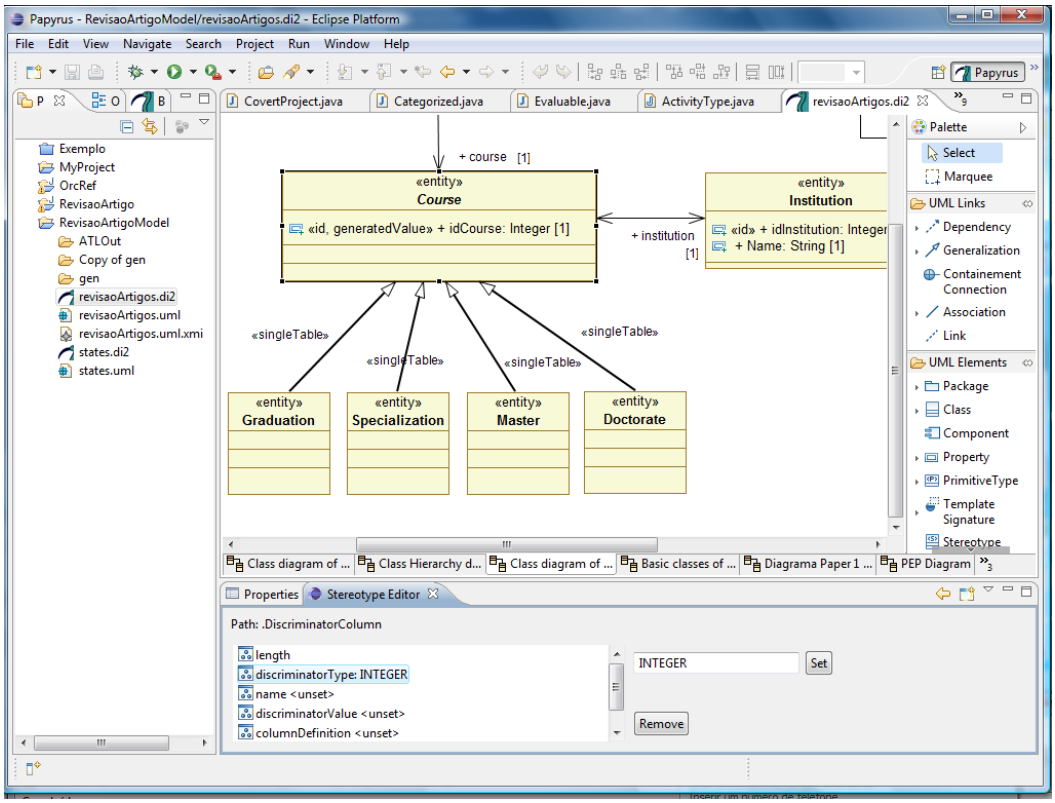


Figura 6.5: Editor de estereótipos em conjunto com a ferramenta Papyrus.

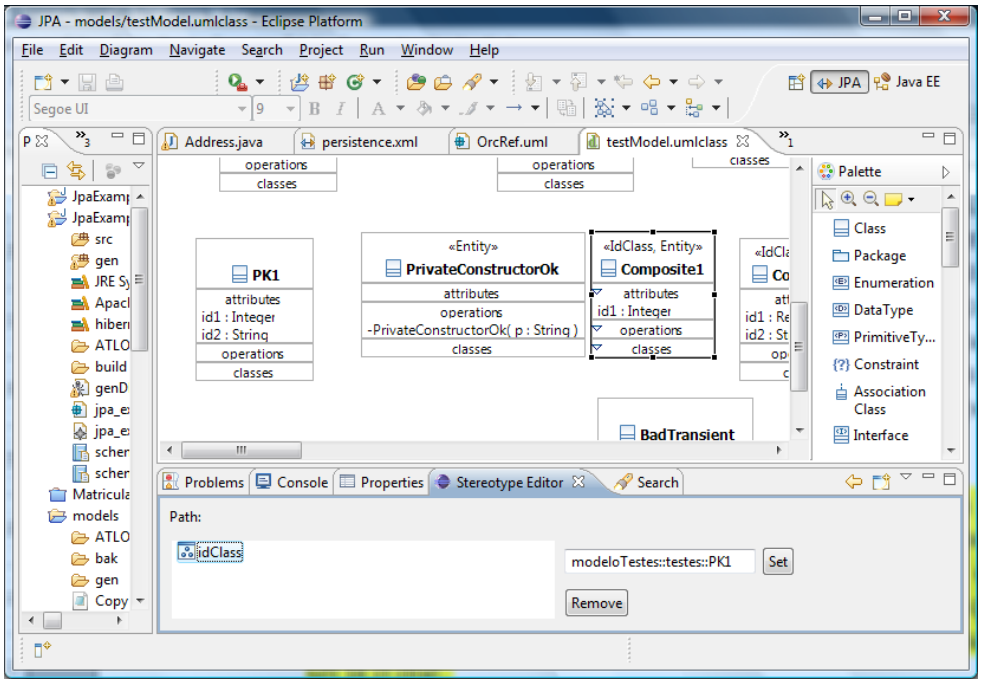


Figura 6.6: Editor de estereótipos em conjunto com o editor de diagramas do Eclipse.

7 CONCLUSÃO

Apesar do sucesso da UML ainda não há uma solução clara, consensual e de fácil adoção para o problema da modelagem de persistência dentro do paradigma orientado a objetos. A utilização de modelos ER e UML, em conjunto, ainda constitui a abordagem mais comum para representação de sistemas, mas não oferece a integração necessária para representar sistemas em uma abordagem DDM. Para tal, os modelos precisam compartilhar conceitos comuns que permitam sua avaliação e utilização como artefatos no desenvolvimento do sistema.

Este trabalho especificou um perfil UML para a modelagem dos elementos persistentes mapeados através do padrão JPA para uma base de dados relacional. O objetivo do perfil MD-JPA é a criação de modelos que descrevem os elementos persistentes e orientados a objeto de maneira integrada. Estes modelos são então verificados com o objetivo de detectar construções incompatíveis com o MOR e são transformados em partes da implementação do software.

O MD-JPA permite, mas não impõe, a utilização dos recursos mais avançados do MOR, também opcionais no padrão JPA. Além disso, preserva a habilidade da UML de criar vários diagramas, com diferentes níveis de abstração ou temas abordados, omitindo os detalhes desnecessários nos diagramas mais abstratos, sem perder as informações representadas em outros diagramas do sistema.

Para avaliar o perfil como ferramenta útil na abordagem DDM foram construídas regras que transformam os modelos em implementação JPA. Estas regras foram testadas com o modelo apresentado no exemplo motivador e com um modelo construído a partir do sistema de exemplo disponível na especificação JPA.

A comparação dos códigos gerados a partir da transformação do modelo proposto com os códigos originais não mostrou diferenças semânticas significativas. O sistema de exemplo representa de maneira compreensiva os conceitos do JPA, mostrando portanto a capacidade das regras de gerar partes da implementação a partir do modelo e a capacidade do perfil de representar os mapeamentos do JPA.

Por fim, foi criada uma ferramenta desenvolvida com o objetivo de disponibilizar o perfil apresentado e seus recursos para a comunidade. A ferramenta MD-JPA funciona como *plugin* para ferramentas de modelagem e desenvolvimento de sistemas e utiliza módulos construídos com código aberto para verificar e transformar modelos.

O *plugin* MD-JPA permite a utilização do perfil proposto de forma integrada a editores UML dentro da plataforma Eclipse. Além disso, disponibiliza as funções de avaliação de modelos e geração da implementação JPA a partir do modelo construído pelo usuário.

Um artigo científico escrito sobre o tema foi aceito na 21ª Conferência em Engenharia de Software e Engenharia do Conhecimento[14]. O título do trabalho é "*MD-JPA Profile: A Model Driven Language for Java Persistence"*. Esta conferência internacional foi classificada pelo *Qualis*[15] com o conceito B2, no ano base de 2007. Um poster denominado "*Um perfil UML para modelagem com persistência Java*" foi anteriormente apresentado no 23ª Simpósio Brasileiro de Bancos de Dados, com os primeiros resultados deste trabalho.

Alguns estudos futuros podem ser relacionados a este trabalho: um estudo sobre as características comuns entre todas as ferramentas MOR; a extensão do perfil para atender construções específicas de outras ferramentas ou padrões MOR; estudo para transformações inversas de código para modelo (engenharia reversa); estudo sobre métricas e avaliação de modelos persistentes em um contexto MDD; geração de testes automatizados a partir de modelos persistentes; e o estudo sobre a representação dos aspectos dinâmicos (tais como implementação das operações) do MOR em modelos dinâmicos da UML. Todos estes trabalhos podem ser integrados ao perfil MD-JPA atual de maneira complementar, facilitando a adoção da abordagem DDM.

14 SEKE 2009 - Software Engineering and Knowledge Engineering. O sítio da conferência é http://www.ksi.edu/seke/seke09.html

15 O Qualis é um sistema de avaliação de periódicos mantido pela CAPES, disponível em http://qualis.capes.gov.br/webqualis/

REFERÊNCIAS

AMBLER, S. **Agile Database Techniques: Effective Strategies for the Agile Software Developer**. 1. ed. USA: Wiley, 2003.

AMBLER, S. **A UML Profile for Data Modeling**. Disponível em: <http://www.agiledata.org/essays/umlDataModelingProfile.html>. Acesso em: jul. 2009.

APACHE FOUNDATION. **Apache OpenJPA**. Disponível em: <http://openjpa.apache.org/>. Acesso em: ago. 2009.

BEYDEDA, S.; BOOK, M.; GRUHN, V. **Model-Driven Software Development**. 1. ed. New York: Springer, 2005.

BRUCK, J. **Eclipse Corner Article: Defining Generics with UML Templates**. Disponível em: <http://www.eclipse.org/articles/article.php?file=Article-Defining-Generics-with-UML-Templates/index.html>. Acesso em: jul. 2009.

CALIARI, G. L.; SILVA, P. S. M. A case study on modeling persistence with MDA tools. In: 12TH CONFERENCE ON SOFTWARE ENGINEERING AND DATABASES, 12., 2007, JISBD 2007, Zaragoza, Spain. **Anais eletrônicos...** 2007, p.51-59.

CHEN, P. P. The entity-relationship model - toward a unified view of data. **ACM Trans. Database Syst.**, v. 1, n. 1, p. 9-36. New York, USA: ACM, 1976.

DEMICHIEL, L.; KEITH, M. **JSR 220: Enterprise JavaBeansTM. Version 3.0. Java Persistence API**. Disponível em: <http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html>. Acesso em: jul. 2009.

ECLIPSE FOUNDATION. **Eclipse.org home**. Disponível em: <http://www.eclipse.org/>. Acesso em: ago. 2009.

ECLIPSE FOUNDATION. **Eclipse Java development tools (JDT)**. Disponível em: <http://www.eclipse.org/jdt/>. Acesso em: ago. 2009.

FISCHER, G.; LUSIARDI, J.; VON GUDENBERG, J. W. Abstract Syntax Trees - and their Role in Model Driven Software Development. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING ADVANCES, 2., 2007, ICSEA 2007, Cap Esterel, France. **Anais...** . Los Alamitos, USA: IEEE Computer Society, 2007, p. 38.

GOSLING, J.; JOY, B.; STEELE, G.; BRACHA, G. **Java(TM) Language Specification, The (3rd Edition)**. 3. ed. USA: Addison Wesley, 2005.

GRANT, E. S.; CHENNAMANENI, R.; REZA, H. **Towards analyzing UML class diagram models to object-relational database systems transformations**. In: 24TH IASTED INTERNATIONAL CONFERENCE ON DATABASE AND APPLICATIONS, 24., 2006, DBA'06, Innsbruck, Austria. **Anais...** . Anaheim, USA: ACTA Press, 2006, p.129-134.

GREEN, T. R. G.; PETRE, M. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. **Journal of visual Languages and Computing**, n.7, p.131-174. [S.l.:s.n], 1996.

INRIA ATLANMOD. **MoDisco Tool - Java Abstract Syntax Discovery Tool**. Disponível em: <http://www.eclipse.org/gmt/modisco/technologies/JavaAbstractSyntax/>. Acesso em: ago. 2009.

JOUAULT, F.; ALLILAIRE, F.; BÉZIVIN, J.; KURTEV, I.; VALDURIEZ, P. ATL: a QVT-like transformation language. In: 21ST ACM SIGPLAN SYMPOSIUM ON OBJECT-ORIENTED PROGRAMMING SYSTEMS, LANGUAGES, AND APPLICATIONS, 21., 2006, OOPSLA 2006, Portland, USA. **Anais...** New York, USA: ACM, 2006, p.719-720.

LARMAN, C. **Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development**. 3. ed. Upper Saddle River, NJ: Prentice Hall PTR, 2004.

LUJÁN-MORA, S.; TRUJILLO, J.; SONG, I. A UML profile for multidimensional modeling in data warehouses. **Data Knowl. Eng.**, v. 59, n. 3, p. 725-769. ACM, 2006.

MELLOR, S. J.; SCOTT, K.; UHL, A.; WEISE, D. **MDA Distilled: Principles of Model-Driven Architecture**. 1. ed. Reading, Massachusetts: Addison-Wesley Professional, 2004.

OLIVEIRA, J. P. M. D.; GALANTE, R. D. M.; MUSA, D. L.; EDELWEISS, N. Uma Proposta para Editoração, Indexação e Busca de Documentos Científicos em um Processo de Avaliação Aberta. In: WORKSHOP EM BIBLIOTECAS DIGITAIS, 1., 2005, WDL2005, Uberlândia, Brasil. **Anais...** . Porto Alegre : Editora da Sociedade Brasileira de Computação, 2005, p. 30-39.

OMG. **OMG's Model Driven Architecture**. Disponível em: <http://www.omg.org/cgi-bin/doc?omg/03-06-01>. Acesso em: ago. 2009.

OMG. **Diagram Interchange Specification, v1.0**. Disponível em: <http://www.omg.org/cgi-bin/doc?formal/06-04-04>. Acesso em: ago. 2009.

OMG. **Meta Object Facility (MOF) Core Specification version 2.0**. Disponível em: <http://www.omg.org/spec/MOF/2.0/PDF/>. Acesso em: ago. 2009.

OMG. **Object Constraint Language, v. 2.0**. Disponível em: <http://www.omg.org/spec/OCL/2.0/PDF/>. Acesso em: ago. 2009.

OMG. **Meta Object Facility (MOF) 2.0 Query/View/Transformation, v1.0**. Disponível em: <http://www.omg.org/spec/QVT/1.0/>. Acesso em: ago. 2009.

OMG. **UML 2.2 Superstructure**. Disponível em: <http://www.omg.org/spec/UML/2.2/Superstructure/PDF/>. Acesso em: ago. 2009.

PALANQUE, P.; BASTIDE, R. UML for Interactive Systems: What is Missing. In: INTERACT 2003 WORKSHOP, 2., 2003, Zürich, Switzerland. **Anais...** Louvain-la-Neuve, Belgium: International Federation for Information Processing (IFIP), 2003, p. 96-99.

PIRES, W.; BRUNET, J.; RAMALHO, F. UML-based design test generation. In: THE 2008 ACM SYMPOSIUM ON APPLIED COMPUTING, 23., 2008, SAC '08, Fortaleza, Brazil . **Anais...** New York, USA: ACM, 2008, p. 735-740.

PRESSMAN, R. **Software Engineering: A Practitioner's Approach**. 5. ed. New York, NY: McGraw-Hill Science/Engineering/Math, 2001.

RED HAT MIDDLEWARE. hibernate.org - Hibernate. Disponível em: <https://www.hibernate.org/>. Acesso em: ago. 2009.

RUMBAUGH, J.; JACOBSON, I.; BOOCH, G. **The Unified Modeling Language Reference Manual**. 1. ed. Reading, Massachusetts: Addison-Wesley Professional, 1999.

SONG, E.; YIN, S.; RAY, I. Using UML to model relational database operations. **Comput. Stand. Interfaces**, v. 29, n. 3, p. 343-354. ACM, 2007.

SUN MICROSYSTEMS. **JavaBeans Spec**. Disponível em: <http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html>. Acesso em: jul. 2009.

WITTHAWASKUL, W.; JOHNSON, R. Specifying Persistence in Platform Independent Models. In: WORKSHOP IN SOFTWARE MODEL ENGINEERING AT THE SIXTH INTERNATIONAL CONFERENCE ON THE UNIFIED MODELING LANGUAGE, 6., 2003, UML 2003, San Francisco, USA. **Anais eletrônicos...** 2003.

APÊNDICE A: RESTRIÇÕES OCL

Este apêndice contém o código fonte, desenvolvido na linguagem OCL, para a avaliação de modelos MD-JPA. Este código fonte é distribuído dentro do *plugin* MD-JPA, no arquivo *persistence.profile.ocl,* podendo ser estendido pelo usuário para realizar verificações sobre o modelo.

```
package persistence

context uml::Element
-- *** constants of this package ************************************************
-- java primitive types + UML primitive types
def: primitiveTypes :Sequence(String) =
Sequence{'byte','char','String','boolean','double','float','short','int','long','Integer
','Boolean'}

-- Classes that can be used with Primary keys without (except Date classes that need
extra check)
def: javaPkClasses :Sequence(String) =
Sequence{'JavaBasic::java::lang::String','JavaBasic::java::lang::Double','JavaBasic::jav
a::lang::Integer',
   'JavaBasic::java::lang::Long','JavaBasic::java::lang::Short','JavaBasic::java::lang::
Character',
   'JavaBasic::java::lang::Byte','JavaBasic::java::lang::Boolean'}

def: javaSerializableStdClasses :Sequence(String) =
-- Std serializable classes of Java
javaPkClasses->union(
Sequence{'JavaBasic::java::math::BigInteger','JavaBasic::java::math::BigDecimal','JavaBa
sic::java::util::Date',
   'JavaBasic::java::util::Calendar','JavaBasic::java::sql::Date','JavaBasic::java::sql:
:Time',
   'JavaBasic::java::sql::Timestamp'})

-- the serializable interface
def: serializableInterface :String = 'JavaBasic::java::io::Serializable'

context Operation
def: isConstructor() : Boolean =
   not(self.extension_Create.oclIsUndefined())

context uml::Element
def: hasStereotype(pname:String) : Boolean =
   not (self.getAppliedStereotype('persistence::'.concat(pname)).oclIsUndefined())

context Property
def: isTransient():Boolean= self.hasStereotype('Transient')

def: verifyPKType() : Boolean =
   if (self.type.oclIsUndefined()) then
      false
   else
   let fullName:String=self.type.qualifiedName in
      ( self.type.oclIsKindOf(uml::PrimitiveType) and
         -- java primitive types + UML primitive types
         primitiveTypes->includes(self.type.name) ) or
      ( self.type.oclIsKindOf(uml::Classifier) and
```

```
         javaPkClasses->includes(fullName)) or
      ( self.type.oclIsKindOf(uml::Classifier) and
         Sequence{'JavaBasic::java::util::Date','JavaBasic::java::sql::Date'}-
>includes(fullName) and
         self.hasStereotype('Date')
      )
   endif


context Persistent
inv rule21_p2:
   base_Class.ownedOperation->forAll(
      (isConstructor() and ownedParameter->isEmpty()) implies
         visibility=uml::VisibilityKind::public
         or
         visibility=uml::VisibilityKind::protected
   )

--The entity class must be a top-level class. An enum or interface should not be
designated as an entity.
--(not nested, not protected, not private, not static)
--(must be an UML Class element. Cannot be Activity, Component nor Stereotype )
--(static concept is absent for classes on UML)
inv rule21_p3 :
   (base_Class.owner.oclIsKindOf(uml::Package) or
base_Class.owner.oclIsKindOf(uml::Model))
   and   not(base_Class.visibility=uml::VisibilityKind::protected or
base_Class.visibility=uml::VisibilityKind::private)
    and base_Class.oclIsTypeOf(Class)
inv rule21_p4 :
   not(base_Class.isLeaf) and
   base_Class.attribute->forAll(not(isLeaf and extension_Transient.oclIsUndefined()))

context Property
--Invalid Type for persistent Field
inv rule211_p10:
   not(class.extension_Persistent.oclIsUndefined()  or isTransient()) implies
      let fullName:String=type.qualifiedName in
         type.hasStereotype('Entity') or
         type.hasStereotype('Embeddable') or
         type.oclIsKindOf(uml::Enumeration) or
         ( type.oclIsKindOf(uml::PrimitiveType) and
            primitiveTypes->includes(type.name) ) or
         ( type.oclIsKindOf(uml::Classifier) and
            javaSerializableStdClasses->includes(fullName) ) or
         type.oclAsType(uml::BehavioredClassifier).interfaceRealization
            ->exists(contract.qualifiedName =serializableInterface)

context Entity
--Every entity must have a primary key.
inv rule214_p1:
   base_Class.getAllAttributes()->exists((class.hasStereotype('Entity') or
class.hasStereotype('MappedSuperclass'))
      and (hasStereotype('Id') or hasStereotype('EmbeddedId') ))

--The primary key must be defined on the entity that is the root of the entity hierarchy
or on a mapped
--superclass of the entity hierarchy. The primary key must be defined exactly once in an
entity hierarchy.
inv rule214_p2:
   base_Class.attribute->exists(not(extension_Id.oclIsUndefined() and
extension_EmbeddedId.oclIsUndefined() )) implies
      base_Class.superClass->forAll(
         not(hasStereotype('Entity'))
         and
         (  not(hasStereotype('MappedSuperclass')) or
            getAllAttributes()->forAll( extension_Id.oclIsUndefined() and
extension_EmbeddedId.oclIsUndefined())
         )
      )
-- if IdClass, check if all ID match PK_CLASS props
--p5: The primary key (or field or property of a composite primary key) should be one of
the following types:
--any Java primitive type; any primitive wrapper type; java.lang.String; java.util.Date;
--java.sql.Date. In general, however, approximate numeric types (e.g., floating point
```

```
types) should
--never be used in primary keys. Entities whose primary keys use types other than these
will not be portable.
--If generated primary keys are used, only integral types will be portable. If
java.util.Date is
--used as a primary key field or property, the temporal type should be specified as
DATE.
--
inv rule214_p5:
   let embeddedId : Sequence(uml::Property)=base_Class.attribute-
>select(hasStereotype('EmbeddedId')) in
   if (embeddedId->isEmpty()) then
      base_Class.attribute->forAll(not(hasStereotype('Id')) or verifyPKType())
   else
      let ebtyp: uml::Type=embeddedId->first().type in
         ebtyp.oclAsType(uml::Class).attribute->forAll(verifyPKType())
   endif

--A composite primary key must either be represented and mapped as an embeddable class
(see
--Section 9.1.14, "EmbeddedId Annotation") or must be represented and mapped to
*multiple*
--fields or properties of the entity class (see Section 9.1.15, "IdClass Annotation").
inv rule214_p7E:
   (base_Class.attribute->forAll(extension_Id.oclIsUndefined())
    and base_Class.attribute->select(hasStereotype('EmbeddedId'))->size()=1
    ) or
    (base_Class.attribute->select(hasStereotype('Id'))->size()>1  --IdClass MUST have
more than one prop
    and base_Class.hasStereotype('IdClass')
    ) or
    (base_Class.attribute->select(hasStereotype('Id'))->size()=1
    and base_Class.attribute->forAll(extension_EmbeddedId.oclIsUndefined())
    and base_Class.extension_IdClass.oclIsUndefined()
    ) or  --NOPK
    (base_Class.attribute->forAll(extension_Id.oclIsUndefined())
    and base_Class.attribute->forAll(extension_EmbeddedId.oclIsUndefined())
    and base_Class.extension_IdClass.oclIsUndefined()
   )


-- p7(F): If the composite primary key class is mapped to multiple fields or properties
of the entity class,
-- the names of primary key fields or properties in the primary key class and those of
the entity
-- class must correspond and their types must be the same
inv rule214_p7F:
   base_Class.hasStereotype('IdClass') implies
      let idprops :
Sequence(uml::Property)=base_Class.extension_IdClass.idClass.attribute in
      let clprops : Sequence(uml::Property)=base_Class.attribute-
>select(hasStereotype('Id')) in
         clprops->forAll(p|idprops->exists(name=p.name and type=p.type)) and
         idprops->forAll(p|clprops->exists(name=p.name and type=p.type))

-- a primary key can be an ordinary class
context uml::Class
def: isPkClass() : Boolean =
   persistence::IdClass.allInstances()->exists(idClass=self)
   or
   persistence::EmbeddedId.allInstances()->exists(base_Property.type=self)
--The primary key class must be public
inv rule214_7A1 :
   isPkClass() implies self.visibility=uml::VisibilityKind::public

--The primary key class must have a public no-arg constructor.
inv rule214_7A2 :
   isPkClass() implies
   self.ownedOperation->forAll(
      (isConstructor() and ownedParameter->isEmpty()) implies
         visibility=uml::VisibilityKind::public
      )
--The properties of the primary key class must be public or protected.
inv rule214_7B :
   isPkClass() implies
```

```
   self.attribute->forAll(visibility=uml::VisibilityKind::public or
visibility=uml::VisibilityKind::protected)

-- Simple rules
context uml::NamedElement
def: hasName() : Boolean =
  (self.name<>'' and not(self.name.oclIsUndefined()))
context Persistent
inv ruleProfile_2 :
   base_Class.name<>'' and base_Class.name<>oclIsUndefined()
inv ruleProfile_3 :
   base_Class.attribute->forAll(hasName())
inv ruleProfile_4:
   base_Class.ownedOperation->forAll(hasName())
inv ruleProfile_8:
   base_Class.attribute->forAll(not(type.oclIsUndefined()))
inv ruleProfile_10:
   base_Class.superClass->size()<=1

-- Profile/UML rules
context uml::Property

def: manyToMany() : Boolean =
   isMultivalued() and getOtherEnd().isMultivalued()
def: oneToMany() : Boolean =
   not(isMultivalued()) and getOtherEnd().isMultivalued()
def: manyToOne() : Boolean =
   isMultivalued() and not(getOtherEnd().isMultivalued())
def: oneToOne() : Boolean =
   not(isMultivalued()) and not(getOtherEnd().isMultivalued())

--context Property
def: isAssociationProp() : Boolean =
   not(association.oclIsUndefined() or
      type.oclAsType(uml::Class).extension_Persistent.oclIsUndefined())

context AssociationMapping

def: isMapCollection() : Boolean =
   collectionType='Map' or collectionType='map' or collectionType='java.util.Map'
def: isListCollection() : Boolean =
   collectionType='List' or collectionType='list' or collectionType='java.util.List'
def: isSimpleCollection() : Boolean =
   collectionType.oclIsUndefined() or collectionType='Collection' or
collectionType='collection' or collectionType='java.util.Collection'
def: isSetCollection() : Boolean =
   collectionType='Set' or collectionType='set' or collectionType='java.util.Set'

-- AssociationMapping only on entity association ends
inv ruleProfile_1:
   base_Property.isAssociationProp()

-- A JoinTable annotation is specified
-- on the owning side of a many-to-many association, or in a unidirectional one-to-many
association
inv rule91_25_p1:
   not(self.joinTable.oclIsUndefined()) implies
      (
         base_Property.oneToMany() and
         not(base_Property.getOtherEnd().isNavigable())
      ) or base_Property.manyToMany()
inv ruleProfile_13:
--JoinTable only one side
not(joinTable.oclIsUndefined()) implies
   base_Property.getOtherEnd().extension_AssociationMapping.joinTable.oclIsUndefined()

inv ruleProfile_11:
--MapKey: on MD-JPA, only used when it's needed to specify what property is the key of
the map
--otherwise, using a Map collection type alone is equivalent to use the MapKey with no
parameters
not (mapKey.oclIsUndefined()) implies
   base_Property.type.oclAsType(uml::Class).attribute->exists(p|p=mapKey)
   and
   self.isMapCollection()
```

```
inv rule91_32_p2:
--... (PrimaryKeyJoinColumn) it may be used in a OneToOne mapping in which the primary
key of
--the referencing entity is used as a foreign key to the referenced entity.
--(PkJoinCols only to oneToOne)
not(pkJoinCols->isEmpty()) implies
   base_Property.oneToOne()


context DiscriminatorColumn
-- The discriminator value must be consistent in type with the discriminator type of the
specified or
-- defaulted discriminator column
-- DiscriminatorValue x DiscriminatorType (on parent)
-- Default type:String / String=>all goes / Char=>len=1 / Int=>toInteger!=Invalid
inv rule91_31_p4A:
   not(self.discriminatorValue.oclIsUndefined()) implies
      (discriminatorType=DiscriminatorType::INTEGER
      and not(discriminatorValue.toInteger().oclIsUndefined()))
      or
      (discriminatorType=DiscriminatorType::CHAR
      and discriminatorValue.size()=1)
      or
      discriminatorType=DiscriminatorType::STRING


context DiscriminableGeneralization
-- The discriminator value must be consistent in type with the discriminator type of the
specified or
-- defaulted discriminator column
-- DiscriminatorValue x DiscriminatorType
-- Default type:String / String=>all goes / Char=>len=1 / Int=>toInteger!=Invalid
-- DiscriminatorValue x DiscriminatorType (on childs)
inv rule91_31_p4B:
   let
dc:persistence::DiscriminatorColumn=base_Generalization.general.oclAsType(uml::Class).ex
tension_DiscriminatorColumn in (
      not(dc.discriminatorType.oclIsUndefined() or discriminatorValue.oclIsUndefined())
implies
       (
         dc.discriminatorType=DiscriminatorType::STRING or
         (dc.discriminatorType=DiscriminatorType::INTEGER
            and not(discriminatorValue.toInteger().oclIsInvalid()))
         or
         (dc.discriminatorType=DiscriminatorType::CHAR
         and discriminatorValue.size()=1
         )
      )
   )

context Entity
inv ruleProfile_5:
-- Only one strategy for each parent
uml::Generalization.allInstances()->select(general=base_Class )->collect(g|
  if (g.hasStereotype('Joined')) then
   'J'
  else
   if (g.hasStereotype('TablePerClass')) then
      'T'
   else
      'S'
   endif
  endif
)->asSet()->size()<2

context Generalization
inv ruleProfile_6:
-- Only one strategy for each Generalization
(self.hasStereotype('Joined') xor self.hasStereotype('TablePerClass') xor
self.hasStereotype('SingleTable'))
or
not (self.hasStereotype('Joined') or self.hasStereotype('TablePerClass') or
self.hasStereotype('SingleTable'))
```

```
def: recurseMapped(cl:uml::Class) : Boolean =
   (general=cl and cl.hasStereotype('MappedSuperclass'))
   or
   general.generalization->select(general.hasStereotype('MappedSuperclass'))-
>exists(recurseMapped(cl))

def: isAncestral(cl:uml::Class) : Boolean =
   (general=cl)
   or
   general.generalization->exists(recurseMapped(cl))

context AssociationOverrides
inv ruleProfile_7:
--Only for mappedsuperclass
base_Generalization.general.hasStereotype('MappedSuperclass')

--The AssociationOverride annotation may be applied to an entity that extends a mapped
superclass
--to override a many-to-one or one-to-one mapping defined by the mapped superclass
inv rule91_12_p2:
overrides->collect(property)->forAll(p|
      p.isAssociationProp() and
      base_Generalization.recurseMapped(p.class) and
      (
         p.manyToOne() or
         p.oneToOne()
      )
   )

context AttributeOverrides
def: baseClass() : uml::Class =
if (self.base_Class.oclIsUndefined()) then
   self.base_Property.class
else
   self.base_Class
endif

inv ruleProfile_9:
--at least one property must be overriden
not(value->collect(property)->isEmpty())


--The AttributeOverride annotation may be applied to an entity that extends a mapped
superclass
--or to an embedded field or property to override a basic mapping defined by the mapped
superclass or
--embeddable class.
inv rule91_10_p2:
--Only for mappedsuperclass/Embed props
if (not(base_Class.oclIsUndefined())) then
   value->collect(property)->forAll(p|
      (  not(p.isAssociationProp()) and
         base_Class.generalization->exists(g|g.recurseMapped(p.class))
      ) or (
         base_Class.generalization->exists(g|g.isAncestral(p.class)) and
         base_Class.hasStereotype('Embeddable')
      )
   )
else
   value->collect(property)->forAll(p|
      (  base_Property.class.generalization->exists(g|g.isAncestral(p.class)) and
         base_Property.class.hasStereotype('Embeddable')
      ) or (
         base_Property.type.hasStereotype('Embeddable') and
         base_Property.type = p.class
      )
   )
endif


context uml::Property
--prevent two properties with same name
inv ruleProfile_14:
   not (self.class.oclIsUndefined()) implies self.class.attribute->forAll(p|p.name =
self.name implies p=self)
```

```
context Column
inv ruleProfile_15:
--Each column name must not conflict with other columns
(base_Property.class.attribute
  ->forAll(p|
      (p.extension_Column.oclIsUndefined() implies
         not (p.name=self.value.name and self.value.table.oclIsUndefined())

      ) and (
      (p.extension_Column.value.name = self.value.name and p<>base_Property) implies
         p.extension_Column.value.table<>self.value.table
      )
   )

)
-- column "table" must exist in the entity mapping
inv ruleProfile_16:
let cl:uml::Class = base_Property.class in
   not (cl.oclIsUndefined()) implies
   cl.extension_Persistent.oclIsTypeOf(Entity) implies
   let entity:Entity = cl.extension_Persistent in
     not (self.value.table.oclIsUndefined()) implies (
         (self.value.table=cl.name and entity.table.oclIsUndefined())
      or
      self.value.table=entity.table
      or
      entity.secondaryTables->exists(st|st.name=self.value.table)
   )

context Transient
inv rule211_p2C:
--Mapping annotations cannot be applied to fields or properties that are transient or
Transient.
--(Transient property does not accept stereotypes)
   base_Property.extension_Column.oclIsUndefined() and
   base_Property.extension_AssociationMapping.oclIsUndefined() and
   base_Property.extension_Id.oclIsUndefined() and
   base_Property.extension_Lob.oclIsUndefined() and
   base_Property.extension_Temporal.oclIsUndefined() and
   base_Property.extension_Embedded.oclIsUndefined() and
   base_Property.extension_EmbeddedId.oclIsUndefined() and
   base_Property.extension_Basic.oclIsUndefined() and
   base_Property.extension_OrderBy.oclIsUndefined() and
   base_Property.extension_Version.oclIsUndefined() and
   not (base_Property.hasStereotype('AttributeOverrides'))

context Class
inv ruleProfile_18:
-- not persistent class cannot have annotations
extension_Persistent.oclIsUndefined() implies
attribute->forAll(p|
   p.extension_Column.oclIsUndefined() and
   p.extension_AssociationMapping.oclIsUndefined() and
   p.extension_Id.oclIsUndefined() and
   p.extension_Lob.oclIsUndefined() and
   p.extension_Temporal.oclIsUndefined() and
   p.extension_Embedded.oclIsUndefined() and
   p.extension_EmbeddedId.oclIsUndefined() and
   p.extension_Basic.oclIsUndefined() and
   p.extension_OrderBy.oclIsUndefined() and
   p.extension_Version.oclIsUndefined() and
   not (p.hasStereotype('AttributeOverrides')) and
   p.extension_Transient.oclIsUndefined()
) and
self.extension_DiscriminatorColumn.oclIsUndefined() and
self.extension_IdClass.oclIsUndefined() and
not (self.hasStereotype('AttributeOverrides')) and
self.extension_Generator.oclIsUndefined() and
generalization->forAll(g|
   g.extension_TablePerClass.oclIsUndefined() and
   g.extension_DiscriminableGeneralization.oclIsUndefined() and
   g.extension_AssociationOverrides.oclIsUndefined()
)
```

```
endpackage
```

APÊNDICE B: TRANSFORMAÇÕES ATL

Este apêndice contém as regras de transformação, desenvolvidas na linguagem ATL, que partem de um modelo UML com o perfil MD-JPA, gerando um modelo segundo a árvore sintática da linguagem Java.

```
module genPersistence;
create AST : JAST from model : UML ;

helper def: profileName : String = 'persistence';

rule genClass {
from
   cl : UML!Class  (not(cl.isEGenericType()))
to
   typ: JAST!TypeDeclaration (
      superclassType<-
      if (cl.superClass->isEmpty()) then OclUndefined      else cl.superClass-
>first().genAnyType(cl) endif,
      superInterfaceTypes<-  cl.interfaceRealization->collect(i|
i.contract.genAnyType(cl)),
      bodyDeclarations<- typ.bodyDeclarations
        ->append(cl.getFeatures()->select(f|f.oclIsKindOf(UML!Property))->collect(f|
thisModule.resolveTemp(f,'typ')))
        ->union(cl.getFeatures()->select(f|f.oclIsKindOf(UML!Property))->collect(f|
thisModule.resolveTemp(f,'getter')))
        ->union(cl.getFeatures()->select(f|f.oclIsKindOf(UML!Property))->collect(f|
thisModule.resolveTemp(f,'setter')))
        ->union(cl.getFeatures()->select(f|f.oclIsKindOf(UML!Operation))->collect(f|
thisModule.resolveTemp(f,'method'))),
      name<-thisModule.genSimpleName(cl)
   ),
   cun: JAST!CompilationUnit (
      package <- thisModule.genPackageDcl(cl)
   )
do {
   thisModule.doGenRules(cl,typ,cun);
}
}
rule doGenRules(cl:UML!Classifier,typ: JAST!TypeDeclaration,cun:JAST!CompilationUnit) {
do {
   for (s in cl.getAppliedStereotypes()) {
      (s.name+' st ').println();
   }
   (cl.name+'=!o= ').println();

   cun.types<-cun.types->append(typ);
   thisModule.addClassAnnotations(cl,typ,cun);
   if (cl.visibility=#public)
      thisModule.genBodyModifiers(typ,'public');
   else if (cl.visibility=#private)
      thisModule.genBodyModifiers(typ,'private')    ;
   else if (cl.visibility=#protected)
      thisModule.genBodyModifiers(typ,'protected');
   if (cl.isAbstract())
      thisModule.genBodyModifiers(typ,'abstract');

   thisModule.addPersistenceImports(cl,cun);

   }
```

```
}
rule addPersistenceImports(cl:UML!Classifier,cun:JAST!CompilationUnit) {
   do {
      if (cl.isEntity())
         thisModule.addImport(cun,'javax.persistence.Entity');
      if (not(cl.getFeatures()->select(f|f.isID())->isEmpty()) )
         thisModule.addImport(cun,'javax.persistence.Id');
      if (cl.isMappedSuperClass())
         thisModule.addImport(cun,'javax.persistence.MappedSuperclass');
      if (cl.hasStereotype('DiscriminatorColumn')) {
            thisModule.addImport(cun,'javax.persistence.DiscriminatorColumn');
            thisModule.addImport(cun,'javax.persistence.DiscriminatorType');
      }

   }
}

rule addImport(cun:JAST!CompilationUnit,pname:String) {
do {
   if (not(cun.hasImport(pname)))
      thisModule.genImport(cun,pname);
}
}

helper context JAST!CompilationUnit def: hasImport( name:String) : Boolean =
   not(self.imports->select(i|i.name.fullyQualifiedName=name)->isEmpty() );

rule genImport(cun:JAST!CompilationUnit,pname:String) {
to
   imp : JAST!ImportDeclaration (
      name <- pname.genNameStr()
      )
do {
   cun.imports<-cun.imports->append(imp);
}
}

helper context String def: genNameStr() : JAST!Name =
   if (self.indexOf('.')>0) then
      thisModule.genQualifiedNameStr(self)
   else
      thisModule.genSimpleNameStr(self)
   endif;

--precondition: MUST have a . in the name!
lazy rule genQualifiedNameStr {
from
   str : String
to
   res : JAST!QualifiedName (
      fullyQualifiedName<-str,
      name<- thisModule.genSimpleNameStr(str.substring(str.lastIndexOf('.')
+2,str.size()) )  ,
      qualifier<- str.substring(1,str.lastIndexOf('.')).genNameStr()

      )
}
helper context UML!Element def: getStereotypeValue(stName: String,valName: String) :
OclAny =
   if ((self.getAppliedStereotype(thisModule.profileName+'::'+stName)).oclIsUndefined())
then
      OclUndefined
   else if
(self.getValue(self.getAppliedStereotype(thisModule.profileName+'::'+stName),valName).oc
lIsUndefined() ) then
         OclUndefined
      else

   self.getValue(self.getAppliedStereotype(thisModule.profileName+'::'+stName),valName)
      endif
   endif;


helper context JAST!ASTNode def: addAnnotation(name: String,values:
Sequence(TupleType(name:String, exp:JAST!Expression))) : JAST!Annotation =
```

```
   if (values->isEmpty()) then
      thisModule.addMarkerAnnotation(Tuple{typ=self,name=name})
   else if (values->size()=1 and values->first().name='') then
         thisModule.addSingleMemberAnnotation(Tuple{typ=self,name=name,exp=(values-
>first().exp)})
      else
         thisModule.addNormalAnnotation(Tuple{typ=self,name=name,pvalues=values})
      endif
   endif;

helper context JAST!TypeDeclaration def: addMarkerAnnotation(name: String) : JAST!
Annotation =
      thisModule.addMarkerAnnotation(Tuple{typ=self,name=name});

lazy rule addMarkerAnnotation {
from
   p : TupleType(typ: JAST!TypeDeclaration,name: String)

to
   res : JAST!MarkerAnnotation (
      typeName <- thisModule.genSimpleNameStr(p.name)
      )
do {
   if (p.typ.oclIsKindOf( JAST!BodyDeclaration )) {
      p.typ.modifiers<-p.typ.modifiers->append(res);
   }
  }
}
helper context JAST!BodyDeclaration def: addSingleMemberAnnotation(name:
String,exp:JAST!Expression ): JAST!Annotation =
   thisModule.addSingleMemberAnnotation(Tuple{typ=self,name=name,exp=exp});

lazy rule addSingleMemberAnnotation {
from
   p : TupleType(typ: JAST!ASTNode,name: String,exp:JAST!Expression )

to
   res : JAST!SingleMemberAnnotation (
      typeName <- thisModule.genSimpleNameStr(p.name),
      value<-p.exp
      )

do {
   if (p.typ.oclIsKindOf( JAST!BodyDeclaration )) {
      p.typ.modifiers<-p.typ.modifiers->append(res);
   }
}
}

lazy rule addNormalAnnotation {
from
   p : TupleType(typ: JAST!ASTNode,name: String,pvalues: Sequence(TupleType(name:String,
exp:JAST!Expression)) )
to
   res : JAST!NormalAnnotation (
      typeName <- thisModule.genSimpleNameStr(p.name)
      )

do {
   if (p.typ.oclIsKindOf( JAST!BodyDeclaration )) {
      p.typ.modifiers<-p.typ.modifiers->append(res);
   }
   for (k in p.pvalues) {
      thisModule.newMemberValuePair(Tuple{na=res,name=k.name,exp=k.exp});
   }
}
}


rule addClassAnnotations(cl:UML!Classifier,typ: JAST!TypeDeclaration,cun:JAST!
CompilationUnit) {
using {
      st_gens: Sequence(UML!Generalization) = UML!Generalization.allInstances()-
>select(g|g.general=cl)->select(g|g.hasStereotype('SingleTable'));
      j_gens: Sequence(UML!Generalization) = UML!Generalization.allInstances()-
```

```
>select(g|g.general=cl)->select(g|g.hasStereotype('Joined'));
      tpc_gens: Sequence(UML!Generalization) = UML!Generalization.allInstances()-
>select(g|g.general=cl)->select(g|g.hasStereotype('TablePerClass'));
      tab: OclAny = cl.getStereotypeValue('Entity','table');
      sectabs: Sequence(UML!Element) =
cl.getStereotypeValue('Entity','secondaryTables');
   }

do {
      if (cl.isEntity())      {


typ.addAnnotation('Entity',Sequence{Tuple{name='name',val=cl.getStereotypeValue('Entity'
,'name')}}
               ->select(tp|tp.val<>OclUndefined)
               ->collect(tp|Tuple{name=tp.name,exp=thisModule.genStringLiteral(tp.val)})

            );
            if (tab<>OclUndefined) {
               typ.addAnnotation('Table',Sequence{
                        Tuple{name='name',val=tab.name},
                        Tuple{name='catalog',val=tab.catalog},
               Tuple{name='schema',val=tab.schema},
               Tuple{name='uniqueConstraints',val=
                        typ.genUniqueConstraints(tab.uniqueConstraints,cun)
                        }
                        }
               ->select(tp|tp.val<>OclUndefined)
               ->collect(tp|Tuple{name=tp.name,exp=
                        if (tp.name='uniqueConstraints') then
                                 tp.val
                        else
                                 thisModule.genStringLiteral(tp.val)
                        endif }
                        )
               );
               thisModule.addImport(cun,'javax.persistence.Table');
               if (not(tab.uniqueConstraints->isEmpty()))
               thisModule.addImport(cun,'javax.persistence.UniqueConstraint');


            }
         if (not(sectabs->isEmpty())) {
            typ.addAnnotation('SecondaryTables',Sequence{
               Tuple{name='value',exp=thisModule.genArrayInitializer(
                        cl.addSecondaryTables(typ,sectabs,cun) )

               }
            });
            thisModule.addImport(cun,'javax.persistence.UniqueConstraint');
            thisModule.addImport(cun,'javax.persistence.SecondaryTable');
            thisModule.addImport(cun,'javax.persistence.SecondaryTables');
            thisModule.addImport(cun,'javax.persistence.PrimaryKeyJoinColumn');
            thisModule.addImport(cun,'javax.persistence.PrimaryKeyJoinColumns');

         }
      }
      if (cl.isMappedSuperClass())
         typ.addMarkerAnnotation('MappedSuperclass');
      if (cl.hasStereotype('Embeddable')) {
         typ.addMarkerAnnotation('Embeddable');
         thisModule.addImport(cun,'javax.persistence.Embeddable');
      }
      if (cl.hasStereotype('IdClass')) {
         typ.addAnnotation('IdClass',Sequence{
                        Tuple{name='',exp=
                        let pkCl:UML!Class=cl.getStereotypeValue('IdClass','idClass') in


   thisModule.genTypeLiteral(pkCl.selectTypeCtx(cl).selectDataType())

                                 }

         });
         thisModule.addImport(cun,'javax.persistence.IdClass');
```

```
      }

      if (cl.hasStereotype('AttributeOverrides')) {


typ.genAttributeOverrides(cun,typ,cl.getStereotypeValue('AttributeOverrides','value'))
   ;
         thisModule.addImport(cun,'javax.persistence.AttributeOverrides');
         thisModule.addImport(cun,'javax.persistence.AttributeOverride');
         thisModule.addImport(cun,'javax.persistence.Column');
      }
      if (cl.hasStereotype('TableGenerator')) {
         cl.addTableGenerator(typ,cun);
         if (not(self.getStereotypeValue('TableGenerator','uniqueConstraints')-
>isEmpty()) )
            thisModule.addImport(cun,'javax.persistence.UniqueConstraint');
         thisModule.addImport(cun,'javax.persistence.TableGenerator');
      }
      if (cl.hasStereotype('SequenceGenerator')) {
         cl.addSequenceGenerator(typ);
         thisModule.addImport(cun,'javax.persistence.SequenceGenerator');
      }

      for (gen in cl.getGeneralizations()->select(g|g.hasStereotype('SingleTable'))) {
         if (gen.getStereotypeValue('SingleTable','discriminatorValue')<>OclUndefined) {
            typ.addAnnotation('DiscriminatorValue',
               Sequence{Tuple{name='',


exp=thisModule.genStringLiteral(gen.getStereotypeValue('SingleTable','discriminatorValue
'))
               }} );

            thisModule.addImport(cun,'javax.persistence.DiscriminatorValue');

         }
      }
      for (gen in cl.getGeneralizations()->select(g|g.hasStereotype('Joined'))) {
         if (gen.getStereotypeValue('Joined','discriminatorValue')<>OclUndefined) {
            typ.addAnnotation('DiscriminatorValue',
               Sequence{Tuple{name='',


exp=thisModule.genStringLiteral(gen.getStereotypeValue('Joined','discriminatorValue'))
               }} );

            thisModule.addImport(cun,'javax.persistence.DiscriminatorValue');
         }


thisModule.genPrimaryKeyJoinColumn(gen,typ,cun,gen.getStereotypeValue('Joined','pkJoinCo
ls'));
      }

      if (typ.genAssociationOverrides(cun,typ,cl.getGeneralizations()
         ->select(g|g.hasStereotype('AssociationOverrides'))
         ->collect(gen|gen.getStereotypeValue('AssociationOverrides','overrides'))
         ->flatten()
         )<>OclUndefined) {
            thisModule.addImport(cun,'javax.persistence.JoinColumns');
            thisModule.addImport(cun,'javax.persistence.JoinColumn');
            thisModule.addImport(cun,'javax.persistence.AssociationOverrides');
            thisModule.addImport(cun,'javax.persistence.AssociationOverride');
      }

   }
}

helper context JAST!ASTNode def: genUniqueConstraints(ucs: Sequence(UML!
Element),cun:JAST!CompilationUnit) : JAST!Expression =
   if (ucs->exists(uc|not(uc.columnNames->isEmpty()) ) )  then
      thisModule.genArrayInitializer(ucs->collect(uc|
cun.addAnnotation('UniqueConstraint',
         Sequence{
```

```
            Tuple{name='columnNames',exp=
               thisModule.genArrayInitializer(uc.columnNames
               ->collect(cname|thisModule.genStringLiteral(cname)) )
            }

         })
      ) )
   else
      OclUndefined
   endif;

helper context UML!Class def: addSecondaryTables(typ: JAST!ASTNode, tabs: Sequence(UML!
Element),cun:JAST!CompilationUnit) : Sequence(JAST!Annotation) =
   tabs->collect(jc|
      cun.addAnnotation('SecondaryTable', Sequence{
         Tuple{name='name',val=jc.name},
         Tuple{name='catalog',val=jc.catalog},
         Tuple{name='schema',val=jc.schema},

   Tuple{name='uniqueConstraints',val=typ.genUniqueConstraints(jc.uniqueConstraints,cun)
},
         Tuple{name='pkJoinColumns',val=
            if (jc.primaryKeyJoinColumn->isEmpty()) then
               OclUndefined
            else


thisModule.genArrayInitializer(self.addPrimaryKeyJoinColumn(cun,jc.primaryKeyJoinColumn)
)
            endif
            }
      }->select(tp|tp.val<>OclUndefined)
       ->collect(tp|Tuple{name=tp.name,exp=
         if (tp.val.oclIsKindOf(JAST!Expression)) then
            tp.val
         else
            thisModule.genStringLiteral(tp.val)
         endif

       })
      )
   );

rule genPrimaryKeyJoinColumn(gen:UML!Element,typ: JAST!TypeDeclaration,cun:JAST!
CompilationUnit,pkcols:Sequence(UML!Element) ) {
do {
   if (pkcols<>OclUndefined) {
      if (pkcols->size()=1) {
         gen.addPrimaryKeyJoinColumn(typ,pkcols);

      } else if (pkcols->size()>1) {
         typ.addAnnotation('PrimaryKeyJoinColumns',Sequence{
            Tuple{name='value',exp=thisModule.genArrayInitializer(
               gen.addPrimaryKeyJoinColumn(cun,pkcols) )
               }
         });
         thisModule.addImport(cun,'javax.persistence.PrimaryKeyJoinColumns');
      }
      thisModule.addImport(cun,'javax.persistence.PrimaryKeyJoinColumn');
   }
}
}

helper context UML!Element def: addPrimaryKeyJoinColumn(cun: JAST!ASTNode, cols:
Sequence(UML!Element)) : Sequence(JAST!Annotation) =
   cols->collect(jc|
      cun.addAnnotation('PrimaryKeyJoinColumn', Sequence{
         Tuple{name='name',val=jc.name},
         Tuple{name='columnDefinition',val=jc.columnDefinition},
         Tuple{name='referencedColumnName',val=jc.referencedColumnName}
      }->select(tp|tp.val<>OclUndefined)
       ->collect(tp|Tuple{name=tp.name,exp=thisModule.genStringLiteral(tp.val)})

      )
   );
```

```
lazy rule newMemberValuePair {
from
   p : TupleType(na: JAST!NormalAnnotation,name: String,exp: JAST!Expression)

to
      pair: JAST!MemberValuePair (
         name<-thisModule.genSimpleNameStr(p.name),
         value<-p.exp
         )
   do {
      p.na.values<-p.na.values->append(pair);
   }
}



helper context UML!Element def: hasStereotype(name:String) : Boolean =
   not (self.getAppliedStereotype(thisModule.profileName+'::'+name).oclIsUndefined());

helper context UML!Element def: isEntity() : Boolean =
if (self.getAppliedStereotype(thisModule.profileName+'::Entity').oclIsUndefined()) then
      false
   else
      true
   endif;

helper context UML!Element def: isMappedSuperClass() : Boolean =
if
(self.getAppliedStereotype(thisModule.profileName+'::MappedSuperclass').oclIsUndefined()
) then
      false
   else
      true
   endif;

helper context UML!Element def: isEGenericType() : Boolean =
if (self.getAppliedStereotype('Ecore::EGenericType').oclIsUndefined()) then
      if (self.templateParameter.oclIsUndefined() ) then
            false
      else
         true
      endif
else
   true
endif;
helper context UML!Element def: isLob() : Boolean =
if (self.getAppliedStereotype(thisModule.profileName+'::Lob').oclIsUndefined()) then
      false
else
   true
endif;


helper context UML!Element def: isWildCardType() : Boolean =
false;

helper context UML!Element def: isID() : Boolean =
if (self.getAppliedStereotype(thisModule.profileName+'::Id').oclIsUndefined()) then
      false
   else
      true
   endif;
helper context UML!Operation def: isConstructor() : Boolean =
if (self.name=self.owner.name) then
      true
   else
      false
   endif;

rule genMethod {
from
   op : UML!Operation (
      not (op.class.oclIsUndefined())
```

```
      )
to
   method: JAST!MethodDeclaration (
      name <- thisModule.genSimpleNameStr(op.name),
      constructor<-op.isConstructor(),
      returnType<-if (op.type.oclIsUndefined()) then
                thisModule.genVoidType(op)
                else
                op.type.genAnyType(op)
                endif,
      parameters<-method.parameters.append(
         op.ownedParameter->select(p|not(p.direction=#return))->collect(p|
thisModule.genMethodParameter(p))
         ),
      body<-thisModule.genMethodBlock(op)

   )
do {
   if (op.visibility=#public)
      thisModule.genBodyModifiers(method,'public');
   else if (op.visibility=#private)
      thisModule.genBodyModifiers(method,'private');
   else if (op.visibility=#protected)
      thisModule.genBodyModifiers(method,'protected');
   if (op.isAbstract())
      thisModule.genBodyModifiers(method,'abstract');
   if (op.isStatic())
      thisModule.genBodyModifiers(method,'static');

}

}

lazy rule genMethodBlock {
from
   op : UML!Operation
to
   block : JAST!Block (
      )
do {
   if (not(op.type.oclIsUndefined()))
      thisModule.genMethodReturn(op,block);
}
}
rule genMethodReturn(op:UML!Operation,block:JAST!Block) {

to
   res : JAST!ReturnStatement (
      expression<-op.genReturnDefaultExpression()
      )
do {
   block.statements <-block.statements->append(res);
}
}
helper context UML!Operation def: genReturnDefaultExpression() : JAST!Expression =
   if (self.type.selectTypeCtx(self).isPrimitiveJava()) then
      thisModule.genNumberLiteral('0')
   else
      thisModule.genNullLiteral(self)
   endif;



helper context UML!Property def: getOtherEndType() : UML!Class  =
   self.type;

helper context UML!Class def: isPropertyAccess() : Boolean  =
   let access:UML!EnumerationLiteral = self.getStereotypeValue('Entity','access') in
      if (access=OclUndefined) then
         false
      else
         access.name = 'PROPERTY'
      endif;
rule genProperty {
from
```

```
   p : UML!Property (
      not (p.class.oclIsUndefined())
      )
using {
      propertyAccess: Boolean = p.class.isPropertyAccess();
   }
to
   typ: JAST!FieldDeclaration (
      fragments<- typ.fragments.append(thisModule.genVariableDcl(p)),
      type<-p.genPropertyType()
   ),
   getter: JAST!MethodDeclaration (
      name <- thisModule.genSimpleNameStr(p.name.asProperty()),
      returnType<-p.genPropertyType(),
      body<-thisModule.genGetterReturnBlock(p)
   ),
   setter: JAST!MethodDeclaration (
      name <- thisModule.genSimpleNameStr(p.name.asSetProperty()),
      returnType<-thisModule.genVoidType(p),
      parameters<-setter.parameters.append(thisModule.genSetterVariable(p)),
      body<-thisModule.genSetterBlock(p)
   )
do {
   if (propertyAccess)
      thisModule.addPropAnnotations(p,typ);
   else
      thisModule.addPropAnnotations(p,getter);

   thisModule.genBodyModifiers(typ,'private');
   if (p.visibility=#public) {
      thisModule.genBodyModifiers(getter,'public');
      if (p.isReadOnly())
         thisModule.genBodyModifiers(setter,'protected');
      else
         thisModule.genBodyModifiers(setter,'public');
   } else if (p.visibility=#private) {
      thisModule.genBodyModifiers(getter,'private');
      thisModule.genBodyModifiers(setter,'private');
   } else if (p.visibility=#protected) {
      thisModule.genBodyModifiers(getter,'protected');
      thisModule.genBodyModifiers(setter,'protected');
   }
   if (p.isStatic()) {
      thisModule.genBodyModifiers(getter,'static');
      thisModule.genBodyModifiers(setter,'static');
   }


}

}

rule genBodyModifiers(typ: JAST!BodyDeclaration, pmod:String) {
to
   modifier: JAST!Modifier (
   )
do {
   if (pmod='public')
      modifier.public<-true;
   if (pmod='abstract')
      modifier.abstract<-true;
   if (pmod='protected')
      modifier.protected<-true;
   if (pmod='private')
      modifier.private<-true;
   if (pmod='static')
      modifier.static<-true;

   typ.modifiers<-typ.modifiers->append(modifier);

}
}

helper context String def: asSetProperty() : String =
   if (self.size()>1) then
```

```
      'set'+self.substring(1,1).toUpper()+self.substring(2,self.size())
   else
      'set'+self
   endif;

helper context String def: asProperty() : String =
   if (self.size()>1) then
      'get'+self.substring(1,1).toUpper()+self.substring(2,self.size())
   else
      'get'+self
   endif;

helper context UML!Property def: getOtherEnd() : UML!Property  =
   self.association.memberEnd->select(e| not(self=e))->first();


helper context UML!Property def: hasAssociation() : Boolean =
   if (self.association.oclIsUndefined()) then
      false
   else
      not(self.type.oclIsUndefined())
   endif;


lazy rule addIdAnnotation {
from
   pcl : UML!Property
to
   res : JAST!MarkerAnnotation (
      typeName <- thisModule.genSimpleNameStr('Id')
      )
}


helper context JAST!ASTNode def: genAttributeOverrides(cun: JAST!CompilationUnit,typ:
JAST!BodyDeclaration,cols: Sequence(UML!Element)) : JAST!Annotation =
   if (cols->size()=1) then
      self.addAttributeOverrides(typ,cols)->first()
   else
      typ.addAnnotation('AttributeOverrides',Sequence{


Tuple{name='value',exp=thisModule.genArrayInitializer(self.addAttributeOverrides(cun,col
s))}
      })
   endif;
helper context JAST!ASTNode def: addAttributeOverrides(cun: JAST!ASTNode, cols:
Sequence(UML!Element)) : Sequence(JAST!Annotation) =
   cols->collect(ao|
         cun.addAnnotation('AttributeOverride', Sequence{
            Tuple{name='name',exp=thisModule.genStringLiteral(ao.property.name)},
            Tuple{name='column',exp=cun.addColumn(cun,ao.column)}
         })
      );
helper context JAST!ASTNode def: genAssociationOverrides(cun: JAST!CompilationUnit,typ:
JAST!BodyDeclaration,cols: Sequence(UML!Element)) : JAST!Annotation =
   if (cols->isEmpty()) then
      OclUndefined
   else
      if (cols->size()=1) then
         self.addAssociationOverrides(typ,cols)->first()
      else
         typ.addAnnotation('AssociationOverrides',Sequence{
            Tuple{name='value',

   exp=thisModule.genArrayInitializer(self.addAssociationOverrides(cun,cols))}

         })
      endif
   endif;
helper context JAST!ASTNode def: addAssociationOverrides(cun: JAST!ASTNode, cols:
Sequence(UML!Element)) : Sequence(JAST!Annotation) =
   cols->collect(ao|
         cun.addAnnotation('AssociationOverride', Sequence{
            Tuple{name='name',exp=thisModule.genStringLiteral(ao.property.name)},
```

```
            Tuple{name='joinColumns',exp=cun.genJoinColumns(cun,cun,ao.joinColumn)}
         })
      );

-- addPropAnnotations:
-- Input: Class or interface properties
rule addPropAnnotations(p: UML!Property,typ: JAST!TypeDeclaration) {
using{
   column:UML!Element=p.getStereotypeValue('Column','value');
   cun: JAST!CompilationUnit = thisModule.resolveTemp(p.owner,'cun');
}
do {

   if (p.isID())
      typ.modifiers<-typ.modifiers->append(thisModule.addIdAnnotation(p));

   if (p.hasStereotype('EmbeddedId')) {
      typ.addAnnotation('EmbeddedId', Sequence{});
      thisModule.addImport(cun,'javax.persistence.EmbeddedId');
   }
   if (p.hasStereotype('Transient')) {
      typ.addAnnotation('Transient', Sequence{});
      thisModule.addImport(cun,'javax.persistence.Transient');
   }
   if (p.hasStereotype('Version')) {
      typ.addAnnotation('Version', Sequence{});
      thisModule.addImport(cun,'javax.persistence.Version');
   }
   if (p.hasStereotype('Lob')) {
      typ.addAnnotation('Lob', Sequence{});
      thisModule.addImport(cun,'javax.persistence.Lob');
   }
   if (p.hasStereotype('AttributeOverrides')) {

   typ.genAttributeOverrides(cun,typ,p.getStereotypeValue('AttributeOverrides','value'))
   ;
      thisModule.addImport(cun,'javax.persistence.Column');
      thisModule.addImport(cun,'javax.persistence.AttributeOverrides');
      thisModule.addImport(cun,'javax.persistence.AttributeOverride');
   }

   if (p.hasStereotype('Basic')) {
      typ.addAnnotation('Basic', Sequence{
         Tuple{name='fetch',val= p.mockNullEnum(p.getStereotypeValue('Basic','fetch'))

         },
         Tuple{name='optional',val=p.getStereotypeValue('Basic','optional')}

      }->select(tp|tp.val<>OclUndefined)
       ->collect(tp|Tuple{name=tp.name,exp=
            if (tp.val.oclIsKindOf(UML!EnumerationLiteral)) then
               (tp.val.enumeration.name+'.'+tp.val.name).genNameStr()
            else
               thisModule.genBooleanLiteral(tp.val)
            endif
       })
      );
      thisModule.addImport(cun,'javax.persistence.Basic');
   }
   if (p.hasStereotype('GeneratedValue')) {
      typ.addAnnotation('GeneratedValue', Sequence{
         Tuple{name='strategy',val=
            let en:UML!EnumerationLiteral =
p.getStereotypeValue('GeneratedValue','strategy') in
               if (en.name='AUTO') then
               OclUndefined
               else
               en
               endif
         },
         Tuple{name='generator',val=p.getStereotypeValue('GeneratedValue','generator')}

      }->select(tp|tp.val<>OclUndefined)
       ->collect(tp|Tuple{name=tp.name,exp=
            if (tp.val.oclIsKindOf(UML!EnumerationLiteral)) then
```

```
               (tp.val.enumeration.name+'.'+tp.val.name).genNameStr()
            else
               thisModule.genStringLiteral(tp.val)
            endif
       })
      );
      thisModule.addImport(cun,'javax.persistence.GeneratedValue');
      if (typ.modifiers->select(m|m.oclIsKindOf(JAST!NormalAnnotation))
         ->exists(m|m.values->exists(v|v.name.identifier='strategy')))
         thisModule.addImport(cun,'javax.persistence.GenerationType');
   }
   if (p.hasStereotype('Enumerated')) {
      typ.addAnnotation('Enumerated', Sequence{
         Tuple{name='value',val=
            let en:UML!EnumerationLiteral = p.getStereotypeValue('Enumerated','value')
in
            if (en.name='ORDINAL') then
               OclUndefined
            else
               en
            endif
         }

      }->select(tp|tp.val<>OclUndefined)
       ->collect(tp|
Tuple{name=tp.name,exp=(tp.val.enumeration.name+'.'+tp.val.name).genNameStr()})
      );
      thisModule.addImport(cun,'javax.persistence.Enumerated');
   }
   if (p.hasStereotype('Embedded')) {
      typ.addAnnotation('Embedded', Sequence{});
      thisModule.addImport(cun,'javax.persistence.Embedded');
   }
   if (p.hasStereotype('Column')) {
      typ.addColumn(typ,column);
      thisModule.addImport(cun,'javax.persistence.Column');
   }
   if (p.hasStereotype('TableGenerator')) {
      p.addTableGenerator(typ,cun);
      if (not(self.getStereotypeValue('TableGenerator','uniqueConstraints')-
>isEmpty()) )
            thisModule.addImport(cun,'javax.persistence.UniqueConstraint');
      thisModule.addImport(cun,'javax.persistence.TableGenerator');
   }
   if (p.hasStereotype('SequenceGenerator')) {
         p.addSequenceGenerator(typ);
         thisModule.addImport(cun,'javax.persistence.SequenceGenerator');
   }
   if (p.hasStereotype('OrderBy')) {
      typ.addAnnotation('OrderBy', Sequence{
         Tuple{name='',val=p.getStereotypeValue('OrderBy','expression')}

      }->select(tp|tp.val<>OclUndefined)
       ->collect(tp|Tuple{name=tp.name,exp=thisModule.genStringLiteral(tp.val)})
      );
      thisModule.addImport(cun,'javax.persistence.OrderBy');
   }
   if (p.hasStereotype('TimeStamp') or p.hasStereotype('Date') or
p.hasStereotype('Time')) {
      if (p.hasStereotype('TimeStamp'))

   typ.addSingleMemberAnnotation('Temporal','TemporalType.TIMESTAMP'.genNameStr());
      else if (p.hasStereotype('Date'))
         typ.addSingleMemberAnnotation('Temporal','TemporalType.DATE'.genNameStr());
      else
         typ.addSingleMemberAnnotation('Temporal','TemporalType.TIME'.genNameStr());
      thisModule.addImport(cun,'javax.persistence.Temporal');
      thisModule.addImport(cun,'javax.persistence.TemporalType');
   }
   if (p.hasAssociation())
      if (p.getOtherEndType().isEntity())
         if (p.isMultivalued() and p.getOtherEnd().isMultivalued())
            -- Many to Many
             thisModule.manyToMany(typ,p);
         else
```

```
            --One to Many
            if (p.isMultivalued() and
               not(p.getOtherEnd().isMultivalued()))
                thisModule.oneToMany(typ,p);
            else
               --Many to One
               if (not(p.isMultivalued()) and
                      p.getOtherEnd().isMultivalued())
               thisModule.manyToOne(typ,p);
               else
               --One To One
               thisModule.oneToOne(typ,p);

}
}
helper context UML!Element def: addTableGenerator(typ: JAST!ASTNode,cun:JAST!
CompilationUnit) : JAST!Annotation =
      typ.addAnnotation('TableGenerator', Sequence{
         Tuple{name='name',val=self.getStereotypeValue('TableGenerator','name')},
         Tuple{name='table',val=self.getStereotypeValue('TableGenerator','table')},
         Tuple{name='catalog',val=self.getStereotypeValue('TableGenerator','catalog')},
         Tuple{name='schema',val=self.getStereotypeValue('TableGenerator','schema')},


Tuple{name='pkColumnName',val=self.getStereotypeValue('TableGenerator','pkColumnName')},


Tuple{name='valueColumnName',val=self.getStereotypeValue('TableGenerator','valueColumnNa
me')},


Tuple{name='pkColumnValue',val=self.getStereotypeValue('TableGenerator','pkColumnValue')
},


Tuple{name='initialValue',val=thisModule.default(self.getStereotypeValue('TableGenerator
','initialValue'),0)},


Tuple{name='allocationSize',val=thisModule.default(self.getStereotypeValue('TableGenerat
or','allocationSize'),50)},
         Tuple{name='uniqueConstraints',val=


typ.genUniqueConstraints(self.getStereotypeValue('TableGenerator','uniqueConstraints'),c
un)
               }


      }->select(tp|tp.val<>OclUndefined)
       ->collect(tp|Tuple{name=tp.name,exp=
            if (tp.val.oclIsKindOf(String) ) then
               thisModule.genStringLiteral(tp.val)
            else
               if (tp.val.oclIsKindOf(Integer)) then
               thisModule.genNumberLiteral(tp.val.toString())
               else
               tp.val
               endif
            endif
       })
      );
helper context UML!Element def: addSequenceGenerator(typ: JAST!ASTNode) : JAST!
Annotation =
      typ.addAnnotation('SequenceGenerator', Sequence{
         Tuple{name='name',val=self.getStereotypeValue('SequenceGenerator','name')},


Tuple{name='sequenceName',val=self.getStereotypeValue('SequenceGenerator','sequenceName'
)},


Tuple{name='initialValue',val=thisModule.default(self.getStereotypeValue('SequenceGenera
tor','initialValue'),1)},
```

```
Tuple{name='allocationSize',val=thisModule.default(self.getStereotypeValue('SequenceGene
rator','allocationSize'),50)}
      }->select(tp|tp.val<>OclUndefined)
       ->collect(tp|Tuple{name=tp.name,exp=
            if (tp.val.oclIsKindOf(String) ) then
               thisModule.genStringLiteral(tp.val)
            else
               if (tp.val.oclIsKindOf(Integer)) then
               thisModule.genNumberLiteral(tp.val.toString())
               else
               tp.val
               endif
            endif
       })
      );
helper context JAST!ASTNode def: addColumn(typ: JAST!ASTNode,column: UML!Element) :
JAST!Annotation =
      typ.addAnnotation('Column', Sequence{
         Tuple{name='name',val=column.name},
         Tuple{name='unique',val=thisModule.default(column.unique,false)   },
         Tuple{name='nullable',val=thisModule.default(column.nullable,true)},
         Tuple{name='insertable',val=thisModule.default(column.insertable,true)},
         Tuple{name='updatable',val=thisModule.default(column.updatable,true)},
         Tuple{name='columnDefinition',val=column.columnDefinition},
         Tuple{name='table',val=column.table},
         Tuple{name='length',val=thisModule.default(column.length,255)},
         Tuple{name='precision',val=thisModule.default(column.precision,0)},
         Tuple{name='scale',val=thisModule.default(column.scale,0)}

      }->select(tp|tp.val<>OclUndefined)
       ->collect(tp|Tuple{name=tp.name,exp=
            if (tp.val.oclIsKindOf(Boolean) ) then
               thisModule.genBooleanLiteral(tp.val)
            else
               if (tp.val.oclIsKindOf(Integer)) then
               thisModule.genNumberLiteral(tp.val.toString())
               else
               thisModule.genStringLiteral(tp.val)
               endif
            endif
       })
      );
lazy rule genMethodParameter {
from
   p : UML!Parameter
to
   res : JAST!SingleVariableDeclaration (
      name <-thisModule.genSimpleNameStr(p.name),
      type <-p.type.genAnyType(p.operation)
      )
}
lazy rule genSetterVariable {
from
   p : UML!Property
to
   res : JAST!SingleVariableDeclaration (
      name <-thisModule.genSimpleNameStr(p.name.asField()),
      type <-p.genPropertyType()
      )
}

lazy rule genVariableDcl {
from
   p : UML!Property
to
   res : JAST!VariableDeclarationFragment (
      name <-thisModule.genSimpleNameStr(p.name.asField())
      )
}
lazy rule genSetterBlock {
from
   p : UML!Property
to
   body : JAST!Block (
```

```
      statements <-body.statements.append(expSt)
   ),
   this : JAST!ThisExpression (
   ),
   leftSide : JAST!FieldAccess (
      expression<-this,
      name<-thisModule.genSimpleNameStr(p.name.asField())
   ),

   assigment : JAST!Assignment (
      operator<-#ASSIGN,
      leftHandSide<-leftSide,
      rightHandSide<-thisModule.genSimpleNameStr(p.name.asField())
   ),
    expSt : JAST!ExpressionStatement (
      expression <-assigment
   )

}
lazy rule genGetterReturnBlock {
from
   p : UML!Property
to
   res : JAST!Block (
      statements <-res.statements.append(thisModule.genGetterReturn(p))
      )
}

lazy rule genGetterReturn {
from
   p : UML!Property
to
   res : JAST!ReturnStatement (
      expression <-thisModule.genSimpleNameStr(p.name.asField())
      )
}

lazy rule genSimpleName {
from
   pcl : UML!Class
to
   res : JAST!SimpleName (
      identifier <- pcl.name,
      fullyQualifiedName <- pcl.name
      )
}

lazy rule genSimpleNameStr {
from
   str : String
to
   res : JAST!SimpleName (
      identifier <- str,
      fullyQualifiedName <- str
      )
}

lazy rule genPackageDcl {
from
   pcl : UML!Classifier
to
   res : JAST!PackageDeclaration (
      name <- thisModule.genSimpleNameStr(pcl.namespace.qualifiedName.removeContext())
      )
}

helper context UML!Type def: genAnyType(cl:UML!TemplateableElement) : JAST!Type =
   if (not(cl.getTemplateBindings()->select(b|b.signature.template=self)->isEmpty()))
then
      thisModule.genGenericType(cl.getTemplateBindings()->select(b|
b.signature.template=self)->first())
   else
       self.selectTypeCtx(cl).selectDataType()
   endif ;
```

```
helper context UML!Property def: genPropertyType() : JAST!Type =
   if (self.type.isEGenericType()) then
      thisModule.genGenericType(self.type.getTemplateBindings()->first())
   else
      if (self.isMultivalued()) then
         if (self.isLob()) then
            thisModule.genArrayType(self)
         else
            thisModule.genParameterizedType(self)
         endif
      else
         self.type.selectTypeCtx(self).selectDataType()
      endif
   endif ;

helper context String def: isPrimitiveJava() : Boolean =
   (self='byte' or self='double' or self='int' or self='long');
helper context String def: selectDataType() : JAST!Type =
if (self.isPrimitiveJava()) then
   thisModule.genPrimitiveType(self)
else
   thisModule.genDataType(self)
endif;

lazy rule genDataType {
from
   typ : String
to
   res : JAST!SimpleType (
      name <- thisModule.genSimpleNameStr(typ)
      )
}
lazy rule genPrimitiveType {
from
   typ : String
to
   res : JAST!PrimitiveType (
      code<-typ
      )
}


lazy rule genWildCard {
from
   p : UML!Type
using {
      upper: UML!Type =
p.getValue(p.getAppliedStereotype('Ecore::EGenericType'),'upperBound');
      lower: UML!Type =
p.getValue(p.getAppliedStereotype('Ecore::EGenericType'),'lowerBound');
   }
to
   res : JAST!WildcardType (
      bound <- if (upper.oclIsUndefined()) then
                lower.selectType().selectDataType()
             else
               upper.selectType().selectDataType()
            endif,
      upperBound <-not(upper.oclIsUndefined())

      )
}


lazy rule genGenericType {
from
   bind: UML!TemplateBinding
using {
   t : UML!Type = bind.boundElement;
   cun: JAST!CompilationUnit = if (t.isEGenericType()) then
                                  thisModule.resolveTemp(t.owner,'cun')
                               else
                                  thisModule.resolveTemp(t,'cun')
                               endif;
```

```
}
to
   res : JAST!ParameterizedType (
      typeArguments<-res.typeArguments->union(
      bind.getParameterSubstitutions()->collect(pa|pa.getActuals())->flatten()
      ->collect(pe|
            if (pe.isWildCardType() ) then
               thisModule.genWildCard(pe)
            else if (pe.isEGenericType()) then
               pe.selectTypeCtx(t).selectDataType()
               else
               pe.selectTypeCtx(t).selectDataType()
               endif
            endif
              )),
      type <- bind.signature.template.name.selectDataType()

      )
do {
   if (not(bind.signature.template.namespace.name=t.namespace.name))
      thisModule.addImport(cun,bind.signature.template.selectType());



   (bind.signature.template.name+'=temp type').println();

}

}

helper context UML!TypedElement def: selectCollectionType() : String =
   if
(self.getAppliedStereotype(thisModule.profileName+'::AssociationMapping').oclIsUndefined
()) then
      'Set' --default collection type
   else
      if
(self.getValue(self.getAppliedStereotype(thisModule.profileName+'::AssociationMapping'),
'collectionType').oclIsUndefined()) then
         'Set'
      else


self.getValue(self.getAppliedStereotype(thisModule.profileName+'::AssociationMapping'),'
collectionType')
      endif
   endif;

lazy rule genParameterizedType {
from
   p : UML!TypedElement
using {
    cun: JAST!CompilationUnit = thisModule.resolveTemp(p.owner,'cun');
}
to
   res : JAST!ParameterizedType (
      typeArguments<-res.typeArguments.append(p.type.selectTypeCtx(p).selectDataType()),
      type <- p.selectCollectionType().selectDataType()

      )
do {
   if (p.type.isEGenericType())
      (p.type.name+'=generic:').println();
   if (res.type.name.fullyQualifiedName='Set')
      thisModule.addImport(cun,'java.util.Set');
   if (res.type.name.fullyQualifiedName='List')
      thisModule.addImport(cun,'java.util.List');

}

}
lazy rule genArrayType {
from
   p : UML!TypedElement
to
```

```
   res : JAST!ArrayType (
      dimensions<-1,
      componentType<-p.type.selectTypeCtx(p).selectDataType()
      )
}
lazy rule genStringLiteral {
from
   lit : String
to
   typ: JAST!StringLiteral (
      literalValue<-lit,
      escapedValue<-'"'+lit+'"'
   )

}
lazy rule genBooleanLiteral {
from
   lit : Boolean
to
   typ: JAST!BooleanLiteral (
      booleanValue<-lit
   )

}
lazy rule genTypeLiteral {
from
   st : JAST!SimpleType
to
   tl: JAST!TypeLiteral (
      type<-st
   )

}


lazy rule genNumberLiteral {
from
   lit : String
to
   typ: JAST!NumberLiteral (
      token<-lit
   )

}
lazy rule genNullLiteral {
from
   e : UML!Element
to
   res : JAST!NullLiteral (
      )
}

lazy rule genVoidType {
from
   f : UML!Feature
to
   res : JAST!PrimitiveType (
      code<-'void'
      )
}
helper context UML!BehavioredClassifier def: selectType() : String =
   self.qualifiedName.removeContext();
helper context UML!Interface def: selectType() : String =
   self.qualifiedName.removeContext();

helper context UML!DataType def: selectType() : String =
   let name:String = self.qualifiedName in
   if (self.qualifiedName='uml::Boolean') then
      'java.lang.Boolean'
   else
      if (self.qualifiedName='uml::Integer') then
         'java.lang.Integer'
      else
         if (self.qualifiedName.endsWith('.byte'))  then
               'java.lang.Byte'
```

```
         else if (self.qualifiedName='uml::Double') then
               'java.lang.Double'
            else
               self.qualifiedName.removeContext()
            endif
         endif
      endif
   endif;

helper context UML!Element def: findContext() : UML!NamedElement =
   if (self.oclIsKindOf(UML!Classifier)) then
      self
   else
      if (self.owner=OclUndefined) then
         self
      else
         self.owner.findContext()
      endif
   endif;
helper context UML!Element def: selectTypeCtx(ctx:UML!NamedElement) : String =
   if (self.namespace=OclUndefined or ctx.findContext().namespace=OclUndefined) then
      self.selectType()
   else
      let str:String = self.namespace.qualifiedName.removeContext() in
      if (ctx.findContext().namespace.name=self.namespace.name or
self.namespace.qualifiedName.removeContext()='java.lang') then
         self.name
      else
         self.selectType()
      endif
   endif;

helper context String def: removeContext() : String =
   if (self.indexOf('::')>0) then
      self.substring(self.indexOf('::')+3,self.size()).removeContext2()
   else
      self
   endif;
helper context String def: removeContext2() : String =
   if (self.indexOf('::')>0) then
      self.regexReplaceAll('::','.')
   else
      self
   endif;
lazy rule genArrayInitializer {
from
   par : Sequence(JAST!Expression)
to
   ai : JAST!ArrayInitializer (
      expressions<-ai.expressions->union(par)
      )
}

-- EMF does not allow optional properties when using Enumerations.
-- this helper transform enum unamed literals in null (OclUndefined) values.
helper context UML!Element def: mockNullEnum(lit:UML!EnumerationLiteral) : UML!
EnumerationLiteral =
   if (lit=OclUndefined) then
      OclUndefined
   else
      if (lit.name='UNDEFINED') then
         OclUndefined
      else
         lit
      endif
   endif;

helper  def: default(value:OclAny,default:OclAny) : OclAny =
   if (value=OclUndefined) then
      OclUndefined
   else
      if (default=value) then
         OclUndefined
      else
         if (value.oclIsKindOf(UML!EnumerationLiteral)) then
```

```
            if (value.name=default) then
               OclUndefined
            else
               value
            endif
         else
            value
         endif
      endif
   endif;

rule oneToMany (typ: JAST!BodyDeclaration,pcl : UML!Property) {
using {
    cun: JAST!CompilationUnit = thisModule.resolveTemp(pcl.class,'cun');
    cascade: Sequence(UML!EnumerationLiteral) =
pcl.getStereotypeValue('AssociationMapping','cascade');
   joinTable :UML!Element=pcl.getStereotypeValue('AssociationMapping','joinTable');
   joinCols :Sequence(UML!
Element)=pcl.getStereotypeValue('AssociationMapping','joinColumns');
   mapKey :UML!Property=pcl.getStereotypeValue('AssociationMapping','mapKey');
}
do {

   typ.addAnnotation('OneToMany',Sequence{
         Tuple{name='mappedBy',val=  if (pcl.getOtherEnd().isNavigable()) then
                                                pcl.getOtherEnd().name
                                          else
                                                OclUndefined
                                          endif
         },
         Tuple{name='fetch',val=
   pcl.mockNullEnum(pcl.getStereotypeValue('AssociationMapping','fetch'))},

         Tuple{name='cascade',val=
            if (cascade=OclUndefined) then
               OclUndefined
            else if (cascade->isEmpty()) then
                           OclUndefined
               else
               thisModule.genArrayInitializer(
                        cascade->collect(c|('CascadeType.'+c.toString()).genNameStr())  )
               endif
            endif
             }
         }
         ->select(tp|tp.val<>OclUndefined)
         ->collect(tp|Tuple{name=tp.name,exp=if (tp.val.oclIsKindOf(String)) then

   thisModule.genStringLiteral(tp.val)
                                                            else if
(tp.val.oclIsKindOf(UML!EnumerationLiteral)) then


   (tp.val.enumeration.name+'.'+tp.val.name).genNameStr()
                                                                  else
                                                                        tp.val
                                                                  endif
                                                            endif



         })


            );

   thisModule.addImport(cun,'javax.persistence.OneToMany');

   if (joinTable<>OclUndefined) {
      pcl.addJoinTable(typ,joinTable);
      thisModule.addImport(cun,'javax.persistence.JoinTable');
      if (not(joinTable.uniqueConstraints->isEmpty()) )
         thisModule.addImport(cun,'javax.persistence.UniqueConstraint');
      if (not(joinTable.joinColumns.isEmpty() and joinTable.inverseJoinColumns.isEmpty()
)) {
         thisModule.addImport(cun,'javax.persistence.JoinColumns');
```

```
         thisModule.addImport(cun,'javax.persistence.JoinColumn');
      }

   }
   if (joinCols<>OclUndefined) {
      typ.genJoinColumns(cun,typ,joinCols);
      thisModule.addImport(cun,'javax.persistence.JoinColumns');
      thisModule.addImport(cun,'javax.persistence.JoinColumn');
   }
   if (mapKey<>OclUndefined) {
      pcl.genMapKey(typ,mapKey);
      thisModule.addImport(cun,'javax.persistence.MapKey');
   }

   if (typ.modifiers->select(m|m.oclIsKindOf(JAST!NormalAnnotation))
         ->exists(m|m.values->exists(v|v.name.identifier='cascade'))
      )
      thisModule.addImport(cun,'javax.persistence.CascadeType');

   if (typ.modifiers->select(m|m.oclIsKindOf(JAST!NormalAnnotation))
         ->exists(m|m.values->exists(v|v.name.identifier='fetch'))
      )
      thisModule.addImport(cun,'javax.persistence.FetchType');
}

}

helper context UML!Property def: genMapKey(typ: JAST!BodyDeclaration,mkey: UML!Property)
: JAST!Annotation =
   typ.addAnnotation('MapKey',Sequence{
            Tuple{name='name',exp=thisModule.genStringLiteral(mkey.name)}
   });

rule manyToMany (typ: JAST!BodyDeclaration,pcl : UML!Property) {
using {
    cun: JAST!CompilationUnit = thisModule.resolveTemp(pcl.class,'cun');
    cascade: Sequence(UML!EnumerationLiteral) =
pcl.getStereotypeValue('AssociationMapping','cascade');
   joinTable :UML!Element=pcl.getStereotypeValue('AssociationMapping','joinTable');
   joinCols :Sequence(UML!
Element)=pcl.getStereotypeValue('AssociationMapping','joinColumns');
   mapKey :UML!Property=pcl.getStereotypeValue('AssociationMapping','mapKey');
}
do {

   typ.addAnnotation('ManyToMany',Sequence{
         --the mappedby side cannot have jointable
         Tuple{name='mappedBy',val=  if (pcl.getOtherEnd().isNavigable()
                                                and joinTable=OclUndefined
                                                and
(pcl.getOtherEnd().getStereotypeValue('AssociationMapping','joinTable')<>OclUndefined or

pcl.getOtherEnd().name>pcl.name) -- alphabetic order otherwise
                                             ) then
                                                pcl.getOtherEnd().name
                                             else
                                                OclUndefined
                                             endif
         },
         Tuple{name='fetch',val=
   pcl.mockNullEnum(pcl.getStereotypeValue('AssociationMapping','fetch'))},

         Tuple{name='cascade',val=
            if (cascade=OclUndefined) then
               OclUndefined
            else if (cascade->isEmpty()) then
                              OclUndefined
               else
               thisModule.genArrayInitializer(
                        cascade->collect(c|('CascadeType.'+c.toString()).genNameStr())  )
               endif
            endif
             }
         }
         ->select(tp|tp.val<>OclUndefined)
```

```
         ->collect(tp|Tuple{name=tp.name,exp=if (tp.val.oclIsKindOf(String)) then

   thisModule.genStringLiteral(tp.val)
                                                                  else if
(tp.val.oclIsKindOf(UML!EnumerationLiteral)) then


   (tp.val.enumeration.name+'.'+tp.val.name).genNameStr()
                                                                        else
                                                                               tp.val
                                                                        endif
                                                                  endif


         })


            );

   thisModule.addImport(cun,'javax.persistence.ManyToMany');

   if (joinTable<>OclUndefined) {
      pcl.addJoinTable(typ,joinTable);
      thisModule.addImport(cun,'javax.persistence.JoinTable');
      if (not(joinTable.uniqueConstraints->isEmpty()) )
         thisModule.addImport(cun,'javax.persistence.UniqueConstraint');
      if (not(joinTable.joinColumns.isEmpty() and joinTable.inverseJoinColumns.isEmpty()
)) {
         thisModule.addImport(cun,'javax.persistence.JoinColumns');
         thisModule.addImport(cun,'javax.persistence.JoinColumn');
      }

   }
   if (joinCols<>OclUndefined) {
      typ.genJoinColumns(cun,typ,joinCols);
      thisModule.addImport(cun,'javax.persistence.JoinColumns');
      thisModule.addImport(cun,'javax.persistence.JoinColumn');

   }
   if (mapKey<>OclUndefined) {
      pcl.genMapKey(typ,mapKey);
   }
   if (typ.modifiers->select(m|m.oclIsKindOf(JAST!NormalAnnotation))
         ->exists(m|m.values->exists(v|v.name.identifier='cascade'))
      )
      thisModule.addImport(cun,'javax.persistence.CascadeType');

   if (typ.modifiers->select(m|m.oclIsKindOf(JAST!NormalAnnotation))
         ->exists(m|m.values->exists(v|v.name.identifier='fetch'))
      )
      thisModule.addImport(cun,'javax.persistence.FetchType');
}

}

helper context JAST!ASTNode def: addJoinColumns(cun: JAST!ASTNode, cols: Sequence(UML!
Element)) : Sequence(JAST!Annotation) =
   cols->collect(jc|
      cun.addAnnotation('JoinColumn', Sequence{
         Tuple{name='name',val=jc.name},
         Tuple{name='referencedColumnName',val=jc.referencedColumnName},
         Tuple{name='unique',val=thisModule.default(jc.unique,false)},
         Tuple{name='nullable',val=thisModule.default(jc.nullable,true)},
         Tuple{name='insertable',val=thisModule.default(jc.insertable,true)},
         Tuple{name='updatable',val=thisModule.default(jc.updatable,true)},
         Tuple{name='columnDefinition',val=jc.columnDefinition},
         Tuple{name='table',val=jc.table}
      }->select(tp|tp.val<>OclUndefined)
       ->collect(tp|Tuple{name=tp.name,exp=
            if (tp.val.oclIsKindOf(Boolean) ) then
               thisModule.genBooleanLiteral(tp.val)
            else
               thisModule.genStringLiteral(tp.val)
            endif
       })
```

```
      )
   );


helper context JAST!ASTNode def: genJoinColumns(cun: JAST!CompilationUnit,typ: JAST!
BodyDeclaration,cols: Sequence(UML!Element)) : JAST!Annotation =
   if (cols->isEmpty()) then
      OclUndefined
   else
      if (cols->size()=1) then

         self.addJoinColumns(typ,cols)->first()

      else
         typ.addAnnotation('JoinColumns',Sequence{

   Tuple{name='value',exp=thisModule.genArrayInitializer(self.addJoinColumns(cun,cols))}
            })
      endif
   endif;
helper context UML!Property def: addJoinTable(typ: JAST!BodyDeclaration,joinTable :UML!
Element) : JAST!Annotation =
let cun: JAST!CompilationUnit = thisModule.resolveTemp(self.class,'cun') in
   typ.addAnnotation('JoinTable',Sequence{
         Tuple{name='name',val=joinTable.name},
         Tuple{name='schema',val=joinTable.schema},
         Tuple{name='catalog',val=joinTable.catalog},
         Tuple{name='uniqueConstraints',val=
            typ.genUniqueConstraints(joinTable.uniqueConstraints,cun)
         },
         Tuple{name='joinColumns',val=
               if (joinTable.joinColumns->isEmpty()) then
               OclUndefined
               else

   thisModule.genArrayInitializer(cun.addJoinColumns(cun,joinTable.joinColumns))
               endif
               },
         Tuple{name='inverseJoinColumns',val=
               if (joinTable.inverseJoinColumns->isEmpty()) then
               OclUndefined
               else

   thisModule.genArrayInitializer(cun.addJoinColumns(cun,joinTable.inverseJoinColumns))
               endif
               }


      }
      ->select(tp|tp.val<>OclUndefined)
      ->collect(tp|Tuple{name=tp.name,exp=if (tp.val.oclIsKindOf(String)) then

   thisModule.genStringLiteral(tp.val)
                                                         else if
(tp.val.oclIsKindOf(UML!EnumerationLiteral)) then


   (tp.val.enumeration.name+'.'+tp.val.name).genNameStr()
                                                               else
                                                                     tp.val
                                                               endif
                                                         endif
      })
      );



rule oneToOne (typ: JAST!BodyDeclaration,pcl : UML!Property) {
using {
    cun: JAST!CompilationUnit = thisModule.resolveTemp(pcl.class,'cun');
    cascade: Sequence(UML!EnumerationLiteral) =
pcl.getStereotypeValue('AssociationMapping','cascade');
   joinCols :Sequence(UML!
Element)=pcl.getStereotypeValue('AssociationMapping','joinColumns');
```

```
}
do {

   typ.addAnnotation('OneToOne',Sequence{
         Tuple{name='mappedBy',val=  if (pcl.getOtherEnd().isNavigable()) then
                                                pcl.getOtherEnd().name
                                          else
                                                OclUndefined
                                          endif
         },
         Tuple{name='optional',val=  if (pcl.getLower()=1) then
                                                thisModule.genBooleanLiteral( false)
                                          else
                                                OclUndefined -- Default condition is
true
                                          endif
         },
         Tuple{name='fetch',val=
   pcl.mockNullEnum(pcl.getStereotypeValue('AssociationMapping','fetch'))},

         Tuple{name='cascade',val=
            if (cascade=OclUndefined) then
               OclUndefined
            else if (cascade->isEmpty()) then
                              OclUndefined
               else
               thisModule.genArrayInitializer(
                        cascade->collect(c|('CascadeType.'+c.toString()).genNameStr())  )
               endif
            endif
             }
         }
         ->select(tp|tp.val<>OclUndefined)
         ->collect(tp|Tuple{name=tp.name,exp=if (tp.val.oclIsKindOf(String)) then

   thisModule.genStringLiteral(tp.val)
                                                                  else if
(tp.val.oclIsKindOf(UML!EnumerationLiteral)) then


   (tp.val.enumeration.name+'.'+tp.val.name).genNameStr()
                                                                        else
                                                                              tp.val
                                                                        endif
                                                                  endif


         })


            );

   thisModule.addImport(cun,'javax.persistence.OneToOne');
   if (joinCols<>OclUndefined) {
      typ.genJoinColumns(cun,typ,joinCols);
      thisModule.addImport(cun,'javax.persistence.JoinColumns');
      thisModule.addImport(cun,'javax.persistence.JoinColumn');

   }


thisModule.genPrimaryKeyJoinColumn(pcl,typ,cun,pcl.getStereotypeValue('AssociationMappin
g','pkJoinCols'));

   if (typ.modifiers->select(m|m.oclIsKindOf(JAST!NormalAnnotation))
         ->exists(m|m.values->exists(v|v.name.identifier='cascade'))
      )
      thisModule.addImport(cun,'javax.persistence.CascadeType');

   if (typ.modifiers->select(m|m.oclIsKindOf(JAST!NormalAnnotation))
         ->exists(m|m.values->exists(v|v.name.identifier='fetch'))
      )
      thisModule.addImport(cun,'javax.persistence.FetchType');
}
```

```
}
rule manyToOne (typ: JAST!BodyDeclaration,pcl : UML!Property) {
using {
    cun: JAST!CompilationUnit = thisModule.resolveTemp(pcl.class,'cun');
    cascade: Sequence(UML!EnumerationLiteral) =
pcl.getStereotypeValue('AssociationMapping','cascade');
   joinCols :Sequence(UML!
Element)=pcl.getStereotypeValue('AssociationMapping','joinColumns');
}
do {
   typ.addAnnotation('ManyToOne',Sequence{
         Tuple{name='fetch',val=
   pcl.mockNullEnum(pcl.getStereotypeValue('AssociationMapping','fetch'))},

         Tuple{name='optional',val=  if (pcl.getLower()=1) then
                                                thisModule.genBooleanLiteral( false)
                                          else
                                                OclUndefined -- Default condition is
true
                                          endif
         },
         Tuple{name='cascade',val=
            if (cascade=OclUndefined) then
               OclUndefined
            else if (cascade->isEmpty()) then
                           OclUndefined
               else
               thisModule.genArrayInitializer(
                      cascade->collect(c|('CascadeType.'+c.toString()).genNameStr())  )
               endif
            endif
             }
         }
         ->select(tp|tp.val<>OclUndefined)
         ->collect(tp|Tuple{name=tp.name,exp=if (tp.val.oclIsKindOf(String)) then

   thisModule.genStringLiteral(tp.val)
                                                      else if
(tp.val.oclIsKindOf(UML!EnumerationLiteral)) then


   (tp.val.enumeration.name+'.'+tp.val.name).genNameStr()
                                                            else
                                                                  tp.val
                                                            endif
                                                      endif
         })
   );

   thisModule.addImport(cun,'javax.persistence.ManyToOne');
   if (joinCols<>OclUndefined) {
      typ.genJoinColumns(cun,typ,joinCols);
      thisModule.addImport(cun,'javax.persistence.JoinColumns');
      thisModule.addImport(cun,'javax.persistence.JoinColumn');

   }

   if (typ.modifiers->select(m|m.oclIsKindOf(JAST!NormalAnnotation))
         ->exists(m|m.values->exists(v|v.name.identifier='cascade'))
      )
      thisModule.addImport(cun,'javax.persistence.CascadeType');

   if (typ.modifiers->select(m|m.oclIsKindOf(JAST!NormalAnnotation))
         ->exists(m|m.values->exists(v|v.name.identifier='fetch'))
      )
      thisModule.addImport(cun,'javax.persistence.FetchType');
}

}
helper context String def: asField() : String =
   if (self.size()>1) then
      self.substring(1,1).toLower()+self.substring(2,self.size())
   else
      self
   endif;
```

```
rule genEnum {
from
   en : UML!Enumeration
to
   typ: JAST!EnumDeclaration (
      enumConstants<- typ.enumConstants ->append(en.getOwnedLiterals()),
      name<-thisModule.genSimpleNameStr(en.name)

   ),
   cun:JAST!CompilationUnit (
      package <- thisModule.genPackageDcl(en)
   )
do {
   (en.name+'=enum= ').println();
   cun.types<-cun.types->append(typ);
   }
}

rule genEnumLiterals {
from
   lit : UML!EnumerationLiteral
to
      typ: JAST!EnumConstantDeclaration (
      name<-thisModule.genSimpleNameStr(lit.name)
   )

}


rule genInterface {
from
   cl : UML!Interface
to
   typ: JAST!TypeDeclaration (
      superInterfaceTypes<- cl.general->collect(i|i.genAnyType(cl)),
      bodyDeclarations<- typ.bodyDeclarations
        ->append(cl.getFeatures()->select(f|f.oclIsKindOf(UML!Property))->collect(f|
thisModule.resolveTemp(f,'getter')))
        ->union(cl.getFeatures()->select(f|f.oclIsKindOf(UML!Property))->collect(f|
thisModule.resolveTemp(f,'setter')))
        ->union(cl.getFeatures()->select(f|f.oclIsKindOf(UML!Operation))->collect(f|
thisModule.resolveTemp(f,'method'))),
      name<-thisModule.genSimpleName(cl),
      interface<-true

   ),
   cun:JAST!CompilationUnit (
      package <- thisModule.genPackageDcl(cl)
   )
do {
   (cl.name+'=int= ').println();
   cun.types<-cun.types->append(typ);
   if (cl.visibility=#public)
      thisModule.genBodyModifiers(typ,'public');

   thisModule.addPersistenceImports(cl,cun);
   }
}
rule genInterfaceAttrs {
from
   p : UML!Property (
       (p.class.oclIsUndefined() and p.owner.oclIsKindOf(UML!Interface) )
      )
to
getter: JAST!MethodDeclaration (
      name <- thisModule.genSimpleNameStr(p.name.asProperty()),
      returnType<-p.genPropertyType()
   ),
setter: JAST!MethodDeclaration (
      name <- thisModule.genSimpleNameStr(p.name.asSetProperty()),
      returnType<-thisModule.genVoidType(p),
      parameters<-setter.parameters.append(thisModule.genSetterVariable(p))
   )
```

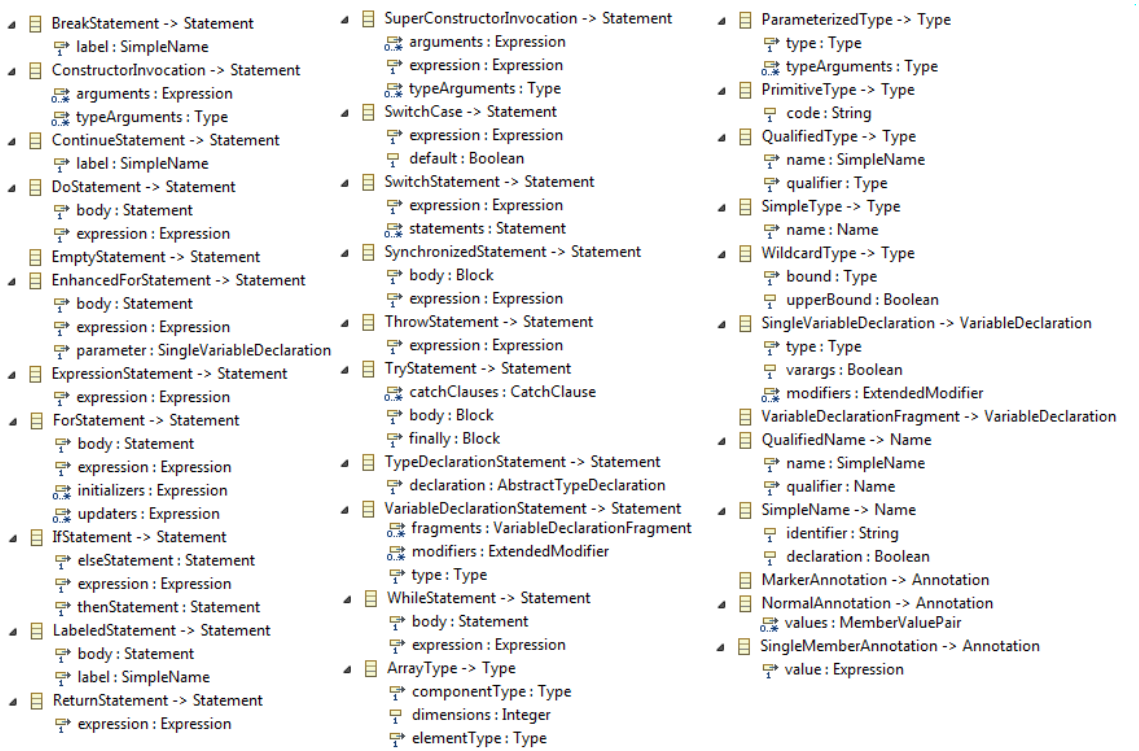
```
do {
   if (p.visibility=#public) {
      thisModule.genBodyModifiers(getter,'public');
      thisModule.genBodyModifiers(setter,'public');
   } else if (p.visibility=#private) {
      thisModule.genBodyModifiers(getter,'private');
      thisModule.genBodyModifiers(setter,'private');
   } else if (p.visibility=#protected) {
      thisModule.genBodyModifiers(getter,'protected');
      thisModule.genBodyModifiers(setter,'protected');
   }
   if (p.isStatic()) {
      thisModule.genBodyModifiers(getter,'static');
      thisModule.genBodyModifiers(setter,'static');
   }
   --method
   thisModule.addPropAnnotations(p,getter);
}
}
rule genInterfaceMethods {
from
   op : UML!Operation (
      (op.class.oclIsUndefined() and op.owner.oclIsKindOf(UML!Interface) )
      )
to
   method: JAST!MethodDeclaration (
      name <- thisModule.genSimpleNameStr(op.name),
      constructor<-op.isConstructor(),
      returnType<-if (op.type.oclIsUndefined()) then
                thisModule.genVoidType(op)
                else
                op.type.genAnyType(op)
                endif,
      parameters<-method.parameters.append(
         op.ownedParameter->select(p|not(p.direction=#return))->collect(p|
thisModule.genMethodParameter(p))
         )

   )
do {
   if (op.visibility=#public)
      thisModule.genBodyModifiers(method,'public');
   else if (op.visibility=#private)
      thisModule.genBodyModifiers(method,'private');
   else if (op.visibility=#protected)
      thisModule.genBodyModifiers(method,'protected');
   if (op.isAbstract())
      thisModule.genBodyModifiers(method,'abstract');
   if (op.isStatic())
      thisModule.genBodyModifiers(method,'static');

}

}

rule genGeneralClass extends genClass {
from
   cl : UML!Class  (
      UML!Generalization.allInstances()->select(g|g.general=cl)
      ->exists(g|g.hasStereotype('Joined') or g.hasStereotype('TablePerClass') or
g.hasStereotype('SingleTable')
         ) and

      not(cl.isEGenericType()))

using {
   discrType: UML!DiscriminatorType =
cl.getStereotypeValue('DiscriminatorColumn','discriminatorType');
   joined: Boolean = UML!Generalization.allInstances()->select(g|g.general=cl)
      ->exists(g|g.hasStereotype('Joined'));
   singleTab: Boolean = UML!Generalization.allInstances()->select(g|g.general=cl)
      ->exists(g|g.hasStereotype('SingleTable'));
   tableClass: Boolean = UML!Generalization.allInstances()->select(g|g.general=cl)
      ->exists(g|g.hasStereotype('TablePerClass'));
```

```
}
to
   inh : JAST!NormalAnnotation (
      typeName <- thisModule.genSimpleNameStr('Inheritance'),
      values<-inh.values->append(
         thisModule.newMemberValuePair(
            Tuple{na=inh,name='strategy',exp=(
               if (singleTab) then
               'InheritanceType.SINGLE_TABLE'
               else
               if (joined) then
                       'InheritanceType.JOINED'
               else
                       'InheritanceType.TABLE_PER_CLASS'
               endif
               endif).genNameStr()})

       )

   )

do {
   thisModule.addImport(cun,'javax.persistence.Inheritance');
   thisModule.addImport(cun,'javax.persistence.InheritanceType');
   typ.modifiers<-typ.modifiers->append(inh);


   if (cl.hasStereotype('DiscriminatorColumn')) {
      typ.addAnnotation('DiscriminatorColumn',
         Sequence{
   Tuple{name='name',val=cl.getStereotypeValue('DiscriminatorColumn','name')},


   Tuple{name='discriminatorType',val=thisModule.default(discrType,'STRING')},

   Tuple{name='length',val=cl.getStereotypeValue('DiscriminatorColumn','length')},


Tuple{name='columnDefinition',val=cl.getStereotypeValue('DiscriminatorColumn','columnDef
inition')}

               }
         ->select(tp|tp.val<>OclUndefined)
         ->collect(tp|Tuple
            {name=tp.name,exp=
               if (tp.val.oclIsKindOf(UML!EnumerationLiteral)) then

               (tp.val.enumeration.name+'.'+tp.val.name).genNameStr()
               else
               if (tp.name='length') then
                       thisModule.genNumberLiteral(tp.val)
               else
                       thisModule.genStringLiteral(tp.val)
               endif
               endif
            })
      );

      if
(cl.getStereotypeValue('DiscriminatorColumn','discriminatorValue')<>OclUndefined) {
            typ.addAnnotation('DiscriminatorValue',
               Sequence{Tuple{name='',


exp=thisModule.genStringLiteral(cl.getStereotypeValue('DiscriminatorColumn','discriminat
orValue'))
               }} );
         thisModule.addImport(cun,'javax.persistence.DiscriminatorValue');
      }
   }
   thisModule.doGenRules(cl,typ,cun); --"super"
   }
```

```
}


rule genTypeParameters {
from
   ctp : UML!ClassifierTemplateParameter (not(ctp.signature.oclIsUndefined()))
using {
   cun: JAST!CompilationUnit = thisModule.resolveTemp(ctp.signature.template,'cun');
   typ:  JAST!TypeDeclaration = thisModule.resolveTemp(ctp.signature.template,'typ');
}
to
   res : JAST!TypeParameter (
      name<-   thisModule.genSimpleNameStr(ctp.parameteredElement.name),
      typeBounds<-if (ctp.constrainingClassifier.oclIsUndefined()) then
               OclUndefined
               else

   ctp.constrainingClassifier.selectTypeCtx(ctp.signature.template).selectDataType()
               endif
      )
do {
   typ.typeParameters<-typ.typeParameters->append(res);

   }

}
```

ANEXO: METAMODELO JAS

Este anexo apresenta o metamodelo da árvore sintática da linguagem java (JAS) utilizado para transformação de modelos MD-JPA em Java. O metamodelo foi capturado na ferramenta Eclipse, relacionando visualmente as classes e propriedades armazenadas em XMI.


BreakStatement -> Statement
label : SimpleName
ConstructorInvocation -> Statement
arguments : Expression
typeArguments : Type
ContinueStatement -> Statement
label : SimpleName
DoStatement -> Statement
body : Statement
expression : Expression
EmptyStatement -> Statement
EnhancedForStatement -> Statement
body : Statement
expression : Expression
parameter : SingleVariableDeclaration
ExpressionStatement -> Statement
expression : Expression
ForStatement -> Statement
body : Statement
expression : Expression
initializers : Expression
updaters : Expression
IfStatement -> Statement
elseStatement : Statement
expression : Expression
thenStatement : Statement
LabeledStatement -> Statement
body : Statement
label : SimpleName
ReturnStatement -> Statement
expression : Expression
SuperConstructorInvocation -> Statement
arguments : Expression
expression : Expression
typeArguments : Type
SwitchCase -> Statement
expression : Expression
default : Boolean
SwitchStatement -> Statement
expression : Expression
statements : Statement
SynchronizedStatement -> Statement
body : Block
expression : Expression
ThrowStatement -> Statement
expression : Expression
TryStatement -> Statement
catchClauses : CatchClause
body : Block
finally : Block
TypeDeclarationStatement -> Statement
declaration : AbstractTypeDeclaration
VariableDeclarationStatement -> Statement
fragments : VariableDeclarationFragment
modifiers : ExtendedModifier
type : Type
WhileStatement -> Statement
body : Statement
expression : Expression
ArrayType -> Type
componentType : Type
dimensions : Integer
elementType : Type
ParameterizedType -> Type
type : Type
typeArguments : Type
PrimitiveType -> Type
code : String
QualifiedType -> Type
name : SimpleName
qualifier : Type
SimpleType -> Type
name : Name
WildcardType -> Type
bound : Type
upperBound : Boolean
SingleVariableDeclaration -> VariableDeclaration
type : Type
varargs : Boolean
modifiers : ExtendedModifier
VariableDeclarationFragment -> VariableDeclaration
QualifiedName -> Name
name : SimpleName
qualifier : Name
SimpleName -> Name
identifier : String
declaration : Boolean
MarkerAnnotation -> Annotation
NormalAnnotation -> Annotation
values : MemberValuePair
SingleMemberAnnotation -> Annotation
value : Expression